ICT-UNPA-38-2012

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

CONEXIONES ASPECTUALES DE REGLAS DE NEGOCIO CON SPRING

Lic. Graciela Beatriz Vidal

gbv1976@yahoo.com.ar

Universidad Nacional de la Patagonia Austral

UARG

2011

Resumen En toda organización existen restricciones, con el fin de que no se lleven a cabo acciones inválidas. Estas restricciones son denominadas Reglas de Negocio, las mismas han adquirido gran interés en la actualidad, llegando a constituir un activo de gran valor para las organizaciones. Por su volatilidad, se considera apropiado mantenerlas separadas de la funcionalidad principal de las aplicaciones, de esta manera se facilita su acceso, mantenimiento e incrementa su reutilización. La Programación Orientada a Aspectos, se ha propuesto como alternativa para encapsular las conexiones que permitan integrar las Reglas de Negocios a la funcionalidad. Existen diferentes enfoques utilizados, el objetivo de este trabajo es utilizar el framework Spring para realizar la integración de un conjunto de reglas definidas sobre una Subasta Electrónica como caso de estudio.

Palabras Clave: *Reglas de Negocio, Programación Orientada a Aspectos, separación de conceptos, AspectJ, Spring.*

ICT-UNPA-38-2012

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

1. INTRODUCCION

En todas las organizaciones existen restricciones, esto es a fin de que no sea posible llevar a cabo acciones inválidas. Estas restricciones son denominadas Reglas de Negocio (RN), pueden ser de tipo legal, político, competitivas, de oportunidad, etc.

Toda organización utiliza aplicaciones software para su funcionamiento, actualmente estas aplicaciones implementan un conjunto de RN para dar respuesta a los requerimientos del cliente. Las RN son muy volátiles, mucho más que la funcionalidad principal, por lo tanto es necesario modificar el software frecuentemente en diferentes módulos para cumplir con estos cambios. Estas modificaciones resultan invasivas y consumen mucho tiempo, lo que significa un mayor costo.

Hasta el momento, se han presentado varios enfoques para llevar a cabo la implementación de estas RN. Los cuales se basan en evaluar la condición de la RN dentro de la funcionalidad principal o invocar la RN desde la funcionalidad principal, causando en ambos casos código disperso y mezclado en los distintos módulos. Esto dificulta el reuso del software y su posterior mantenimiento.

La Programación Orientada a Aspectos (POA) [1], fue propuesta para mejorar la integración de RN con la funcionalidad principal. Se ha demostrado que la encapsulación de las conexiones en *aspectos,* mejora el reuso y posterior mantenimiento. Estos trabajos se han centrado en proveer soporte de implementación con lenguajes POA de características programáticas como AspectJ [2] y Jasco[3]. Sin embargo estas nuevas alternativas, también plantean nuevas limitaciones.

El presente trabajo apunta a explorar la implementación de conexiones con RN con el framework Spring [4], un contenedor liviano que se basa en la técnica de Inversión de Control. Spring ofrece varias alternativas para la implementación de aspectos, para este trabajo se seleccionó el soporte de aspectos POJOS puros, denominado soporte de esquemas, por ser declarativo y simple. El objetivo es analizar este tipo de soporte y la posibilidad superar las alternativas de conexión existentes, buscando mayor flexibilidad y planteando nuevas estrategias.

Se ha desarrollado una Subasta Electrónica como caso de estudio, para llevar adelante la experimentación. En principio se ha desarrollado la funcionalidad base, a la cual se añadieron diversas RN, que fueron integradas utilizando el soporte de esquemas de Spring POA.

Este trabajo, concluye los aspectos más importantes de una tesina de grado, realizada en el marco del Proyecto de Investigación "Estrategias para la Integración y Conexión de Reglas de Negocio con Aspectos" (29/A242), en la Universidad Nacional de la Patagonia Austral.

La estructura de este informe está organizado de la siguiente manera: Sección 1 corresponde a la introducción; en la Sección 2 se define el Marco Conceptual en el cual se desarrolla la definición de Reglas de Negocios y su clasificación, además el Enfoque de RN, ventajas de su utilización y al final de esta sección el Bussines Rules Group; en la Sección 3 el Marco Teórico, el cual profundiza en la POA, AspectJ y Spring; en la Sección 4 se presenta Marco de Referencia, se mencionan los enfoques existente para implementar RN; en la Sección 5 Materiales y métodos, se da a conocer la metodología utilizada para el desarrollo de este trabajo; en la Sección 6 se realiza un Análisis de Resultados y para finalizar en la Sección 7 se presentan las conclusiones.

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

2. MARCO CONCEPTUAL

2.1 REGLAS DE NEGOCIOS

En todos los ámbitos de la sociedad, como pueden ser el de la salud, sectores económico-financieros, medios de comunicación, la educación, por mencionar solo algunos, se interactúa con aplicaciones software de distinto tipo. Estas aplicaciones proveen la funcionalidad principal requerida por cada uno de estos ámbitos.
Estas aplicaciones generalmente requieren modificaciones hasta lograr su incorporación total dentro de la organización, sin embargo existe otro tipo de cambios que la afectarán permanentemente, que están directamente relacionados con los objetivos de la empresa.
El código necesario para llevarlos a cabo aparecer en varios lugares y por lo tanto cada vez que se produzca una modificación, será necesario acceder a ellos para realizar la actualización.
Investigaciones al respecto, demuestran que lo conveniente es identificar estos cambios para luego encapsularlos y separarlos de la funcionalidad para representarlos como Reglas. Es así como aparece el concepto de Reglas de Negocio (RN), muy popular en los últimos tiempos dentro del ámbito del desarrollo de aplicaciones.
Se puede definir una RN como una directiva, una restricción o una decisión que se aplica a la lógica de negocios de una empresa y que afecta la aplicación que utiliza la misma. Esta sería una definición algo informal.
El denominado Business Rules Group (2000) define a una RN, como una declaración que define o restringe algún aspecto del negocio de manera que nunca sea posible llevar a cabo acciones no válidas [5], pretendiendo reafirmar la estructura de negocios, controlar la conducta y su comportamiento.
Las RN se aplican en puntos bien definidos de una aplicación y tienen como finalidad realizar alguna acción que aporte al cumplimiento de los objetivos de la organización y que no permita realizar acciones no válidas, esto puede referirse a estrategias de mercado, aprovechar oportunidades, introducir nuevos productos o servicios, atraer nuevos clientes.
La estructura general de una RN puede representarse mediante una sentencia de la forma *"if condition then action"* es decir, si se cumple una condición simple o compleja, se lleva a cabo una acción específica.

2.2 CLASIFICACION

Existen muchos tipos de clasificación de RN [6][7][8], teniendo en cuenta la definición formal antes mencionada. Sin embargo, no existe un único esquema para categorizar las RN.
Un esquema de clasificación de RN es de gran ayuda para descubrir reglas, analizarlas y proyectarlas.
Un esquema de clasificación de reglas sirve para cumplir con los siguientes objetivos [9]:

- Exponer la gama completa de reglas que la organización considera capturar.
- Simplificar y guiar los negocios de manera eficientemente de tal forma que se pueda descubrir nuevas reglas.
- Permitir expresar las reglas de negocios por tipo utilizando algún patrón para una mayor claridad.

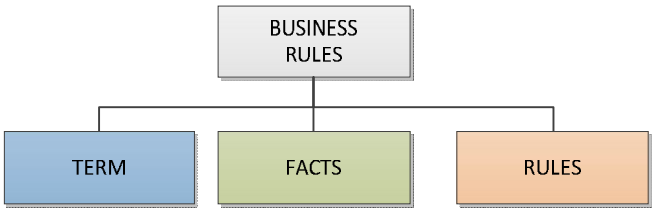

Figura 1. Esquema de clasificación de alto nivel.

La clasificación representada en Figura 1 es muy conocida, de alto nivel y se describe en detalle a continuación:
Term: se refiere a un sustantivo o una frase sustantiva con una definición sobre algo. Los *terms* se refieren a otro tipo de reglas de negocios. Un *term* puede definir alguno de los siguientes elementos:

- Un concepto, a cerca de un cliente.
- Una propiedad de un concepto, tal como límite de crédito de un cliente.
- Un valor de un dato.
- El valor de un grupo de datos.

Facts (structural assertions): es una declaración o afirmación que conecta *terms* como una preposición o frase de acuerdo a observaciones sobre acciones relevantes del negocio. Algunos ejemplos son:

- Un cliente puede hacer un pedido especial.
- Poner un artículo en línea.
- Clientes clasificados por límite de crédito de un cliente.

Los *terms* y *fact* son la semántica detrás de las reglas. Permite expresar una regla en lenguaje natural.

Rules: en esta tercer clasificación, se despliegan o ponen a prueba las capacidades del enfoque de RN antes mencionado. Se sabe que una regla es una declaración que aplica la lógica o la computación sobre determinada información. El resultado de este tipo de reglas sirve para descubrir nueva información o tomar decisiones y realizar una determinada acción.
Dentro de esta última categoría, podemos distinguir otra clasificación, la misma se representa en Figura 2.

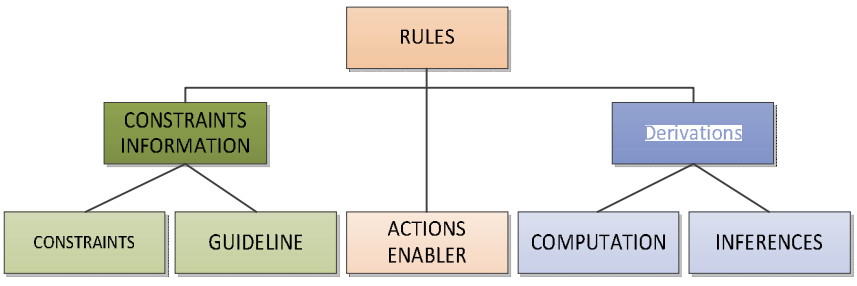

Figura 2. Esquema de clasificación de RN.

