ACVF: Un Framework para Visualizar Patrones de Diseño Distribuidos y Concurrentes implementado en Programación Orientada a Aspectos

Diego M. S. Erdödy[1], María Feldgen[1] y Osvaldo Clúa[1]
[1] Facultad de Ingeniería, Universidad de Buenos Aires, Paseo Colón 850,
Ciudad Autónoma de Buenos Aires, Argentina
{ erdody@gmail.com, mfeldgen@ieee.org, oclua@acm.org }

Abstract. En este trabajo se hace un análisis de una implementación de patrones de diseños distribuidos y concurrentes utilizando el paradigma de programación orientada a aspectos. Se modela y desarrolla el framework ACVF "Aspectual Component Visualization Framework", para poder integrar gráficamente los componentes de cada patrón implementado y observar la interacción resultante. Cada patrón se visualiza como una mini-aplicación. Se realiza un análisis de las ventajas reales de la aplicación de aspectos a patrones de diseño distribuidos y concurrentes y se verifica tanto la modularización, transparencia e independencia de los mismos así como también el grado de localidad en el código fuente resultante usando las mismas metodologías que para patrones de diseño tradicionales.

Keywords: Framework, Patrones de diseño, Programación Orientada a Aspectos.

1 Introducción

La estructura e interacción de clases en un diseño orientado a objeto puede ser muy complejo y por lo tanto, difícil de desarrollar y entender. Los patrones de diseño reducen este nivel de complejidad a partir de una base de casos típicos que ayude a los desarrolladores a resolver problemas recurrentes [1] y soluciones prefabricadas que demostraron ser eficientes para determinado tipo de problema. Es por este motivo, que mientras más independientes y fáciles de "ensamblar" sean estas soluciones, mayor será el tiempo ahorrado en el diseño, la implementación y el mantenimiento.

Un patrón de diseño identifica las clases participantes y sus instancias, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón corresponde a un problema particular de diseño orientado a objetos. Describe cuando se puede aplicar y cuando no, las consecuencias y los beneficios de su uso. Un patrón de diseño también provee código ejemplo que muestra una implementación [2].

Otro de los problemas diseño es la abstracción de código "transversal", que se encuentra distribuido en varias clases sin relación funcional con ellas (por ejemplo, instrucciones de seguridad o de bitácora). Al momento de la implementación usando paradigmas de programación orientada a objetos, requiere dispersar este código

dentro de las clases. La programación orientada a aspectos (POA)[3] permite abstraer estos fragmentos en módulos independientes, llamados aspectos, que se aplican dinámicamente cuando es necesario. Esto se logra definiendo lugares específicos, llamados puntos de corte o "*pointcuts*", en el modelo de objetos donde el código transversal debe ser aplicado. Este código es insertado dentro de las clases en tiempo de ejecución o en tiempo de compilación. Se agregan nuevas funcionalidades dentro de los objetos sin la necesidad de que ellos tengan conocimiento de dicha introducción.

En los patrones de diseño tradicionales, también llamados GoF (Gang of Four) [2] se ha demostrado que se puede introducir aspectos [3]. En el presente trabajo se muestra, que lo mismo ocurre con patrones de diseño distribuidos y concurrentes, es decir, los que tratan con problemas en donde intervienen más de una máquina o hilo de ejecución.

En grandes proyectos de software se construyen arquitecturas semi-terminadas, llamadas frameworks, para reusar diseños probados, abstracciones y código. Un framework, en este contexto, está representado por un conjunto de clases abstractas y la forma en que interactúan [7], o sea, patrones de diseño. Es un esqueleto que puede ser reusado o adaptado de acuerdo a reglas bien definidas para resolver un familia de problemas relacionados.

