ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

La Programación Distribuida de Aplicaciones desde la Perspectiva de la Programación Orientada a Aspectos: Estudio y Demostración de uso.

AUTORA: María Francisca Rosique Contreras.
DIRECTOR: Pedro Sánchez Palma.
Diciembre / 2003

Autor	María Francisca Rosique Contreras
E-mail del Autor	paquirosique@wanadoo.es
Director(es)	Pedro Sánchez Palma.
E-mail del Director	pedro.sanchez@upct.es
Título del PFC	La Programación Distribuida de Aplicaciones desde la Perspectiva de la Programación Orientada a Aspectos: Estudio y Demostración de Uso.
Descriptores	Programación Orientada a Aspectos, Programación Distribuida.

Resumen:

En el desarrollo de sistemas distribuidos existen numerosos problemas causados por el dinamismo y evolución de dichos sistemas. Ante esta situación, estos sistemas deben ser diseñados para poder ser adaptados fácilmente al entorno evolutivo que les rodea que hace que sus requisitos.

Por este motivo, ha surgido en los últimos años la programación orientada a aspectos (POA) como una magnífica alternativa para implementar adaptabilidad y reutilización de sistemas en general

Este proyecto además de presentar los conceptos principales de la programación orientada a aspectos (POA) y de AspectJ como el lenguaje orientado a aspectos, está realizado para evaluar los beneficios de esta tecnología en el desarrollo de aplicaciones distribuidas y las ventajas que supone su utilización frente a la tecnología convencional.

Titulación	Ingeniero Técnico de Telecomunicación, especialidad Telemática.
Departamento	TIC
Fecha de Presentación	Diciembre - 2003

Agradecimientos

Gracias a todas las personas que han hecho posible, de una manera o de otra, la realización de este proyecto fin de carrera, a todas ellas gracias y muy especialmente a los siguientes:

- Pedro Mártinez Pérez- por la paciencia que ha tenido durante estos últimos días aguantando a mi lado.

- A mi familia, en especial mi padre y mi madre- por el apoyo moral y técnico recibido día a día, gracias de todo corazón.

- Laura Vera Amat- por su compañía incondicional y por no rendirse nunca.

- A mis compañeros de laboratorio- por esos momentos de relax.

Índice

Índice de Figuras

Índice de Tablas

Índice de Códigos

Introducción

Si se echa un vistazo a la historia de la ingeniería del software, prestando particular atención en cómo ha evolucionado, se puede observar que los grandes progresos se han obtenido gracias a la aplicación de uno de los principios fundamentales a la hora de resolver cualquier problema (incluso de la vida cotidiana), la descomposición de un sistema complejo en partes más pequeñas y más manejables, es decir, gracias a la aplicación del dicho popular conocido como "*divide y vencerás"*.

En las primeras etapas de desarrollo de los lenguajes de programación se tenía un código en el que no había separación de requisitos. Datos y funcionalidad se mezclaban por todo el programa sin ninguna distinción. A esta etapa se la conoce como la del *código spaghetti*, ya que se tenía una maraña entre datos y funcionalidad.

En la siguiente etapa se pasó a aplicar la llamada *descomposición funcional,* que pone en práctica el principio de "divide y vencerás" identificando las partes más manejables como funciones que se definen en el dominio del problema. La principal ventaja que proporciona esta descomposición es la facilidad de integración de nuevas funciones, aunque también tiene grandes inconvenientes, como son el hecho de que las funciones quedan algunas veces poco claras debido a la utilización de datos compartidos, y los datos quedan esparcidos por todo el código, con lo cual, normalmente el integrar un nuevo tipo de datos implica que se tengan que modificar varias funciones.

Intentando solucionar estas desventajas con respecto a los datos, se dio otro paso en el desarrollo de los sistemas software. La *Programación Orientada a Objetos (POO)*[1] ha supuesto uno de los avances más importantes de los últimos años en la ingeniería del software para construir sistemas complejos utilizando el principio de descomposición, ya que el modelo de objetos subyacente se ajusta mejor a los problemas del dominio real que la descomposición funcional. La ventaja que tiene es que es fácil la integración de nuevos datos, aunque también quedan las funciones esparcidas por todo el código. Aun así, tiene los inconvenientes de que, con frecuencia, para realizar la integración de nuevas funciones hay que modificar varios objetos, y que se produce un enmarañamiento de los objetos en funciones de alto nivel que involucran a varias clases.

En la *Figura 1* se representa mediante un esquema las distintas etapas de la evolución de los sistemas software. En este esquema se refleja de forma clara la mezcla de requisitos o funcionalidades que se produce en cada una de las etapas de la evolución (etapas comúnmente conocidas como "generaciones"). Cada una de las distintas formas que aparecen dibujadas (triángulo, cuadrado, trapecio, elipse) representa a un tipo de datos distinto, y cada color o tonalidad representa a una función distinta.

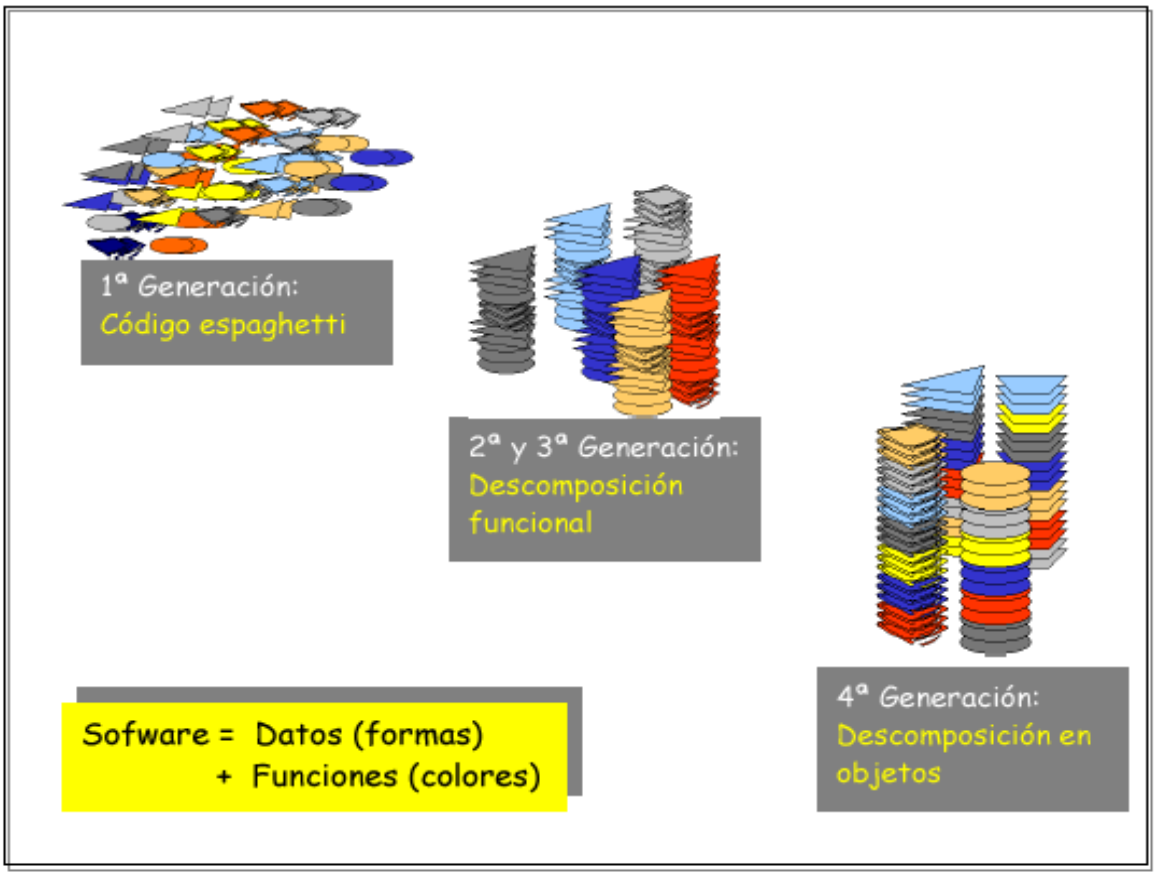


Figura 1: Esquema de la evolución de la ingeniería del software

Esta evolución del software ha tenido como principal objetivo mejorar las características tanto del proceso de desarrollo de software como del software en sí mismo. En este sentido, es evidente que el software cada vez es más sencillo de desarrollar, de adaptar, de corregir, de mantener, de evolucionar. Todas estas características se hacen indispensables en un entorno industrial o empresarial como es ya el del desarrollo de software. El desarrollo de software pertenece cada vez más al mundo industrial o empresarial, en el que la ley básica es "ahorrar costes". Por lo tanto, las buenas prácticas de programación deben verse reflejadas en una metodología de desarrollo, unas tecnologías, unos lenguajes y unas herramientas que simplifiquen el proceso y minimicen los costes.

Frecuentemente, los sistemas software grandes y complejos tienen que hacer frente a continuos cambios en sus requisitos a lo largo de su vida. En consecuencia, estos sistemas tienden a crecer y a cambiar después de haber sido desarrollados. Este requisito fundamental provoca la necesidad de que el sistema presente una alta capacidad de adaptación. Sin embargo, lograr un software altamente adaptable no es una tarea sencilla. La evolución de los sistemas no se produce de manera homogénea en todos los aspectos que lo componen, sino que en algunos se da con mayor acentuación que en otros. Por lo tanto, la evolución supone adaptar algunos aspectos del sistema a los requisitos cambiantes del contexto mientras se mantiene el resto inalterable.

Parece obvio que si se puede tratar cada uno de estos aspectos de manera independiente al resto del sistema, se consigue aumentar la adaptabilidad de todo el sistema. Para ello es necesario que cada uno de estos aspectos aparezca correctamente modularizado dentro del sistema. La modularización supone no sólo la separación del código referente a un aspecto del resto, sino también la especificación de una interfaz que defina la interacción de éste aspecto con el resto de aspectos que componen el sistema.

El aumento del grado de moduralidad del software es uno de los principales objetivos perseguidos por el mundo del desarrollo de software a través de conceptos como la programación procedural, la programación estructurada, la programación funcional, la programación lógica y la programación con tipos abstractos de datos. Cada uno de estos pasos de la tecnología de programación ha introducido un mayor nivel de modularidad en el código fuente.

Actualmente, el paradigma de programación dominante es la Programación Orientada a Objetos. Este paradigma describe el comportamiento de una aplicación informática en términos de objetos y de sus relaciones. Por consiguiente, los sistemas orientados a objetos definen un conjunto de objetos y los relacionan con el propósito de obtener un objetivo final. El mecanismo de paso de mensajes es el encargado de establecer las relaciones entre los objetos. Este paradigma de programación tiene su reflejo en todo el espectro tecnológico del desarrollo de software: metodologías, herramientas de análisis y diseño, herramientas de testeo y lenguajes orientados a objetos. El desarrollo de aplicaciones complejas manteniendo un código fuente comprensible ha sido posible gracias a la programación orientada a objetos.

Muchos sistemas tienen propiedades que no se alinean necesariamente con los componentes funcionales de un sistema orientado a objetos. Restricciones de sincronización, persistencia, protocolos de distribución, replicación, coordinación, restricciones de tiempo real, etc., son aspectos del comportamiento de un sistema que afectan a grupos de componentes funcionales. Su programación, utilizando los actuales lenguajes orientados a objetos, resulta, por un lado, una dispersión de código con el mismo propósito entre diferentes componentes y, por otro lado, cada componente tiene que implementar código para propósitos diferentes. De esta forma, el código fuente se convierte en una mezcla de instrucciones para diferentes fines.

Ante esta situación, se incrementan las dependencias entre componentes, se introducen más posibilidades de cometer errores de programación y los componentes se hacen menos reutilizables. En definitiva, se hace difícil de razonar, desarrollar y modificar el código fuente. La introducción de nuevos requerimientos que afectan a un aspecto puede implicar la modificación de otros aspectos, expandiendo de esta forma el número de componentes que deben ser cambiados o modificados. Y esta situación está en contra de la definición de adaptabilidad dada anteriormente.

Uno de los principales inconvenientes que aparecen al aplicar estas descomposiciones ya tradicionales es que muchas veces se tienen ejecuciones ineficientes debido a que las unidades de descomposición no siempre van acompañadas de un buen tratamiento de aspectos tales como la sincronización, coordinación, persistencia, manejo de excepciones, gestión de memoria, la distribución, las restricciones de tiempo real,...

En el desarrollo de un sistema software además del diseño y la implementación de la funcionalidad básica, se recogen otros aspectos tales como la sincronización, la distribución, el manejo de errores, la optimización de la memoria, la gestión de seguridad, etc. Mientras que las descomposiciones funcionales y orientada a objetos no plantean ningún problema con respecto al diseño y la implementación de la funcionalidad básica, estas técnicas no se comportan bien con los otros aspectos. Es decir, que se está ante un problemas de programación en los cuales la orientación a objetos no es suficiente para capturar las decisiones de diseño que el programa debe implementar.

Con las descomposiciones tradicionales no se separan bien estos otros aspectos, sino que quedan esparcidos por distintos lugares de todo el sistema, enmarañando el código que implementa la funcionalidad básica, yendo en contra de la claridad del mismo. Se puede afirmar entonces que las técnicas tradicionales no soportan bien la separación de competencias para aspectos distintos de la funcionalidad básica de un sistema, y que esta situación claramente tiene un impacto negativo en la calidad del software.

Figura 2: Ejemplos de código con mala modularidad

En la *figura 2*, se puede observar cómo la estructura del código se ve afectada por la implementación de una propiedad (líneas rojas) que no se haya correctamente modularizada.

Los sistemas software con una mala moduralidad presentan una serie de inconvenientes graves, que se pueden resumir en:

- **Código redundante y enmarañado**. Por un lado, ciertos fragmentos de código aparecen repetidos en diferentes partes del sistema. Por otro, las porciones de código que implementan una cierta propiedad aparecen mezcladas con otras porciones relacionadas con propiedades muy distintas.

- **Difícil razonar acerca del mismo.** El código no presenta una estructura explícita, debido a la redundancia y a la mezcla de código con diferentes propósitos.

- **Difícil de modificar o adaptar.** Es necesario encontrar todo el código relacionado, asegurarse de cambiarlo consistentemente y, todo ello, sin la ayuda de las herramientas orientadas a objetos.

La separación de propósitos (separation of concerns, SOC) es fundamental para poder manejar todos los aspectos heterogéneos de una aplicación software. La separación de propósitos es un principio clave de la ingeniería software. Consiste en la habilidad para identificar, encapsular y manipular solamente aquellas partes del software relevantes para un concepto o fin particular. En resumen, aumentar la moduralidad del software.

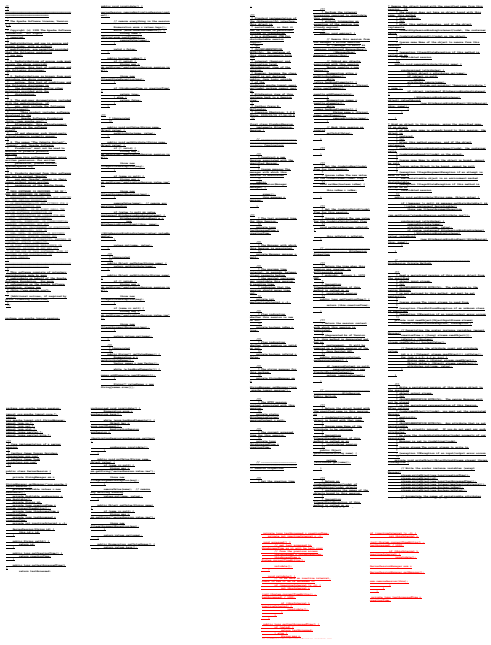


Figura3: Ejemplo de código con buena modularidad.

Dentro de la comunidad software, han surgido diferentes modelos que intentan seguir las ideas introducidas por la separación de propósitos; sin embargo, cada uno considera la separación desde diferentes puntos de vista. Una de las propuestas que más relevancia ha alcanzado es la *Programación Orientada a Aspecto*s.

Introducción a POA

La *Programación Orientada a Aspectos (POA) o separación avanzada de propósitos (MDSAC)* es un nuevo paradigma que intenta mejorar la calidad del software eliminando esta mezcla de aspectos del resto del programa, separando los requisitos funcionales (funcionalidad básica o principal) y no funcionales (aspecto) que afectan a diferentes partes del sistema[5].

Esta nueva forma de descomposición se basa en una nueva abstracción, el aspecto o concepto, para recoger los requisitos no funcionales que se dispersan y se mezclan con la funcionalidad básica de un sistema. La idea es identificar los distintos aspectos del dominio del problema, representado estos aspectos como componentes de forma aislada y dejando la tarea de componerlos para la construcción de aplicaciones finales a las herramientas orientadas a aspectos.

En definitiva, suponemos que es más fácil razonar sobre un sistema si se especifican cada uno de los aspectos que lo forman por separado, dejando a la programación orientada a aspectos definir los mecanismos para componer estos aspectos. (*Figura 4*). Esta especificación por separado tiene un claro impacto en la comprensión, trazabilidad, evolución y reutilización del software. Se puede implementar así una aplicación de forma eficiente y fácil de entender, permitiendo cambiar o agregar funcionalidad mas fácilmente.

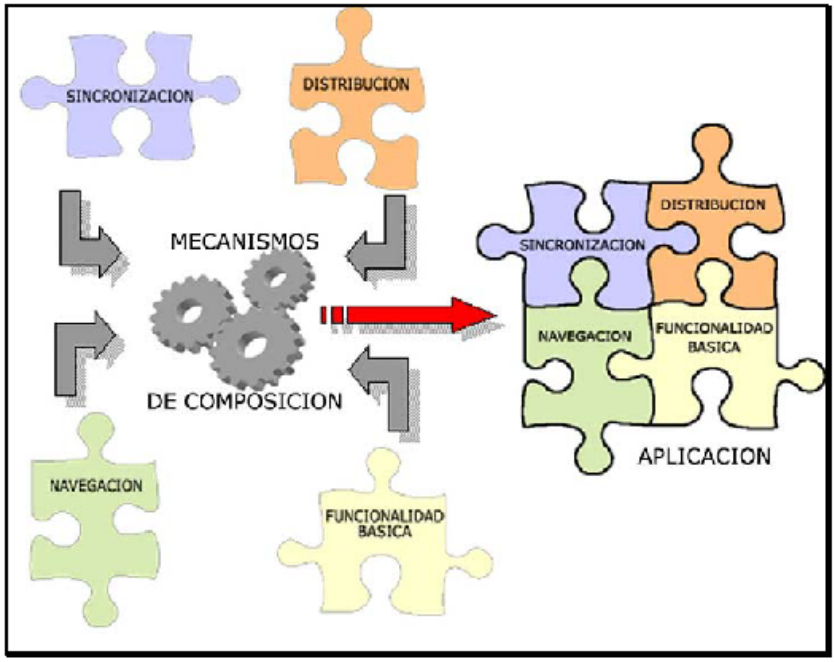


Figura 4: Principio de programación orientada a aspectos.

La programación orientada a aspectos (POA) es un desarrollo que sigue al paradigma de la orientación a objetos, y como tal, soporta la descomposición orientada a objetos, además de la propia y la descomposición funcional. Pero, a pesar de esto, POA no se puede considerar como una extensión de la POO, ya que puede utilizarse con los diferentes tipos de programación, tanto orientada a objetos como no orientada a objetos.

En la *Figura 5* se representa la forma en que la programación orientada a aspectos descompone los sistemas. Se sigue el razonamiento empleado en la *Figura 1*, donde los datos se identificaban con la forma de las figuras y las funcionalidades con el color o la tonalidad.

En este esquema se observa que la disociación de los de los distintos conjuntos se realiza tanto en base a la forma (datos) como a las tonalidades (funciones). Además, se indica que las distintas funcionalidades están relacionadas de alguna manera. Esto se representa utilizando figuras transparentes para indicar este tipo de relación.

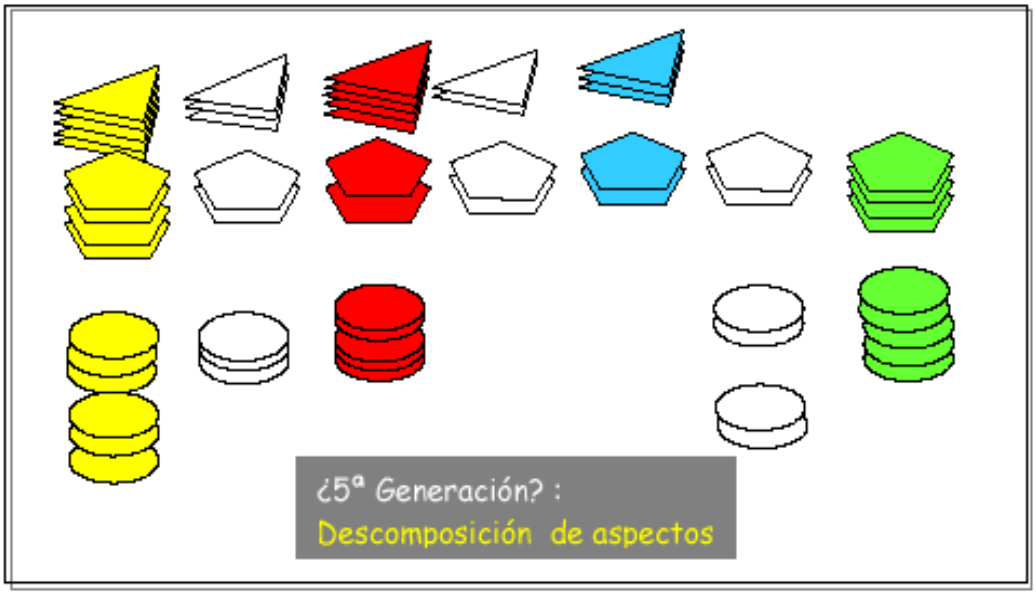


Figura 5: Descomposición en aspectos ¿La quinta generación?

El concepto de programación orientada a aspectos fue introducido por Gregor Kiczales y su grupo de trabajo, aunque el equipo Demeter había estado utilizando ideas orientadas a aspectos antes incluso de que se acuñara el término.[11]

El trabajo del grupo Demeter estaba centrado en la programación adaptativa, que no es más que una instancia temprana de la programación orientada a aspectos. La programación adaptativa se introdujo alrededor de 1991. Aquí los programas se dividían en varios bloques de cortes. Inicialmente, se separaban la representación de los objetos del sistema de cortes. Luego se añadieron comportamientos de estructuras y estructuras de clases como bloques constructores de cortes. Cristina Lopes propuso la sincronización y la invocación remota como nuevos bloques.

Pero no fue hasta 1995 cuando se publicó la primera definición del termino aspecto, realizada también por el grupo Demeter. Gracias a la colaboración de Cristina Lopes y Karl J. Lieberherr con Gregor Kiczales y su grupo, se introdujo el término de *Programación Orientada a Aspectos*.

2.1 Conceptos Básicos

El nuevo paradigma de la programación orientada a aspectos es soportado por los llamados *lenguajes de aspecto*s, que proporcionan constructores para capturar los elementos que o bien se dispersan en múltiples clases de la aplicación, o bien se mezclan con otros elementos que recogen otros requisitos. Estos elementos se llaman ***aspectos***. Para poder entender la programación orientada a aspectos antes se debe entender qué es un aspecto[6].

Una de las primeras definiciones que aparecieron del concepto de aspecto fue publicada en 1995 y se describía de la siguiente manera: *Un aspecto es una unidad que se define en términos de información parcial de otras unidade*s.

La definición actual de trabajo la dio G. Kiczales en 1999:

"Un aspecto es una unidad modular que corta la estructura de otra unidad modular"

Los aspectos pueden existir tanto en diseño como en implementación. Un aspecto de diseño en una unidad modular del diseño que corta la estructura modular de otras partes del diseño. Un aspecto en el código o programa es una unidad modular del programa que corta la estructura modular de otras unidades modulares del programa.

De manera más informal podemos decir que los aspectos son la unidad básica de la programación orientada a aspectos (POA) y pueden definirse como las partes de una aplicación que describen las cuestiones claves relacionadas con la semántica esencial o el rendimiento. También pueden verse como los elementos que se diseminan por todo el código y que son difíciles de describir localmente con respecto a otros componentes.

En un sistema podemos diferenciar entre un componente y un aspecto:

- Un **componente** es aquel módulo software que puede ser encapsulado en un procedimiento (un objeto, un método, procedimiento o API). Los componentes serán unidades funcionales en las que se descomponen el sistema.