Constraints (action assertions): se refiere a restricciones y/o controles sobre el comportamiento de la organización. Se analiza el resultado de una condición y de acuerdo a este resultado, se realiza una acción determinada.
Guideline: es una declaración que expresa alguna advertencia, sobre alguna circunstancia que puede ser verdadera o falsa.
Actions Enabler: es una declaración que verifica alguna condición, si su resultado es verdadero inicia otro evento, muestra un mensaje o realiza otra actividad.
Derivations: este tipo de reglas proporcionan el resultado de alguna operación, algún tipo de información sobre la cual posteriormente se realiza una acción. Dentro de esta categoría se pueden distinguir dos más:

- **Computation**: es una afirmación que provee de un algoritmo para obtener algún valor, este algoritmo puede contener operaciones matemáticas.
- **Inferences**: es una declaración completa que testea una condición y sobre resultados verdaderos, establece como verdaderos nuevos datos.

2.3 ENFOQUE DE RN

El enfoque de RN [9] es simplemente una manera formal de administrar y automatizar las RN de una organización. Es una metodología mediante la cual se captura, analiza, publica, automatiza y cambia las RN de acuerdo a las estrategias de negocios de la organización.
El enfoque de RN propone cumplir con cuatro objetivos: *separar la regla de la aplicación principal, hacer el seguimiento de las reglas para la política y toma de decisiones, exteriorizar reglas al público por medio de audiencias y posicionar las reglas al cambio* [9].
Si se logra cumplir con estos cuatro objetivos, las RN pueden utilizarse dentro de un Sistema de soporte a Negocios. Por lo general estos sistemas están implementados utilizando técnicas Orientadas a Objetos (OO)[10].
Este enfoque tiene además, tres aspectos únicos que lo caracterizan, el primero es la introducción al seguimiento de las reglas, esto permite manejarlas como un componente lógico, separado, exteriorizado de las interacciones y los datos de la aplicación.
El segundo aspecto único, es que representa la integración de la orientación a objetos, la ingeniería de información y las reglas representadas formalmente.
El tercer aspecto es que esta metodología proporciona una base de reglas para la gestión que motivarán el cumplimiento de objetivos, desarrollo de estrategias y puesta en marcha de tácticas. La organización puede evaluar las reglas aplicadas y verificar si los resultados finales son los deseados.
El resultado de utilizar esta metodología es un sistema de RN, un sistema automatizado en el cual las RN se mantienen en forma separada, lógica y tal vez físicamente de otros aspectos del sistema. El objetivo del enfoque de RN es el reconocimiento de las RN como un activo de gran valor para las organizaciones.

2.3.1 VENTAJAS DEL ENFOQUE DE RN

Algunas de las ventajas de la utilización del Enfoque de RN en una organización se exponen a continuación:

- **Simplicidad**: este enfoque es simple de comprender tanto, para las personas de negocios como para quienes desarrollan aplicaciones. El concepto de regla es sencillo y la clasificación de las mismas es bastante intuitiva.
- **Base teórica**: el concepto de regla tiene como origen el modelo relacional. Existen muchas *experiencias teóricas y prácticas* con reglas en el campo de la ingeniería del conocimiento.
- **Pocos conceptos no técnicos**: el centro del enfoque de RN es la regla en sí misma. Hay pocos conceptos adicionales alrededor de las reglas, tales como decisiones, patrones de reglas, familias de reglas y clausulas de reglas.
- **Desarrollo fácil de aplicaciones**: con la aparición de la tecnología de RN, también surgieron varias clases de procesamiento fácil de reglas. Con algunos productos, la aplicación llama al producto regla cuando la aplicación necesita tomar alguna decisión basada en la ejecución de una regla. Con otros productos, el producto regla apoya la ejecución de las reglas como consecuencia de cambio de datos. En ambos casos, el profesional de reglas de negocios define las reglas una vez y son utilizadas, en las aplicaciones cuando es necesario.
- **Independencia**: permite expresar las reglas utilizando sintaxis independiente de la tecnología de desarrollo de la aplicación.
- **Reusabilidad**: una regla puede ser utilizada en varias transacciones, interactuar con sistemas de bases de datos, browser y otros tipos de software. Es decir, se define e implementa una regla o un conjunto de ellas, solo una vez y están disponibles para varios propósitos.
- **Diseño simple de sistemas**: el enfoque de RN apunta a separar el procesamiento principal de la ejecución de reglas. Se crean dos flujos, uno para las reglas y otro para la aplicación principal.
- **Reglas dinámicas**: si bien los cambios en las reglas no pueden aplicarse instantáneamente, es posible cambiar sistemas de RN fácilmente. Un desarrollador de reglas puede cambiar una o varias reglas a la vez y hacer que estos cambios, estén disponibles para aquellas transacciones relevantes de la aplicación. De este modo, un sistema de RN puede convertirse en una plataforma de cambios de negocios. Las reglas en sí mismas, pueden ser un instrumento de adaptabilidad de negocios.
- **Performance**: los productos de reglas son diseñados específicamente para administrar y ejecutar reglas. Por lo tanto, contienen internamente la lógica para hacerlo de la mejor manera y así entregar productos de alta performance.
- **Entregas incrementales del sistema**: un sistema de RN puede ser entregado fácilmente realizando entregas incrementales. Si la primera incluye una base sólida de datos, las próximas entregas en el futuro irán mejorando y agregando la funcionalidad de la estructura existente.

2.4 BUSINESS RULES GROUP

El Business Rules Group[5], también conocido como GUIDE Business Rules Project se organizó en 1993 con el fin de formalizar un enfoque para identificar y articular las reglas que definen la estructura y controlan las operaciones de una organización.

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

Por mucho tiempo los analistas de sistemas fueron capaces de definir una organización en base a los datos y funciones que la misma requiere para su funcionamiento. Sin embargo han desatendido algunas acciones bajo las cuales funcionan las organizaciones. Por lo general estas acciones son identificadas, en el momento en que es necesario realizar la modificación en el software. Algunas de estas reglas y funciones que forman parte de la estructura de una empresa, son documentadas pero otras no. El Business Rules Project propone llevarlo a cabo, incluyendo todas aquellas RN que no fueron documentadas en el pasado.

La tarea de definir las RN es conocida, sin embargo se desarrollaron técnicas y herramienta para brindar apoyo cuando surjan problemas en esta definición. Estas técnicas incluyen métodos formales para describir reglas rigurosamente y herramientas que traducen estos formalismos, directamente al código fuente de la aplicación. El GUIDE Business Rules Project fue creado con cuatro propósitos específicos:

- Definir y describir RN, como así también, conceptos asociados para entonces poder determinar qué es y qué no es una RN.
- Definir un modelo conceptual de RN para expresar, en términos significativos para profesionales de la informática, solamente que es una RN y como debe ser aplicada.
- Proporcionar una base rigurosa para modificar RN existentes en las aplicaciones.
- Proporcionar una base rigurosa para generar nuevos sistemas basados en RN definidas formalmente.

3. MARCO TEÓRICO

3.1 PROGRAMACION ORIENTADA A ASPECTOS

La programación ha transitado por diferentes escenarios desde sus orígenes hasta la actualidad. Cada etapa se asocia a un paradigma, en el cual nuevos conceptos se incorporan y por lo tanto, la implementación cambia, dando lugar a la utilización de nuevos tipos de lenguajes y mecanismos que permiten llevar a cabo la integración de estos nuevos componentes de las aplicaciones software.
En un principio no existía separación de conceptos, datos y funcionalidad se mezclaban sin una línea divisoria clara. Posteriormente, se aplicó la descomposición funcional que pone en práctica el principio de "*divide y vencerás*" identificando las partes manejables como funciones que se definen en el dominio del problema. Luego surgió la Programación Orientada a Objetos (POO) [10], considerada un gran paso en la ingeniería del software, permitiendo construir sistemas complejos, utilizando el principio de descomposición.
La Programación Orientada a Aspectos (POA)[1] es un paradigma de programación relativamente nuevo, su propósito es lograr una adecuada modularización de aplicaciones y posibilitar una mejor separación de conceptos. La POA permite capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada eliminando las dependencias inherentes entre cada uno de los módulos. De esta manera se logra razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Los objetivos que propone la POA principalmente son, separar conceptos y minimizar las dependencias entre ellos. En consecuencia se obtienen las siguientes ventajas:

- Un código menos enmarañado, más natural y reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y mantener.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusables y que se puede acoplar y desacoplar cuando sea necesario.

La POA incorpora el concepto de *aspecto*. A continuación de presentan algunas definiciones:

> *Un aspecto es una unidad que se define en términos de información parcial de otras unidades [11].*

> *Un aspecto es una unidad modular, que se dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular de diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa.[12]*

Un aspecto no se puede encapsular claramente en un procedimiento con los lenguajes tradicionales. No pueden ser unidades de descomposición funcional del sistema, sino propiedades que afectan al rendimiento o la semántica de los componentes. Los aspectos son la unidad básica de la POA y pueden definirse como las partes de una aplicación que describen las cuestiones claves relacionadas con la semántica esencial o el rendimiento.
Respecto a una aplicación y su implementación basada en un lenguaje basado en procedimientos, una aplicación se implementará como:

- un componente, aquel que puede ser encapsulado de forma clara en un procedimiento, como pueden ser los objetos, métodos, procedimientos, API. Se entiende de forma clara, está bien localizada, es de fácil acceso y que se deja componer como sea necesario.
- un aspecto, aquel que no se puede encapsular de forma clara en un procedimiento. Los aspectos no suelen ser unidades funcionales que obtengamos al descomponer el sistema, son propiedades que afectan el rendimiento o el comportamiento de componentes. Como ejemplos de aspectos, se pueden citar: patrones de acceso a memoria, sincronismo de objetos concurrentes, control de errores y manejo de excepciones, utilización de caché.

Estos conceptos permiten establecer cuál es el objetivo de la POA: proporcionar técnicas para separar componentes y aspectos unos de otros, dotando de mecanismos que hacen posible abstraer estos para después componerlos dando resultado al sistema final.
Está claro que hay una relación entre los componentes y los aspectos, por lo tanto, el código de los componentes y de estas nuevas unidades de programación tiene que interactuar de alguna manera. Para que ambos (aspectos y componentes) se puedan componer, deben tener algunos puntos comunes, conocidos como puntos de enlace y debe existir algún modo de componerlos. Los puntos de enlace, son una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. El responsable de realizar este proceso de mezcla se conoce como tejedor, "*weaver*". El tejedor se encarga de mezclar los diferentes mecanismos de abstracción y composición que aparecen en los lenguajes de aspectos y componentes ayudándose de los puntos de enlace. Las clases y aspectos se pueden entrelazar de dos formas distintas: de manera estática o dinámica.