En este trabajo, se describe como se desarrolló el framework ACVF ("*Aspectual Component Visualization Framework*"). El framework provee un conjunto de mini-aplicaciones clásicas o casos de estudio simples, que representan un patrón de diseño distribuido o concurrente al cual se le introducen aspectos. Cada uno se pueden estudiar individualmente o en conjunto desde el punto de vista de las siguientes características [3]:

- *Reusabilidad*. Indica si el mismo código fuente del patrón del diseño es aplicable en otro entorno.
- *Transparencia de composición*. Es la habilidad de combinar patrones de diseño sin que ello afecte el comportamiento que presenta cada uno por separado.
- *Independencia*. Es el nivel de acoplamiento que hay entre los roles que componen el patrón de diseño y las probabilidades de supervivencia que tiene cada rol fuera del patrón.
- *Localidad del código*. Es la proximidad física del código fuente que implementa el patrón de diseño. Si el patrón de diseño puede ser implementado con una sola unidad de implementación dentro del lenguaje elegido (por ejemplo un aspecto en el caso de AspectJ o una clase en Java) entonces se dice que la localidad es absoluta. Si en cambio, el código se encuentra diseminado en varias unidades de implementación entonces la localidad es menor. Mientras más unidades de implementación se encuentren involucradas, menor será el grado de localidad de la implementación del patrón.

Tomando en cuenta que las ciencias cognitivas enfatizan que la visualización aumenta la comprensión y el aprendizaje en resolución de problemas, se desarrolla un framework que visualiza la arquitectura del patrón del diseño. El framework permite integrar gráficamente los componentes de cada patrón y observar la interacción resultante. O sea, permite visualizar, configurar y relacionar los componentes pertenecientes a cada patrón y al mismo tiempo analizar los resultados de su ejecución. A continuación se describen los patrones elegidos y a modo de ejemplo, la

descripción, el análisis e implementación o desarrollo de un patrón en particular.

2 Patrones de diseño distribuidos y concurrentes

Para la descripción de los patrones de diseño se analizaron las metodologías utilizadas por Gamma [2], Schmidt [4] y Powel Douglass [5] y se eligió una combinación de estas metodologías. En este caso, el objetivo principal no es analizar los beneficios del patrón en sí, sino explicar el uso del patrón y analizar los beneficios o desventajas de la implementación orientada a aspectos.

Los patrones a analizar se han dividido en las siguientes categorías: *Concurrencia y Sincronización, Manejo de Eventos* y *Seguridad y Fiabilidad* [5][4]. Se analizan los patrones sin tener en cuenta el estudio de las restricciones temporales.

- *Patrones de Concurrencia y Sincronización*: Son los patrones destinados a resolver problemas derivados de la sincronización de recursos compartidos y tareas concurrentes. En estos ambientes se debe garantizar la integridad de los datos y evitar *deadlock* o *starvation* de las tareas.
- *Patrones de Manejo de Eventos*: Son los patrones que permiten la comunicación distribuida entre tareas utilizando el modelo de eventos. Los eventos permiten independizar el comportamiento y las responsabilidades de los participantes de un sistema distribuido.
- *Patrones de Seguridad y Fiabilidad*: Son los patrones encargados de la integridad de un sistema distribuido. Proveen soluciones para tolerancia a fallos tanto aleatorios como sistemáticos. Los fallos aleatorios se basan generalmente en mal funcionamiento de hardware o condiciones externas. Los sistemáticos en cambio, se deben habitualmente a errores en el diseño o construcción del sistema. Estos patrones se basan en proveer redundancia

Algunos de los patrones que se han elegido, son representantes de un grupo de patrones que poseen características similares (Tabla 1).

Tabla 1: Patrones de diseño distribuidos y concurrentes

Categoría	*Patrón*	*Notas*
Concurrencia y Sincronización	Rendezvous	
	Observer	*Versión distribuida (patrón GoF)*
	Optimistic Locking	
	Balking	*Patrones similares*: Guarded Suspension, Single Threaded Execution
Manejo de Eventos	Reactor	*Patrones similares*: Proactor, Acceptor, Half Synch/Half Asynch, Leader/Followers
Seguridad y Fiabilidad	Watchdog	Patrones similares: Monitor Actuator, Sanity Check Safety Executive

El lenguaje de modelado que se utiliza para describir la implementación orientada a aspectos de cada caso de estudio está basado en UML[6].