- Un **aspecto** es aquel módulo software que no puede ser encapsulado en un procedimiento con los lenguajes convencionales. No son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afecta en la ejecución o semántica de los componentes. Se tiene que tener bien claro la diferencia entre "aspecto-de" y "parte-de", una rueda es una "parte de" un carro, mientras que resistencia al viento es un "aspecto-de" un carro

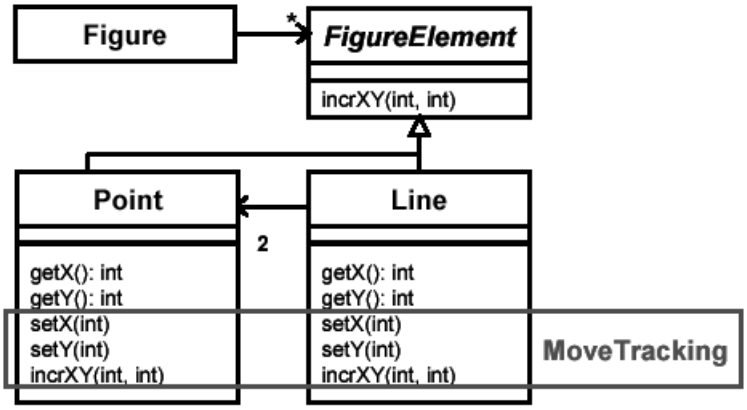


Figura 6: Ejemplo de aspecto, Move Tracking representa un aspecto que corta transversalmente las clases Point y Line.

Algunos ejemplos de aspectos son, distribución, coordinación, persistencia, navegación, los patrones de acceso a memoria, la sincronización de procesos concurrentes, el manejo de errores, etc.

En la *Figura 7*, se muestra un programa como un todo formado por un conjunto de aspectos más un modelo de objetos. Con el modelo de objetos se recoge la funcionalidad básica, mientras que el resto de aspectos recogen características de rendimiento y otras no relacionadas con la funcionalidad esencial del mismo.

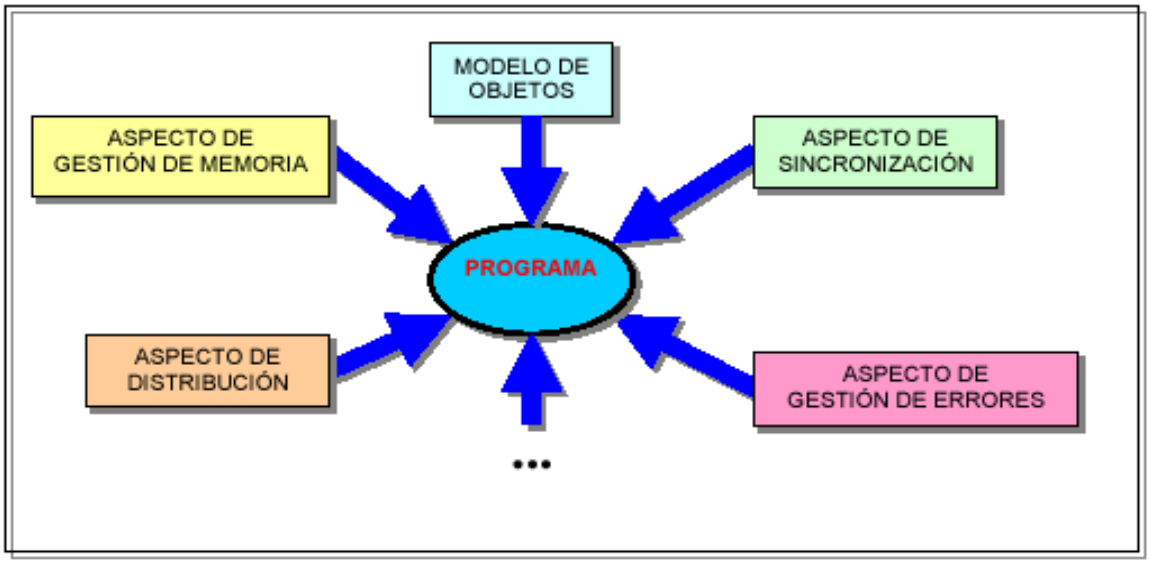


Figura 7: Estructura de un programa orientado a aspectos

Teniendo esto en cuenta, podemos realizar una comparativa de la apariencia que presenta un programa tradicional con la que muestra un programa orientado a aspectos. Esta equiparación se puede apreciar en la *Figura 8,* en la que se confrontan la estructura de un programa tradicional (parte izquierda) con la de uno orientado a aspectos (parte derecha).

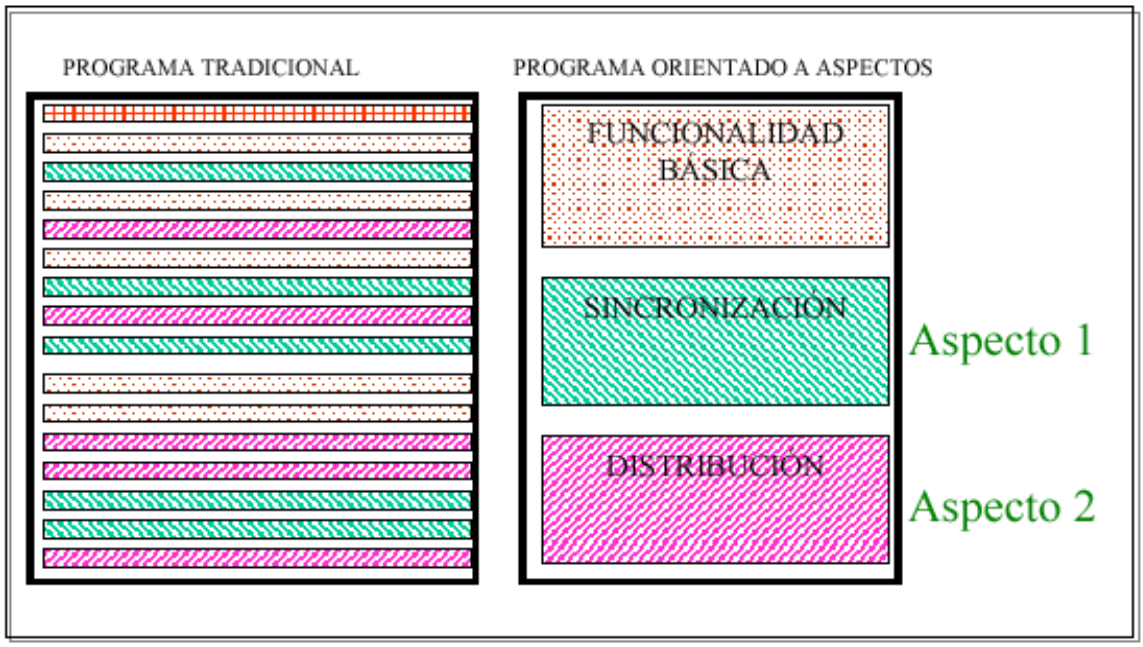


Figura 8: Comparativa de la forma de un programa tradicional con uno orientado a aspectos.

Si el rectángulo de traza más gruesa representa el código de un programa, se puede apreciar que la estructura del programa tradicional está formada por una serie de bandas horizontales. Cada banda está rellena utilizando una trama distinta., y cada trama representa una funcionalidad.

El programa orientado a aspectos, sin embargo, esta formado por tres bloques compactos, cada uno de los cuales representa un aspecto o competencia dentro del código.

Como se puede observar, en la versión tradicional estos mismos bloques de funcionalidad quedan esparcidos por todo el código, mientras que en la versión orientada a aspectos tenemos un programa más compacto y modularizado, teniendo cada aspecto su propia competencia..

Con todo lo visto hasta ahora, se puede decir que con las clases se implementa la funcionalidad principal de una aplicación (como por ejemplo, la gestión de un almacén), mientras que con los aspectos se capturan conceptos técnicos tales como la persistencia, la gestión de errores, la sincronización o la comunicación de procesos. Estos aspectos se escriben utilizando lenguajes especiales de descripción de aspectos.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular las funcionalidades que atraviesan o cortan (“*croscutting”)* todo el código, estas funcionalidades que atraviesan todo el código del sistema se llaman ***preocupaciones transversales(crosscutting concerns)***. AOP permite a los programadores separar las preocupaciones transversales(crosscutting concerns) y encapsularlas en los aspectos, motivo por el cual estas funcionalidades son más conocidas como ***aspectos*** en si.

En POO la unidad natural con la que se modulariza es la clase, una preocupación transversal(un crosscutting conerns) suele atravesar varias clases, esto es lo que provoca los mayores inconvenientes de POO. Desde este punto de vista POA añade un nuevo nivel de modularización con los aspectos.

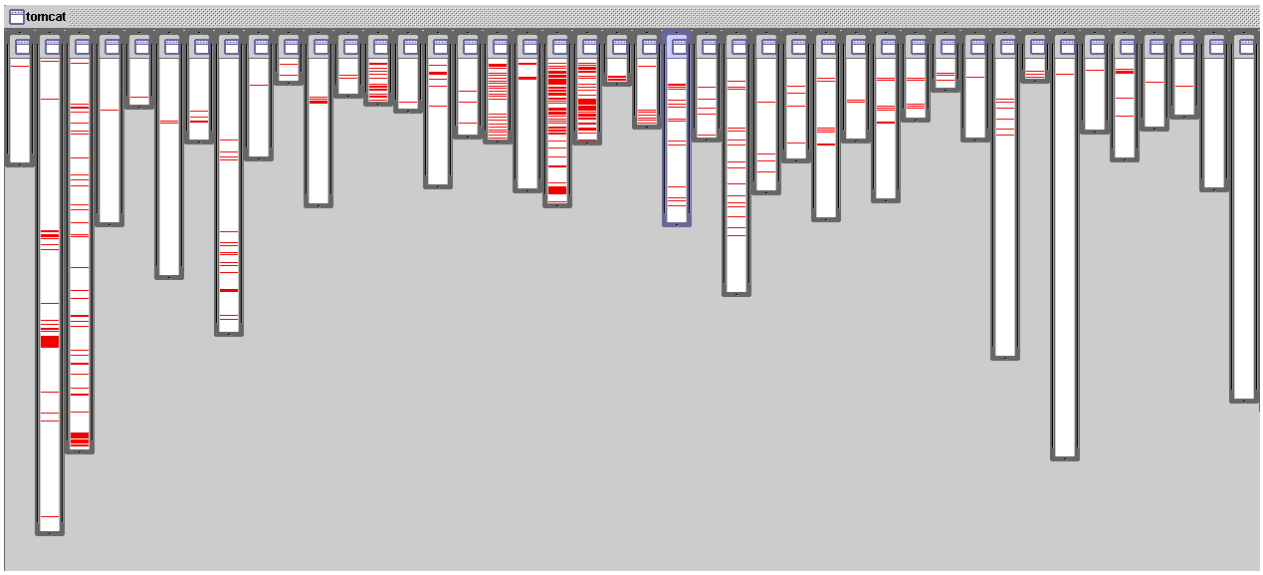


Figura 9: Ejemplo de crosscutting concern disperso.

En la *Figura 9* se puede observar cómo la funcionalidad pintada de rojo esta dispersa por todo el código del programa, cortando transversalmente las clases. Lo que provoca una mala modularidad.

Lo que se quiere conseguir es mejorar esta modularidad, agrupando todo este código que se encuentra disperso(crosscutting concern) en una unidad modular, en un aspecto.

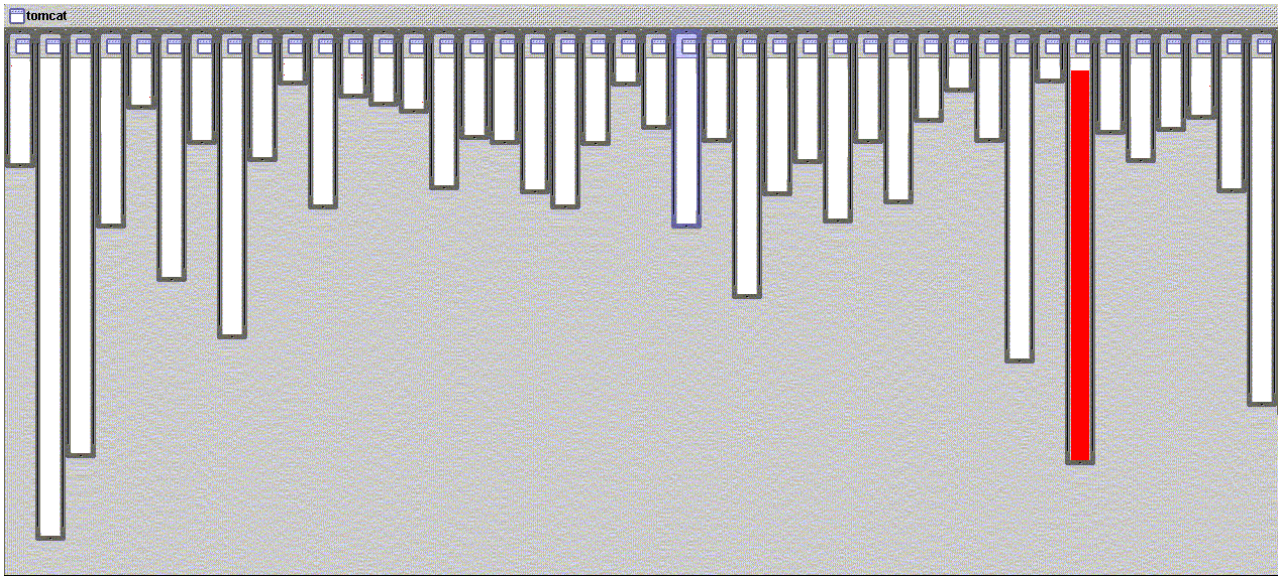


Figura 10: Ejemplo de crosscutting concern agrupado en un aspecto.

En la *Figura 10* se puede observar como la funcionalidad pintada de rojo ahora se encuentra agrupada en una sola unidad, sin atravesar el resto de clases, con lo que conseguimos claramente una mejora de modularidad.

Además, los lenguajes orientados a aspectos deben soportar la separación de aspectos como la sincronización, la distribución, el manejo de errores, la optimización de memoria, la gestión de seguridad, la persistencia. De todas formas, estos conceptos no son totalmente independientes, y es evidente que hay una relación entre los componentes y los aspectos y por lo tanto, el código de los componentes y de estas nuevas unidades de programación tienen que interactuar de alguna manera.

Para que ambos (aspectos y componentes) puedan interactuar entre ellos, deben tener algunos puntos comunes, que son los que se conocen como ***puntos de unión (Join Point)***.

Un ***Join Point (punto de unión)*** es un punto bien definido en la ejecución de un programa. Para otros autores, el conjunto posible de puntos de unión incluye todas las localizaciones en el código de un componente o aplicación, es decir, se podría definir puntos de unión a nivel de sentencia, de este modo se podría describir que ocurre en cualquier parte del código. Los puntos de unión más extendidos o los que con más frecuencia se usan son: invocaciones a métodos, manejo de excepciones, acceso a atributos, constructores, etc.

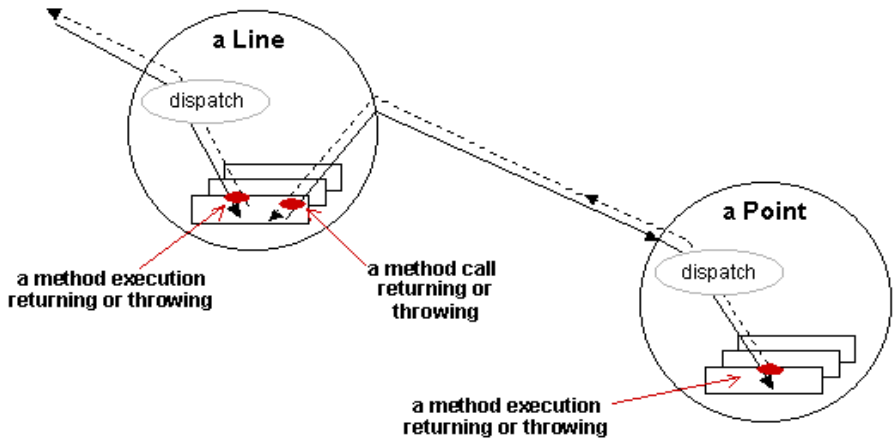


Figura 11: Join Points de un flujo de ejecución entre dos objetos

Los lugares definidos por los puntos de unión son las zonas del código donde se puede aumentar con comportamientos adicionales. Estos comportamientos se especifican en los aspectos. Y por supuesto debe haber algún modo de mezclar los componentes y aspectos, el encargado de realizar este proceso de mezcla se conoce como ***"weaver" (tejedor)***.

El **tejedor (weaver)** 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 unión. Genera el código fuente compilable de la aplicación final.

2.2 Aplicación Orientada a Aspectos

Para tener una aplicación o programa orientado a aspectos se necesitan definir los siguientes elementos:

- **Lenguaje base**, un lenguaje para definir la funcionalidad básica. Suele ser un lenguaje de propósito general, tal como C++ o Java. En general, se podrían utilizar también lenguajes no imperativos.

- Uno o varios ***lenguajes orientados a aspectos***. El lenguaje de aspectos define la forma de los aspectos. Por ejemplo, los aspectos de AspectJ se programan de forma muy parecida a las clases.

- Un ***tejedor de aspectos***. El tejedor se encargará de combinar los lenguajes. El proceso de mezcla se puede retrasar para hacerse en tiempo de ejecución, o hacerse en tiempo de compilación. (TEJEDOR)

En la *Figura 12* se aprecia la forma en la que se trabaja con las aplicaciones tradicionales, y cómo será esta forma de operar en una aplicación orientada a aspectos en la *Figura 13*.

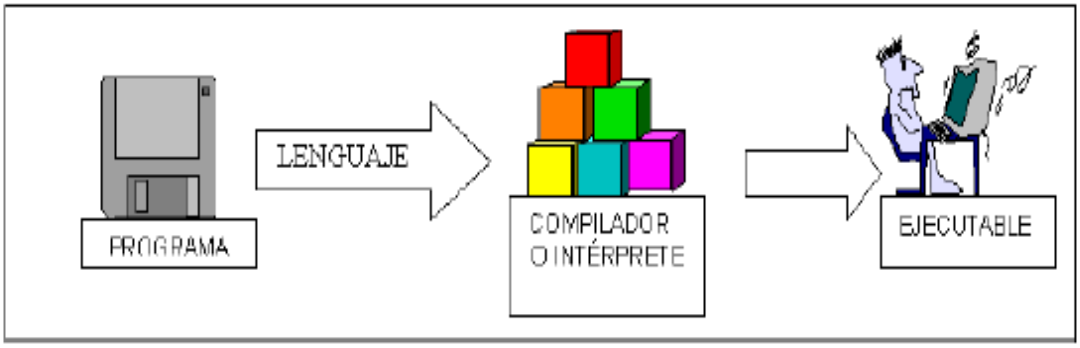


Figura 12: Estructura de una implementación en los lenguajes tradicionales.

En las aplicaciones tradicionales bastaba con un compilador o intérprete que tradujera nuestro programa escrito en un lenguaje de alto nivel a un código directamente entendible por la máquina.

En las aplicaciones orientadas a aspectos, sin embargo, además del compilador se ha de tener un *tejedor*, que combine el código que implementa la funcionalidad básica con los distintos módulos que implementan los aspectos, pudiendo estar cada aspecto codificado con un lenguaje distinto.

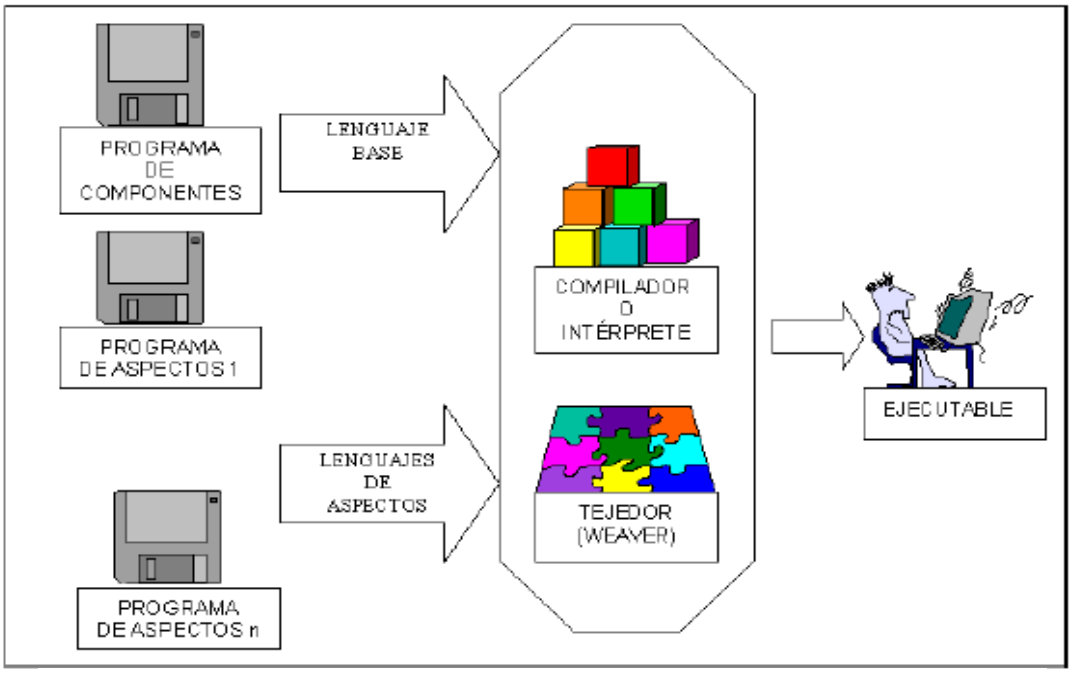


Figura 13: Estructura de una implementación en los lenguajes de aspectos.

2.2.1 Lenguaje base

El lenguaje base se utiliza para definir la funcionalidad básica o principal del sistema. Suele ser un lenguaje de propósito general, entre otros podrían ser:

- C++
- Java
- Smalltalk
- etc...

Pero para el diseño de los lenguajes de aspectos hay dos alternativas relativas al lenguaje base:

- Una sería diseñar un nuevo lenguaje base junto con el lenguaje de aspectos.
- Y la otra sería tomar un lenguaje ya existente como base, lo cual es posible, ya que la base ha de ser un lenguaje de propósito general.

Esta última opción tiene las ventajas de que se puede trabajar menos en el diseño y en la implementación de lenguajes para los entornos orientados a aspectos, se pueden utilizar lenguajes ya utilizados y que el programador solamente tiene que aprender el lenguaje de aspectos no el lenguaje para la funcionalidad básica, que todavía constituye la mayor parte de los programas orientados a aspectos.

Hasta ahora, tanto los lenguajes orientados a aspectos de propósito específico y los de propósito general se han diseñado para utilizarse con lenguajes base existentes.

Con respecto a los lenguajes de dominio específico, puede tener bastante importancia el hecho de escoger un lenguaje base. Se tiene que tener en cuenta que los puntos de enlace solamente pueden ser los que se identifiquen en el lenguaje base. Así que no se es completamente libre para diseñar los puntos de enlace.

Si se necesita separar las funcionalidades se debe recordar que el lenguaje base debe restringirse después de que se hayan separado los aspectos. Esta es la parte más difícil, ya que se tienen que quitar elementos de un sistema complejo, el lenguaje base. Aunque el diseño de un lenguaje de programación es una tarea difícil y compleja, aún lo es más el hacerle cambios a un lenguaje, que no fue diseñado para tal propósito. Por ejemplo, las restricciones no deben ser tan fuertes como en el caso de COOL que solo soporta los aspectos para los que fue diseñado en un principio.

Los lenguajes de aspectos de propósito general son menos difíciles de implementar por encima de un lenguaje de programación existente, ya que no necesitan restringir el lenguaje base. Aparte de esto, la situación es la misma que con los lenguajes de dominio específicos, es decir, en el diseño de los puntos de enlace, uno se limita a los que se pueden definir en el lenguaje base.

2.2.2 Lenguajes orientados a Aspectos

En este apartado se comentan las distintas tendencias que se siguen en los lenguajes de aspectos.

Hasta ahora se han distinguido dos enfoques diferentes en el diseño de los lenguajes de aspectos: *los lenguajes de aspectos de dominio específico* y *los lenguajes de aspectos de propósito general.*

Los ***lenguajes de aspectos de dominio específico*** soportan uno o más de estos sistemas de aspectos que se han ido mencionando en las secciones anteriores (distribución, coordinación, manejo de errores,...), pero no pueden soportar otros aspectos distintos de aquellos para los que fueron diseñados. Los lenguajes de aspectos de dominio específico normalmente tienen un nivel de abstracción mayor que el lenguaje base y por tanto expresan los conceptos del dominio específico del aspecto en un nivel de representación más alto.

Estos lenguajes normalmente imponen restricciones en la utilización del lenguaje base. Esto se hace para garantizar que los conceptos del dominio del aspecto se programen utilizando el lenguaje diseñado para este fin y evitar así interferencias entre ambos. Se quiere evitar que los aspectos se programen en ambos lenguajes lo cual podría conducir a un conflicto. Como ejemplos de lenguajes de dominio específico están COOL, que trata el aspecto de sincronización, y RIDL, para el aspecto de distribución.