3.2 AspectJ

AspectJ es una extensión compatible de Java TM orientada a aspectos y de propósito general, con una nueva clase de módulos llamado *aspectos* [13]. Los aspectos pueden interceptan las clases, las interfaces y a otros aspectos. Mejoran la separación de competencias permitiendo localizar de forma limpia los conceptos de diseño del corte. A diferencia de las clases, no se pueden crear instancias de los aspectos. Un aspecto es una clase, exactamente igual que las clases Java, pero con la diferencia que pueden contener constructores de corte, que no existen en Java, como se muestra en la Figura 3.

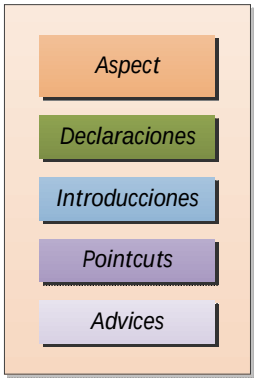

Figura 3. Componentes de un aspecto en *AspectJ*.

Los *Aspects* (aspectos) son constructores que trabajan al cortar de forma transversal la estructura de las clases de forma limpia y cuidadosamente diseñada.

Pointcuts : también llamados cortes, capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, señalización y gestión de excepciones. Los cortes no definen acciones, simplemente describen eventos. Un *Pointcut* está formado por dos partes, separadas por dos puntos. En la parte izquierda (a) se define el nombre del corte y el contexto del corte. La parte derecha, del Ejemplo 1, (b,c) define los eventos de corte.

```
pointcut corte(): call(void Line.moveBy(int, int));
      (a)              (b)              (c)
```

Ejemplo 1.

Los cortes se utilizan para definir el código de los aspectos utilizando avisos. A los descriptores de eventos de la parte derecha de la definición del corte se les llama designadores. Un designador puede ser:

- Un método.
- Un constructor.
- Un manejador de excepciones.

Se pueden componer utilizando los operadores o (“||”), y (“&”) y no (“!”).

Advices: definen la asociación de los “*pointcuts*” para insertar la implementación del aspecto que se ejecuta en esos puntos bien definidos. Estos puntos pueden identificarse por cortes con nombre (b), en el Ejemplo 2 o por cortes anónimos.

```
   (a)          (b)
before() : corte() {

System.out.println(“call-before sobre Line.moveBy()”);(c)
}
```

Ejemplo 2.

El cuerpo de un *advice* se puede añadir en distintos puntos del código(a), cada uno de los cuales se define de mediante alguna de las siguientes palabras clave:

before: se ejecuta justo antes que las acciones asociadas con los eventos del corte.
after: se ejecuta justo después de que lo hayan hecho las acciones asociadas con los eventos del corte.
catch: se ejecuta durante la ejecución de las acciones asociadas con los eventos definidos en el corte se ha producido una excepción del tipo definido en la apropiada cláusula catch.
finally: se ejecuta justo después de la ejecución de las acciones asociadas con los eventos del corte, incluso aunque se haya producido una excepción durante la misma.
around: atrapan la ejecución de los métodos designados por el evento. La acción original asociada con el mismo se puede invocar utilizando thisJoinPoint.runNext( ).
Los ***advices*** se pueden declarar con el modificador "*static*", esto indica que se ejecuta el mismo aviso para todas las instancias de las clases designadas. Si se omite indica que solamente se ejecuta para ciertas instancias.
Introducciones y declaraciones: Las introducciones "*Introduction*" se utilizan para añadir elementos completamente nuevos, permitiendo modificar la estructura de una clase dada. Es decir afecta la estructura estática de las clases. Entre los elementos que se pueden añadir se encuentran:

- Un nuevo método a la base.
- Un nuevo constructor.
- Un atributo.
- Varios de los elementos anteriores a la vez.
- Varios de los elementos anteriores en varias clases.

3.3 SPRING

Spring [14] es un framework de código abierto para desarrollo de aplicaciones Java. Spring logró popularizar ciertas técnicas de diseño como la Inversión de Control [4]. En el año 2004 tuvo una muy buena aceptación al ofrecer su propio framework de POA (Aspect Oriented Programming, AOP), consiguiendo hacer más popular el paradigma de programación en la comunidad Java.
Las aplicaciones Java típicamente, están compuestas por un número de objetos que colaboran entre sí para formar una aplicación apropiada. Los patrones de diseño, Inversión de Control e Inyección de Dependencia (Dependency Injection), proporcionan solución a los problemas de dependencias (acoplamiento) entre objetos de una aplicación. La utilización de interfaces y la aparición de frameworks fue el primer paso para minimizar estas dependencias entre componentes.
La Inversión de Control (IoC) es una técnica que invierte el flujo de control tradicional, delegando el control de la aplicación a una API o sistema externo. Este concepto se encuentra en el modelo de programación orientado a eventos Swing y AWT. Formalmente la Inversión de Control se representa con la frase "*No me llames yo te llamo*"[4]. El sistema externo es el que llama a la aplicación.

Spring se identifica con una forma de IoC denominada *Inyección de Dependencia* (Dependency Injection) que consiste, no solamente en ceder la gestión de instancias de objetos, sino también la inyección de dependencias (sub-partes) en la cual los servicios de gestión e inyección se identifican y localizan por medio de mecanismos no programáticos, externos al código de la aplicación, por ejemplo un archivo XML.
Spring framework, toma las mejores prácticas que han sido probadas durante los últimos años en numerosas aplicaciones y formalizadas como patrones de diseño codificándolos como clases que se pueden integrar a las aplicaciones propias.
El uso de patrones, contribuye a formalizar un vocabulario común entre los programadores, estandarizar el modo en que se realiza el diseño y facilita el aprendizaje a futuras generaciones de diseñadores condensando el conocimiento existente.

3.3.1 POA SPRING

Uno de los componentes clave de Spring es POA framework. No existen dependencias entre el contenedor y POA. Spring POA es implementado en Java puro y no necesita ningún proceso especial de compilación**.**
El soporte POA con Spring presenta cuatro alternativas [15]:

- Spring POA basados en proxies en tiempo de ejecución (disponible en todas las versiones de Spring).
- Aspectos AspectJ dirigidos por anotaciones (sólo disponible en la versión 2.x de Spring).
- Aspectos POJO puros (sólo disponible en la versión 2.x de Spring).
- Aspectos AspectJ inyectados (disponible en todas las versiones de Spring).

En las tres primeras propuestas Spring POA soporta solamente intercepción a nivel de ejecución de métodos, no implementa intercepción de campos o atributos, mientras que la última alternativa de aspectos AspectJ inyectados, permite la inyección a nivel de constructor y atributos.
Spring POA se enfoca precisamente en permitir una integración entre la implementación POA y Spring IoC para ayudar a resolver problemas comunes de una aplicación. Por lo tanto es frecuente utilizar la implementación POA con el contenedor IoC de Spring. Los aspectos son configurados utilizando la sintaxis normal para definir *beans*, esta es una diferencia crucial con respecto a otras implementaciones POA. Podría suponerse que Spring intenta competir con AspectJ, para alcanzar una completa solución POA, sin embargo se cree que ambos framework son válidos y que más que competidores, son complementarios.
La configuración basada en Aspectos POJOS puros, también denominada basada en esquemas XML, es tal vez la más utilizada, esto puede estar relacionado a la manera declarativa en que se realiza la definición. Un aspecto es simplemente un objeto Java definido como un bean en el contexto de aplicación Spring. El comportamiento es capturado en los atributos y métodos del objeto. La información de los *pointcut* y *advice* se obtiene de la configuración XML.
Los elementos utilizados para definir esta configuración se muestran en la Tabla 1.

Tabla 1. *Elementos para la configuración POA XML*

Elemento	Propósito
<aop:after>	Define un aviso after
<aop:after-returning>	Define un aviso after-returning
<aop:after-throwing>	Define un aviso after-throwing
<aop:around>	Define un aviso around
<aop:before>	Define un aviso before
<aop:aspect>	Define un aspecto
<aop:config>	Elemento de mayor nivel. Contiene al resto de los elementos
<aop:pointcut>	Define un pointcut

Los aspectos, advices y pointcuts deben estar ubicados entre los elementos `<aop:config>…</aop:config>`, pueden existir varias `<aop:config>` dentro de un mismo archivo.
Un aspecto se declara utilizando el elemento `<aop:aspect>`, Listado 1, se especifican los atributos 'id' y 'ref', donde 'id' permite identificar el aspecto y 'ref' indica el bean al cual debe referenciar al aspecto, que en definitiva es una clase simple.

```
<aop:config>
                <aop:aspect id="myAspect" ref="aBean">
                                        ...
                </aop:aspect>
</aop:config>
```

Listado 1.

Esta configuración, permite implementar la DI descripta anteriormente. Distintos aspectos, beans, pueden referenciar, por lo tanto inyectar, esta misma dependencia.
Un pointcut puede ser declarado dentro de un elemento `<aop:config>`, como se representa en la Listado 2, permitiendo que esta definición sea utilizada, en consecuencia, entrecruzada por distintos aspectos.

```
<aop:config>
      <aop:pointcut id="businessService"
                expression="execution(* com.xyz.myapp.service.*.*(..))"/>
</aop:config>
```

Listado 2.

Se observa que en `'expression'` se utiliza AspectJ para definir el *pointcut*. Otra forma de declarar un *pointcut* puede ser dentro del aspecto. El 'id' identifica el *pointcut*, este puede ser obtenido y ser pasado al advice, como se observa.

Los *pointcut* pueden contener expresiones simples o compuestas, para obtenerlas combinan sub-expresiones, utilizando los 'and', 'or' y 'not', que reemplazan a los símbolos, "&&","||" y "!" respectivamente.
Todos los *advices* mencionados anteriormente pueden declararse utilizando la misma sintaxis, deben ser definidos entre los elementos `<aop:aspect>`, como se muestra en el Listado 3.

```
<aop:aspect id="beforeExample" ref="aBean">
                <aop:before
                           pointcut-ref="dataAccessOperation"
                           method="doAccessCheck"/>
                        ...
</aop:aspect>
```

Listado 3.