3 El Framework de Visualización ACVF

3.1 Características del Framework desarrollado

La herramienta de POA elegida para este trabajo debía ofrecer la mayor expresividad posible en la definición de *pointcuts* y la mayor cantidad de recursos posibles. AspectJ[8] es la más apropiada, seguida por CaesarJ [9], ya que ambas usan el mismo modelo de *pointcuts*. AspectJ es una implementación en Java de POA creada en los laboratorios de XEROX Parc y se convirtió en el lenguaje orientado a aspectos más popular. El proyecto ha sido integrado a la fundación Eclipse dándole soporte dentro de su entorno de desarrollo, donde se conoce como AspectJ Development Tools [10].

El framework de visualización ACVF se implementó en forma de plug-in (extensión) de Eclipse 3.1+. Sus principales características se describen a continuación.

3.1.1 Componentes

El framework es "orientado a componentes". El componente es el elemento atómico fundamental que representa visualmente un rol o un actor dentro del caso de estudio. El segundo elemento importante, es el diagrama, en el cual los componentes pueden ser desplegados y asociados entre sí a través de relaciones. Una vez desplegados, relacionados y configurados apropiadamente, el diagrama puede ser "ejecutado" a través de una simulación, que muestra la interacción entre los componentes.

El plug-in que se desarrolló como implementación del framework, posee una paleta principal asociada al diagrama, que contiene una lista de los componentes disponibles, agrupados por categorías (Fig. 1). Los componentes pueden ser arrastrados al diagrama para armar un entorno que represente el patrón a estudiar y pueden tener asociados los siguientes tipos de elementos:

3.1.1.1 *Propiedades comunes:* Son atributos comunes para todos los componentes que controlan la visualización básica.

3.1.1.2 *Propiedades propias del componente:* Son propiedades inherentes al componente. Estas se pueden dividir en dos tipos:

- *Estáticas:* Son los atributos que controlan el comportamiento del componente y son de escritura/lectura. Se visualizan dentro del panel de propiedades bajo una agrupación particular en forma de pestaña. (*RoboArm*).
- *Dinámicas:* Son las propiedades de salida del componente cuya evolución se puede ver a lo largo de una simulación, interactiva o no, en tiempo real.

3.1.1.3 *Acciones*: Son pedidos que se le pueden hacer al componente en el modo de simulación interactiva.

3.1.1.4 *Consola:* Es un panel dentro del componente para mostrar mensajes informativos de su estado.

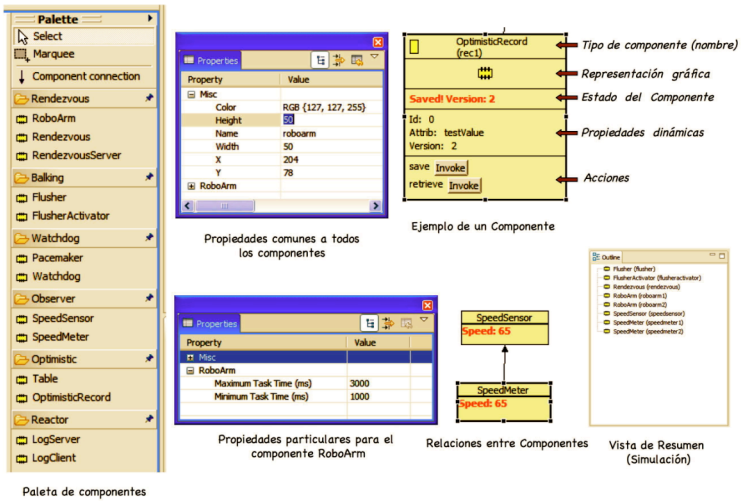

Fig. 1. Framework de visualización ACVF.

3.1.1.5 *Representación gráfica:* Cada componente puede reflejar su estado también por medio de una representación gráfica. Dicha representación constituye un modelo gráfico del componente que provee una respuesta visual a los cambios de estado y hace al componente más intuitivo y didáctico (Fig. 2)

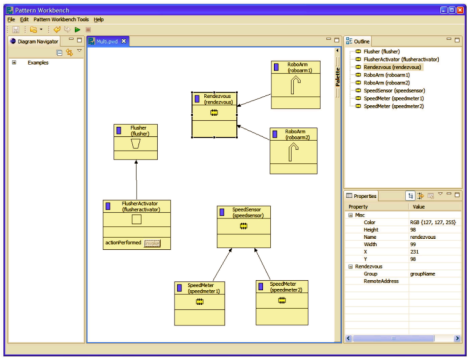

Fig. 2. Vista de la edición de un diagrama con varios componentes

3.1.2 Relaciones

Los componentes interactúan con otros por medio de relaciones. Las relaciones se indican con flechas dirigidas que unen componentes.