Los ***lenguajes de aspectos de propósito general*** se diseñaron para ser utilizados con cualquier clase de aspecto, no solamente con aspectos específicos. Por lo tanto, no pueden imponer restricciones en el lenguaje base. Principalmente soportan la definición separada de los aspectos proporcionando unidades de aspectos. Normalmente tienen el mismo nivel

de abstracción que el lenguaje base y también el mismo conjunto de instrucciones, ya que debería ser posible expresar cualquier código en las unidades de aspectos.
Como ejemplo de este tipo de lenguajes se puede mencionar:

- ***AspectJ***, que utiliza Java como base, y las instrucciones de los aspectos también se escriben en Java. AspectJ considera la funcionalidad básica más importante que el resto de los aspectos. Fue el primer lenguaje orientado a aspectos, creado por Xerox. Es un lenguaje de propósito general de programación orientado a aspectos en lenguaje Java. Es el más extendido.

- **HyperJ**, también utiliza Java como base, es un lenguaje orientado a aspectos muy similar a AspectJ, creado por IBM. Pero a diferencia de AspectJ, HyperJ considera de igual importancia la funcionalidad básica como el resto de aspectos.

Pero existen versiones similares a AspectJ para otros lenguajes de programación (ver anexo C), lo que permite extrapolar los resultados a otros lenguajes, como:

- AspectC++ (C++).
- AspectR (Ruby).
- AspectS (Smalltalk).
- AspectC (C).
- AspectC# (C#).
- Aspect.pm (Perl).
- l Pythius (Python).

Si se contrastan estos dos enfoques, propósito general versus propósito específico, se tiene que los lenguajes de aspectos de propósito general no pueden cubrir completamente las necesidades. Tienen un severo inconveniente: permiten la separación del código pero no garantizan la separación de funcionalidades, es decir, que la unidad de aspecto solamente se utiliza para programar el aspecto. Sin embargo, esta es la idea central de la programación orientada a aspectos. En comparación con los lenguajes de aspectos de propósito general, los lenguajes de aspectos de dominio específico fuerzan la separación de funcionalidades.

Si se hace esta comparación desde el punto de vista empresarial, siempre les será más fácil a los programadores el aprender un lenguaje de propósito general, que el tener que estudiar varios lenguajes distintos de propósito específico, uno para tratar cada uno de los aspectos del sistema.

2.2.3 "Weaver": Tejedor de Clases y Aspectos

Los aspectos describen apéndices al comportamiento de los objetos. Hacen referencia a las clases de los objetos y definen en qué punto se han de colocar estos apéndices. Puntos de unión que pueden ser tanto métodos como asignaciones de variables.

Las clases y los aspectos se pueden tejer (entrelazar) de dos formas distintas: de manera estática o bien de manera dinámica.

- ***Entrelazado estático.***

El *entrelazado estático* implica modificar el código fuente de una clase insertando sentencias en los puntos de unión. Es decir, que el código del aspecto se introduce en el de la clase. Un ejemplo de este tipo de tejedor es el de AspectJ.

La principal ventaja de esta forma de entrelazado es que se evita que el nivel de abstracción que se introduce con la programación orientada a aspectos se derive en un impacto negativo en el rendimiento de la aplicación. Pero, por el contrario, es bastante difícil identificar los aspectos en el código una vez que éste ya se ha tejido, lo cual implica que si se desea adaptar o reemplazar los aspectos de forma dinámica en tiempo de ejecución aparece un problema de eficiencia, e incluso imposible de resolver a veces.

- ***Entrelazado dinámico.***

Una condición necesaria para que se pueda realizar un *entrelazado dinámico* es que los aspectos existan de forma explícita tanto en tiempo de compilación como en tiempo de ejecución.

Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Dado una interfaz de reflexión, el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica, si así se desea, durante la ejecución. Un ejemplo de este tipo de tejedores es el Tejedor AOP/ST, que no modifica el código fuente de las clases mientras se tejen los aspectos. En su lugar utiliza la herencia para añadir el código específico del aspecto a sus clases.

El principal inconveniente de este enfoque es el rendimiento y que se utiliza más memoria con la generación de todas estas subclases.

Una de las primeras clasificaciones de las formas de combinar el comportamiento de los componentes y los aspectos fue dada por John Lamping :

*1. Yuxtaposició*n. Consiste en la intercalación del código de los aspectos en el de los componentes. La estructura del código mezclado quedaría como el código base con el código de los aspectos añadidos en los puntos de enlace. En este caso, el tejedor sería bastante simple.

2. Mezcla. Es lo opuesto a la yuxtaposición, todo el código queda mezclado con una combinación de descripciones de componentes y aspectos.

3. *Fusión.* En este caso, los puntos de enlace no se tratan de manera independiente, se fusionan varios niveles de componentes y de descripciones de aspectos en una acción simple.

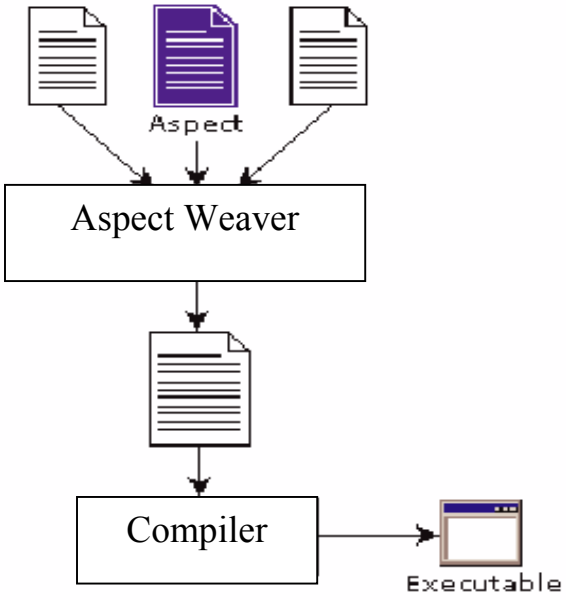


Figura 14: Esquema de funcionamiento del entrelazado

2.2.4 Características de una Aplicación POA

Entre los objetivos que se ha propuesto la programación orientada a aspectos están principalmente el de *separar los aspectos* y el de *minimizar las dependencias entre ellos*. Con el primer objetivo se consigue que cada cosa esté en su sitio, es decir, que cada decisión se tome en un lugar concreto, con el segundo se tiene una pérdida del acoplamiento entre los distintos elementos.

De la consecución de estos objetivos se pueden obtener las siguientes *ventajas*:

- La programación orientada a aspectos aumenta la **modularidad**, ya que se especifican cada uno de los conceptos que forman el sistema por separado y con una dependencia mínima. Separando el código que implementa funciones especificas que afectan a diferentes partes del sistema llamadas *preocupaciones transversales (crosscutting concern).* Se está frente un código menos mezclado, y más natural.

- Con la modularidad proporcionada por la programación orientada a aspectos conseguimos que sea más fácil razonar sobre un sistema, ya que se especifican cada uno de los conceptos que lo forman por separado y con una dependencia mínima entre ellos. Podemos decir que esta programación tiene un claro impacto en la **comprensión y complejidad del software**.

- Si el código es más simple es obvio que tanto su **desarrollo** como su **mantenimiento** será **más fácil** y en consecuencia su potencial de **reutilización** será **mayor**.

- Dotamos al código de mayor **inmunidad frente a cambios**. 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 reutilizable** y que se puede acoplar y desacoplar cuando sea necesario. Simplificando la integración de nuevas funcionalidades y desintegración de las ya existentes.

- La programación orientada a aspectos(POA) procura solucionar la ineficiencia en capturar algunas características importantes que un sistema debe implementar antes limitado en programación orientada a objetos(POO).

- A la funcionalidad básica o principal se le pueden **añadir los aspectos** necesarios no previstos inicialmente sin necesidad de modificar la funcionalidad básica. Pudiendo ser interceptada cualquier clase **sin ninguna preparación previa** para ello.

- Desde siempre el desarrollador de software ha buscado mejores niveles de reutilización y mantenimiento aumentando la **productivida**d de desarrollo y soportando los **cambios** de requisitos. Con la única finalidad de conseguir **reducir el esfuerzo** de desarrollo y sobre todo **reducir costes**. Pues con la programación orientada a aspectos se ha conseguido.

Pero no todo son ventajas, la programación orientada a aspectos también tiene unos pequeños *inconvenientes*.

- La programación orientada a aspectos (AOP) es una **tecnología** relativamente **joven**.

- Dada su escasa vida, **no** se ha **comprobado** y menos aún documentado en su totalidad.

- Hoy día AOP es **sólo una teoría**, no se ha probado en el mundo real con proyectos de gran escala, no existe todavía una garantía de que la teoría se pueda llevar tan fácilmente a la practica.

- Aspectj es la herramienta más evolucionada para AOP, hay otras, pero en versiones iniciales o muy inmaduras, y la **falta de herramientas** genera un rechazo a su adopción.

- Es difícil estimar realmente el riesgo que supone adoptar AOP, ya que todavía no se ha experimentado lo suficiente.

- Por todo ello se puede reducir las desventajas a una única y principal, su escaso tiempo de vida (que con el tiempo se solucionará).

AspectJ

Como ya se menciono anteriormente, AspectJ es un lenguaje de propósito general orientado a aspectos que usa Java como lenguaje base y es de libre distribución.

Este es el lenguaje orientado a aspectos que se utilizara en este proyecto. La razón por la que ha sido seleccionado AspectJ es que ha sido desarrollado por Xerox Parc, el mismo equipo que desarrolló POA y además es el más extendido, con la mayor cantidad de usuarios activos, con mas herramientas disponibles y sobre todo porque está orientado al lenguaje Java, asumiendo todas las ventajas e inconvenientes que esto supone, siendo capaz de ejecutarse sobre cualquier maquina virtual ya existente[4].

Las construcciones del lenguaje de AspectJ amplía el lenguaje de programación Java, de manera que cada programa válido en Java es también un programa válido para AspectJ.

3.1 Componentes del Lenguaje de AspectJ

Con sólo unas pocas estructuras nuevas [7], AspectJ provee un soporte para una implementación modular en cuanto a las preocupaciones transversales (crosscutting concern).

3.1.1. Los Aspectos

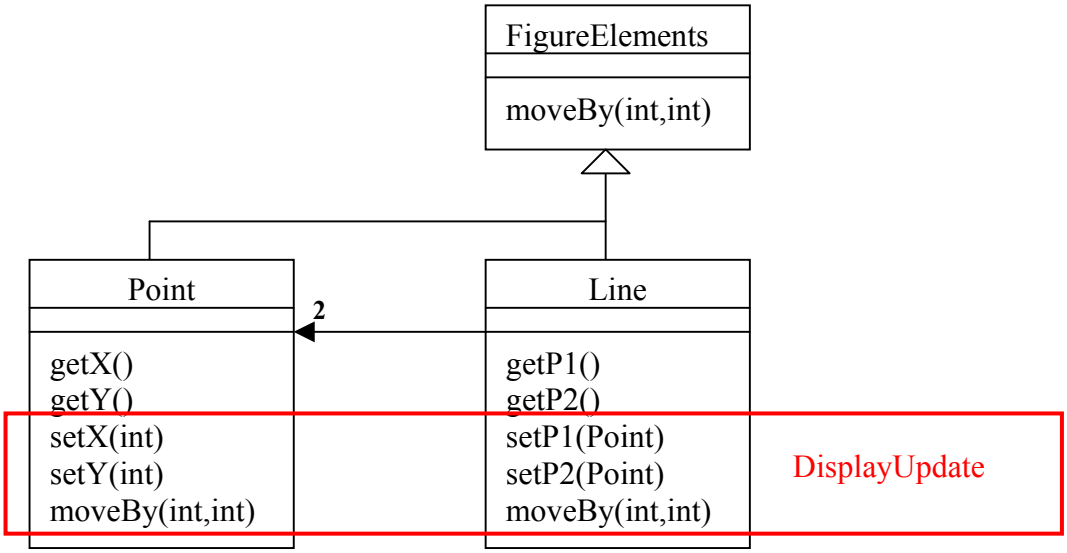
La principal construcción de AspectJ es el ***aspecto***. Cada aspecto define una función específica que puede afectar a varias partes de un sistema, como distribución, sincronismo, seguridad, etc...

Un aspecto, como se dijo en apartados anteriores, es la unidad funcional que implementa un crosscutting (corte) o varios.

Si para definir una clase en Java se utilizaba la cláusula "*class*", para definir un aspecto con AspectJ se utiliza la cláusula "*aspect*". Se definen en términos de ***introductions, join points, pointcuts y advices***, que son los elementos de los que está formado un aspecto.

Un aspecto al igual que una clase Java puede definir variables, métodos y una jerarquía de aspectos a través de aspectos especializados.

Un aspecto, como se puede comprobar, es muy similar a una clase de Java, pero la principal diferencia es que un aspecto afecta y modifica varias clases.




```
public class Line implements FigureElements{

  private Point _p1, _p2;

  public Point getP1(){return _p1;}
  public Point getP2(){return _p2;}

  public void setP1(Point p1){_p1=p1;}
  public void setP2(Point p2){_p2=p2;}

  public void moveBy(int dx, int dy) { ... }
}
```

```
public class Point implements FigureElements{

  private int _x=0,_y=0;

  public int getX()     {return _x;}
  public int getY()     {return _y;}

  public void setX(int x)       {_x=x;}
  public void setY(int y)       { _y=y;}

  public void moveBy(int dx, int dy) { ... }
}
```

```
aspect  DisplayUpdate{

  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() returning: move() {
    Display.update();
  }
}
```

Figura 15: Ejemplo de aspecto, DisplayUpdate representa un aspecto que corta transversalmente las clases Point y Line.

Características similares a las clases:

- Los aspectos pueden tener diferentes tipos
- Los aspectos pueden extender de clases o de otros aspectos.
- Los aspectos pueden ser abstractos o concretos.
- Los aspectos que no son abstractos pueden ser instanciados.
- Los aspectos pueden comportarse de forma estática o no estática.
- Los aspectos pueden ser campos, métodos, o otros tipos de elementos.
- Los elementos de aspectos que no sean privileged siguen las mismas reglas que las clases.

Características diferentes a las clases:

- Los aspectos añaden a una parte de las declaraciones de elementos comunes a las clases, otro tipo de declaraciones propias de los aspectos, los pointcut, advice, etc...
- Los aspectos se pueden clasificar según el contexto en el que se especifican.
- Los aspectos no disponen de constructores ni finalizadores, no se pueden crear con el operador "new". Los aspectos están disponibles automáticamente cuando se necesitan.
- Los aspectos "privileged" tienen acceso a los elementos "private" de otros tipos.

Las principales diferencias y similitudes entre clases y aspectos las podemos resumir en la siguiente tabla.

Características	Clases	Aspectos
Puede extender clases	Sí	Sí
Puede implementar interfaces	Sí	Sí
Puede extender aspectos	No	Sí [1]
Puede ser declarado internamente	Sí	Sí [2]
Puede ser instanciado directamente	Sí	No

Tabla 1: Diferencias y similitudes entre clases y aspectos.

Los aspectos pueden alterar la estructura estática de un sistema añadiendo variables, métodos, etc., a una clase, alterando la jerarquía del sistema, y convirtiendo una excepción chequeada en no chequeada(excepción runtime). Esta característica de alterar la estructura estática de un programa es llamada *static crosscutting*.

[1] Sólo puede extender aspectos abstractos.
[2] Se debe declarar estático.

Además de afectar a la estructura estática, un aspecto también puede afectar a la estructura dinámica de un programa, esto es posible a través de intercepciones en puntos de flujos de ejecución, los llamados ***puntos de unión (join points***), añadiendo comportamientos antes, durante, o después de los mismos, mediante el control total de estos puntos de ejecución del programa.

3.1.2 Join point (Punto de Unión)

Aunque cualquier punto que se pueda identificar en la ejecución de un programa es un punto de unión, AspectJ limita los puntos de unión disponibles a aquellos que son usados de una manera sistemática.

También es posible definir join points como resultado de la composición de varios join points. AspectJ provee los siguientes tipos de join points.

- Llamada a métodos (method call)
- Ejecución de métodos (method execution)
- Llamada a constructores (constructor call)
- Ejecución de constructores (constructor execution)
- Ejecución del manejo de excepciones (exception handler execution)
- Inicialización de objetos (object initialization)
- Pre-inicialización de objetos(object preinitialization)
- Ejecución de inicializaciones (initializer execution)
- Ejecución de inicializaciones estáticas (static initializer execution)
- Escritura/lectura de campos de objetos, clases o interfaces (get field, set field)

El lenguaje de AspectJ define una construcción sintáctica para detectar estos puntos de unión, son los **pointcuts**. La diferencia entre un join point y un pointcut es fundamental: el join point es un concepto y el pointcut es la construcción que AspectJ provee para canalizar dicho concepto.

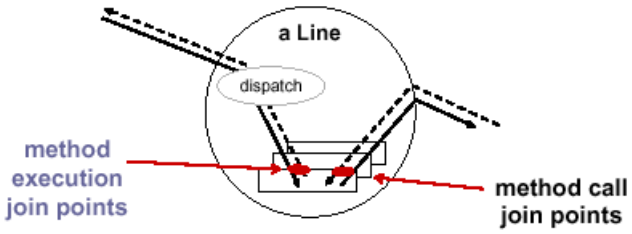


Figura 16: Ejemplo de distintos tipos de Join Points

3.1.3 Pointcut (Cortes)

Los pointcut (cortes) corresponden a la definición sintáctica de un conjunto de puntos de unión (join points) y algunas veces algunos valores del contexto de ejecución de estos puntos de unión. El pointcut puede utilizar los operadores &&, ||, ! para agrupar los join points, o bien puede estar formado por un solo join point. Utilizando pointcuts podemos obtener valores de argumentos de métodos, objetos en ejecución y atributos o excepciones de los join points.

Se puede declarar un pointcut como un elemento de un aspecto o de una clase, se pueden declarar públicos, privados o finales, pero no se pueden sobrecargar. También se pueden declarar pointcuts abstractos, pero estos solo dentro de aspectos abstractos.

Un pointcut está formado por una parte izquierda y una parte derecha, separadas ambas por dos puntos. En la parte izquierda se define el nombre del pointcut (también puede ser anónimo y no tener nombre) y el contexto del pointcut. La parte derecha los designadores(designators) de eventos del pointcut.

pointcut nombre(contexto) : pointcut_designators;

A los descriptores de eventos de la parte derecha de la definición del pointcut se les llama designadores(designators). Un designador de corte captura en ejecución varios puntos de enlace, por ejemplo el designador:

call(void NumComplejo.ingresar_parte_real(real))

Este designador captura todas las llamadas al método *ingresar_parte_real* de la clase *NumComplejo* con un argumento de clase *real* (ver *Tabla 2*).

Los designadores de pointcut son:

Métodos y constructores	
call(Signature)	Selección de join points en llamadas (call) a métodos o constructores basado en la firma estática (signature).
execution(Signature)	Selección de join points en ejecuciones (execution) de métodos o constructores basado en la firma estática (signature).
initialization(Signature)	Selección de join points en inicialización de objetos con el constructor con la siguiente firma estática(signature).

Tabla 2: Designador de métodos y constructores.

Los pointcuts de llamadas a métodos y constructores, primero evalúan la firma(Signature), pero sin que llegue a llamar al método y después seleccionan los join points correspondientes.

Los pointcuts de ejecuciones de métodos y constructores seleccionan la ejecución del método o constructor teniendo el control total de dicha ejecución.

Manejador de excepciones	
handler(ExceptionTypePattern)	Selección de manejadores de excepciones (bloque catch de Java) de cualquiera de los bloques ExceptionTypePattern que sean de tipo Throwable.

Tabla 3: Designador de manejadores de excepciones.

Acceso a campos	
Get(Signature)	Selección de join points en lectura de campos(get field) basado en la firma estática (signature).
set(Signature)	Selección de join points en escritura de campos (sep fiel) basado en la firma estática (signatura.

Tabla 4: Designador de acceso a campos.

Condicionales	
if(BooleanExpression)	Selección de join points donde la expresión booleana es evaluada verdadera(true)

Tabla 5: Designador condicional.

Objetos	
this(TypePattern or Id)	Selección de join points donde el objeto ejecutado es instancia de TypePattern.
args(TypePattern or Id, ...)	Selección de join points donde los argumentos son del tipo TypePattern.
Target(TypePattern or Id)	Selección de join points donde el objeto donde se invoca un método es del tipo TypePattern. Es decir el objeto destino es instancia de TypePattern.

Tabla 6: Designador de objetos.

Léxico	
Within(TypePattern)	Selección de join points pertenecientes al código de cualquier clase definida en TypePattern.
withincode(Signature)	Selección de join points interiores al código de métodos o constructores basados en la firma estática(signature)

Tabla 7: Designador léxico.

Control de flujo	
cflow(Pointcut)	Selección de todos los join points del control de flujo seleccionado por el pointcut, incluyendo el método que genera el flujo.
cflowbelow(Pointcut)	Selección de todos los join points del control de flujo seleccionado por el poincut, sin incluir el método que genera el flujo.

Tabla 8: Designador de control de flujo.

Inicializaciones estáticas	
staticinitialization(TypePattern)	Selección de join points en ejecuciones de inicializaciones estáticas dentro de TypePattern.

Tabla 9: Designador de inicializaciones estáticas.

Combinaciones	
PointcutId(TypePattern or Id, ...)	Selección de join points que han sido seleccionados por el designador del pointcut definido por el usuario con nombre *PointcutId.*
! Pointcut	Selección de join points que no han sido seleccionados por Pointcut
Pointcut0 && Pointcut1	Selección de join points que son seleccionados por Pointcut0 y por Pointcut1
Pointcut0 || Pointcut1	Selección de join points que son seleccionados por Pointcut0 o por Pointcut1
(Pointcut)	Selección de join points que son seleccionados por el Pointcut entre paréntesis.

Tabla 10: Designador combinacional.

Existen dos formas de definir un pointcut:

- Una es dándole un nombre explícito, de esta manera se puede reutilizar el pointcut para definir otros pointcuts o para definir parte de un aviso(advice), anular pointcut, etc.

- La otra manera de definir un pointcut es de forma anónima, se definen en el momento en el que va a ser utilizado, o dentro de la especificación de un aviso(advice) o de otro pointcut. Como es lógico, este tipo de pointcut no pueden ser reutilizados. Este tipo de pointcut se suele utilizar con cflow( )

Con el uso de los operadores vistos anteriormente ("|"), ("&") y ("!"), se pueden combinar tanto pointcuts con nombre como los anónimos, por separado o conjuntamente.

También se puede hacer uso de caracteres especiales para definir pointcuts. Los más usados son "*****" ,"**..**", "+" y "[ ]".

- El primer carácter especial es quizás el más conocido, también se le llama comodín. Si aparece * solamente, indica todos los tipos, incluidos los tipos primitivos.

  Ejemplo:

  call(void foo(*))

  Selecciona todos los join point de llamadas a métodos llamados foo, que devuelvan void y con "cualquier tipo de argumentos de entrada".

  Si lo encontramos dentro de un identificador indica cualquier secuencia de caracteres sin incluir "**.**". Si va entre medio, indica cualquier secuencia que comience y termine con la secuencia indicada.

  Ejemplo:

  *handler(java.util.*Map)*

  Selecciona todos los tipos de java.util.Map, entre otros el java.util.HashMap.

  En definitiva según en el lugar donde se encuentre y en el contexto que se encuentre significa una cosa u otra, pero siempre generalizando.

- El carácter especial "+" debe ser utilizado junto al nombre de un tipo, para representar el conjunto de todos los subtipos del tipo utilizado.

- El carácter especial "**..**" en un identificador, indica cualquier secuencia de caracteres que empiecen y terminen por "**.**".

- El carácter especial "[]" representa el tipo array.

3.1.4 Static Crosscutting.: Introductions

Hasta ahora se han visto distintas construcciones de AspectJ para implementar crosscutting dinámicos. AspectJ también permite implementar *crosscutting estáticos* para cambiar la estructura estática de un programa, estas construcciones se llaman ***Introductions (introducciones)***.

Las *introducciones (introductions)* se utilizan para introducir elementos completamente nuevos en las clases dadas de manera estática. Entre estos elementos podemos añadir:

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

También se usa para modificar la jerarquía de las clases, ya que agrega nuevos elementos y altera la relación de la herencia entre las clases, como ocurre en el siguiente ejemplo, donde la jerarquía de la clase Point es modificada.

```
public class Point {
      private int x, y;

      public Point(int x, int y) { this.x = x; this.y = y; }

      public int getX() { return this.x; }
      public int getY() { return this.y; }

      public void setX(int x) { this.x = x; }
      public void setY(int y) { this.y = y; }

      public static void main(String[] args) {
            Point p = new Point(3,4);
            Point q = (Point) p.clone();
   }
}
```

Código 1: Código fuente de la clase Point

```
aspect CloneablePoint {

      declare parents: Point implements Cloneable;
      declare soft: CloneNotSupportedException: execution(Object
clone());
      Object Point.clone() { return super.clone(); }
}
```

Código 2: Código fuente del aspecto CloneablePoint que modifica la jerarquía de la clase Point.