En todos los ejemplos presentados, el atributo `'method'` hace referencia al método dentro del aspecto, que ejecutará una determinada tarea, siempre que se ejecute la expresión especificada en el *pointcut.*
Se conoce que *around* es más poderoso que los demás tipos de *advice*. Debido a que puede actuar en lugar de la ejecución de un método. Se declara utilizando un elemento `<aop:around>`. Este *advice* proporciona argumentos con información referente al contexto en el cual es utilizado.
El primer argumento de este *advice* es del tipo ProceedingJoinPoint. Dentro del cuerpo del advice se debe llamar al método `proceed`(), cuando la ejecución del método interceptado deba finalizar correctamente.
El método `proceed`() puede ser llamado solamente, cuando el método en el cual es invocado reciba como parámetro un objeto de tipo `ProceedingJoinPoint`, como se muestra en Listado 4, primero la definición del *advice*:

```
<aop:aspect id="aroundExample" ref="aBean">
            <aop:around
                        pointcut-ref="businessService"
                        method="doBasicProfiling"/>
                      ...
</aop:aspect>
```

Listado 4.

La definición del método `doBasicProfiling` debería ser:

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws
Throwable {
//
                Object retVal = pjp.proceed();
//
return retVal;
}
```

Listado 5.

En ocasiones los métodos especificados en el atributo `'method'`, requieren de argumentos, los que pueden ser detallados explícitamente en la definición del *advice*, se utiliza el atributo `'arg-names'`, que acepta una lista de argumentos, todos estos separados por coma.
Teniendo en cuenta que los *pointcuts* pueden definirse una vez y varios aspectos pueden interceptarlo, es necesario un mecanismo que permita ordenar la ejecución de estos aspectos, de manera que sean ejecutados correctamente y los resultados obtenidos sean los esperados.
Para manejar estas situaciones entre aspectos existen varias alternativas, la primera es que los aspectos en cuestión implementen la interface `Ordered`. Aquel aspecto cuyo getOrder() sea mayor, será el aspecto con mayor prioridad y por lo tanto el que se ejecute primero. Otra alternativa es definirlo directamente sobre la configuración XML, especificando una prioridad asignando un valor al atributo `'order'` del aspecto.
Las Introducciones, conocidas como declaración de inter-type en AspectJ, permiten declarar qué objetos implementarán una Interface, tendrán un nuevo 'padre', que proporcionará nuevos atributos o métodos en un determinado momento (*advice*). Se debe codificar la Interface, y la clase que la implementará con los atributos y los métodos que serán agregados a objetos específicos. Una introducción se hace utilizando el elemento `<aop:declare-parent>`, definido siempre dentro de los elementos `<aop:aspect>`. El elemento `<aop:declare-parent>` posee tres atributos más en los cuales se especificará, la clase que recibirá los nuevos atributos y métodos, la interface que implementará esta clase y la clase donde se encuentran los atributos y métodos a introducir. Esta introducción se lleva a cabo, cuando se produce un evento particular en la aplicación, por lo tanto junto al elemento `<aop:declare-parent>` se debe definir el *advice* que indica cuando hacer la introducción.

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">
    <aop:declare-parents
         types-matching="com.xzy.myapp.service.*+"
         implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
       default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>
    <aop:before
              pointcut="com.xyz.myapp.SystemArchitecture.businessService()
                        and this(usagTracked)"
             method="recordUsage"/>
</aop:aspect>
```

Listado 6.

En Listado 6, se define que los objetos que implementen la interface service, también implementan la interface UsageTracked.
En el caso de estudio de este trabajo, se ponen en práctica los conceptos expuestos en esta sección con el fin de indagar sobre esta alternativa de implementación, para luego analizar los resultados.
En tal caso se describirán ventajas y desventajas, se realizarán comparaciones con otros enfoques, proponiendo alternativas de solución que darán lugar a trabajos futuros.

4. . MARCO DE REFERENCIA

4.1 ENFOQUES PARA IMPLEMENTAR RN

Como se mencionó anteriormente, existen varios enfoques para la implementación de RN en una aplicación, a continuación se detallan brevemente algunos de ellos:

Tradicional: puede considerarse el más antiguo, por ser el que se utilizó hasta hace un tiempo, pero también es el que presenta más problemas. Justamente con el fin de solucionarlos es que aparecieron los que se desarrollarán más adelante. En este caso, la RN se encuentra dentro del módulo en la cual será aplicada, junto al código de la funcionalidad principal, codificada con algún lenguaje O.O como el resto de la aplicación. Este código se encuentra repetido en muchos módulos, si realizamos un cambio en la especificación de la RN, es necesario modificar todos los módulos involucrados, lo que resulta engorroso y demanda mucho tiempo.
Debido a que las RN son muy volátiles, su mantenimiento se dificulta ocasionando problemas para la reutilización, esto se puede observar en el Listado 7.

```
public Factura facturarCompra(Vector ítems)
{
       int value;
        void metodoI(){
             if (condicion)
                // aplicar descuento
             System.out.println("Se aplicó el Descuento");
               }
        void metodoII(){
             if (condicion)
               // asignar puntos
             System.out.println("Se asignó puntaje");
         }
}
```

Listado 7.

Enfoque de Motores de Reglas: Un motor de RN es un componente, el cual a partir de información inicial y un conjunto de reglas, detecta cuales se deben aplicar y en qué momento, todo esto proporcionando una infraestructura desacoplada del código fuente [16]. Básicamente, un motor de RN consta de tres elementos: un conjunto de reglas, el espacio de trabajo (o contexto que tiene) y el procesador de reglas. Las reglas son sentencias de la forma if *condition*() then *action*(). El espacio de trabajo es la información que el motor utilizará para decidir que reglas deben aplicarse. Los motores pueden ser capaces de manejar conflictos entre reglas, asignando prioridades a cada regla y utilizando técnicas FIFO o LIFO. Algunos motores que podemos mencionar son Jess[17] para Java, utilizado con fines académicos, Blaze Advisor [18] un producto flexible y abierto, y JRule[19].

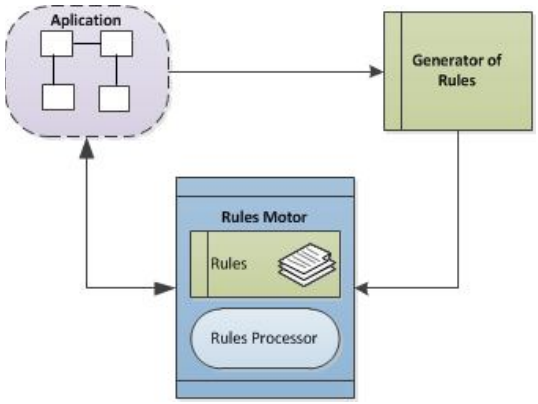

Figura 4.

Patrones de objetos regla: este enfoque fue creado para representar las RN como objetos [20]. Cada regla es representada como una clase simple, que implementa los métodos *condition*(), action() y apply(), lo que se representa en Listado 8. Este patrón puede llegar a ser bastante elaborado, si se utiliza con otros patrones de diseño como Composite, Strategy[21], etc. con el propósito de minimizar las dependencia y mejorar la reutilización. Sin embargo esta alternativa tiene por desventaja que requieren demasiados objetos intermedios, lo que se torna difícil de manejar.

```
class RN {
          boolean condition (..) {}
          void apply()
          {
                  if condition()
                      action();
          }
          void action() {}
}
```

Listado 8.

```
class REDescuento extends RN
{
          boolean condition (..){}
          if (condicion())
                  return true,
          else
                  return false;
          }
         void action () {
                 ……
            }
```

Listado 9.

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

```
class X {
      RN rn;
      void metodo XX ( … )
      {
            rn.apply(…);
       }
```

Listado 10.

Los últimos dos enfoques apuntan a hacer las reglas explícitas y separadas de la funcionalidad principal, Sin embargo, se observa que el problema del código entremezclado en la funcionalidad principal, no es solucionado en su totalidad por ninguno de ellos. Aunque las RN se encuentran físicamente separadas del código de la funcionalidad principal, el código que realiza la conexión de las mismas, se encuentra aún disperso por el código de la aplicación.

Para lograr separar este código, es necesario encapsularlo. Lo que permite el desarrollo de alternativas de integración cuando las RN, la funcionalidad principal o ambas cambien debido al mantenimiento, reúso o evolución de la aplicación.

5. MATERIALES Y MÉTODOS

5.1 POA PARA CONECTAR RN

Por sus características y objetivos se utiliza el enfoque POA [1] para realizar la conexión entre las RN y la funcionalidad principal. La POA se enfoca en lograr una mejor *separación de conceptos* [22], aunque por otro lado afirma que existen algunos conceptos de las aplicaciones que no pueden ser modularizados limpiamente ya que se encuentran aislados y mezclados por distintas partes del sistema.
La POA propone encapsular este código en un módulo denominado *aspecto*. Un aspecto actúa como conector entre la funcionalidad principal y la RN, más precisamente, dentro de un aspecto se crea un objeto de tipo RN, Listado 11, el cual invoca al método que implementa el código de la RN , Listado 12 . Asimismo, se debe definir en el código XML un bean para cada aspecto y RN, como así también la IoC y entonces lograr la conexión a las RN, Listado 13 y 14 . Encapsular este código no es una tarea sencilla, existen muchas cuestiones que necesitan ser tratadas con detenimiento para realizarlo correctamente. Cibran se ha destacado por enfocarse particularmente en AspectJ, gracias a su arduo trabajo [23][24][25], logró identificar una serie de requerimientos a la hora de encapsular una conexión de RN como así también, varios inconvenientes presentes en este lenguaje, ambos aportes se describen a continuación.

```
package connectors;

public class ConnectorAssignPoints  {

      AssignPoints points;

      public void triggerRule(JoinPoint obj) {

            OfferI offer = (OfferI)obj.getThis();

            if (points.apply(offer))
            {
                  System.out.println("SE APLICO REGLA AssignPoints, SE
ACTUALIZO PUNTAJE ");
                  System.out.println();
            }

      }
      public AssignPoints getPoints() {
            return points;
      }
      public void setPoints(AssignPoints points) {
            this.points = points;
      }

}
```