3.1.3 Simulación

El objetivo del diagrama es la ejecución e interacción visible de los componentes. La simulación se inicia con el botón "Run" de la barra de herramientas, que se habilita cuando se está editando un diagrama. La simulación se detiene con el botón "Stop".

Cuando la simulación se está ejecutando, cada componente se comporta de la manera preestablecida a partir de los valores proporcionados para sus propiedades específicas, o sea, la lógica propia del componente. Ciertos componentes tienen dos modos de actuar frente a una simulación: preestablecido o interactivo.

- En el modo preestablecido, el componente ejecuta la rutina que tiene prefijada y muestra su comportamiento pero no se puede interactuar con el componente.
- En el modo interactivo el componente espera a que el usuario ejecute alguna de sus acciones. Sólo los componentes que tienen definidas acciones, tienen la propiedad "Interactive" en donde se puede definir el modo de ejecución.

3.1.4 Vista de Resumen

El panel de vista de resumen ("outline" en inglés) provee un listado de los componentes existentes en el diagrama. (Fig. 1)

3.2 Arquitectura y Diseño

El framework está implementado sobre la base del subproyecto de Eclipse denominado GEF (Graphical Editing Framework)[11]. (Fig. 3).

GEF implementa el patrón *Model-View-Controller* para el desarrollo de editores gráficos permitiendo una clara separación de responsabilidades. Se integra con el Workbench de Eclipse lo que posibilita el manejo de diagramas dentro de un entorno integrado. GEF pone énfasis en desacoplar las distintas capas de un editor gráfico y brinda soporte para la implementación del controlador. El controlador es el intermediario entre la vista y el modelo. El esquema de trabajo de alto nivel es el siguiente:

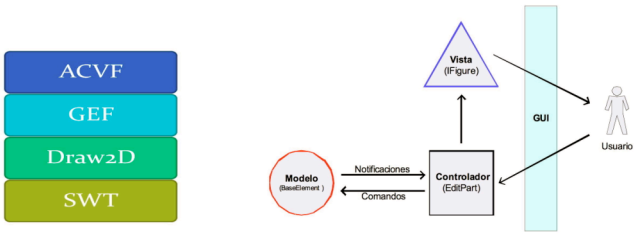

Fig. 3. Arquitectura del Framework

- *El Controlador*: Tiene el papel de mediador entre el usuario, la vista y el modelo.
- *El Modelo*: Consta de la información de base que se quiere almacenar en un editor. Los cambios realizados por el Controlador sobre el modelo son representados por Comandos.

- *La Vista:* Es la capa visual del editor, solamente contiene información acerca de la representación visual de los elementos.

3.3 Ejemplo de descripción, análisis e implementación de un patrón: *Rendezvous*

Para la descripción, el análisis y la implementación de los patrones de diseño se eligió una combinación de las metodologías utilizadas por Gamma[2], Schmidt[4] y Powel Douglass[5]. Las secciones que se describen por cada patrón son las siguientes:

3.3.1 Resumen:
Este patrón de diseño tiene como objetivo modelar las precondiciones para la sincronización o encuentro de distintos hilos de ejecución. Es un patrón general y fácil de aplicar para asegurar que las condiciones requeridas (las cuales pueden ser complejas) sean cumplidas en tiempo de ejecución. El modelo de comportamiento es el siguiente: Un hilo que está listo para "encontrarse" con los otros hilos, se registra ante el *Rendezvous* y éste lo bloquea. El *Rendezvous* libera a los participantes registrados al cumplirse las restricciones de concurrencia, por ejemplo al llegar a un número predefinido de participantes registrados.

3.3.2 Otras denominaciones: Punto de encuentro

3.3.3 Problema
El problema que soluciona el patrón de diseño es la abstracción de precondiciones en la ejecución de una acción dada que se encuentra compartida entre más de un componente.