Después de aplicar el aspecto, la clase Point implementara la interfaz Cloneable, tendrá un nuevo método llamado clone que devuelve un nuevo objeto exactamente igual al Point y puede ocurrir una excepción no chequeada del tipo CloneNotSupportedException durante la ejecución del método clone().

La introducción altera la estructura estática de un sistema y trabaja en tiempo de compilación. Esta nueva capacidad de alterar las declaraciones de las clases y en definitiva la jerarquía, es un nuevo crosscutting concern porque afecta a múltiples clases. Existen las siguientes introducciones:

- **Introducción de un nuevo método:**

Cualquier aspecto puede añadir un método nuevo a cualquier objeto utilizando la siguiente declaración:

```
[Modificador] Type TypePattern.Id(Formals) { Body }
```

Esta introducción hace que los objetos del tipo TypePattern soporten el nuevo método que se ha introducido.

También podemos introducir métodos abstractos.

```
[abstract] [Modificador] Type TypePattern.Id(Formals);
```

En estas declaraciones los modificadores son modificadores de acceso (public, private, etc...). Type es el tipo que devuelve el método. Body es el código que implementa el método.

- **Introducción de un nuevo constructor:**

Cualquier aspecto puede añadir un nuevo constructor a cualquier objeto utilizando la siguiente declaración:

```
[Modificador] TypePattern.new(Formals) { Body }
```

Esta introducción hace que los objetos del tipo TypePattern soporte el nuevo constructor que hemos introducido. Hay que tener en cuenta que no podemos aplicar esta introducción sobre una interfaz, de manera que si TypePattern incluye una interfaz, daría un error.

- **Introducción de un nuevo campo:**

Cualquier aspecto puede añadir un campo nuevo a cualquier clase usando la siguiente declaración:

```
[Modificador] TypePattern.Id  [=expresion]

[Modificador] Type TypePattern.Id;
```

En esta declaración los modificadores pueden ser o modificadores de acceso o de almacenamiento (final, static,...)

Se puede asignar un valor inicial al campo mediante una expresión de inicialización optativa.

Reglas de acceso.

Como se ha visto, las introducciones de métodos, constructores y campos hacen uso de modificadores de acceso, tenemos que tener en cuenta que las introducciones no soportan modificadores de acceso "*protected*".

Los modificadores de acceso se aplican al aspecto, no al destino del aspecto. De manera que si dentro de un aspecto se introduce un elemento privado, este elemento sólo será accesible dentro de dicho aspecto, no en las clases en las que actúa el aspecto.

El acceso por defecto para introducir elementos es el mismo que tiene Java por defecto, es decir el paquete **protected**.

- **Extensión:**

  Además de poder introducir nuevos elementos, AspectJ permite introducir a una clase una nueva superclase, una nueva clase padre de la que hereda, así conseguimos cambiar la jerarquía estática de una clase. La sintaxis para esta introducción es la siguiente:

  ```
  declare parents: TypePattern extends TypeList;
  ```

  Donde TypePattern será la clase afectada y TypeList especifica la nueva clase que hereda (sólo pude ser una).

- **Implementación:**

  Si introducir una nueva clase padre no es suficiente, se puede hacer que una clase cualquiera implemente una nueva interfaz (una o varias):

  ```
  declare parents: TypePattern implements TypeList;
  ```

  Donde TypePattern será la clase afectada, y TypeList especifica la nueva interfaz que implementara (una o varias).

- **Warnings y errores:**

  El compilador de AspectJ proporciona un servicio para declarar *warnings y errores*. Puede detectar en tiempo de compilación dichos warnings o errores. Si se trata de un warning muestra un mensaje de texto y continua con la compilación, mientras que si es un error muestra un mensaje de texto y abandona la compilación.

  Para declarar los warnings o errores se usan las siguientes declaraciones:

  ```
  declare error: Pointcut: String;
  ```

  ```
  declare warning: Pointcut: String;
  ```

Donde String es el mensaje que se muestra cuando se detecta un warning o un error, y Pointcut es un pointcut estático determinado. Este pointcut selecciona una serie de join points que serán localizados en el código fuente, en el lugar correspondiente a estos join points durante la ejecución del programa.

- **Excepciones:**

  AspectJ puede convertir una excepción chequeada en una no chequeada del tipo *org.aspectj.lang.SoftException* que encapsula la excepción original.

  ```
  declare soft: TypePattern: Pointcut;
  ```

  Cualquier excepción de un tipo en TypePattern que sea lanzada en un join point seleccionado por el Pointcut (pointcut estático determinado anteriormente) será encapsulada en una excepción no chequeada del tipo *org.aspectj.lang.SoftException.* De esta manera la posibilidad de que suceda la excepción no provoca un error de compilación si el método desde donde fue lanzada no contiene la clausura "throws TypePattern" en la firma del método o si esta excepción no es tratada por los bloques "try /catch".

- **Introducciones libres:**

  Además de las introducciones públicas AspectJ permite las introducciones privadas y de pakage-protected. Las introducciones privadas significan privadas con respecto al aspecto no respecto a las clases afectadas por el aspecto.

  Estas introducciones suelen ser por ejemplo como la siguiente: *private int Foo.x;*

3.1.5 Avisos(Advices).

Los avisos son construcciones que definen código adicional que deberá ser ejecutado en los puntos de unión.

Las declaraciones de avisos (advices) definen partes de la implementación del aspecto que se ejecutan en puntos de unión bien definidos (join points). Estos puntos pueden venir dados bien por un pointcut (corte) con nombre o bien por un pointcut anónimo. En los dos siguientes recuadros se puede ver el mismo aviso, primero definido mediante un pointcut con nombre (*Figura 17)* y después mediante uno anónimo (*Figura 18)*.

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                       (call(void setX(int) ||
                                       call(void setY(int)));

before (Point p1, int newval): setter(p1, newval) {
        System.out.println("P1:" + p1 );
}
```

Figura 17: Ejemplo aviso definido mediante un corte con nombre.

```
before (Point p1, int newval): target(p1) && args(newval)
                                (call(void setX(int)) ||
                                call(void setY(int))) {

        System.out.println("P1: " + p1 );
}
```

Figura 18: Ejemplo aviso definido mediante un corte anónimo.

AspectJ tiene varios tipos de avisos (advices) para definir el código adicional que se deberá ejecutar en los join points. Los avisos definen con que puntos de unión van a interaccionar y en que momento. Los puntos de unión son definidos, como ya se explico anteriormente, a través de los pointcuts. Hay tres tipos básicos de avisos: advice before, advice after y advice around.

- **Advice before**. Este aviso es ejecutado justo "*antes*" de que las acciones asociadas con los eventos del pointcut (corte) sean ejecutadas. Es decir el cuerpo (*Body*) del aviso es ejecutado antes de que un punto de unión sea seleccionado por el pointcut asociado a dicho aviso.

  El advice before tiene la siguiente forma:

  before(Formals): Pointcut { Body }

  Donde *Formals* son los parámetros de entrada con los que podrá trabajar el aviso, *Pointcut* es el pointcut (corte) asociado al aviso y *Body* es el cuerpo del aviso, es decir el código de ejecución del aviso.

- **Advice after**. La finalidad del Advice after es ejecutar el cuerpo del aviso justo "*después*" de que el punto de unión asociado sea seleccionado por el Pointcut. Es decir se ejecuta justo después de que lo hayan hecho las acciones asociadas con los eventos del Pointcut.

  El problema es que no en todos los casos el "after (después)" se crea igual. ¿Que pasa si mientras la ejecución del aviso se lanza una excepción al punto de unión?

  AspectJ dispone de tres modalidades diferentes para este aviso, una simple y dos especiales.

  - **Advice after:** Esta es la forma simple, se ejecuta sea cual sea las circunstancias.

    Su sintaxis es similar a la del advice before:

    after(Formals) : Pointcut { Body }

    Donde al igual que en el advice before *Formals* son los parámetros de entrada con los que podrá trabajar el aviso, *Pointcut* es el pointcut (corte) asociado al aviso y *Body* es el cuerpo del aviso, es decir el código de ejecución del aviso.

- **Advice after returning:** Esta versión del Advice after sólo se ejecuta si el código del punto de unión no lanza ninguna excepción, y se retorna normalmente. Su sintaxis le capacita para retornar objetos:

  after(Formals) returning [(Formal)]: Pointcut { Body }

  Además de los parámetros que provee el pointcut el aviso puede tener uno o más, son los parámetros *Formals*. Puede devolver un objeto al join point a través del parámetro *Formal* opcional que permite el acceso al valor retornado.

- **Advice after throwing:** Esta versión del Advice after se ejecuta precisamente cuando no lo hace el advice after returning, es decir cuando se lanza una excepción al join point. Su sintaxis le capacita para lanzar excepciones:

  after(Formals) throwing [(Formal)]: Pointcut { Body }

- **Advice around**.- Atrapa la ejecución de los métodos designados por el evento. La acción original asociada con el mismo se puede invocar utilizando thisJoinPoint.runNext().

  Type around(Formals) [throws TypeList] : Pointcut { Body }

  Los avisos pueden tener en su interior declaraciones de retorno, pero tienen que devolverlo solamente para los avisos before y after. Una declaración de retorno dentro de un aviso around, debe devolver el tipo declarado.

  Los avisos se pueden declarar con el modificador static, indicado 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.

Método especial proceed:

Dentro de un aviso (advice) around con parámetros (*Formals*), existe un método *proceed* que acepta los mismos parámetros que los que acepta el advice around.

Llamando a proceed con valores diferentes a los actuales, se realizará el advice restante para estos nuevos valores. Este método particular del advice around se suele utilizar para asegurar que las variables con las que se trabaja están bien inicializadas y con los valores previstos.

En el siguiente ejemplo podemos ver cómo el advice around hace uso del método proceed para asegurarse que los valores que reciben los métodos son siempre cero ó mayores de cero y en el caso de no ser así le asigna un cero como parámetro de entrada, quedando así la variable inicializada correctamente.

```
Around(int nv) returns void:  receptions(void Point.setX(nv)) ||
             receptions(void Point setY(nv))
{
      proceed(Math.max(0,nv));
}
```

Variables especiales:

En el cuerpo de los avisos también pueden utilizarse tres variables especiales, thisJointPoint, thisJointPointStaticPart y thisEnclosingJointPointStaticPart.

Estas variables permiten declarar avisos(advices) de una forma más simple. Con estas variables AspectJ proporciona un acceso reflexivo a la información correspondiente al join point en curso.

- **thisJointPoint:** Dentro del cuerpo(body) de la declaración de un aviso(advice), la variable especial thisJoinPoint representa a un objeto que envuelve al join point en curso. Este objeto suministra información correspondiente al join point, como de que tipo de join point se trata, el lugar donde se esta ejecutando, etc.

- **thisJointPointStaticPart:** Esta variable es muy similar a thisJoinPoint, sólo se diferencia en la información que suministra. Esta variable proporciona una información más pobre, se suele utilizar cuando los requerimientos de memoria lo exijan.

- **thisEnclosingJointPointStaticPart:** Esta variable engloba la parte estática del join point al que representa, sólo esta disponible la información correspondiente a la parte estática.

thisJointPoint representa un objeto del tipo `org.aspectj.lang.JoinPoint`, mientras que thisJointPointStaticPart usa objetos con interfaces del tipo `org.aspectj.lang.JoinPoint.StaticPart`.

Una forma de tener más control sobre cómo afecta la declaración de los avisos al comportamiento de los objetos individuales es con las instancias de los aspectos. Para obtener una instancia de un aspecto se trabaja de la misma forma que para tener una instancia de una clase en Java. Lo más interesante de esto es componer instancias de aspectos con instancias de clases normales, donde juegan un papel muy importante los cortes.

Esta composición se puede realizar de dos formas:

1. Con una composición explícita que es soportada por las acciones de corte estáticas. Aquí todas las partes de estado y comportamiento se capturan por los objetos normales. Los aspectos pegan estas partes separadas.

2. Con un mecanismo de composición automático soportado por las acciones de corte no estáticas. Este mecanismo tiene una naturaleza más dinámica que el estático. Se ejecuta para los aspectos que tengan al objeto invocado en su dominio. Los objetos se insertan en el dominio de los aspectos utilizando el método addObject disponible para todos los aspectos.

3.2. Herramientas de AspectJ

A pesar de los numerosos beneficios que proporciona la programación orientada a aspectos aún no se ha adoptado totalmente como una nueva forma de trabajo. AspectJ supone una de las mayores herramientas de trabajo en programación orientada a aspectos y así se refleja en la *Figura 19*, donde se puede observar cómo en un pequeño plazo de tiempo el uso de AspectJ ha ido incrementándose notablemente.

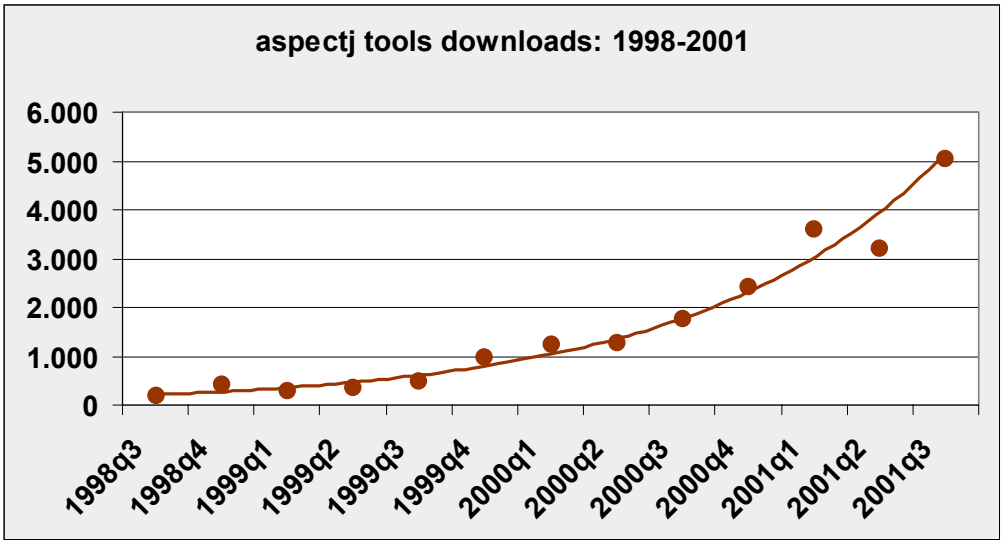


Figura 19: Evolución de la adopción de AspectJ como herramienta de desarrollo

En los primeros proyectos de AspectJ los usuarios encontraban dos barreras a la hora de adoptar AspectJ como herramienta de trabajo. La primera era que la mayoría de usuarios se preguntaban si su entorno de desarrollo soportaría AspectJ, y la segunda y la más importante era cómo saber que aspecto le afectaba a su sistema. Como respuesta AspectJ presento sus herramientas y su estructura para múltiples plataformas.

Actualmente las herramientas o componentes que presenta AspectJ están muy evolucionadas, la clara evolución de la adopción de AspectJ va ligada con la evolución y perfeccionamiento de las herramientas que ofrece a los usuarios.

El entorno de desarrollo de AspectJ (AJDE) se encuentra dentro del entorno de desarrollo IDE que es el utilizado por Java, pero también se puede utilizar por líneas de comandos.

3.2.1 AJC – AspectJ Compiler –

La herramienta ***ajc*** es un tejedor y compilador para el lenguaje AspectJ, puede ser invocado en un sistema operativo Windows y también Linux. También se puede poner a ejecutar directamente a través de la clase *org.aspectj.tools.ajc.main* (es decir, su método main()).

El comando ***ajc*** compila los archivos de AspectJ y Java, teje los aspectos necesarios para producir los archivos .class compilables en cualquier maquina virtual de Java (1.1 en adelante). Para tejer el bytecode también se acepta como entrada clases y aspectos en forma binaria.

El formato general de su invocación es el siguiente comando:

ajc [*options*] [*file...* | *@file...* | *-arglist file...*]

Los argumentos después de las opciones especifican los ficheros o archivos fuente a compilar (también se puede especificar las clases con la opción –injars). Los ficheros o archivos fuente pueden ser listados directamente en la línea de comandos o listados en un archivo. El @file y –arglist file son formas equivalentes, representan todos los ficheros o archivos fuente que están listados en un archivo. Cada línea de estos archivos debe contener una opción o el nombre de un fichero, los comentarios al igual que en java comienzan con // y terminan al final de la línea. Las distintas opciones que acepta ajc quedan recogidas en el *Anexo D*.

3.2.2 AJBROWSER – AspectJ program structure browser –

En programación orientada a objetos, las herramientas de desarrollo más usuales permitían al programador estructurar las clases de su programa de forma fácil gracias a un browser, este soporte permite al programador ver la herencia y sustituir su programa, gracias al soporte gráfico que proporciona un browser.

Para AspectJ se desarrolló una herramienta similar a la que pose la programación orientada a objetos. Mediante un browser se puede visualizar la estructura de los aspectos, las estructuras que se entrecortan(crosscutting) en un programa, etc. Este soporte muestra las uniones bidireccionales entre los aspectos (y sus avisos) y las clases (y sus elementos) que son afectadas por dichos aspectos o avisos.

AspectJ presenta una interfaz gráfica (GUI) para compilar los programas con ajc y un navegador que muestra la estructura transversal del programa (crosscutting).

El browser de AspectJ ajbrowser, puede editar los archivos fuente del programa, compilarlos usando el compilador de AspectJ (ajc) y navegar gráficamente en la estructura transversal del programa

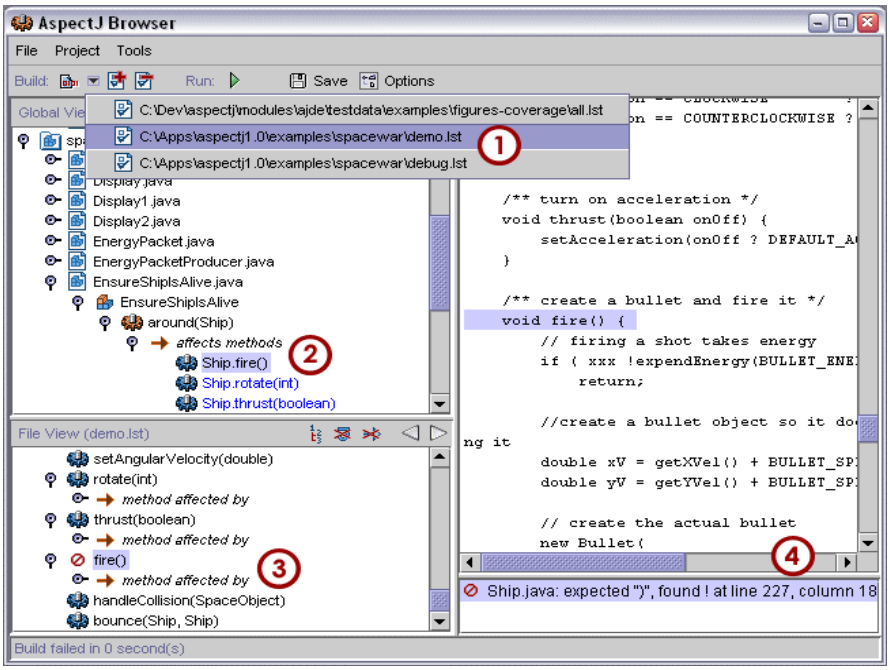


Figura 20: AspectJ Browser

3.2.3 AJDB – AspectJ Debugger –

Ajdb es el depurador(debugger) de AspectJ. Algunas versiones de AspectJ no lo incluyen, pero tampoco es una de las herramientas más necesarias ya que el compilador tiene una opción que crea un código sin interlineado, que es capaz de funcionar inicialmente con cualquier depurador de Java. Además la mayoría de los depuradores están comenzando a trabajar según el JSR-45, "depuración por otros lenguajes". Algunas versiones de AspectJ ya vienen preparadas para soportar JSR-45, otras versiones no disponen de dicho soporte inicialmente.

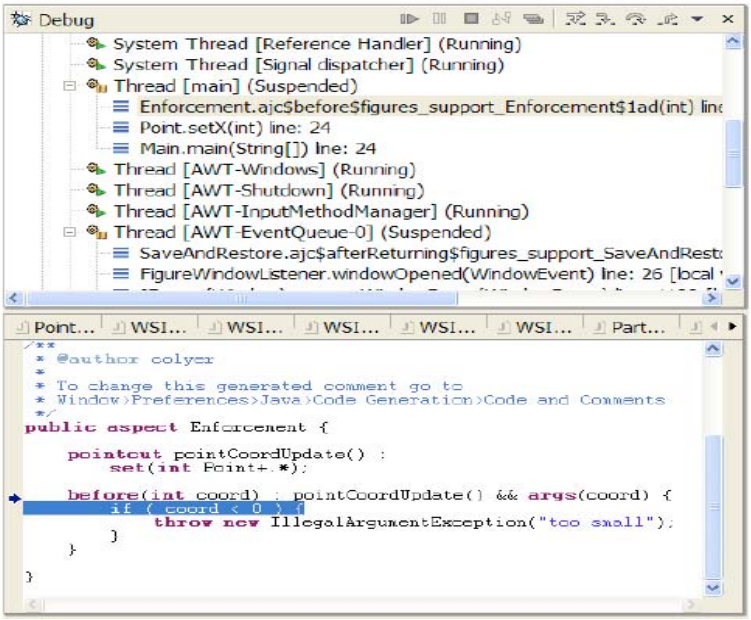


Figura 21: Aspect Debugger

3.2.4 AJDOC – AspectJ Documentation Generator –

La herramienta ajdoc genera una documentación (API) en formato HTML. Ajdoc está basado en la herramienta javadoc y por tanto acepta todos los parámetros estándar de javadoc.

El formato general de su invocación es el siguiente comando:

ajdoc [*options*] [*packagenames*] [*sourcefiles*] [*@files*]

Las principales diferencias entre ajdoc y javadoc son las siguientes:

- *ajdoc* genera información para crosscuts, advices e introducciones.
- *ajdoc* agrega enlaces a los elementos afectados por advices e introducciones.
- *ajdoc* enlaza los campos introducidos con sus correspondientes introducciones.
- *ajdoc* enlaza los métodos afectados por los advices con sus correspondientes advices.
- *ajdoc* necesita todos los archivos fuente listados en la línea de comandos o incluidos en el fichero files list.

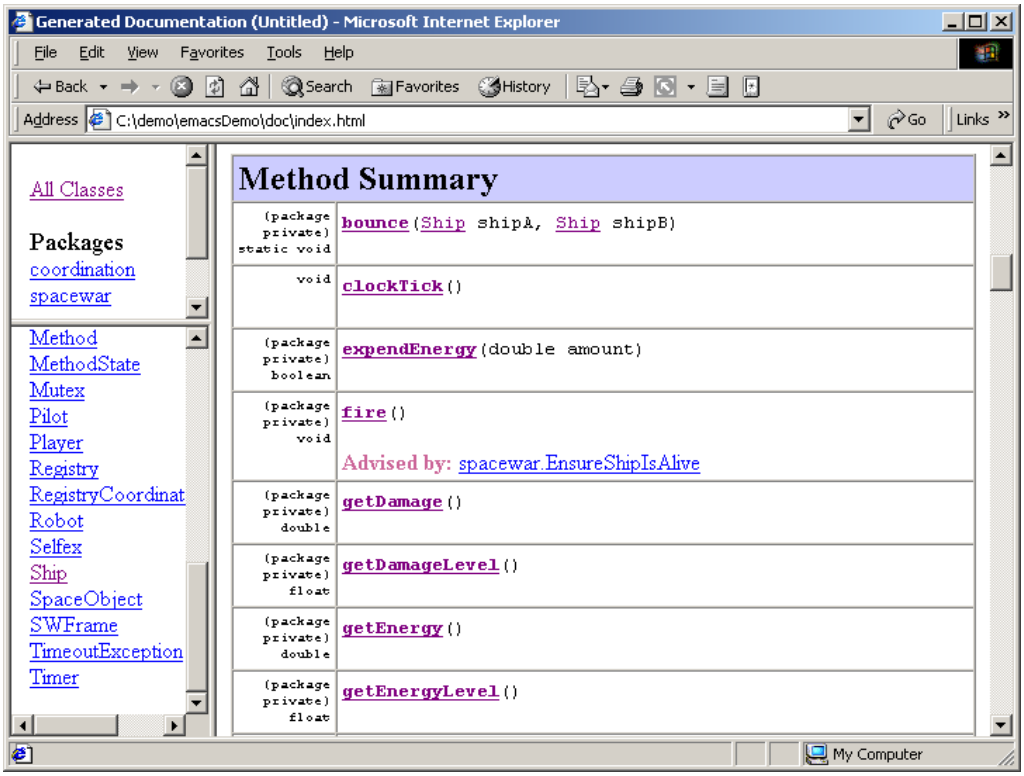


Figura 22: Aspect documentation generator.

3.3. Compatibilidad de AspectJ

Pero de nada sirve que AspectJ proporcione unas herramientas si no son compatible con las aplicaciones que los usuarios utilizan, por lo que el nuevo reto era crear diferentes extensiones de las herramientas de AspectJ para que sean lo más compatibles posibles, de esta manera la adopción de AspectJ como herramienta de trabajo se vera incrementada, porque hay que ofrecer unas buenas herramientas pero compatibles en múltiples plataformas y entornos de desarrollo.

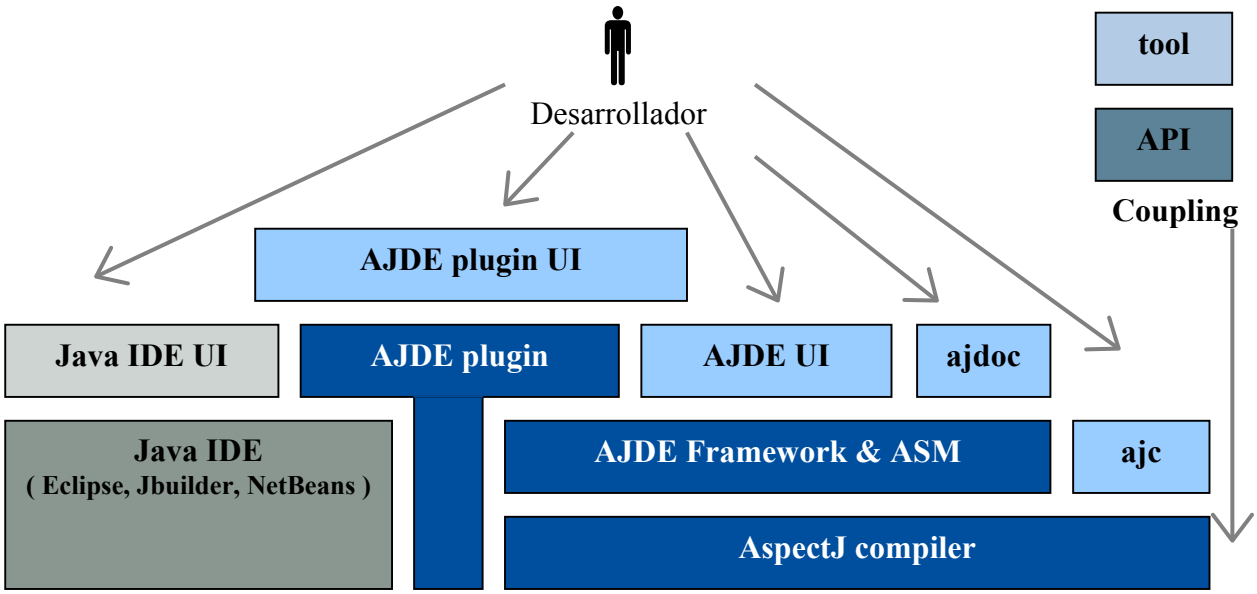


Figura 23: Visión global de la arquitectura de las herramientas

Los entornos de desarrollo soportados por AspectJ son :

- Eclipse
- JBuilder
- Emacs
- Forte/SunOne Studio 4
- NetBeans

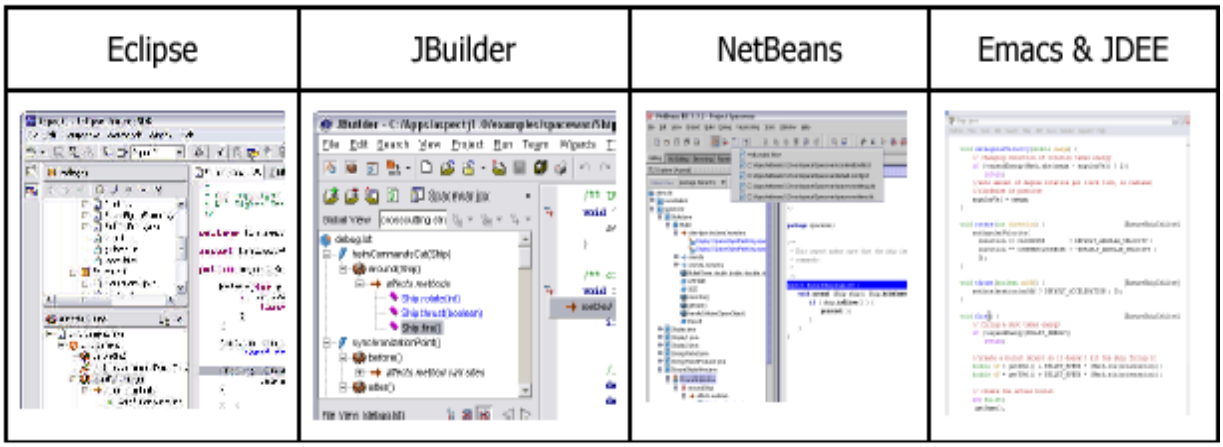


Figura 24: Extensiones de AspectJ para diferentes entornos de desarrollo(AJDE).

3.4 Ejemplo de Aplicación AspectJ

Come ejemplo se va a realizar la gestión de una cola circular. Se verá con un ejemplo práctico de cómo se reflejan las características de AspectJ en el código de las aplicaciones.

En primer lugar se implementará la cola circular en Java sin el aspecto de sincronización. Luego, se le añadirán las líneas de código necesarias para gestionar este aspecto y se podrá apreciar claramente cómo estas se diseminan por todo el programa, quedando un código poco claro. Después el mismo ejemplo se implementará con AspectJ.

A continuación se muestra el código Java de una cola circular, en principio sin restricciones de sincronización. La clase *ColaCircular* se representa con un array de elementos. Los elementos se van añadiendo en la posición *ptrCola* (por atrás) y se van borrando por *ptrCabeza* (por delante). Estos dos índices se ponen a cero al alcanzar la capacidad máxima del array. Esta inicialización se consigue utilizando el resto obtenido al dividir la posición a tratar entre el número total de elementos del array. Esta clase solamente contiene un constructor y dos métodos: *Insertar* para introducir elementos en la cola, y *Extraer*, para sacarlos.