Listado 11.

```
package rules;
…
```

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

```
public class AssignPoints implements BusinessRule
{
	private UserI user;
	private OfferI offer;
	private ManagerUserI manageruser;
	public void action() {
		user.setPoints(user.getPoints()+10);
		System.out.println("REGLA AssignPoints SE ACTUALIZO PUNTAJE ");
		System.out.println();
	}

	public boolean apply(Object obj) {
		offer = (OfferI)obj;
		if (user!=null)
		{
			if (this.condition())
				{
					this.action();
					return true;
				}
				else
				{
				System.out.println("REGLA AssignPoints NO HAY PUJA
GANADORA ");
					System.out.println();
					return false;
				}
		}
		else
		{
		System.out.println("REGLA AssignPoints NO HAY PUJA GANADORA ");
			System.out.println();
			return false;
		}
	}
….	public boolean condition() {

		user = this.offer.getWinner().getBidder();

		if ( user != null)
			return true;
		else
			return false;
	}
}
```

Listado 12

```
<bean id="connectorAssignPoints" class="connectors.ConnectorAssignPoints"
scope="prototype" >
		<property name="points" ref="asignarPuntos" />

	</bean>
```

Listado 13

```
<bean id="asignarPuntos" class="rules.AssignPoints" >
            <property name="manageruser" ref="gestorUsuarios" />
            <property name="user" ref="usuario" />
            <property name="offer" ref="oferta" />
      </bean>
```

Listado 14

5.2 REQUERIMIENTOS PARA ENCAPSULAR UNA CONEXIÓN DE RN

A continuación se describen los requerimientos necesarios para encapsular limpiamente este código de integración de una regla. Estos requerimientos son independientes del lenguaje de implementación y de la tecnología de desarrollo que se utilice:

- Encapsulamiento de *crosscutting*: las reglas son aplicadas en puntos bien definidos de una aplicación y el código de integración es idéntico en todos estos puntos. Esto implica agregar código para obtener información que será empleada por las reglas y adicionar parámetros que posibiliten que estos datos estén disponibles cuando la regla sea aplicada. Como resultado de esa integración, se obtiene código mezclado y esparcido por toda la funcionalidad principal. Por lo tanto, es necesaria la encapsulación de este código en módulos separados.
- Identificación de eventos dinámicos: los eventos en los cuales es necesario emplear una regla son puntos dinámicos de la aplicación, tales como la invocación a un método o el acceso a un atributo. Como las reglas siempre cambian, se agregan o se eliminan permanentemente, en general no es posible anticiparse a los eventos en los cuales ellas necesitarán ser aplicadas. Es necesario un mecanismo que permita especificar eventos dinámicos en tiempo de ejecución.
- Capturar y exponer información en eventos dinámicos: algunas reglas necesitan información global del sistema, en estos casos el acceso a la misma es directo. Otras necesitan información de objetos que se encuentran dentro del alcance del evento que activa la regla. Es decir los objetos están disponibles en los puntos en los cuales se activa la regla, pero no son accedidos directamente por las reglas. Es necesario un mecanismo que capture y pase estos objetos a las reglas.
- Introducir información anticipada en comportamiento: las reglas pueden necesitar información específica para ser aplicadas correctamente. Algunas reglas requieren información que no está prevista al momento de su diseño e implementación. En tales casos es necesario un mecanismo que introduzca nuevos objetos, atributos y operaciones en la funcionalidad principal sin tener que alterar manualmente el código original. Estas adiciones deberían ser encapsuladas para poder ser reusables y modificadas fácilmente.
- Compartir información de contexto: cuando se obtiene la información para la activación de una regla, es necesario pasarla al evento en el cual la regla será aplicada. Esto requiere de un mecanismo de comunicación entre los diferentes módulos que encapsulan la integración de la regla, de manera que la información sea compartida cuando sea requerida.

- Controlar inicialización e instanciación: debe ser posible controlar la inicialización e instanciación de las reglas, teniendo en cuenta que estas son muy volátiles, éste es un requerimiento muy importante.
- Especificación de estrategias de precedencia, combinación y ejecución: puede suceder que algunas reglas interfieran sobre otras o que dependan de otras reglas. Es necesario controlar esta colaboración entre reglas, a fin de que sean aplicadas correctamente. Se asignan prioridades a las reglas para especificar precedencia en la ejecución.
- Actualización dinámica de crosscutting: la volatilidad de las reglas, demanda que estas sean modificadas frecuentemente, pero esto no debería paralizar la aplicación. La actualización e integración se debe realizar de manera transparente a los demás componentes de la aplicación.

5.3 PROBLEMAS CON ASPECTJ

En cuanto a los problemas identificados por Cibrán [23][24][25], están directamente relacionados con la conexión entre las RN y la funcionalidad principal en aplicaciones OO, lo que se detallan a continuación:

- AspectJ permite encapsular la conexión de una RN en un aspecto, con lo cual se logra una gran reusabilidad e independencia, aunque por otro lado, genera una proliferación de aspectos que resulta difícil de manejar.
- Por lo general, las relaciones entre aspectos se expresan dentro de los aspectos en cuestión, por lo tanto esto disminuye la reusabilidad y composición.
- La reusabilidad del código es posible en AspectJ con la herencia, también se observa que posee características muy poderosas de bajo nivel para solucionar diversos problemas. Sin embargo, en muchas ocasiones es posible solucionar con las mismas características, cuestiones semánticamente diferentes, con lo cual se dificulta la comprensión y portabilidad de los programas.
- Si bien, la herencia permite la reutilización de código, los *pointcuts* en AspectJ son frágiles al definirse directamente sobre un evento concreto en la ejecución, por lo tanto son menos reusables.
- AspectJ es estático, esto significa que cualquier modificación sobre el código fuente de los aspectos y los componentes que éstos interceptan requiere su posterior recompilación.

Estos son algunos de los inconvenientes que presenta AspectJ en cuanto a la implementación de conexiones de RN con la funcionalidad principal. A tal fin en este trabajo se proponen nuevas estrategias que se enfocan en tratar de dar solución o al menos mejorar las deficiencias que presenta AspectJ.

5.4 CASO DE ESTUDIO

La metodología utilizada para el desarrollo de este trabajo, se enfocó en la experiencia sobre un caso de estudio, la integración y conexión de RN con la funcionalidad principal de una aplicación.
Las RN son restricciones existentes en cualquier organización y afectan directamente a sus aplicaciones, cada modificación demanda tiempo y por ende significa incrementar costos. Estos son algunos de los inconvenientes que se presentan al implementar RN en una aplicación, es por ello que se está trabajando desde hace un tiempo en la difícil tarea de encapsular las RN y separarlas de la funcionalidad principal, de manera que sea más fácil y rápido su mantenimiento e integración. Se considera la POA como una alternativa para realizar esta integración, con el fin de minimizar dependencias y acoplamiento, obteniendo de esta manera, aplicaciones más reutilizables. Por lo expuesto se pretende experimentar estrategias para la integración de las RN con aspectos utilizando Spring, un contenedor liviano que se basa en la técnica de Inversión de Control y una implementación de desarrollo según el paradigma Orientado a Aspectos.
Para este trabajo se ha seleccionado el soporte de aspectos POJOS puros, también conocido como soporte de esquemas, por ser el más declarativo y simple en su configuración. El principal objetivo es tratar de superar las alternativas de conexión existentes, buscando mayor flexibilidad de integración como así también la adquisición de nuevos conocimientos en el desarrollo de este tipo de estrategias.
El caso de estudio es una “Subasta Electrónica” sobre la cual se deberán, de ser posible, integrar RN que se aplican en este ámbito. Por su comportamiento, una subasta puede cambiar sus RN continuamente, varias veces al mes, a la semana y en algunos casos dentro de un mismo día. Es un caso interesante para analizar, no solo porque la cantidad de RN es importante sino también porque estas pueden variar mucho en poco tiempo, lo que requiere de su modificación e integración permanentemente.
La aplicación es una Subasta que gestiona un conjunto de Ofertas y en la cual se crean Usuarios y Ofertas. Tanto para crear una Oferta como para Pujar, se requiere estar registrado como usuario, quienes pueden realizar Pujas sobre las Ofertas activas en la Subasta, solo son aceptadas las Pujas cuyo valor sea mayor al precio base de la Oferta. El valor de una Puja puede ser modificado y solo pueden ser canceladas mientras la Oferta esté activa. Cada Oferta tiene como atributos usuario, producto, valor base, tipo, estado, pujas, pre-ganador, ganador, fecha y hora de inicio y final y ganancia.
Transcurrido el tiempo de cada Oferta se verifica si existe una Puja ganadora, se asigna un ganador, se cierra la oferta y se calcula la ganancia. El ganador es aquel que ofertó el mayor valor en su Puja.
Únicamente se pueden modificar los datos de una Ofertas cuando su estado es inactiva.
Se utilizarán RN para verificar si los usuarios están registrados, calcular ganancias de la subasta, aplicar descuentos, asignar categorías de usuarios, asignar puntos a los ganadores según su categoría, definir categorías de usuarios, controlar posibles valores extremos de pujas, controlar duración de ofertas activas, definir puja ganadora en caso de empate, definir duración máxima de una puja, extender duración de la oferta si no existieran pujas, controlar el tiempo para cancelar una puja realizada, etc.
El diagrama de Casos de Uso, en la Figura 5 representa el funcionamiento de la Subasta Electrónica sobre la cual se aplicarán las RN con Spring.