3.3.4 Solución
La solución es delegar la sincronización en un objeto externo, llamado *Rendezvous*, que administra en forma centralizada la información de cada una de las partes involucradas. Cada hilo de ejecución se registra con el *Rendezvous* y pide permiso para proseguir en el momento en que la sincronización es necesaria. El *Rendezvous*, con la ayuda de una política de sincronización encapsulada en otro componente, le informa a cada hilo de ejecución, cuándo puede proceder.

3.3.5 Caso de Estudio
Como ejemplo ilustrativo del patrón se utiliza una línea de montaje compuesta por brazos robóticos. En este caso, el punto de encuentro está indicado por la finalización de las tareas de cada etapa, para comenzar la etapa siguiente, en cada ciclo de procesamiento. De esta forma todos los brazos empezarán su tarea simultáneamente, sin importar la duración de cada tarea.

3.3.6 Estructura
La estructura clásica del patrón es la sugerida por Douglass [5] (Fig. 4). Tiene una clase central llamada *Rendezvous* que representa el punto de encuentro de los objetos

a sincronizar. Los participantes están representados por la clase *ClientThread* y el mecanismo para liberarlos por la interface *Callback*. El conjunto de precondiciones que necesita verificar el Rendezvous para poder liberar a los integrantes es *Synch Policy*. La política más sencilla es simplemente contar los integrantes registrados y liberarlos al llegar al número preestablecido de miembros.

3.3.7 Estrategia de Implementación Orientada a Aspectos

La implementación tiene dos versiones: local o simple, en la cual los brazos robóticos están en la misma máquina; y la distribuida, que puede sincronizar los brazos en distintas máquinas.

El núcleo del patrón de diseño se encuentra en el aspecto abstracto *RendezvousProtocol* junto con las interfaces *Rendezvous* y *RendezvousMembe*r. La Figura 4 representa el diagrama de estructura de dichos elementos:

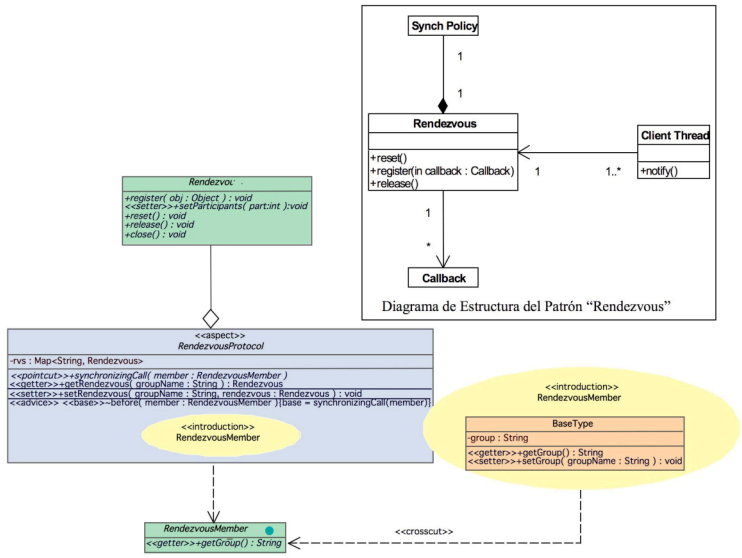

Fig. 4. Núcleo de la estructura Orientada a Aspectos del Patrón "*Rendezvous*"

La idea principal es la de desacoplar la lógica de sincronización del objeto a sincronizar por medio del punto de corte *syncronizingCall()* definido en el aspecto abstracto, que representa el conjunto de lugares donde la sincronización es requerida. El objeto a sincronizar, en nuestro caso el brazo robótico, no contiene la lógica asociada a la sincronización, ni sabe de su existencia.

El *Rendezvous* es el objeto que aplica la política de sincronización a los participantes. Los participantes implementan la interface *RendezvousMembe***r** que identifica el grupo de sincronización al que pertenecen. Un grupo de sincronización es una entidad lógica que permite al aspecto poder correlacionar un participante con un *Rendezvous* específico y de esta forma poder servir a más de un Rendezvous a la vez.