```
public class ColaCircular
{
	private Object[] array;
	private int ptrCola = 0, ptrCabeza = 0;
	private int eltosRellenos = 0;

	public ColaCircular (int capacidad)
	{
		array = new Object [capacidad];
	}

	public void Insertar (Object o)
	{
		array[ptrCola] = o;
		ptrCola = (ptrCola + 1) % array.length;
		eltosRellenos++;
	}

	public Object Extraer ()
	{
		Object obj = array[ptrCabeza];
		array[ptrCabeza] = null;
		ptrCabeza = (ptrCabeza + 1) % array.length;
		eltosRellenos--;
		return obj;
	}
}
```

Código 3: Cola circular con Java sin restricciones de sincronización

El problema del *Código 3* es que si varios hilos solicitan ejecutar métodos del objeto ColaCircular, estos no están sincronizados, pudiéndose dar el caso de que la cola haya cubierto su capacidad y se sigan haciendo peticiones de inserción de elementos, y al contrario, es decir, que la cola esté vacía y se hagan peticiones de extracción de elementos. En estos casos, el hilo que hace la solicitud debería esperar a que se extraiga algún elemento, en el primer caso, o a que se inserte alguno en el segundo. Es decir, que se necesita código adicional para eliminar estas inconsistencias.

Java permite coordinar las acciones de hilos de ejecución utilizando *métodos e instrucciones sincronizada*s. A los objetos a los que se debe coordinar el acceso se le incluyen métodos sincronizados. Estos métodos se declaran con la palabra reservada **synchronized**. Un objeto solamente puede invocar a un método sincronizado al mismo tiempo, lo que impide que los métodos sincronizados en hilos de ejecución entren en conflicto.

Todas las clases y objetos tienen asociados un *monito*r, que se utiliza para controlar la forma en que se permite que los métodos sincronizados accedan a la clase u objeto. Cuando se invoca un método sincronizado para un objeto determinado, no puede invocarse ningún otro método de forma automática. Un monitor se libera automáticamente cuando el método finaliza su ejecución y regresa. También se puede liberar cuando un método sincronizado ejecuta ciertos métodos, como wait(). El subproceso asociado con el método sincronizado se convierte en no ejecutable hasta que se satisface la situación de espera y ningún otro método ha adquirido el monitor del objeto. Los métodos notify () se emplean para notificar a los hilos de ejecución en espera que el tiempo de espera se ha acabado. En Java, por defecto, no se define ninguna estrategia de sincronización.

La nueva versión sincronizada del ejemplo se muestra en el siguiente código. Donde se ha introducido la exclusión mutua y condiciones con guardas. El código que se introduce para la sincronización es el sombreado.

Como se puede observar los métodos Insertar y Extraer se han mezclado con el código correspondiente a la sincronización de los hilos. Además el código añadido se esparce por todo el método; no está localizado en una sola parte del código.

```
public class ColaCircular
{
	private Object[] array;
	private int ptrCola = 0, ptrCabeza = 0;
	private int eltosRellenos = 0;

	public ColaCircular (int capacidad)
	{
		array = new Object [capacidad];
	}
	public synchronized void Insertar (Object o)
	{
		while (eltosRellenos == array.length){
		try {
				wait ();
			}catch (InterruptedException e) {}
		}
		array[ptrCola] = o;
		ptrCola = (ptrCola + 1) % array.length;
		eltosRellenos++;
		notifyAll();
	}

	public synchronized Object Extraer ()
	{
		while (eltosRellenos == 0){
			try {
				wait ();
			}catch (InterruptedException e) {}
		}
		Object obj = array[ptrCabeza];
		array[ptrCabeza] = null;
		ptrCabeza = (ptrCabeza + 1) % array.length;
		eltosRellenos--;
		notifyAll();
		return obj;
	}
}
```

Código 4: Cola circular en Java con restricciones de sincronización

Ahora se implementará la cola circular con aspectos, en AspectJ. Primero se creará un aspecto para tratar la sincronización de la *Cola Circular* .

```
aspect ColaCirSincro
{
    private int eltosRellenos = 0;

    pointcut insertar(ColaCircular c):instanceof (c) &&
              receptions(void Insertar(Object));

    pointcut extraer(ColaCircular c):instanceof (c) &&
              receptions (Object Extraer());

    before(ColaCircular c):insertar(c)
    {antesInsertar(c);}

    protected synchronized void antesInsertar(ColaCircular c)

    {
           while (eltosRellenos == c.getCapacidad())
           {
                 try { wait(); }
```

```
                    catch (InterruptedException ex) {};
                }
        }
        after(ColaCircular c):insertar(c) { despuesInsertar();}

        protected synchronized void despuesInsertar ()
        {
                eltosRellenos++;
                notifyAll();
        }

        before(ColaCircular c):extraer(c) {antesExtraer();}

        protected synchronized void antesExtraer ()
        {
                while (eltosRellenos == 0)
                {
                        try { wait(); }
                        catch (InterruptedException ex) {};
                }
        }

        after(ColaCircular c):extraer(c) {despuesExtraer();}

        protected synchronized void despuesExtraer ()
        {
                eltosRellenos--;
                notifyAll();
        }
}
```

Código 5: Aspecto para definir la estrategia de sincronización mediante AspectJ.