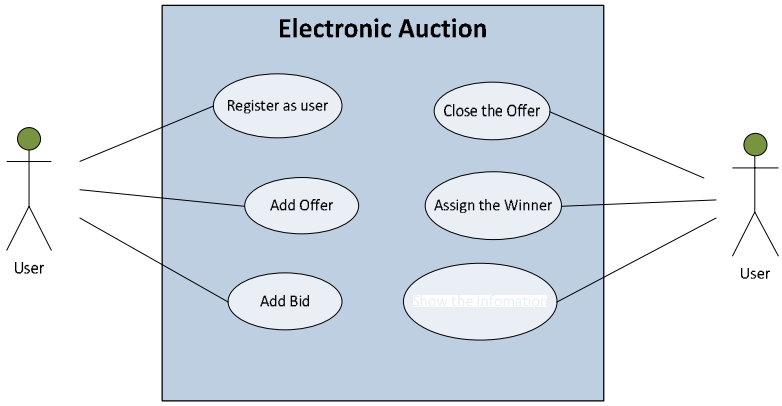

Figura 5. *Diagrama de Casos de Uso de la Subasta Electrónica.*

Otra forma de representar las relaciones entre los componentes de la Subasta Electrónica es a través de un diagrama de Clases, en el cual se representan las clases que conforman la aplicación, con sus atributos y operaciones.

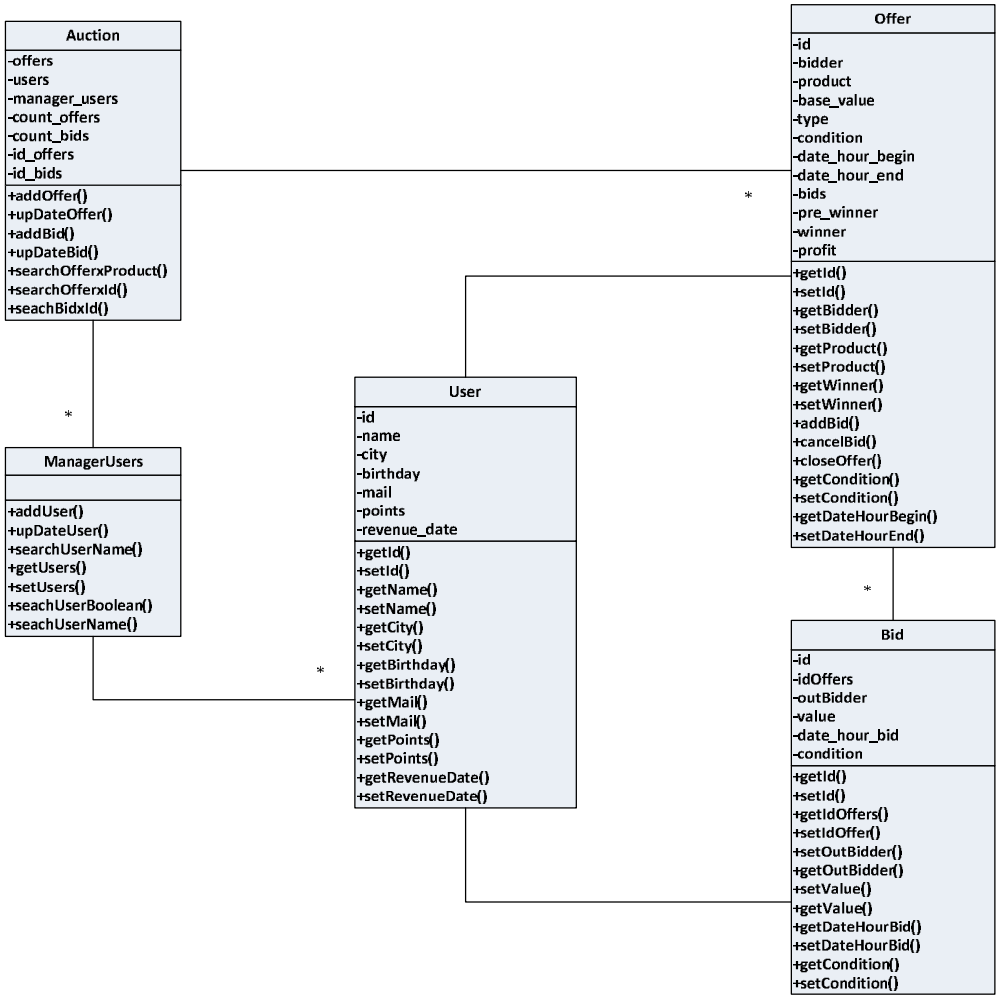

Figura 6. *Diagrama de Clases de la Subasta Electrónica*.

5.5 DEFINICION DE RN

Existen muchas formas de definir formalmente las RN de una organización, dependiendo de la cantidad, su naturaleza y complejidad . Asimismo se reconocen varias categorías en las cuales se pueden encuadrar. Cada una define distintas categorías y esquemas que nos permiten documentar el conjunto de RN de una organización. Esto facilitará el mantenimiento de las mismas, el acceso a su descripción y funcionamiento, además contribuirá a realizar modificaciones rápidamente cuando sea necesario. Seleccionar una categoría y esquema adecuado para definir un conjunto de Reglas, facilita la comprensión de las mismas.
Algunas de las RN definidas sobre la Subasta Electrónica son:

- Se controla que el usuario este registrado para agregar Ofertas y Pujas.
- Si el valor de una Puja supera el 200% del valor base de la Oferta, la Puja no es agregada.
- Al cerrar una Oferta, si existe un ganador, se calcula una ganancia del 3% sobre el valor base de la Oferta.
- Al cerrar una Oferta, si existe un ganador, y el valor de la Puja ganadora es mayor a $10000, se calcula una ganancia del 1% sobre el valor base de la Oferta.
- Si cierra una Oferta y no existen Pujas, se puede extender el la duración hasta un máximo establecido.

Teniendo en cuenta la clasificación de RN expuesta en Sección 2.1, se definirán los *term* y *fact* que conforman las RN.

Term

```
<auction> IS DEFINED AS <entidad subasta>
<offer> IS DEFINED AS <entidad oferta>
<bid> IS DEFINED AS <entidad puja>
<user> IS DEFINED AS <entidad usuario>
<base_value_offer> IS DEFINED AS <valor base que toma una oferta>
<date_hour_begin_offer> IS DEFINED AS <fecha y hora que se inicia una
oferta>
<date_hour_end_offer> IS DEFINED AS <fecha y hora que finaliza una oferta>
<profit> IS DEFINED AS <ganancia de la oferta que queda para la subasta>
<winner_bid> IS DEFINED AS <puja que ganadora de la oferta>
<bid_value> IS DEFINED AS <valor que toma una puja>
<bid_condition> IS DEFINED AS <estado en que se encuentra una puja>
<bid_condition> MUST BE IN LIST <”activa”,”cancelada”>
<date_hour_bid> IS DEFINED AS <fecha y hora que se crea una puja>
<points_user> IS DEFINED AS <puntaje del usuario>
<category_user> IS DEFINED AS <categoría del usuario>
<category_user> MUST BE IN LIST <”oro”,”plata”,”bronce”>
<date_hour_system> IS DEFINED AS <fecha y hora actual del sistema>
<max_duration_offer> IS DEFINED AS <la duración máxima de una oferta es de
7 días >
```

Fact

```
<offer> HAS A PROPERTY OF <base_value_offer>
<offer> HAS A PROPERTY OF <date_hour_begin_offer>
<offer> HAS A PROPERTY OF <date_hour_end_offer>
<offer> HAS A PROPERTY OF <winner_bid>
<offer> HAS A PROPERTY OF <profit>
<bid> HAS A PROPERTY OF <bid_value>
<bid> HAS A PROPERTY OF <bid_condition>
<bid> HAS A PROPERTY OF <date_hour_bid>
<user> HAS A PROPERTY OF <points_user>
<user> HAS A PROPERTY OF <category_user>
```

Y por último, se definen los eventos sobre los cuales se aplicarán las RN:

Event

```
<createOffer> IS DEFINED AS <cuando se crea una oferta por un usuario>
<closeOffer> IS DEFINED AS <cuando el sistema cierra la oferta al cumplirse
el tiempo de duración de la misma>
<createBid> IS DEFINED AS < cuando se crea una puja por un usuario>
<cancelBid> IS DEFINED AS < cuando el usurario cancela una puja>
<winningBidExists> IS DEFINED AS <cuando existe una puja ganadora>
<registreredUser> IS DEFINED AS <cuando el usuario está registrado>
<assingPointUser> IS DEFINED AS <cuando se asignan puntos al usuario>
<addAtributeUser> IS DEFINED AS <cuando se agrega un atributo al usuario>
```

Posteriormente se utiliza el lenguaje natural y un esquema con palabras reservadas lo que ayuda considerablemente en su implementación. Dentro de este esquema, se utilizan los elementos antes definidos. Además serán numeradas para una fácil localización y acceso, la Tabla 3 representa, como podría organizarse un catálogo de RN de una organización.

Nombre	**Lenguaje Natural**	**Nro.**
Ganancia Base	*Al cerrar una oferta si hay puja ganadora, se obtiene una ganancia del 3% del valor base.*	**RN #1**
WHEN <closeOffer> **IF** <hayPujaGanadora> **THEN** <profit> = <profit> + (<base_value_offer> * 3 / 100)		
Nombre	**Lenguaje Natural**	**Nro.**
Ganancia Superior	*Al cerrar una oferta, hay puja ganadora y si el valor de la puja supera los 10000 pesos se obtiene una ganancia del 1% del valor base.*	**RN #2**
WHEN <closeOffer> **IF** <bid_value> > 10000 **THEN** <profit> = <profit> + (<base_value_offer> / 100)		
Nombre	**Lenguaje Natural**	**Nro.**
Puja Máxima	*Al crear una puja si el valor de la puja supera el 200% del valor base de la oferta, la puja se cancela.*	**RN #3**
WHEN <createBid> **IF** <bid_value> >= <base_value_offer> + (<base_value_offer> * 200 / 100) **THEN** <bid_condition> **MUST BE** "cancelada"		
Nombre	**Lenguaje Natural**	**Nro.**
Asignar Puntos	*Al cerrar una oferta según la categoría del usuario que gano la puja, se cargará una cantidad de puntos determinada.*	**RN #4**
WHEN <close Offer> **IF** < winningBidExists > **THEN** <points_user> = <points_user> + 10		

Nombre	Lenguaje Natural	Nro.
Categoría Usuario	*Al cerrar una oferta si el usuario que gano la puja supera los 50 puntos obtiene una categoría. Esta se irá modificando de acuerdo a la cantidad de puntos.*	**RN #5**
WHEN <closeOffer> **IF** < winningBidExists > & <points_user> > 50 **THEN**<addAtributeUser> <category_user> **MUST BE** "bronze" **IF** < winningBidExists > & <points_user> > 100 **THEN** <category_user> **MUST BE** "silver"		
Nombre	**Lenguaje Natural**	**Nro.**
Verificar Usuario	*Se verifica si el usuario esta registrado antes de agregar una oferta o una puja.*	**RN #6**
WHEN <createOffer> ||<createBid > **IF** <registreredUser> **THEN** <crearOferta>||<crearPuja>		

Tabla 3. *Catálogo de RN*.