El aspecto abstracto implementa los métodos *getGroup()* y *setGroup()*. A continuación se detalla cada elemento:

Rendezvous: Es la interface de un objeto de sincronización para registrarse, por medio del método *register()* y suspender su ejecución hasta la sincronización. El método *reset()* permite volver al *Rendezvous* a su estado original y el *release()* libera a los participantes registrados. El método *close()* realiza todas las operaciones necesarias para finalizar el ciclo de vida del rendezvous.

RoboArm: Representa al brazo robótico. Tiene un identificador (atributo id) y se puede configurar el límite superior e inferior del tiempo de su acción principal (atributos minTaskTime y maxTaskTime). En ejecución, el tiempo real de duración de la tarea será un número aleatorio comprendido entre dichos límites. Posee un método *run()* para iniciar la simulación y otro *stop()* para detenerla.

RendezvousHandler: Es el aspecto concreto que agrega los detalles necesarios para aplicar el patrón definido en el aspecto abstracto. Determina cuales clases se utilizarán como miembro del *Rendezvous* y cuál será el punto de sincronización. En nuestro caso es la clase *RoboArm* y su método *execute()*.

RendezvousMember: Es la interface que identifica a un miembro capaz de registrarse ante un *rendezvous*. La implementación de esta interface, que para aspectos se denomina "introducción", se efectúa en el aspecto abstracto que implementa el protocolo. La realización, en este caso por parte de la clase *RoboArm*, se lleva a cabo en el aspecto concreto permitiendo así indicar cuales clases pueden ser miembros del *Rendezvous* sin que ésta tenga que ser alterada con código específico del patrón de diseño.

RendezvousProtocol: Aspecto abstracto encargado de definir las interacciones comunes del patrón de diseño. El punto de corte abstracto llamado *synchronizingCall()* es el punto de sincronización del patrón (que no requiere código adicional en el miembro) para registrarse y la llamada queda a cargo del aspecto (abstracción del mecanismo). Los métodos *getRendezvous()* y *setRendezvous* especifican el *Rendezvous* que sincroniza al grupo. El *advice* que se agrega sobre el punto de corte obtiene el *Rendezvous* especificado para el miembro que se está procesando y llama al método *register()*. Por último, se implementa la interface *RendezvousMember* que posee métodos para administrar el grupo del miembro.

3.3.8 Componente gráfico

Hay tres componentes implementados:

- *RoboArm*: Brazo robótico. Se pueden configurar los límites de tiempo de la duración de su tarea así como también el nombre de la instancia.
- *Rendezvous:* Es la representación del *Rendezvous*. Si no se especifica una dirección de un servidor actúa en modo "simple" o local. De lo contrario, se conecta con el servidor especificado y actúa en forma remota sincronizándose con todos los *Rendezvous* conectados a ese servidor. Si hay mas de un *Rendezvous* requiere un nombre de grupo.
- *RendezvousServer*: Servidor coordinador de llamadas de *Rendezvous* remotas.

3.3.9 Análisis

El aspecto abstracto encapsula la lógica general del patrón. Es un elemento reutilizable y localizado. Esto implica, que al momento de aplicar el patrón, solo se debe indicar sobre qué lugares del sistema actuarán los puntos de corte definidos en el aspecto abstracto y distribuir los roles especificados por el mismo. Se analizan cada una de las siguientes características:

- *Localidad del código*: Todo el código relacionado al manejo del patrón *Rendezvous* se encuentra concentrado en el protocolo así como también en las implementaciones específicas de la interface.
- *Reusabilidad*: Dada la abstracción que se logró del protocolo principal, el aspecto abstracto es totalmente reutilizable e incluso configurable con una implementación particular de la interface.
- *Independencia*: Se puede aseverar que es totalmente independiente del patrón, ya que el brazo robótico no tiene ningún conocimiento de las restricciones que se le está aplicando.
- *Transparencia de composición*: Un brazo puede participar en más de un *Rendezvous* sin necesidad de agregar restricciones adicionales ni afectar el comportamiento del mismo.

Para analizar la capacidad de cumplir con cada una de las características analizadas, se introduce la cantidad de roles únicos y superpuestos que componen cada patrón, al igual que en el trabajo hecho sobre los patrones GoF [3].