5.6 CLASIFICACION DE RN

Para las RN utilizadas en este caso de estudio se aplicó la clasificación mencionada en sección 2.2.

Nro.	Nombre	Categoría
RN#1	Ganancia Base	*Computation*
RN#2	Ganancia Superior	*Computation*
RN#3	Puja Máxima	*Constraint*
RN#4	Asignar Puntos	*Computation*
RN#5	Categoría Usuario	*Constraint*
RN#6	Verificar Usuario	*Constraint*

5.7 DISEÑO DE RN CON SPRING

Habiendo definido las RN, se procede a la implementación de algunas de ellas, se propone Spring como lenguaje para llevarla a cabo. Se utiliza la definición de esquemas para realizar la conexión entre las reglas y la funcionalidad principal. Esta propuesta responde al siguiente diagrama, en el cual se puede observar la estructura de una aplicación que implementa sus RN utilizando conexiones con Spring.

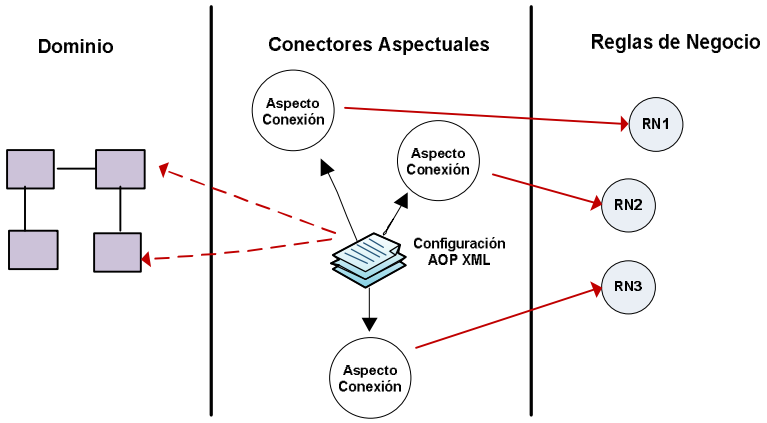

Figura 7. *Diagrama de los elementos con los que interactúan los conectores aspectuales*

Por un lado se implementan las clases que componen la aplicación principal, en la cual se llevan a cabo los eventos que activan las RN, en la Figura 7, el Dominio, para el cual se utiliza Java. Por otro lado están definidas las RN, para que ambos componentes se relacionen entran en escena los Conectores Aspectuales y la configuración AOP XML.
Los conectores se basan en la configuración POA definida en los archivos XML para llevar a cabo la ejecución de las RN cuando corresponda.

5.8 IMPLEMENTACION DE RN

Para llevar a cabo la implementación utilizada en este caso de estudio, se definió una Plantilla y tres pasos que serán expuestos en detalle a continuación.
Algunas de la RN implementadas sobre la Subasta Electrónica son:

RN#1	*Al cerrar una Oferta, si existe un ganador, se calcula una ganancia del 3% sobre el valor base de la Oferta.*
Evento	*La RN#1 se activa cuando cierra la Oferta.*
Implementación de regla: BaseProfitBR	

```
public class BaseProfitBR implements BusinessRule
{
	private OfferI offer;

	public boolean apply(Object offer)
	{
		this.offer = (OfferI)offer;

		if (this.condition())
		{
			this.action();
			return true;
		}
		else
			return false;
```

```
	}

	public boolean condition()
	{
		if (offer.getWinner() != null)
			return true;
		else
			return false; }

	public void action()
	{
	offer.setProfit(offer.getProfit()+(float)(offer.getBaseValue()*0.03));
	}

	public void applyProceed(Object obj) {

	}
}
```

Implementación del conector: ConnectorProfitRule

```
public class ConnectorProfitRule implements Serializable
{
	BaseProfitBR profit;
	OfferI offer;
	JoinPoint jp;
	public void triggerRule(JoinPoint obj) {

		offer = (OfferI) obj.getThis();
		profit.apply(offer);
	}

	public OfferI getOffer()
	{
		return this.offer;
	}
	public BaseProfitBR getProfit() {
		return profit;
	}
	public void setProfit(BaseProfitBR profit) {
		this.profit = profit;
	}
	public void setOffer(OfferI offer) {
		this.offer = offer;
	}
}
```

Configuración XML del conector ConnectorProfitRule

```
<bean id="connectorProfitApply" class="connectors.ConnectorProfitRule" >
	<property name="profit" ref="baseprofit" />
</bean>

<bean id="baseprofit" class="rules.BaseProfitBR" />
```

Configuración POA XML

```
<aop:config>

  < aop:pointcut id="applyProfit" expression="execution(*
                                    model.Offer.closeOffer(..))" />
 < aop:aspect ref="connectorProfitApply"  order="5" >
```

```
        < aop:after pointcut-ref=" applyProfit " method="triggerRule" />
 </ aop:aspect>

</ aop:config>
```

Es necesario definir *pointcut* y aspectos entre elementos del tipo <aop:config>…<aop:config>. El *pointcut applyProfit* define que el evento sobre el cual se aplicará la regla es al ejecutarse el método *closeOffer()* de la clase *Offer*. El aspecto hace referencia al *bean connectorProfitApply* que define al conector y se indica el momento en el cual se aplicará la regla, en este caso *after*, después de cerrar la oferta.
Luego, se asocia este *advice* con el *pointcut applyProfit* y en el atributo 'method', se especifica el método del conector que se ejecutará. El atributo 'order' del aspecto se explicará más adelante en esta sección.
La segunda RN también se activa cuando se cierra la Oferta, en este caso verifica el valor de la Puja ganadora, si este supera un valor X, se calcula una ganancia adicional a la anterior.

RN#2	*Al cerrar una Oferta, si existe un ganador, y el valor de la Puja ganadora es mayor a $10000, se calcula una ganancia del 1% sobre el valor base de la Oferta.*
Evento	*La RN#2 también se activa cuando se cierra la Oferta.*

Implementación de la regla UpperProfitBR

```
public class UpperProfitBR implements BusinessRule
{
	private OfferI offer;

	public boolean apply(Object offer)
	{

		this.offer = (OfferI)offer;
		if (this.condition())
		{
			this.action();
			return true;
		}
		else
		{
			return false;
		}
	}

	public boolean condition()
	{
	if (offer.getWinner()!= null && offer.getWinner().getValue()  > 10000)
			return true;
		else
			return false;
	}

	public void action()
	{
		float calculation = (float)(offer.getBaseValue() * 0.01);
		offer.setProfit(offer.getProfit()+calculation);
```

```
	}
}
```

Implementación del conector ConnectorProfitUpper

```
public class ConnectorProfitUpper
{

	ProfitUpper profitUpper;

	public void triggerRule(JoinPoint obj) {

		OfferI offer = (OfferI) obj.getThis();
		profitUpper.apply(offer);

	}
	public boolean condition() {
		return false;
	}

	public ProfitUpper getProfitUpper() {
		return ProfitUpper;
	}

	public void setProfitUpper(ProfitUpper ProfitUpper) {
		this.ProfitUpper = ProfitUpper;
	}
}
```

Configuración XML del conector ConnectorProfitUpper

```
<bean id="connectorUpperProfit" class="connectors.ConnectorProfitUpper" >
	<property name="upperprofit" ref="upperProfitBR" />
</bean>

<bean id="upperProfitBR" class="rules.UpperProfitBR" />
```

Configuración POA XML

```
<aop:config>

<aop:pointcut id="applyProfit" expression="execution(*
                                    model.Offer.closeOffer(..))" />
    <aop:aspect ref="connectorUpperApply" order="6">
        <aop:after pointcut-ref=" applyProfit " method="triggerRule" />
    </aop:aspect>

</aop:config>
```

En la configuración POA XML, se presentada se observa el atributo ‘order’. Las reglas *UpperProfitBR* y *BaseProfitBR* se aplican sobre el mismo *pointcut applyProfit*, cerrar oferta. Spring permite controlar la ejecución de las reglas utilizando el atributo ‘order’, asignando un valor a este atributo en aquellos aspectos que se asocian al mismo *pointcut* se puede definir la precedencia de unos sobre otros. Aquel aspecto que tenga mayor valor se ejecuta en primer lugar. Si es necesario cambiar este orden, se modifica simplemente este valor.

5.9 PROBLEMAS EN LA IMPLEMENTACION CON SPRING

Un requerimiento importante para poder encapsular el código de la conexión de una RN es la capacidad de guardar algún dato de un objeto dentro de un aspecto para ser utilizado posteriormente.
En ocasiones, esta información puede ser vital para aplicar una RN correctamente. No se trata de aplicar una RN, en este caso el mecanismo para almacenar y recuperar la información está asociado a eventos sobre los cuales se aplican las Reglas.
Se guarda un atributo de Offer para ser utilizado en otros eventos. Sobre el conector *ConnectorRuleValidateUsers*, que verifica los usuarios antes de agregar una oferta, se agregan los métodos para almacenar y recuperar la información necesaria.
Esta información se obtiene del objeto de tipo JoinPoint que será definido en la configuración XML.

Evento	*Cuando se agrega una oferta a la subasta, se guarda un atributo del objeto Offer.*

Implementación del conector ConnectorRuleValidateUsers

```
public class ConnectorRuleValidateUsers
{
       UsersValidate controlUser;
       OfferI Offer, info;
       BidI Bid;
       private UserI User;

       ..................
       public void retrieve (JoinPoint jp)
       {
              this.info = (OfferI) jp.getArgs()[0];
       }
       public OfferI getInfo() {

             return info;

       }      ..................
}
```

Se define el *poincut "guardInfo"* sobre el método *addOffer()*. Se definen *advice* dentro del aspecto que referencia al conector *ConnectorRuleValidateUsers* en el cual se definieron los métodos *retrieve()* y *getInfo()*.
Estos *advice* referencian distintos *pointcut*, *"guardInfo"* y *"applyProfit"* (definido anteriormente para aplicar ganancia), que están relacionados a otras reglas, esto se hace al solo fin de comprobar si realmente la información es almacenada y luego puede ser recuperada.