Los *roles únicos* son los roles del patrón que son implementados por un componente que no tiene funcionalidad más allá del rol que le es asignado, es decir que sólo existe para cumplir el rol dentro del patrón y, por lo tanto, es definido por él. En cambio, los *roles superpuestos* son los roles cuyos componentes comparten la funcionalidad con roles fuera del patrón. Por ejemplo, el rol de Sujeto, en el patrón *Observer*, es un rol superpuesto, ya que quien lo implemente, siempre va a cumplir uno o más roles adicionales. Si no fuera así, no habría nada que observar. Por otro lado, la implementación del rol *Rendezvous* en el patrón del mismo nombre, sólo cumple esa función, no tiene ninguna responsabilidad fuera del patrón.

4 Resultados y Conclusiones

La descripción y análisis de los patrones de diseño restantes se realizó siguiendo la metodologías antes mencionadas de la misma forma que el patrón de Rendezvous. Este trabajo se encuentra en la tesis de uno de los autores[12]. En el framework se pueden analizar los beneficios de aplicar aspectos a los distintos patrones y los resultados que se encontraron se resumen a continuación:

En el trabajo de Hannemann [3], se dividen los patrones en tres grupos según los tipos de roles que poseen: (a) los que sólo poseen roles superpuestos, (b) los que poseen roles de ambos tipos y por último, (c) los que únicamente poseen roles únicos. En este trabajo no se han identificado patrones del grupo c, aunque el más cercano por cantidad de roles únicos, es el patrón *Reactor*.

Los resultados obtenidos (Tabla 2) para los patrones analizados es similar al encontrado por Hannemann. Para los patrones del grupo (a), todas las propiedades se

cumplen completamente. Esto indica que la modularización ha sido total. Para el grupo (b), el resultado varía levemente en algunos patrones, en nuestro caso para el patrón *Optimistic*, aunque el grado de modularidad es alto. Por último, para el grupo (c), el beneficio de aplicar aspectos es limitada o nula. El patrón *Reactor* se puede tomar como ejemplo de este caso.

Tabla 2. Resultados comparativos del análisis de patrones de diseño

Patrón	***Localidad***	***Reusabilidad***	***Transparencia***	***Independencia***	***Roles***	
					Únicos	***Sup***
Balking	SI	SI	SI	SI	0	2
Observador	SI	SI	SI	SI	0	2
Rendezvous	SI	SI	SI	SI	1	1
Watchdog	SI	SI	SI	SI	1	1
Optimistic	Alta	Alta	SI	Alta	1	2
Reactor	NO	NO	SI	NO	3	1

Nota: La clasificación "Alta" indica que faltaron algunos detalles para lograr el objetivo de la propiedad. No se intenta utilizar una escala de graduación para cada propiedad. Sólo si se cumplió o no, con la salvedad de la calificación "Alta" para identificar casos cercanos al Sí

Referencias

1. Schmidt, D. C., Buschmann, F.: Patterns, frameworks, and middleware: their synergistic relationships, en Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Portland, Oregon, (2003) 694 – 704.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of reusable Object-Oriented Software, Addison- Wesley (1995)
3. Hannemann J., -Kiczales G.: Design Pattern Implementation in Java and AspectJ, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, (2002) 161-173
4. Schmidt, D. C., Stal M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2, Wiley & Sons, (2000).
5. Douglass, B. P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison-Wesley (2002)
6. http://www.uml.org (2006)
7. Jonson, R. E.: Components, Frameworks, Patterns, Proceedings of the 1997 Symposium on Software reusability, Boston, Massachusetts, United Status, ACM Press (1997) 10-17
8. http://www.eclipse.org/aspectj (2006)
9. Mezini, M., Ostermann, K.: Conquering aspects with Caesar, Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts, ACM Press (2003), 90-99
10. http://www.eclipse.org/ajdt (2006)
11. http://www.eclipse.org/gef/ (2006)
12. Erdödy, D.: ACVF: Un Framework para la Visualización de Patrones de Diseño Distribuidos y Concurrentes implementados con POA. Tesis de grado en Ingeniería en Informática, Facultad de Ingeniería, Universidad de Buenos Aires, disponible en http://www.fi.uba.ar/materias/7500/ (2007)