Configuración XML

```
<aop:pointcut id="guardInfo" expression="execution(*
                              model.Auction.addOffer(..))" />
```

Configuración POA XML

```
<aop:config>

<aop:aspect ref="connectorUsersValidate" >
      <aop:after pointcut-ref="guardInfo" method="retrieve" />
</aop:aspect>

<aop:aspect ref="connectorUsersValidate">
      <aop:after pointcut-ref="applyProfit" method="getInfo" />
</aop:aspect>

</aop:config>
```

Las pruebas demostraron que la información se almacena, pero luego se pierde, es decir que no se mantiene asociada al aspecto. Esto podría estar asociado a la instanciación de objetos en Spring. Por cada referencia a un *bean* definido en la configuración XML, se crea un nuevo objeto con lo cual se pierden las instancias anteriores.
Este inconveniente de Spring podría resolverse mediante la implementación de Aspectos AspectJ dirigidos por anotaciones, disponible en la versión 2.x de Spring y sería tema de estudio para trabajos futuros.

6. RESULTADOS

6.1 ANALISIS DE RESULTADOS

Uno de los principales objetivos de este trabajo era implementar nuevas alternativas de integración de RN a la funcionalidad principal de una aplicación, para realizar un análisis de los resultados obtenidos es necesario hacer una comparación de algunos conceptos relevantes con las alternativas existentes. A tal fin se toma como referente el enfoque que propone AspectJ, el lenguaje Orientado a Aspectos más popular y maduro en la actualidad, que extiende de Java.

- ***Modificar e integrar una RN***

Sabiendo que las RN son muy volátiles, se requiere que la modificación e integración de las mismas sea fácil y rápida de realizar. Cambiar una regla por otra en AspectJ requiere modificar el código fuente del conector aspectual que dispara la RN, en Spring esto no es necesario. Gracias a la IoC la RN es inyectada al conector sin necesidad de modificar el código fuente del aspecto.
Otra alternativa es modificar en la configuración XML, la referencia al bean en el cual se define la RN. Esto sucede porque los desarrolladores de AspectJ no tienen control sobre la inicialización e instanciación de objetos, en Spring esto se maneja desde el Application Context.

- ***Desacoplamiento y reutilización***

AspectJ permite desacoplar las diferentes partes que conforman el conector de las RN en aspectos separados, lo que provoca una proliferación de aspectos difícil de manejar.
En Spring la separación de las distintas partes del conector, se realizan mediante su modelo de esquemas. Los *pointcuts* se definen en la configuración POA XML, permitiendo diferentes formas de reutilización y las distintas sentencias de ejecución en el aspecto. Lo que significa que la reutilización de *pointcuts* en Spring no genera costos adicionales de mantenimiento.
En AspectJ la reutilización se consigue gracias a la herencia, sin embargo los *pointcut* son frágiles debido a que se definen sobre un evento concreto (*join-point*) y el pasaje de contexto en la ejecución son menos reusables.
Spring tiene la capacidad de manejar el contexto de forma muy transparente en la definición de *poincuts* y *advices*, esto sin dudas mejora considerablemente la reutilización.

- ***Compilación de código fuente***

Una modificación en AspectJ, requiere re-compilación del código fuente tanto de los aspectos como de los componentes interceptados, en Spring algunas modificaciones no requieren modificar el código fuente, limitándose a pequeños cambios en la configuración XML, sin necesidad de volver a compilar.

- ***Control del orden de ejecución***

Cuando dos o más RN están relacionadas al mismo evento, en AspectJ esto se puede controlar mediante la sentencia *"declare precedence"* dentro del aspecto. Cualquier cambio requiere modificar este código y volver a compilar.

ISSN: 1852-4516

Aprobado por Resolución Nro. 0263/12-R-UNPA

En Spring, el orden de ejecución se puede manejar desde la configuración del aspecto de conexión cuando el *bean* es declarado, utilizando el atributo 'order'. Si existen cambios posteriores, no es necesario modificar el código fuente ni recompilar.

- ***Pasaje de información***

Se han realizado pruebas con respecto al pasaje de información de contexto. En ese sentido Spring presenta una notable debilidad, no es posible acceder información de otro contexto, dentro de un aspecto. Esto puede estar relacionado con que la instanciación de objetos. Cada vez que se instancia una *bean*, se crea un objeto del tipo especificado, con lo cual no es posible mantener la relación entre la información guardada y su objeto correspondiente.

La Tabla 4 resume los elementos que deben ser modificados en diferentes operaciones, teniendo en cuenta la volatilidad de las Reglas de Negocios.

Tabla 4. *Resumen de cambios necesarios sobre elementos de la aplicación*.

Requerimientos	**Modifica**		**Estático / Dinámico**
	XML	**Aspecto**	
Definir el orden de ejecución de dos o más aspectos asociados al mismo evento(pointcut)	*Si*	*No*	*Dinámico*
Cambiar (configuración) o remover reglas de negocios existentes	*Si*	*No*	*Dinámico*
Agregar reglas de negocios	*Si*	*Si*	*Estático*
Agregar o remover atributos y operaciones a los aspectos de conexión	*No*	*Si*	*Estático*
Cambiar el advice en una regla	*Si*	*No*	*Dinámico*
Asociar reglas de negocios existentes a nuevos eventos	*Si*	*No*	*Dinámico*

7. CONCLUSIONES

Se han presentado algunas alternativas para la definición de conectores aspectuales para integrar RN a la funcionalidad principal de una aplicación utilizando el modelo de esquemas que ofrece Spring Framework.

Spring presenta características interesantes en este sentido, presenta un estilo declarativo para definir eventos y RN, permite realizar modificaciones y/o adaptaciones sin tener que manipular código fuente, reutilización parcial de los esquemas para los eventos (*pointcuts*) y RN, tiene la capacidad de asociar aspectos de manera sencilla.

Estas características han permitido conectar con el soporte POA basado en el modelo de esquemas de manera flexible. El modelo instanciación de aspectos *Singleton* es una restricción muy importante, sobre todo cuando se trata de implementar conexiones complejas.

La recuperación de información en diferentes contextos, también es una restricción que presenta Spring, esto podría solucionarse tal vez con Aspectos AspectJ dirigidos por anotaciones.

8. REFERENCIAS

[1] Kiczales G., Lamping L., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., "Aspect-Oriented Programming" (1997). In Proceedings ECOOP'97.Object-Oriented Programming, 11th European Conference, Finland, Springer-Verlang.

[2] Kiczales G., Hildale E., Hugunin J., Kersten M., Palm J. and Griswold W. "An overview of AspectJ" (2001). In Proceeding of the 15th European Conference on Object-Oriented Programming (ECOOP'01).

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

[4] Johnson R., Hoeller J., Arendsen A., Sampaleanu C., Harrop R., Risberg T., Davison D., Kopylenko D., Pollack M., Templier T., Vervaet E., Tung P., Hale B., Colyer A., Lewis J., Leau C., Fisher M., Brannen S., Laddad R., Poutsma A., Beams Ch., Rabbo T.A . "Spring Reference Documentation". (2004-2009)

[5] BRG "Defining Business Rules: What Are They Really?" (2001). Business Rule Group. http://www.businessrulesgroup.org/.

[6] Ross, Ronald G., "The BRS Rule Classification Scheme" (2001).DataToKnowledge Newsletter (29:5).www.BRCommunity.com/a2001/b086.html.

[7] Date, C. J. "What Not How: The Business Rules Approach to Application Development " (2000). Reading, Mass.: Addison-Wesley Longman Inc.

[8] Home Page: http://www.versata.com

[9] Von Halle B., "Business Rules Applied Business Better Systems Using the Business Rules Approach"(2002).

[10] Home Page: http://www.versata.com

[11] Home Page del Grupo Demeter: http://www.ccs.neu.edu/research/demeter/biblio/dembook. html

[12] Home Page: http://www.cs.ubc.ca/~gregor/

[13] Kiczales G., Hildale E., Hugunin J., Kersten M., Palm J. and Griswold W. "An overview of AspectJ" (2001). In Proceeding of the 15th European Conference on Object-Oriented Programming (ECOOP'01).

[14] Home Page : http://www.springsource.com

[15] Lieberherr K. “Adaptive Object-Oriented Software” (1996) . The Demeter Meted College of Computer Science, Northeastern University Boston.

[16] Page: http://rincew.blogspot.com/2005/11/drools-i-introduccin-los-motores-de.html

[17] Home Page: http://www.jessrules.com

[18] Fair I.,Pauls V., “Blaze Advisor Structured Rules Language “ (2005) .A commercial rules representation. http://www.w3.org/2004/12/rules-ws/paper/55/

[19] Home Page : http://www.ilog.com/product/jrules

[20] Arsanjani, A. Rule object 2001: “A pattern language for adaptive and scalable business rule construction” (2001). Technical report, IBM T.J. Watson Research Centre.

[21] Gamma E., Helm R., Johnson R., & Vlissides J., “Design Patterns, Elements of reusable Object-Oriented Software”(1995). Addison-Wesley.

[22] Dijkstra, E.W. “A Discipline of Programming” (1976). Prentice-Hall.

[23] Cibrán, M.A., Suvée, D., D’Hondt M., Vanderperren V., Jonckers V. “ Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming” (2004). System and Software Engineering Lab (SSEL) Vrije Universiteit Brussel (VUB) Pleinlaan 2 1050. Brussels .Belgium

[24] Cibrán, M.A., Suvée, D., D’Hondt M., Vanderperren V., Jonckers V. “Aspect-Oriented Programming for Connecting Business RuleS”. (2003) System and Software Engineering Lab. Vrije Universiteit Brussel Belgium.

[25] Cibrán, M.A., D’Hondt M. “ Composable and Reusable Business Rules Using AspectJ” (2003). System and Software Engineering Lab. Vrije Universiteit Brussel Belgium.