```
public class ColaCircular{
      private Object[] array;
      private int ptrCola = 0, ptrCabeza = 0;
      public ColaCircular (int capacidad){
            array = new Object [capacidad];
      }
      public void Insertar (Object o){
            array[ptrCola] = o;
            ptrCola = (ptrCola + 1) % array.length;
      }
      public Object Extraer ()   {
            Object obj = array[ptrCabeza];
            array[ptrCabeza] = null;
            ptrCabeza = (ptrCabeza + 1) % array.length;
            return obj;
      }
      public int getCapacidad(){
            return capacidad;
      }
}
```

Código 6: Clase ColaCircular para un aspecto en AspectJ.

Se puede observar que la información que solamente interesa para la sincronización se ha pasado al código del aspecto, como es el caso del atributo *eltosRellenos*, la clase ColaCircular asociada al aspecto sólo a de conocer por donde se inserta y por donde se extrae.

La idea que se ha seguido para la creación del aspecto es bastante simple. Se han creado dos *pointcuts* (cortes), uno para capturar las llamadas al método *Insertar* y otro para capturar las llamadas al método *Extraer*.

Luego se ha definido cuatro *advices*(avisos), dos por cada pointcut. A cada pointcut le corresponde una advice *before* y otro *after*.

En el de tipo *before* se comprueba si se dan las condiciones necesarias para que se ejecute el método. En el caso de la inserción, la comprobación de que el buffer no esté lleno, y en el caso de la extracción que no esté vacío. Si alguna de estas condiciones no se cumple, el hilo que ejecuta el objeto se pone en espera hasta que se cumpla la condición.

En los avisos de tipo after se incrementa o decrementa el número de elementos del buffer, dependiendo si estamos en la inserción o la extracción, y se indica a los demás hilos que pueden proseguir.

Al dispararse el join point(eventos) asociado a los cortes(pointcut) se ejecuta el código definido en el bloque asociado al advice(aviso).

Como se comprobar con este ejemplo la aplicación queda más clara al aplicar la estrategia de sincronización usando los aspectos con AspectJ que si utilizáramos la programación Java tradicional. Además con AspectJ se pueden añadir tantos aspectos como la aplicación requiera, dotando a la aplicación de una mayor flexibilidad.

AspectJ es un lenguaje orientado a aspectos que está basado en el lenguaje Java. Tan sólo con las nuevas construcciones del lenguaje AspectJ (aspect, pointcut, advice, introduction) se ha conseguido dotar a la aplicación original en Java del requisito de sincronización, sin necesidad de realizar ninguna modificación previa para ello. Todo el código correspondiente a la sincronización a quedado recogido dentro de un módulo (aspecto) bien diferenciado. La aplicación tiene una mayor modularidad y una mayor adaptabilidad a cambios futuros.

Esta aplicación será compilada con las herramientas que AspectJ proporciona para dicho fin. Se puede utilizar el compilador ajc para generar el fichero .class correspondiente o también se puede hacer de una manera más gráfica desde el ajbrowser. Y todo esto sin preocupase del entorno donde se trabaja, porque AspectJ es compatible con múltiples plataformas y entornos de desarrollo.

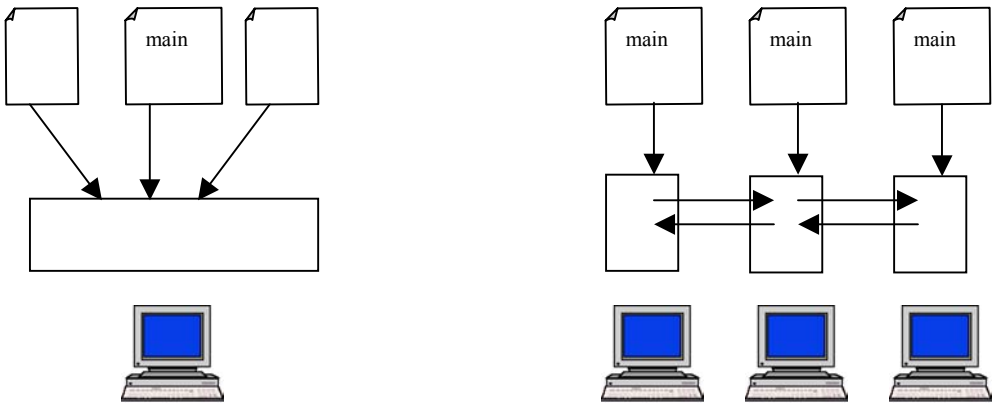
Introducción a la Programación Distribuida.

La gran mayoría de los sistemas de hoy en día requieren de los procedimientos remotos, de las computadoras distribuidas, de la programación distribuida, esto se debe tanto a distancias geográficas como a requerimientos de computo, ya que seria iluso pensar que las necesidades de computo de TODO un sistema fueran satisfechas por *una sola* computadora.

El concepto de sistema distribuido surge a partir de la necesidad que implica la comunicación y el deseo de compartir información y recursos a través de la red, la cual no se limitará únicamente al servicio de usuarios que geográficamente se encuentren dispersos, sino que además permitirá la evolución o retraso del sistema que se requiera. De esta manera los sistemas distribuidos solucionan las necesidades de:

- Repartir el volumen de información.
- Compartir recursos, ya sea en forma de software o hardware.

En un sistema distribuido el código del programa queda distribuido en varios programas que se ejecutan independientemente.



Aplicación no distribuida *Aplicación distribuida*

Figura25: Diferencia entre aplicaciones.

Para que un sistema se considere distribuido, debe presentar ciertas características desde el inicio hasta la aplicación final del sistema. Dichas características coinciden en la *transparencia*.

- *Transparencia al acceso:* indica que el sistema debe posibilitar la obtención de los recursos sin importar su acceso, ya sea local o remoto.

- *Transparencia en escalabilidad:* Debe permitir como el incremento o disminución de su tamaño

- *Transparencia en la migración:* Permitirá que existan movimientos en los objetos que componen al sistema, sin tener consecuencia alguna en los usuarios o las aplicaciones.

- *Transparencia en la ubicación del sistema:* No importa la geografía de los recursos existentes, ya que la transparencia en la ubicación del sistema, lo verá como una entidad. Por lo que también se podrá operar al mismo tiempo, sin que el trabajo de un interfiera en el otro.

- *Transparencia frente a fallos*: Aunque se presenten fallos en el hardware o software, un sistema distribuido procurará evitar pérdida en las tareas de los usuarios. Un buen tratamiento de los fallos aumenta la disponibilidad del sistema.

En general se pueden encontrar nociones de transparencia de acceso, de ubicación, de concurrencia, de replicación, frente fallos, de movilidad, de prestaciones y al escalado.

Como podemos comprobar los sistemas distribuidos por definición poseen grandes ventajas que facilitan el trabajo al usuario:

- Una de ellas es el *costo-rendimiento*, el cual es prácticamente reducido, ya que con el avance tecnológico las computadoras se vuelven cada vez más necesarias y el rendimiento cada vez más elevado, debido a los sistemas de comunicación que han permitido la implementación de protocolos que permiten la efectiva y no prolongada transmisión de los datos.

- Un sistema distribuido es también *escalable*, ya que tiene la posibilidad de crecer, incrementando su capacidad de procesamiento agregando servidores y procesadores, lo cual lo hace también *escalable*.

- La *modularidad* es también otra de las ventajas de los sistemas distribuidos, ya que éste deja de ser centralizado y permite que cada entidad sea independiente y programada cuidadosamente, de tal manera que tenga un optimo desempeño dentro de la comunidad donde se ejercerán los servicios.

- Por último, la *disponibilidad* representa otra ventaja, debido a que cada entidad se encuentra programada dentro del concepto de redundancia y así los servicios permanecen al alcance de quien los solicite, a pesar de que ocurra algún fallo en ellos.

Finalmente es necesario señalar que, para que un sistema tenga estas ventajas, es imprescindible que se encuentre respaldado por una tecnología que le permita localizar los objetos remotos, comunicarse con ellos y obtener de ellos los procesos que se soliciten.

En la mayoría de los casos, las aplicaciones distribuidas funcionan siguiendo un modelo *cliente/servidor*.

4.1. El Modelo Cliente/Servidor

La arquitectura cliente/servidor es la plataforma abierta por excelencia, por la variedad de combinaciones de clientes y servidores que permiten conectar en red. Sin embargo, elegir las plataformas, las herramientas, los proveedores y las bases de administración de esta arquitectura, además de la tecnología de creación, es una decisión difícil de toma.

Esta arquitectura es un modelo para el desarrollo de sistemas de información, en las que las transacciones se dividan en elementos independientes que cooperan entre sí para intercambiar información, servicios o recursos.

La arquitectura posee un conjunto de procesos clientes y servidores. El proceso de cada uno de los usuarios, llaman a un cliente e inicializan un diálogo: produce una demanda de información o solicita recursos. El proceso que responde a la demanda del cliente, se le conoce como servidor.

El modelo de cliente-servidor precisa además de unos recursos (software), todos los estos recursos son manejados por los procesos servidor. Cliente y Servidor deben hablar el mismo lenguaje para conseguir una comunicación efectiva, el primero solicita al segundo unos recursos y este último los concede, le hace esperar o lo deniega según los permisos que tenga. Uno o más servidores crean unos objetos locales y luego atienden peticiones de acceso sobre esos objetos provenientes de clientes situados en lugares remotos de la red.

Bajo este modelo cada usuario tiene libertad de obtener la información que requiera en un momento dado proveniente de una o varias fuentes locales o distantes y procesarla como según le convenga.

Los clientes y los servidores pueden estar conectados a una red local o a una red amplia, como la que se puede implementar en una empresa o una red mundial como lo es Internet. Cliente/Servidor es el modelo de iteración más común entre aplicaciones de una red.

Asimismo esta arquitectura posee una infraestructura versátil modular basada en mensajes, que pretende mejorar la portabilidad, la interoperabilidad y la escalabilidad del computo. Representa una parte fundamental del sistema a desarrollar para cumplir con el objetivo de poner a disposición los servicios de procesamiento de imágenes a una comunidad de usuarios distribuidos.

Aquí resulta útil señalar que los servicios que se van a implementar se encontrarán físicamente en el *servidor*

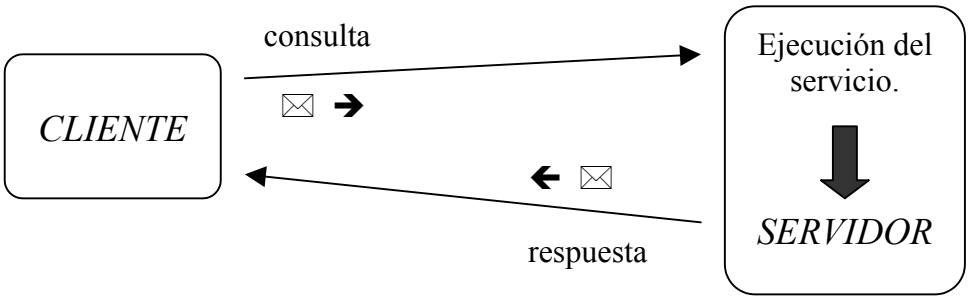


Figura 26: Modelo cliente/servidor

En el modelo Orientado a Objetos hay una serie de objetos que solicitan servicios (clientes) a los proveedores de los servicios (servidores) a través de una interfaz de encapsulación definida. Un cliente envía un mensaje a un objeto (servidor) y éste decide qué tecnología ejecutar.

Sin duda, una de las tecnologías más populares (hasta el momento) ha sido *CORBA* (Common Object Request Broker Architecture). *CORBA* es una tecnología creada y mantenida por el *OMG* (Object Management Group), un consorcio formado por un grupo de empresas implicadas en el uso de programación distribuida. La principal característica de *CORBA* es su independencia del lenguaje de programación mediante el uso de una interfaz de programación común denominada *IDL* (*CORBA* Interface Definition Language).

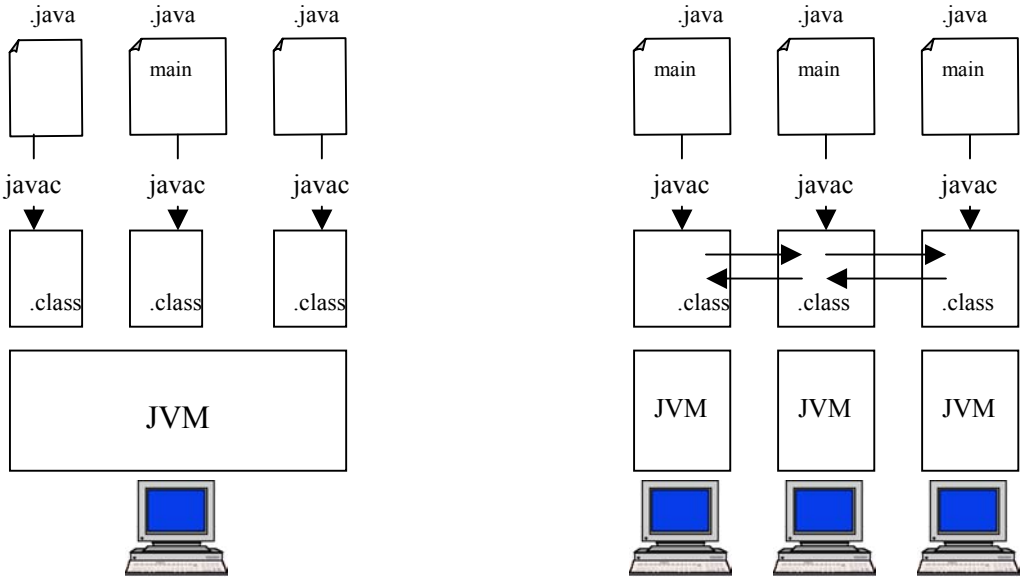
Desde Java es posible usar *CORBA* mediante Java *IDL* y aprovechar todas sus características, sin embargo, los ingenieros de *SUN* debieron pensar que esta solución era poco elegante y decidieron implementar una tecnología propia de programación distribuida orientada a objetos totalmente basada en *JAVA*. Esta tecnología es conocida como *RMI* (Remote Method Invocation).

RMI proporciona una potente herramienta al programador *JAVA* poniendo a su alcance una capacidad de programación distribuida 100% *JAVA*. La idea básica de *RMI* es que, objetos ejecutándose en una VM (Virtual Machine) sean capaces invocar métodos de objetos ejecutándose en VM's diferentes. Haciendo notar que las VM's pueden estar en la misma máquina o en máquinas distintas conectadas por una red.

4.2. El Modelo de Objeto Distribuido de JAVA

Cuando se construyen aplicaciones distribuidas es posible aprovechar en toda su extensión el paradigma de la programación orientada a objetos. Los objetos no son solo elementos independientes, sino que además, se encuentran distribuidos en diversos ordenadores conectados a través de una red. Se puede decir, que se acerca un poco más a como funciona el mundo real, donde los objetos que se manejan diariamente no suelen estar en el mismo sitio.

El modelo de objeto distribuido que usa Java permite que los objetos que se ejecutan en una *Máquina Virtual de Java (JVM)* invoquen a los métodos de objetos que se ejecutan en otras JVM. Estas otras JVM pueden ejecutarse como proceso separado en la misma computadora o en otras computadoras remotas. El objeto que hace la invocación del método se denomina objeto cliente o objeto local, mientras que el objeto cuyos métodos se están invocando se denomina objeto servidor o objeto remoto.



Aplicación Java no distribuida

Aplicación Java distribuida

Figura 27: Diferencia entre aplicaciones Java.

El modelo de objetos distribuidos de Java se desarrolló teniendo como meta acercarlo lo más posible al modelo de objetos locales de Java. Como es de esperar, un objeto remoto no puede ser exactamente igual a uno local, pero es similar en dos aspectos muy importantes:

- Se puede pasar una referencia a un objeto remoto, como argumento o como valor de retorno en la invocación de cualquier método, ya sea local o remoto.

- Se puede forzar una conversión de tipos de un objeto remoto a cualquier interfaz remota, mediante la sintaxis normal de Java que existe para este propósito.

Sin embargo el modelo de objetos distribuidos difiere con el modelo de objetos locales en los siguientes aspectos:

- Los clientes de los objetos remotos interactúan con las interfaces remotas y nunca con las clases que implementan dichas interfaces.

- Los argumentos de los métodos remotos, así como los valores de retorno, son pasados por copia y no por referencia.

- Los objetos remotos se pasan por referencia y no mediante la copia de la implantación del objeto.

- La semántica de algunos métodos definidos por la clase java.lang.Object, está especializada para el caso de los objetos remotos.

- Los clientes deben tener en cuenta excepciones adicionales referentes a la invocación remota de los métodos.

Una aplicación implementada según el modelo de objeto distribuido de Java se compone de unos elementos principales :

- Interfaces remotas.
- Implementación de los objetos.
- Referencias a objetos remotos.
- Sistema Runtime.
- Acceso a los objetos(obtención de las referencias remotas).

- Interfaces Remotas:

  En el modelo de objeto distribuido de Java, un objeto cliente nunca hace referencia directa a un objeto remoto. En lugar de ello, hace referencia a una interfaz remota que implementa el objeto remoto. Esta interfaz habrá sido acordada entre el cliente y el servidor y contiene los tipos de objetos del servidor que van a ser accesibles desde el cliente, los métodos incluidos en esta interfaz son los que podrá invocar el cliente.

  El uso de interfaces remotas permite:

  - Que los objetos servidor diferencien entre sus interfaces locales y remotas.

  - Que los objetos servidor presenten modos diferentes de acceso remoto.

  - Que la posición del objeto servidor dentro de una clase jerárquica se abstraiga de la manera en que esta se utiliza, lo que permite compilar los objetos cliente por medio de la interfaz remota, sin necesidad de que los archivos de clases del servidor estén presentes localmente durante el proceso de compilación.

- Implementación de los objetos:

  La implementación de los objetos forma parte del programa servidor, esta implementación debe concordar con la interfaz, también puede extenderla si lo necesita. Y por supuesto la implementación de los métodos se ejecutara en el servidor.

- Referencia a objetos remotos:

  La noción de referencia a objeto se extiende para permitir que cualquier objeto que pueda recibir una invocación a método remoto tenga una referencia remota. Una referencia a objeto remoto es un identificador que puede usarse a lo largo de todo un sistema distribuido para referirse a un objeto remoto particular único.

  A estas referencias se les suele llamar *Stubs* , son objetos que residen en el cliente y representan a objetos remotos (host + puerto + ID del objeto).

  Su clase es generada automáticamente a partir de la interfaz, los objetos de estas clases implementan dicha interfaz por lo que se puede invocar los métodos sobre ellos.

  Las referencias a los objetos remotos son análogas a las locales en cuanto a que:

  - El objeto remoto donde se recibe la invocación de método remoto se especifica mediante una referencia a objeto remoto.
  - Las referencias a objetos remotos pueden pasarse como argumentos y resultados de las invocaciones de métodos remotos.

- Sistema Runtime:

  Este sistema actúa de mediador entre los stubs y los objetos remotos.

- Acceso a los objetos:

  El acceso a los objetos remotos es la obtención de la referencia de los objetos remotos, es decir obtener los stubs.

  Podemos obtener los stubs de los objetos remotos mediante un sistema de nombrado(lo más usual) o mediante algún método que envíe los objetos.

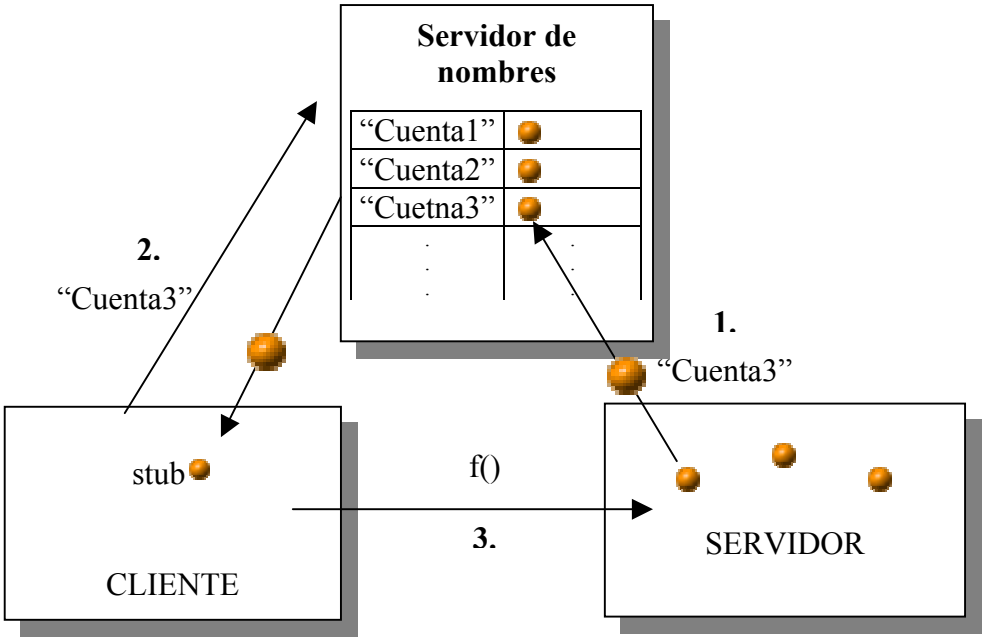


Figura 28: Obtener stub: servicio de nombrado.

A la hora de construir una aplicación distribuida en Java podemos seguir varios enfoques:

- Sockects TCP: Java los admite, y con ellos podrá construir aplicaciones tradicionales cliente/servidor. Aunque para ello deberá definirse un protocolo que le permita interpretar los mensajes que se envían durante la comunicación entre el cliente y el servidor.

- Entorno de computación distribuido (DCE): Basado en la RPC (llamadas de procedimientos remotos), constituye un enfoque orientado a procedimientos para el desarrollo de aplicaciones distribuidas. Las RPC no se acoplan bien con las aplicaciones distribuidas orientadas a objetos. El enfoque de la invocación remota de métodos que admite Corba se adapta mucho mejor al modelo de objetos de Java.

- Modelo de objeto componente distribuido (DCOM): Se basa en la RPC de DCE, pero ofrece posibilidades de programación orientadas a objetos a través de objetos, interfaces y métodos del COM. Además, el DCOM proporciona servicios de seguridad amplios. El entorno de desarrollo Java de Microsoft, Visual J++, ofrece la posibilidad de acceder a objetos Com y Dcom desde Java. No obstante, esta posibilidad constituye mas bien un puente a las tecnologías de herencia que una extensión distribuida del modelo de objetos de Java.

- Arquitectura de intermediación de solicitud de objetos comunes (CORBA): Proporciona un enfoque excelente a la construcción de aplicaciones distribuidas orientadas a objetos, ya que está orientada a objetos y es una solución no vinculada a ningún sistema operativo. Java admite CORBA, sin embargo, Corba esta diseñada para admitir un modelo de objetos independiente del lenguaje. La arquitectura RMI de Java posee todas las ventajas de Corba, pero está especialmente adaptada al modelo de objetos Java. Esto hace que RMI de Java sea mucho más eficaz y fácil de usar que Corba en lo que respecta a aplicaciones de Java puro.

Invocación de métodos remotos Java RMI

En la actualidad la programación distribuida ocupa un lugar importante tanto en las ciencias de las comunicaciones como en la industria debido a que muchos de los problemas a los que se enfrentan son esencialmente distribuidos. De la misma manera, las tecnologías orientadas a objetos se han consolidado como una de las herramientas más eficaces en el desarrollo de software debido principalmente a su capacidad de describir los problemas en el dominio del problema más que en el dominio de la solución.

Dentro del ámbito del tratamiento de la información distribuida se incorpora fuertemente la tecnología orientada a objetos debido a que en el paradigma basado en objetos el estado de un programa ya se encuentra distribuido de manera lógica en diferentes objetos, lo que hace que la distribución física de estos objetos en diferentes procesos o computadoras sea una extensión natural.

RMI [2] fue el primer framework que apareció con la idea de crear sistemas distribuidos para Java, proporcionando una potente herramienta al programador Java que encuentra a su alcance una capacidad de programación distribuida cien por cien Java. Además, viene integrado en cualquier máquina virtual Java posterior a la versión 1.1

La idea básica de RMI es que, objetos ejecutándose en una VM(*Máquina Virtual*) sean capaces de invocar métodos de objetos ejecutándose en VM's diferentes. Haciendo notar que las VM's pueden estar en la misma máquina o en máquinas distintas conectadas por una red.

RMI es una forma de RPC (Remote Procedure Call). La invocación de métodos remotos permite que un objeto que se ejecuta en una máquina puede invocar métodos de un objeto que se encuentra en ejecución bajo el control de otra máquina (por supuesto no hay problemas para las relaciones entre los objetos cuando ambos son locales). En definitiva, RMI permite crear e instanciar objetos en máquinas locales y al mismo tiempo crearlos en otras máquinas (máquinas remotas), de forma que la comunicación se produce como si todo estuviese en local.

RMI se convierte así en una alternativa muy viable para realizar aplicaciones distribuidas, proporcionando una serie de particularidades destacables:

- RMI permite abstraer las interfaces de comunicación a llamadas locales, no se necesita fijarse en el protocolo y las aplicaciones distribuidas son de fácil desarrollo.

- RMI te permite trabajar olvidándote del protocolo.

- RMI es flexible y extensible, destaca su recolector de basura.

- Permite disponer de objetos distribuidos utilizando Java.

- Un objeto distribuido se ejecuta en una JVM remota.

- RMI facilita que un cliente interactúe con un objeto remoto por referencia, o por su valor descargándolo y manipulándolo en el ambiente de ejecución local. Esto se debe a que todos los objetos en RMI son objetos Java. RMI utiliza las capacidades de serialización del objeto, que proporciona el lenguaje Java, para transportar los objetos desde el servidor al cliente.

- Es similar a RPC, si bien el nivel de abstracción es más alto. Ambos realizan llamadas a procedimientos no locales.

- Se puede generalizar a otros lenguajes utilizando JNI.

- RMI pasa los parámetros y valores de retorno utilizando serialización de objetos.

- Dado que RMI es una solución cien por cien Java para objetos remotos, proporciona todas las ventajas de las capacidades de Java, puede ser utilizado sobre diversas plataformas, desde mainframes a cajas UNIX hasta máquinas Windows que soporten dispositivos mientras haya una implementación de JVM para esa plataforma.

- Cualquier tipo de información es manejada por el mismo objeto la cuál puede ser buscada utilizando reflexión e introspección.

5.1. Aplicación Genérica RMI

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente, que necesitan de un tercero, el registro de objetos. Una aplicación servidor típica crea un montón de objetos remotos, hace accesibles unas referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Una aplicación cliente típica obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos.

- Servidor:

  - Crea objeto remoto servidores
  - Crea referencia al objeto remoto, haciéndolas accesibles.
  - Espera a que un cliente invoque un método del objeto remoto

- Cliente:

  - Obtiene referencia a un objeto remoto en el servidor
  - Invoca un método del objeto remoto del servidor (petición de servicio)

- Registro de objetos remotos:

  El registro de objetos remotos proporciona servicios para que:
  - Los servidores se registren, de forma que estén accesibles referencias a los mismos.
  - Los clientes obtengan referencias de los servidores.

RMI proporciona el mecanismo por el que se comunican y se pasan información del cliente al servidor y viceversa. Cuando es una aplicación algunas veces se refiere a ella como *Aplicación de Objetos Distribuidos.*

Las aplicaciones de objetos distribuidos con RMI en definitiva necesitan.

- **Localizar Objetos Remotos:** Las aplicaciones pueden utilizar uno de los dos mecanismos para obtener referencias a objetos remotos. Puede registrar sus objetos remotos con la facilidad de nombrado de RMI *rmiregistry*. O puede pasar y devolver referencias de objetos remotos como parte de su operación normal.

- **Comunicar con Objetos Remotos:** Los detalles de la comunicación entre objetos remotos son manejados por el RMI; para el programador, la comunicación remota se parecerá a una llamada estándar a un método Java.

- **Cargar Bytecodes para objetos que son enviados:** Como RMI permite al llamador pasar objetos Java a objetos remotos, RMI proporciona el mecanismo necesario para cargar el código del objeto, así como la transmisión de sus datos.

Una de las principales características de RMI es la habilidad de descargar los **bytecodes** (o simplemente, **código**) de una clase de un objeto si la clase no está definida en la máquina virtual del recibidor. Los tipos y comportamientos de un objeto, anteriormente sólo disponibles en una sola máquina virtual, ahora pueden ser transmitidos a otra máquina virtual, posiblemente remota. RMI pasa los objetos por su tipo verdadero, por eso el comportamiento de dichos objetos no cambia cuando son enviados a otra máquina virtual. Esto permite que los nuevos tipos sean introducidos en máquinas virtuales remotas, y así extender el comportamiento de una aplicación dinámicamente.

RMI utiliza el mecanismo de serialización de objetos de Java para transmitir datos entre máquinas, pero además agrega la información de localización necesaria para permitir que las definiciones de las clases se puedan cargar a la máquina que recibe los objetos.

La siguiente ilustración muestra una aplicación RMI distribuida que utiliza el registro para obtener referencias a objetos remotos. El servidor llama al registro para asociar un nombre con un objeto remoto. El cliente busca el objeto remoto por su nombre en el registro del servidor y luego llama a un método. Esta ilustración también muestra que el sistema RMI utiliza una servidor Web existente para cargar los bytecodes de la clase Java, desde el servidor al cliente y desde el cliente al servidor, para los objetos que necesita.

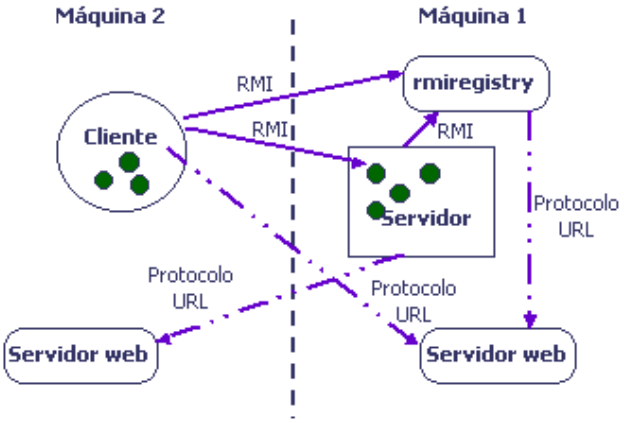


Figura 29: Invocación remota de objetos, utilizando el servicio de nombres rmiregistry

El sistema RMI utiliza un servidor Web para cargar los bytecodes de la clase Java, desde el servidor al cliente y desde el cliente al servidor.

5.2. Arquitectura y Comunicación de JAVA RMI

La arquitectura de RMI puede verse como un modelo de cuatro capas:

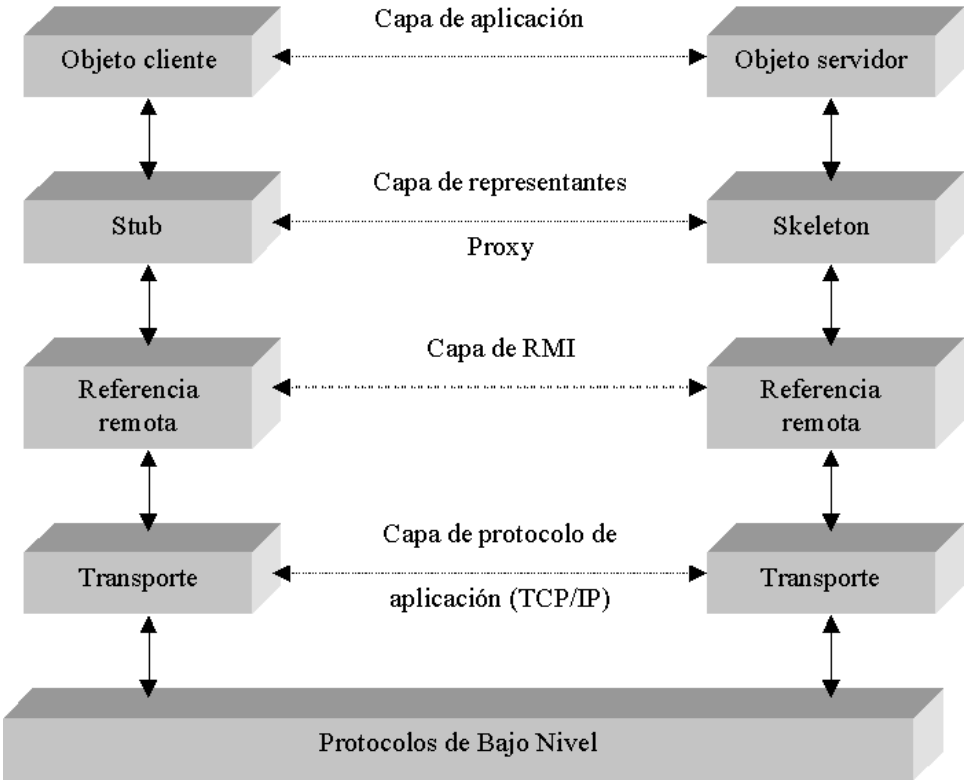


Figura 30: Arquitectura de RMI.

Cada capa es independiente de las otras y tiene definido su propia interfaz y protocolo, de forma que una capa puede ser cambiada sin afectar a las otras. Por ejemplo, la capa de transporte puede utilizar distintos protocolos como TCP, UDP… sin que el resto de las capas se vean afectadas en su funcionamiento.

La comunicación entre las capas se realiza por medio de la abstracción de stream o flujos de datos.

5.2.1. Capa de Aplicación

La primera capa es la *capa de aplicación* y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos.

Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda de java.rmi.Remote. Dicha interfaz se usa básicamente para marcar un objeto como remotamente accesible.

Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende de la clase UnicastRemoteObject(paquete java.rmi.server), o puede hacerse de forma explícita con una llamada al método exportObject() del mismo paquete.

5.2.2. Capa de Representantes Proxy

La segunda capa es la capa proxy o capa stub-skeleton. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

El punto de contacto de la aplicación cliente con el objeto remoto se hace por medio de un stub local. Este stub implementa la interfaz del objeto remoto y gestiona toda la comunicación con el servidor a través de la capa de referencia remota. A todos los efectos, el stub es la representación local del objeto remoto.

RMI utiliza el mismo mecanismo que los sistemas RPC para implementar la comunicación con los objetos remotos, el basado en skeletons y stubs.

Los stubs forman parte de las referencias y actúan como representantes de los objetos remotos ante sus clientes. En el cliente se invocan los métodos del stub, quien es el responsable de invocar de manera remota al código que implementa al objeto remoto. En RMI un stub de un objeto remoto implementa el mismo conjunto de interfaces remotas que el objeto remoto al cual representa.

Cuando se invoca algún método de un stub, se realizan las siguientes acciones:

- Inicializar una conexión con la Máquina Virtual(MV) que contiene al objeto remoto.

- Serializar (*marshalls*) y transmitir los parámetros de la invocación a la MV remota.

- Esperar el resultado de la invocación.

- Deserializar (*unmarshalls*) el valor de retorno o la excepción.

- Devolver el valor a quien lo solicito.

Los stubs se encargan de ocultar la serialización de los parámetros, así como los mecanismos de comunicación empleados.

En la parte del servidor, es decir en la máquina virtual remota, cada objeto posee un skeleton correspondiente, el skeleton es el equivalente al stub en el cliente. Se encarga de traducir las invocaciones a los objetos remotos que provienen de la capa de referencia remota, así como de gestionar las respuestas.

Un skeleton realiza las siguientes acciones cuando recibe una invocación:

- Deserializar los parámetros necesarios para la ejecución del método remoto.

- Invocar el método de la implantación del objeto remoto.

- Serializar los valores de retorno y enviarlos de vuelta al cliente.

5.2.3. Capa de Referencia Remota

La tercera capa es la capa RMI o la capa de referencia remota. Esta capa es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de conexiones perdidas.

En esta capa se espera una conexión del tipo *stream (stream-oriented connection)* desde la capa de transporte.

5.2.4. Capa de Transporte

Esta última capa es la capa de transporte o capa de protocolo de aplicación. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP *(Java Remote Method Protocol)*,que solamente es comprendido por programas Java.

5.2.5. Comunicación RMI

En la comunicación de RMI participan tres procesos: servidor, cliente y registro de objetos remotos llamado registro de nombres.

Primero el servidor registra el objeto remoto en el registro de nombres, el cliente obtiene una referencia del objeto registrado por el servidor, el cliente envía una petición al objeto remoto (que se encuentra en el servidor o es el propio servidor), el objeto remoto recibe la petición y ejecuta el procedimiento correspondiente, el objeto remoto envía la respuesta al cliente y por último el cliente recibe la respuesta.

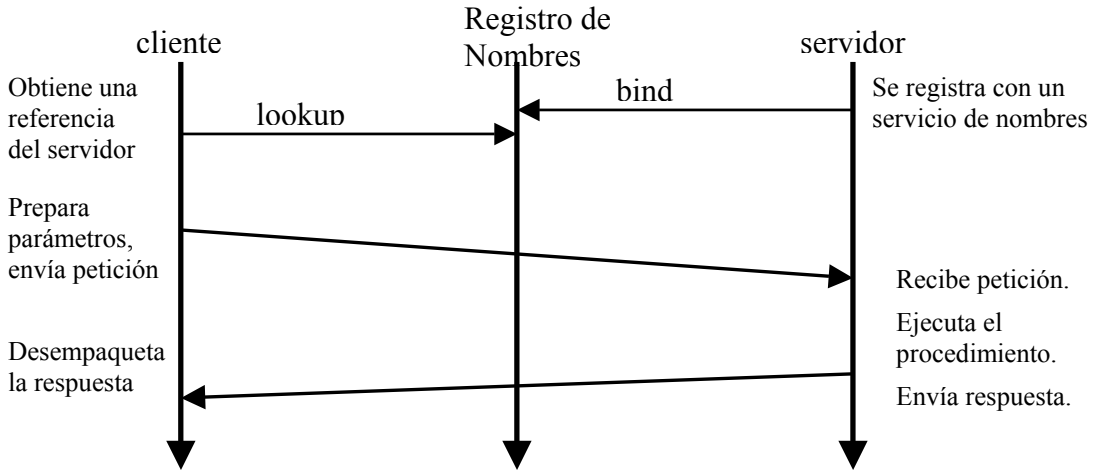


Figura 31: Diagrama de comunicación RMI.

5.3.Creación de Aplicaciones RMI

Cuando se utiliza RMI para desarrollar una aplicación distribuida, se deben seguir unos pasos generales:

1. Diseñar e implementar los componentes de la aplicación distribuida, interfaces, objetos remotos, clientes, servidores, etc.

2. Compilar los archivos fuentes, es decir compilar los archivos .java y después generar los stubs y skeletons correspondientes de cada objeto remoto.

3. Hacer las clases accesibles a la red registrándolas en el registro de nombres.

4. Arrancar la Aplicación.

Este proceso de desarrollo queda resumido gráficamente en la *Figura 32*.

5.3.1. Diseñar e Implementar los Componentes de la Aplicación Distribuida

Primero se decide la arquitectura de la aplicación y se determina qué componentes son objetos locales y cuales deberían ser accesibles remotamente. Este paso incluye:

- **Definir las Interfaces Remotas.** Una interface remota especifica los métodos que pueden ser llamados remotamente por un cliente. Los clientes programan los interfaces remotos, no la implementación de las clases de dichos interfaces. Parte del diseño de dichos interfaces es la determinación de cualquier objeto local que sea utilizado como parámetro y los valores de retorno de esos métodos; si alguno de esos interfaces o clases no existen aún también tenemos que definirlos. El servidor remoto de objetos debe declarar sus servicios por medio de una interfaz, extendiendo la interfaz de *java.rmi.Remote*. Cada método de la interfaz remota debe lanzar una excepción *java.rmi.RemoteException.*

- **Implementar los Objetos Remotos.** Los objetos remotos deben implementar uno o varios interfaces remotos. La clase del objeto remoto podría incluir implementaciones de otros interfaces (locales o remotos) y otros métodos (que sólo estarán disponibles localmente). Si alguna clase local va a ser utilizada como parámetro o cómo valor de retorno de alguno de esos métodos, también debe ser implementada. El servidor remoto debe implementar la interfaz remota y extender la clase *java.rmi.UnicastRemoteObject.*.

- **Implementar los Clientes.** Los clientes que utilizan objetos remotos pueden ser implementados después de haber definido los interfaces remotos, incluso después de que los objetos remotos hayan sido desplegados. El cliente debe usar la clase *java.rmi.Naming* para localizar el objeto remoto. Entonces, el cliente puede invocar los servicios de los objetos remotos entablando comunicación a través del stub.

5.3.2. Compilar los Archivos Fuentes y Generar Stubs y Skeletons.

- **Compilar los archivos fuentes.** Para compilar los archivos fuentes se usará el compilador de Java *javac,* estos archivos contienen las implementaciones de las interfaces remotas, las clases del servidor y del cliente.

- **Generar los stubs y skeletons.** El generador o compilador de stubs que acompaña a RMI es *rmic*. Este compilador se aplica al código compilado (.class) para generar los stubs del cliente y skeletons del servidor. Tanto los stubs como los skeletons actúan de mediadores en la comunicación, son los responsables de traducir los objetos a una representación apropiada para que se pueda realizar la invocación al método remoto. RMI utiliza una clase stub del objeto remoto como un proxy en el cliente de manera que los clientes puedan comunicarse con un objeto remoto particular. El compilador rmic toma los mismos parámetros de la línea de comandos que toma javac.

5.3.3. Hacer Accesibles las Clases en la Red.

En este paso, tenemos que hacer que todos los ficheros de clases Java asociados con los interfaces remotos, los stubs, y otras clases que necesitemos descargar en los clientes sean accesibles a través de un servidor Web.

- **Registrar el objeto remoto con el registro**. Todas las instancias de los objetos se deben registrar ante el registro RMI de modo que puedan ser conocidos por los clientes. Para lograrlo, se deben usar los métodos de la clase *java.rmi.Naming*, la cual asocia un nombre al servidor. Esta clase es la infraestructura de registro RMI para almacenar los nombres. Los servidores, una vez registrados, ya pueden ser conocidos e invocados por los clientes.

5.3.4. Arrancar la Aplicación.

Arrancar la aplicación incluye ejecutar el registro de objetos remotos de RMI, el servidor y el cliente.

- **Arrancar el registro RMI en el servidor**:

  RMI define interfaces para un servicio de nombrado no persistente llamado Registry. RMI tiene una implementación de este objeto que permite recuperar y registrar servidores usando nombres simples.

  Cada servidor puede soportar su propio registro o tener un solo registro independiente que admita a todas las máquinas virtuales disponibles en la computadora del servidor. Para arrancar un objeto registro en el servidor se lanza el comando *rmiregistry*.

  Puesto que las bases de datos del registro esta vacía cuando el servidor comienza, todos los objetos remotos que se construyan se deben insertar.

- **Iniciar la ejecución del Servidor:**

  Antes de comenzar la ejecución, se debe cargar la clase del servidor y entonces crear las instancias de los objetos remotos.

- **Iniciar la ejecución del Cliente:**

  Antes de comenzar la ejecución, se debe cargar la clase del cliente así como los stubs. RMI también ofrece mecanismos de seguridad para descargar por demanda a diferentes representantes provenientes del servidor.

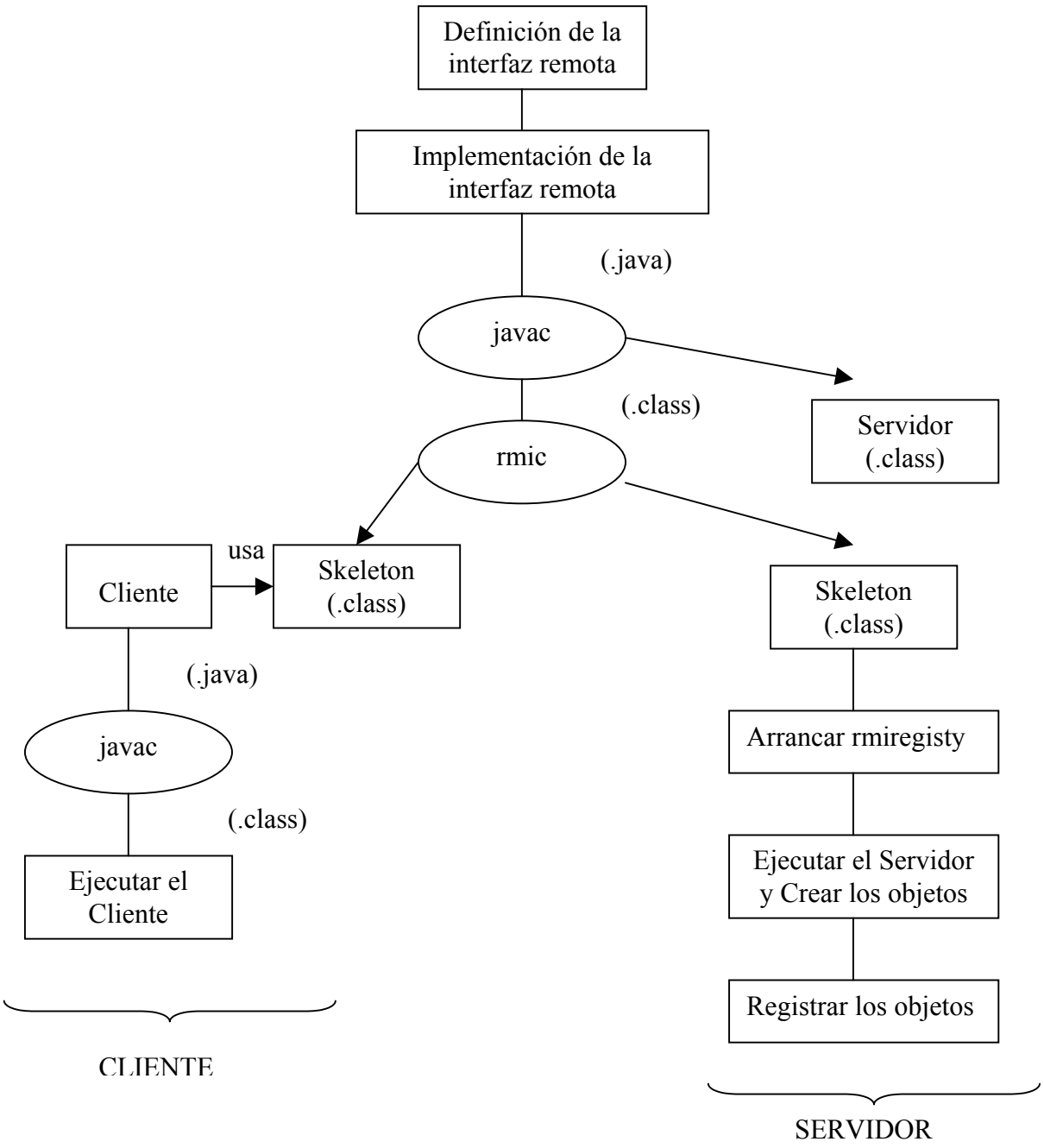


Figura 32 : Proceso de desarrollo de una aplicación RMI

5.4. Ejemplo de Aplicación RMI

Vamos a ver un sencillo ejemplo donde un servidor remoto saluda "Hello" a sus clientes, es el famosísimo "Hello World" pero esta vez remoto.

Este ejemplo consiste en un servidor donde se encuentra instanciado un objeto remoto con un único método y un cliente que accede al objeto del servidor e invoca su método. La invocación de este método le devuelve al cliente el mensaje "Hola Mundo" y este lo visualiza por pantalla. El siguiente esquema muestra el funcionamiento de este ejemplo.

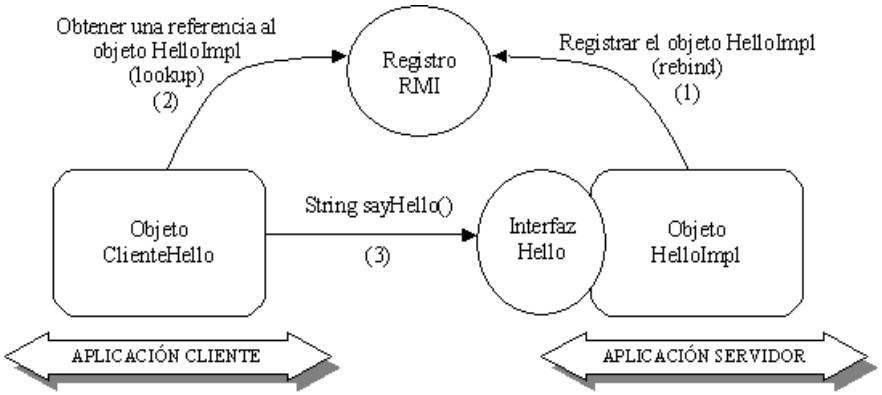


Figura 33: Funcionamiento del ejemplo Hello World

Los elementos involucrados en este ejemplo son los siguientes:

- Interfaz remota Hello que el objeto remoto HelloImpl debe implementar.

- Aplicación servidor que esta formada únicamente por el objeto remoto HelloImpl.

- Aplicación cliente formada por el objeto local ClienteHello.

Para terminar este ejemplo se compilarán los fichero fuentes .java de todos los elementos involucrados, se generarán los stubs y skeletons de los objetos remotos y se ejecutara el sistema con la aplicación cliente y aplicación servidor en máquinas separadas.

5.4.1 Interfaz Remota

- **Definición de la interfaz remota:**

Una interfaz remota define los métodos que van a poder ser invocados remotamente por los clientes. En nuestro ejemplo, se definirá una interfaz remota llamada *hello* que contendrá un único método llamado *sayHello()*.

Para definir la interfaz remota se deberán tener en cuenta los siguientes aspectos:

- Toda interfaz remota debe derivar de la interfaz *java.rmi.Remote*. Extendiendo de esta interfaz, la interfaz *Hello* queda marcada como una interfaz remota y por tanto sus métodos podrán ser invocados remotamente.

  public interface Hello extends java.rmi.Remote

- Todos los métodos de la interfaz deben estar preparados para poder lanzar la excepción *java.rmi.RemoteException*. Esta excepción será lanzada en la invocación de un método remoto cuando se produzca algún fallo en la comunicación. El método *sayHello()*, por tanto, deberá tenerla declarada en su prototipo.

  String sayHello() throws java.rmi.RemoteException;

A continuación, se presenta el código completo de la interfaz remota *Hello*:

```
package hello;
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface Hello extends Remote{
   String sayHello() throws RemoteException;
}
```

Código 7: Interfaz remota Hello con RMI.

- **Implementación de la interfaz remota**

Una vez que tenemos definida la interfaz remota deberemos definir una clase que la implemente. En nuestro ejemplo, definiremos la clase *HelloImpl* para que implemente la interfaz remota *Hello*.

Un objeto remoto es aquel que implementa una interfaz remota. En nuestro caso, todos los objetos de la clase *HelloImpl* implementaran la interfaz remota *Hello*, siendo por tanto objetos remotos que podrán recibir invocaciones remotas a los métodos de la interfaz por parte de los clientes.

A la hora de definir una clase que implemente una interfaz remota habrá que seguir los siguientes pasos:

- Declarar las interfaces remotas que va a implementar. Nuestra clase *HelloImpl* solamente va a implementar el interfaz remoto *Hell*o.

```
public class HelloImpl implements Hello
{ ...}
```

- Derivar de la clase *java.rmi.server.UnicastRemoteObjec*t. Ésta es una clase de utilidad que redefine algunos de los métodos que todas las clases heredan de la clase *Object* y los implementa acorde a las características que deben tener los objetos remotos. La clase *java.rmi.server.UnicastRemoteObjec*t, además, define un constructor sin parámetros que al ser invocado provoca que el objeto remoto sea *exportad*o, esto es, que el objeto remoto se prepare para escuchar peticiones de los clientes. En nuestro ejemplo, la clase *HelloImpl* extenderá de esta clase.

```
public class HelloImpl extends java.rmi.server.UnicastRemoteObject
                                                    implements Hello
{ ...}
```

- Implementar los métodos de las interfaces remotas. En nuestro caso, la interfaz *Hello* sólo contiene el método *sayHello()* y, por tanto, es el único método que tendremos que implementar.

```
public String sayHello()
{
        return "Hello World!";
}
```

- Proporcionar un constructor. Como cualquier clase, la clase *HelloImpl* deberá proporcionar un constructor que permita crear las instancias de ella que serán, en definitiva, los objetos remotos. El constructor de la clase *HelloImpl* lo único que hace es invocar al constructor sin parámetros de la superclase (la clase *java.rmi.server.UnicastRemoteObjec*t) para que así el objeto remoto que se crea sea *exportad*o. El constructor de la superclase puede lanzar la excepción *java.rmi.RemoteExcepcion* con lo cual nuestro constructor también propagará esta excepción.

```
public HelloImpl() throws java.rmi.RemoteException
{
        super();
}
```

A continuación, se muestra el código completo de la clase *HelloImp*l:

```
package hello;
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject implements Hello{

      public HelloImpl() throws RemoteException{
            super();
      }

      public String sayHello(){
            return "Hello World!";
      }
}
```

Código 8: Código de HelloImpl con RMI.

5.4.2. Aplicación Servidor

La aplicación servidor del ejemplo se encargará de instanciar un único objeto remoto y de registrarlo en el registro RMI. Este objeto remoto implementará una interfaz remota con un único método que al ser invocado por los clientes devolverá el mensaje "Hello World!".

- **Implementación del proceso servidor**

Una vez que se tiene implementada la clase que define los objetos remotos, es el momento de desarrollar el proceso servidor que se encargará de crear los objetos remotos y de hacerlos accesibles a los clientes.

Para crear el proceso servidor se deben realizar los siguientes pasos:

- Crear e instalar el *SecurityManage*r. Todos los programas que usen RMI deben instalar un *SecutityManager* que proporcione seguridad en el acceso a los recursos locales por parte del código descargado de otra maquina. El *SecurityManager* asegura que las operaciones realizadas por el código descargado cumplen una serie de normas de seguridad, no permitiendo que este código acceda a cualquier recurso del sistema y pueda producir daños. En nuestro ejemplo, se ha usado el *RMISecurityManage*r.

```
if (System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
}
```

- Crear los objetos remotos. En nuestro ejemplo, se creará un único objeto remoto. Para ello, se llamará al constructor de la clase *HelloImp*l.

```
Hello objHello = new HelloImpl();
```

Cabe destacar que se ha definido una variable *objHello* de tipo *Hello* para hacer referencia al objeto creado y no de tipo *HelloImpl* como cabría esperar. En realidad se podría haber definido una variable de tipo *HelloImpl* pero no se ha hecho así para recalcar el hecho de que lo que van a ver los clientes del objeto creado va a ser un objeto de tipo *Hello* y no de tipo *HelloImp*l, ya que éstos sólo van a poder invocar los métodos de la interfaz remota *Hello* y no todos los métodos del objeto *HelloImp*l.

- Registrar los objetos remotos en el registro RMI. Java RMI proporciona un servicio de nombres particular, el registro RMI, que permite a los clientes obtener referencias a los objetos remotos del servidor a través de su nombre. En nuestro ejemplo, para que el objeto remoto que hemos creado en el servidor pueda ser localizado por el cliente mediante el registro RMI, previamente deberemos haberlo registrado en él con un determinado nombre. Todas las operaciones que tanto el cliente como el servidor vayan a realizar sobre el registro RMI, las harán a través del API proporcionado por la interfaz *java.rmi.Naming*. En este caso, usaremos el método *rebind(name, objRef)* para registrar el objeto referenciado por *objRef* con el nombre *nam*e. El parámetro *name* será una cadena texto con la estructura ***"//Host:Port/Nombre***", donde:

    - **Host:** maquina donde se está ejecutando el registro RMI.
    - **Port:** puerto en el que escucha el registro RMI.
    - **Nombre:** el nombre con el que queremos registrar nuestro objeto en el registro RMI.

En caso de que se omita el *Host* y el *Por*t, se tomará como máquina por defecto la máquina local y como puerto por defecto el puerto 1099 que es el puerto en el que habitualmente se encuentra escuchando el registro RMI. Además, por razones de seguridad una aplicación solamente podrá registrar objetos en el registro RMI local y no en uno remoto, con lo cual el parámetro *Host* no es de gran utilidad.

A continuación, se muestra el código del ejemplo que registra el objeto remoto creado anteriormente con el nombre "objHello":

```
String name = "//127.0.0.1:1099/Hello";
java.rmi.Naming.rebind(name,objHello);
```

El código del proceso servidor debe de ser ejecutable, por lo que es necesario que tenga un método main. En este ejemplo el proceso servidor es a la vez el objeto remoto por lo que el código se introducirá dentro del método *main* de de la propia clase *HelloImpl*. A continuación se muestra el código completo de clase HelloImpl:

```
package hello;
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject implements Hello
{
	private int cont = 0;

	public HelloImpl() throws RemoteException
	{
		super();
	}

	public String sayHello()
	{
		cont++;
		System.out.println("Cliente numero: " + cont);
		return "Hello World!";
	}

	public static void main(String[] args)
	{
		if (System.getSecurityManager() == null)
			System.setSecurityManager(new RMISecurityManager());
		String name = "//127.0.0.1:1099/Hello";
		try
		{
			Hello objHello = new HelloImpl();
			Naming.rebind(name,objHello);
			System.out.println("Servidor en espera...");
		}
		catch(Exception e)
		{
			System.err.println("Hello exception:"+e.getMessage());
			e.printStackTrace();
		}
	}
}
```

Código 9: Código HelloImpl con excepciones en RMI.

A todo lo anterior se le ha añadido el manejo de las excepciones. Tanto el constructor que crea el objeto *HelloImpl* como el método *rebind* pueden lanzar excepciones y por tanto, se tiene que añadir un bloque *try/catch* que se encargue de capturarlas.

Hacer notar, también, que no es necesario crear un thread que se encargue de mantener vivo el proceso servidor ya que el registro RMI mantiene una referencia al objeto remoto que hemos registrado y mientras existan referencias al objeto del servidor, éste se seguirá ejecutando.

5.4.3 Aplicación Cliente

El proceso cliente del ejemplo se encargará de obtener una referencia al objeto remoto del servidor, a través del registro RMI, y de invocar el método *sayHello()* que contiene este objeto. La invocación de este método le devolverá el mensaje "Hello World!" y lo visualizará por pantalla.

- **Implementación del proceso cliente:**

La aplicación cliente de nuestro ejemplo se verá reducida a un único proceso cliente que se encargará de invocar el método *sayHello()* del objeto remoto del servidor y de visualizar su resultado.

Los pasos a seguir para desarrollar el proceso cliente son los siguientes:

- Crear e instalar el *SecurityManage*r. Al igual que en el servidor, el cliente debe instalar un *SecurityManager* que le garantice que el código descargado del servidor es inofensivo.

```
if (System.getSecurityManager() == null){
        System.setSecurityManager(new RMISecurityManager());
}
```

- Obtener una referencia al objeto remoto del servidor. Para ello se hará uso del registro RMI. Usaremos el método *lookup(name)* de la interfaz *java.rmi.Naming* para obtener una referencia al objeto registrado con el nombre *nam*e. El parámetro *name* será una cadena texto con la estructura ***"//Host:Port/Nombre***", donde:

  - **Host:** maquina donde se está ejecutando el registro RMI.
  - **Port:** puerto en el que escucha el registro RMI.
  - **Nombre:** el nombre con el que fue registrado el objeto en el registro RMI.

En caso de que se omita el *Host* y el *Por*t, se tomará como maquina por defecto la máquina local y como puerto por defecto el puerto 1099. Con el método *looku*p, y al contrario de lo que ocurría con el método *rebin*d, sí podemos buscar objetos remotos en registros RMI que no sean locales a nuestra maquina.

A continuación, se muestra el código de nuestro ejemplo que obtiene una referencia al objeto registrado con el nombre "Hello":

```
String name = "//127.0.0.1:1099/Hello";
Hello objHello = (Hello) java.rmi.Naming.lookup(name);
```

En nuestro caso, tanto el cliente como el servidor estarán ubicados en la misma máquina y por tanto el registro RMI al que accedemos a buscar el objeto será local.

El método *lookup* devolverá una referencia de tipo *java.rmi.Remote* (la interfaz de la que derivan todas las interfaces remotas) y por tanto, será necesario hacer un "casting" de esa referencia al tipo *Hello* que es el tipo de la interfaz remota con la que va a trabajar el cliente.

- Invocar el método del objeto remoto. Una vez obtenida la referencia al objeto, la invocación remota de métodos se hace exactamente igual que si fuese una invocación de métodos normal.

  objHello.sayHello();

A continuación, se muestra el código completo de la aplicación cliente ClienteHello:

```
package cliente;
import java.rmi.*;
import hello.*;

public clas ClienteHello
{
	public static void main(String[] args)
	{
		if (System.getSecurityManager() == null)
		{
			System.setSecurityManager(new RMISecurityManager());
		}
		try
		{
			String name = "//127.0.0.1:1099/Hello";
			Hello objHello = (Hello) java.rmi.Naming.lookup(name);
			System.out.println(objHello.sayHello());
		}
		catch(Exception e)
		{
			System.err.println("Hello exception:"+e.getMessage());
			e.printStackTrace();
		}
	}
}
```

Código 10: Código ClienteHello en RMI.

Como se puede observar el código del cliente se ha introducido en el método *main* de una clase, tal y como se hizo con el servidor. Se ha añadido un bloque *try/catch* para el manejo de las excepciones ya que la invocación de un método remoto puede lanzar la excepción *java.rmi.RemoteExcepcion* (tal y como se definió en la interfaz remota). El método *lookup* también puede lanzar excepciones.

5.4.4 Compilación del ejemplo.

Las clases asociadas al servidor *(Hello* y *HelloImp*l) se encuentran dentro del paquete *hello* y por tanto deben residir dentro de un directorio llamado *hell*o. Se supondrá que este directorio cuelga del raíz (C:\hello).

Las clases asociadas al cliente (únicamente la clase *ClienteHello*) se encuentran dentro del paquete *cliente* y deben residir dentro de un directorio llamado *client*e. Se supondrá de nuevo que este directorio cuelga del raíz (C:\cliente).

Partiendo de los supuestos anteriores, la compilación del ejemplo seguiría los siguientes pasos:

- Configurar el *classpath*. Siempre se deberá colocar el *classpath* un directorio por encima del que se indica en el paquete. En este caso:

  set classpath = C:

- Compilar todos los ficheros .java de los paquetes cliente y servidor.

  javac C:\hello.java*
  javac C:\cliente.java*

- Generar el *Stub* y *Skeleton* del objeto remoto. Para ello se dispone del comando *rmic* que recibe como parámetro la clase que implementa el objeto remoto y genera las clases *Stub* y *Skelet*on. Se deberá ejecutar desde un directorio por encima de donde se quieren dejar los ficheros .class generados.

  rmic hello.HelloImpl

  En este caso se generará la clase *HelloImpl_Stub* y la clase *HelloImpl_Ske*l, ambas dentro del paquete *hell*o.

5.4.5 Ejecución del ejemplo.

Para ejecutar la aplicación se deberá abrir dos sesiones de MS-DOS, una sesión para el servidor y otra para el cliente. Los pasos a seguir son los siguientes:

- Configurar el *classpath*. El *classpath* debe estar configurado en ambas sesiones de MS-DOS. Se configurará de la misma manera que en la compilación .

  set classpath=C:

- Activar el registro RMI. Se podrá activar en cualquiera de las dos sesiones MS-DOS ya que automáticamente iniciará una nueva sesión para él ubicada en la máquina donde se ejecuta. Usaremos el comando *rmiregistry <port>* que pone en marcha el registro en el puerto especificado con el parámetro *<port>*. En caso de que no se indique ningún puerto, el registro se ejecutará por defecto en el puerto 1099. En nuestro caso tomamos esta última opción:

  start rmiregistry

- Ejecutar la aplicación servidora. Para poder ejecutar tanto el cliente como el servidor es necesario usar un fichero *policy* que conceda una serie de permisos. El *RMISecurityManager* que instalamos es muy restrictivo y no permite que se establezcan conexiones con puertos superiores al 1024, como tanto el cliente como el servidor se tienen que conectar al puerto 1099 (donde está el registro RMI) es necesario usar un fichero *policy* que dé permisos para conectarse a este puerto. El fichero *policy* usado se llama *...java.policy* y se encuentra en el directorio del cliente y en el del servidor. En la sesión MS-DOS del servidor se escribirá lo siguiente:

  java –Djava.security.policy=C:\hello\java.policy hello.HelloImpl

- Ejecutar la aplicación cliente. En la sesión MS-DOS del cliente se realizará algo similar a lo que se hizo para ejecutar el servidor:

  java –Djava.security.policy=C:\cliente\java.policy cliente.ClienteHello

  El contenido del fichero .java.policy , es el siguiente:

```
grant
{
        permission java.security.AllPermission;
};
```

Es un fichero de políticas de acceso, en el caso anterior se le están dando todos los permisos, pero se podrían especificar los permisos que se quieren dar según el número de puerto, según el usuario, según las clases a las que se quieren acceder, etc ....

Este ejemplo se ha ejecutado en una misma máquina física, pero en distintas máquinas virtuales, cuando se decide ejecutar en distintas máquinas físicas se siguen los mismos pasos pero el servidor de nombres se deberá ejecutar en la máquina del servidor y desde un directorio distinto de donde residan los ficheros del servidor, de lo contrario el registro tendría carácter local en vez de remoto.

La Distribución Como un Aspecto más.

Implementar una aplicación que hace uso de múltiples computadoras, es decir implementar un sistema distribuido, es complicado y además requiere una alta modularidad. Lo que se persigue es poder implementar un programa de forma local como si fuera a funcionar en una sola máquina y después repartir el uso y ejecutarlo en múltiples máquinas de forma distribuida.

Las propiedades de la distribución de una aplicación es otro requisito no funcional de un sector de aplicaciones en crecimiento. Estaría bien si se pudiera utilizar aspectos para describir cómo la aplicación debe ser distribuida.

La mayoría de las aplicaciones distribuidas siguen un patrón de diseño especifico. Cuando se quiere implementar una aplicación distribuida utilizando RMI, hay una serie de pasos comunes en la creación de todas las aplicaciones que diferencia a una aplicación distribuida de una no distribuida. Como se vio en el apartado anterior las consideraciones a tener en cuenta para crear una aplicación distribuida son:

- **Definir los Interfaces Remotos.** Un interface remoto especifica los métodos que pueden ser llamados remotamente por un cliente. Los clientes programan los interfaces remotos, no la implementación de las clases de dichos interfaces. Parte del diseño de dichos interfaces es la determinación de cualquier objeto local que sea utilizado como parámetro y los valores de retorno de esos métodos; si alguno de esos interfaces o clases no existen aún también tenemos que definirlos. El servidor remoto de objetos debe declarar sus servicios por medio de una interfaz, extendiendo la interfaz de *java.rmi.Remote*. Cada método de la interfaz remota debe lanzar una excepción *java.rmi.RemoteException*.

- **Implementar los Objetos Remotos.** Los objetos remotos deben implementar uno o varios interfaces remotos. La clase del objeto remoto podría incluir implementaciones de otros interfaces (locales o remotos) y otros métodos (que sólo estarán disponibles localmente). Si alguna clase local va a ser utilizada como parámetro o cómo valor de retorno de alguno de esos métodos, también debe ser implementada. El servidor remoto debe implementar la interfaz remota y extender la clase *java.rmi.UnicastRemoteObject*.

- **Implementar los Clientes.** Los clientes que utilizan objetos remotos pueden ser implementados después de haber definido los interfaces remotos, incluso después de que los objetos remotos hayan sido desplegados. *El cliente debe usar la clase java.rmi.Naming para localizar el objeto remoto*. Entonces, el cliente puede invocar los servicios de los objetos remotos entablando comunicación a través del stub.

Se pude observar con mayor precisión el código del siguiente ejemplo sombreado de rojo que se encarga de asegurar que nuestra aplicación es una aplicación distribuida y como dicho código no se encuentra localizado en ningún modulo en concreto, sino todo lo contrario, se encuentra esparcido a lo largo de todo el código de la aplicación. Es un claro ejemplo de aspecto: **el aspecto de distribución**.

```
import java.rmi.*;

public interface HolaMundoRmiI extends Remote
{
    String objRemotoHola(String cliente) throws RemoteException;
}
```

Código 11: implementación de la interfaz remota.

```
import java.rmi.*;
import java.rmi.server.*;
public class HolaMundoRmiO extends UnicastRemoteObject implements
HolaMundoRmiI
{
// Constructor del objeto remoto
        public HolaMundoRmiO() throws RemoteException
        {
                super();
        }
        public String objRemotoHola() throws RemoteException
        {
                return("Hola Mundo" );
        }
}
```

Código 12: implementación del objeto remoto.

```
import java.rmi.*;
import java.rmi.server.*;
public class HolaMundoRmiS
{
        public static void main(String args[])
        {
                try{
// Se instala el controlador de seguridad
                        if (System.getSecurityManager() == null)
                                System.setSecurityManager(new RMISecurityManager());
                        HolaMundoRmiO objRemoto = new HolaMundoRmiO();
                        Naming.rebind("ObjetoHola", objRemoto);
                        System.out.println("Objeto remoto preparado");
                } catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

Código 13: implementación del proceso servidor remoto.

```
import java.rmi.*;
import java.io.*;

public class HolaMundoRmiC
{
	public static void main(String[] args)
	{
// Dirección de la maquina remota, en este caso la maquina local. si se va a ejecutar en
// una maquina diferente, se deberá cambiar a algo semejante a:
// "rmi://www.servidor.com"
		String direccion = "rmi://localhost/";
		try
		{
			HolaMundoRmiI hm =
			(HolaMundoRmiI)Naming.lookup(direccion + "ObjetoHola");
			System.out.println(hm.objRemotoHola("Mundo REMOTO!"));

		} catch(Exception e)
		{
			e.printStackTrace();
		}
		System.exit(0);
	}
}
```

Código 14: implementación del proceso cliente remoto.

Si se elimina de la aplicación remota el código sombreado de rojo obtenemos una aplicación local que realiza las llamadas de forma local, una aplicación donde cliente y servidor serán ejecutados en un único proceso.

```
public interface HolaMundoI
{
	String objHola(String cliente);
}
```

Código 15: implementación de la interfaz local.

En esta versión local la interfaz remota no se utilizará.

```
public class HolaMundoO implements HolaMundoI
{
// Constructor del objeto
        public HolaMundoO()
        {
                super();
        }
        public String objHola(String cliente)
        {
                return("Hola" + cliente);
        }


}
```

Código 16: implementación del objeto local.

```
import java.io.*;

public class HolaMundoS
{
        HolaMundoO hmo;

        public HolaMundoS()
        {
                hmo=new HolaMundoO();
        }
}
```

Código 17: implementación del servidor local.

```
import java.io.*;
public class HolaMundoC
{
        public HolaMundoC(HolaMundoO hm)
        {
                        System.out.println(hm.objHola("Mundo!! "));
        }
}
```

Código 18: implementación del cliente local.

```
import java.io.*;

public class HolaMundo
{
	public static void main(String[] args)
	{
		//Creamos un servidor
		HolaMundoS hms = new HolaMundoS();

		//obtenemos la referencia del objeto
		HolaMundoO hm=hms.hmo;

		//Creamos un cliente
		HolaMundoC hmc =new HolaMundoC(hm);
		System.exit(0);
	}
}
```

Código 19: implementación del proceso principal local.

Pero en realidad lo que se quiere separar y agrupar en un módulo diferenciado es el código que se ha eliminado, la verdadera utilidad del aspecto de distribución no es crear aplicaciones locales a partir de una aplicación remota, lo que se persigue es crear una aplicación distribuida a partir de una aplicación local. Para este propósito haremos uso de un patrón PaDA (Pattern for Distribution Aspects)[12]: el patrón de distribución con aspectos.

6.1. PaDA: Patrón de Distribución con Aspectos

Cuando se implementa un sistema distribuido, éste requiere una alta modularidad, el sistema debe tener una distribución independiente del resto de requisitos. Para lograr una mejor separación de los requisitos debemos usar programación orientada a aspectos. Un aspecto delimita un requisito no funcional, por ejemplo la distribución se teje automáticamente a un sistema cambiando su conducta original. Por eso, se debe ejecutar el sistema con un lenguaje de programación que sea compatible con la programación orientada a aspectos, en nuestro caso con Java.

Así el código mezclado(código con preocupaciones diferentes entrelazados el uno con otro) y código esparcido(código con respecto a una preocupación distribuido en varias unidades del sistema) reducen la modularidad del sistema. Por eso, el mantenimiento y la extensibilidad se ven disminuidas también.

PaDA (Patrón para Distribución con Aspectos) proporciona una estructura para implementar un código distribuido mediante programación orientada a aspectos, logrando así una mejor separación de aspectos. Aumentando la modularidad, extensibilidad y mantenimiento del sistema .

Este patrón es el resultado del trabajo de Sergio Soares y Paulo Borba, dos autores de renombre en el mundo de la programación orientada a aspectos. PaDA fue presentado por primera vez en la segunda conferencia latinoamericana “Pattern Languages Programming” celebrada en Rio de Janeiro en Agosto de 2002. El documento presentado en dicha conferencia tiene el titulo original “PaDA: A Pattern for Distribution Aspects”[12].

6.1.1. Necesidades

Para distribuir un sistema, PaDA tiene en cuenta una serie de necesidades, alguna de ellas propias de los sistemas distribuidos:

- **Comunicación Remota.** La comunicación entre dos componentes de un sistema debe ser remota para permitir que varios clientes accedan al sistema, considerando que la interfaz del usuario es la parte distribuida del sistema.
- **API independiente.** El sistema debe ser completamente independiente de la API de comunicaciones y el middleware, para facilitar el mantenimiento del sistema. No se mezcla el código con las comunicaciones o el código interfaz usuario. También permite cambiar la API de comunicación sin modificar otro código del sistema.
- Un mismo sistema puede emplear middlewares diferentes al mismo tiempo. Por ejemplo, dos clientes acceden el sistema, uno usa RMI y el otro CORBA.
- **Cambio dinámico de middleware.** El sistema debe permitir cambia el middleware sin cambiar o recompilar su código fuente.
- **Facilitar pruebas funcionales.** La prueba funcional facilita la comprobación del sistema con su versión local; por eso, los errores del código de la distribución no afectarán a las pruebas.

Para resolver el problema previamente presentado, PaDA usa programación orientada a aspectos para definir aspectos de la distribución que se pueden tejer al código fuente del sistema central. Esta separación de requisitos se consigue definiendo aspectos que implementan requisitos específicos. Después los aspectos se pueden tejer automáticamente con el código fuente del sistema creándose así una versión del sistema con los requerimientos del aspecto.

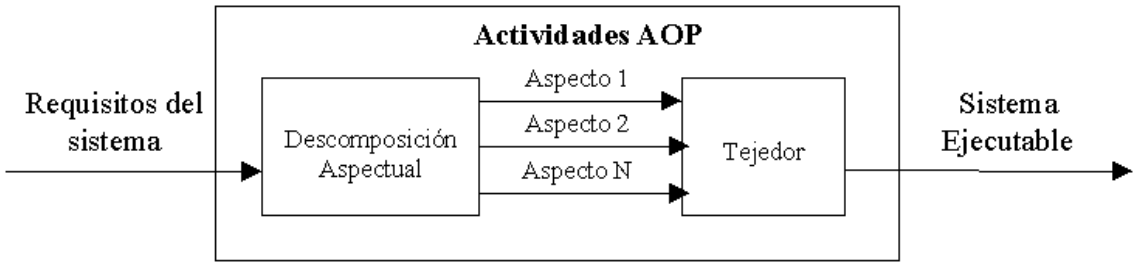


Figura 34: Fases de desarrollo de AOP.

La *Figura34* ilustra la descomposición aspectual que identifica los requisitos no funcionales implementados como aspectos de un sistema y la recomposición aspectual que teje los aspectos con la funcionalidad básica del sistema, obteniendo la versión decisiva con las funciones requeridas. En este caso PaDA define un único requisito que es la distribución. Este requisito será implementado con tres aspectos como se vera a continuación.

6.1.2. Estructura de PaDA.

El patrón PaDA define tres aspectos: uno que atraviesa o corta el módulo servidor, otro que atraviesa o corta las clases del cliente y un tercero que atraviesa o corta a ambos, éste último provee el manejo de excepciones tal y como se muestra en la *Figura35*. El aspecto que atraviesa al módulo servidor también atraviesa o corta otras clases que son tipos de argumentos o tipos de retorno del método sobre el que actúa.

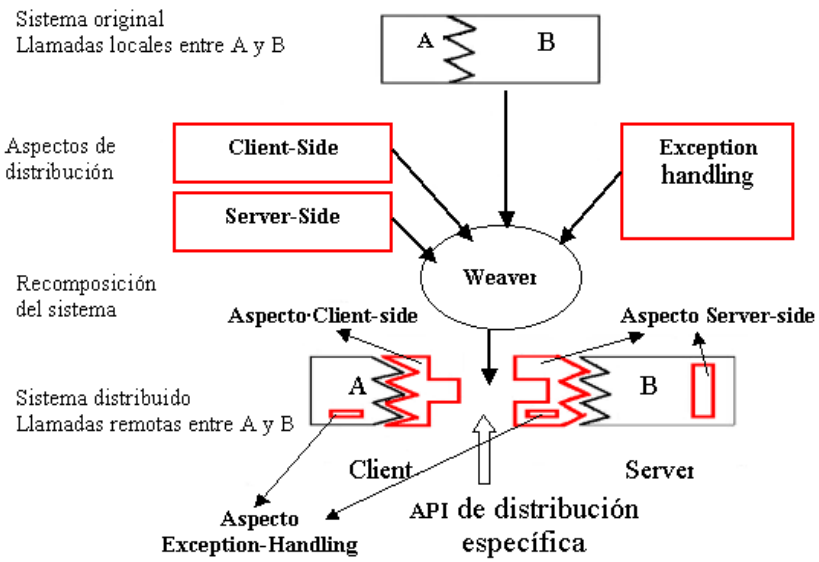


Figura 35: Estructura PaDA

La *Figura 36* representa un diagrama UML que muestra los aspectos y los componentes que permiten el acceso remoto.

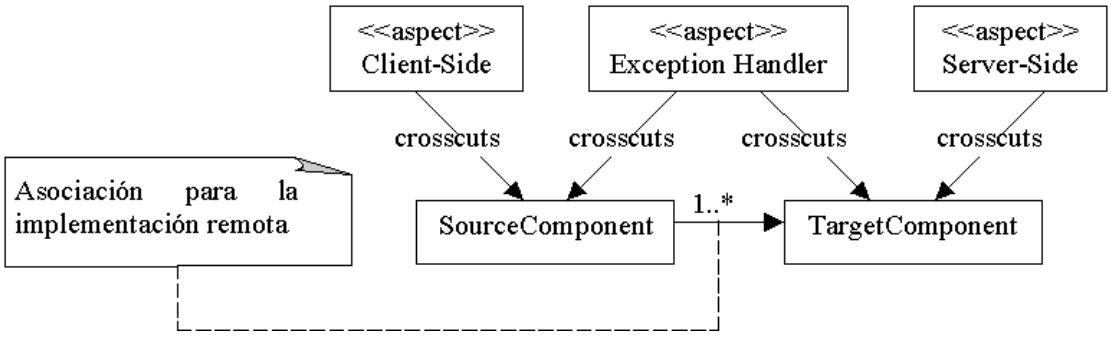


Figura 36: Diagrama UML de las clases PaDA.

6.1.3. Dinámicas de PaDA.

La *Figura 37* muestra en un diagrama de secuencias cuál es el comportamiento original del sistema: una instancia de SourceComponent ejecuta una llamada local a unos métodos de una instancia de TargetComponent.

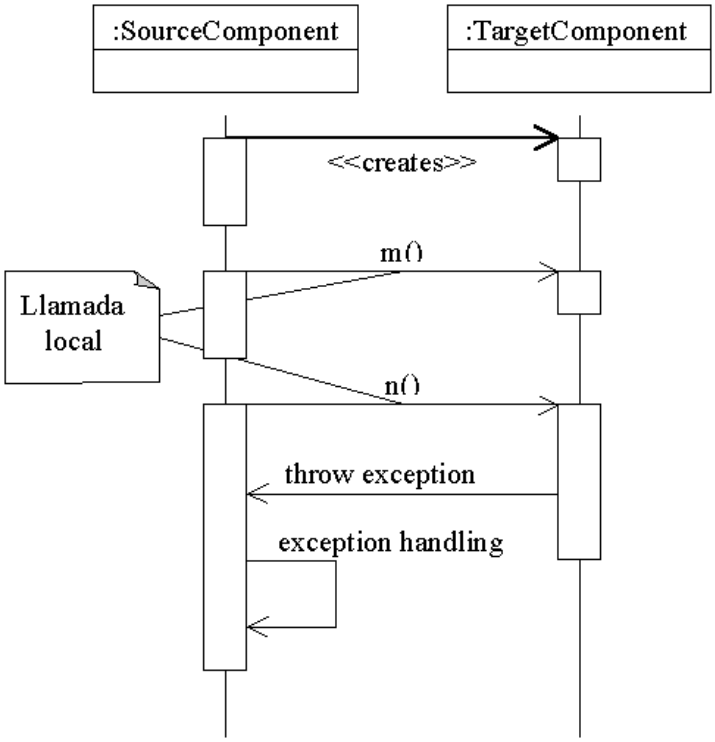


Figura 37: Comportamiento original del sistema

La *Figura 38* muestra el diagrama de secuencias del comportamiento del sistema después de que se le haya tejido los aspectos de distribución. Las llamadas locales de *SourceComponent* son interceptadas por el aspecto *ClientSide* que obtiene la referencia de la instancia remota y redirecciona la llamada local a dicha instancia. El aspecto *ServerSide* crea y ejecuta la instancia remota (una instancia de *TargetComponent*) disponible para las respuestas a llamadas remotas.

Si la llamada remota causa una excepción, como ocurre en la invocación al método n(), el aspecto *ExceptionHandler* trata la excepción como una excepción no chequeada (unchecked) y la lanza en el *ServerSide*. El mensaje que trata y lanza la excepción no chequeada es un mensaje dirigido al aspecto *ExceptionHandler*, porque el aspecto *ExceptionHandler* es responsable de capturar la excepción no chequeada y suministrar el manejo necesario de la excepción en el *ClientSide* (SourceComponent).

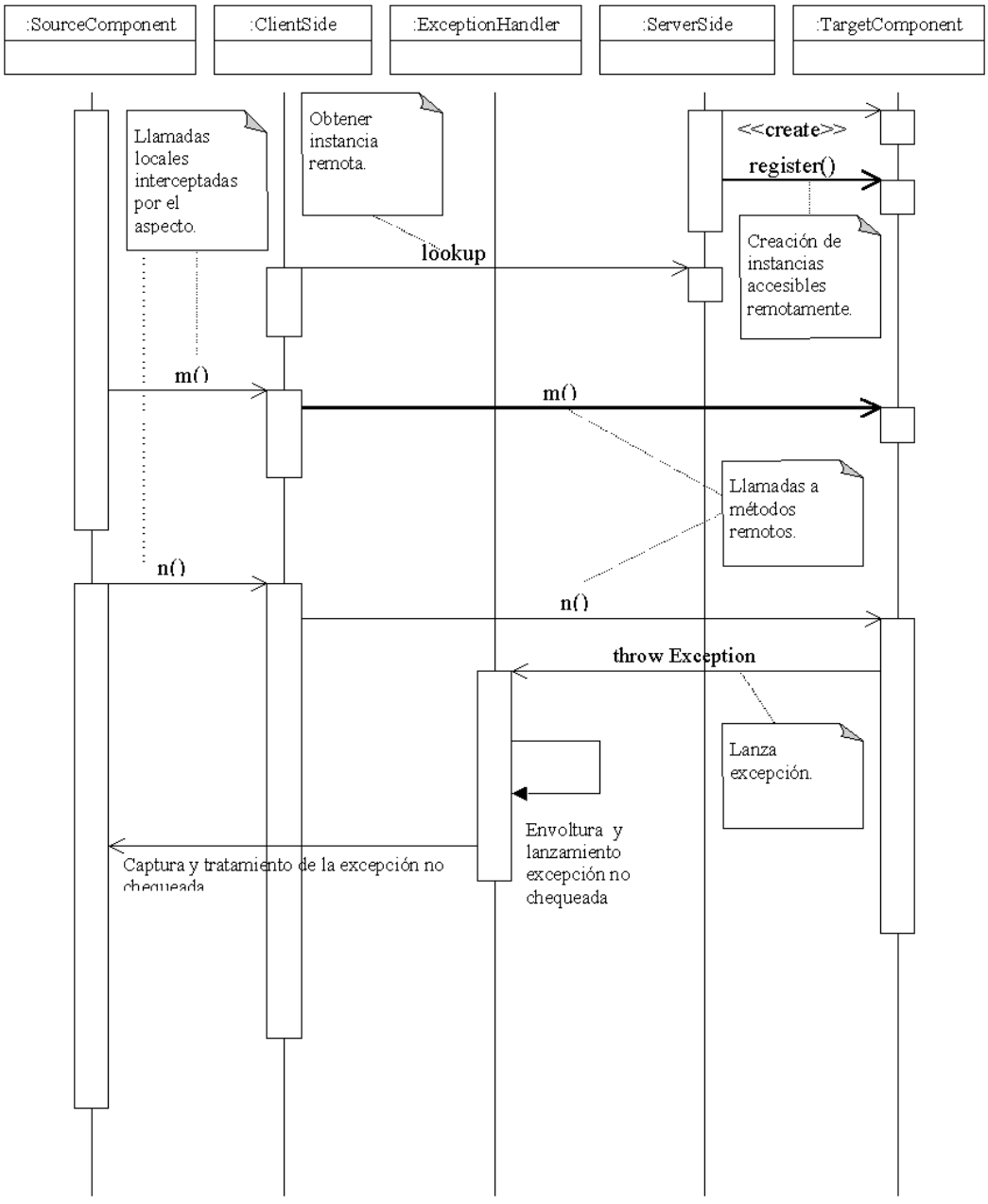


Figura 38: Comportamiento del sistema después de aplicarle PaDA.

6.1.4. Ventajas e Inconvenientes de PaDA.

El patrón PaDA presenta las siguientes ventajas:

- *Implementación Distribuida*. El patrón proporciona comunicación remota entre dos módulos de un sistema; podemos convertir un sistema cualquiera en distribuido simplemente aplicando PaDA.

- *Modularidad*. En la estructura PaDA el código de los aspectos de distribución están completamente separado del código del sistema, ejecutando el código fuente del sistema API independientemente;

- *Mantenimiento y extensibilidad*. Como el código de la distribución esta completamente separado del código del sistema, cambiar la API comunicación es más simple y no causa ningún impacto en el código del sistema. Los programadores sólo deben escribe otros aspectos de la distribución, para el nuevo API especificado, o cambiar los aspectos ya implementados para corregir errores y tejerlos al código fuente del sistema original.

- *Separación adicional de preocupaciones*. La estructura PaDA define el manejo de excepciones como un nuevo aspecto a tratar que no se realiza por técnicas de la programación orientada a objetos. Por eso, se puede cambiar el manejador de la excepción sin modificar el código fuente del sistema original y en los aspectos de la distribución, o en los otros aspectos que implementan los requerimientos del sistema.

- *Facilita comprobación de requisitos funcionales*. Se pueden hacer pruebas de los requisitos funcionales fácilmente si ejecutamos el sistema sin la distribución. La completa separación de aspectos preserva el código fuente del sistema original. Esto significa que se añaden los aspectos de distribución al sistema sólo si se realiza la composición aspectual (weaving).

El patrón PaDA también tiene una serie de inconvenientes :

- *Nuevo paradigma de la programación*. El patrón usa una nueva técnica de la programación que implica saber un nuevo paradigma de la programación para usar el patrón. Otro cambio de tener un nuevo paradigma de la programación es con respecto a la separación del código que usualmente estaba junto en el mismo módulo. El programador de los requisitos funcionales no puede ver el código resultante que implementara el aspecto requerido, decreciendo la legibilidad del código y desconfianza del usuario.

- *Aumento del número de módulos*. PaDA añade tres módulos nuevos en el sistema, aumentando la complejidad del manejo de los módulos.

- *Aumento del bytecode*. Cada aspecto definido dará por resultado una clase después de tejerlo al sistema, que aumentará el bytecode del sistema.

- *Dependencia del nombre*. La definición del aspecto depende de las clases del sistema, métodos, atributos y nombres del argumento, que disminuyen la reutilización de los aspectos. De cualquier modo, se esta investigando en herramientas que puedan crear la definición de los aspectos automáticamente, aumentando la productividad de los aspectos y la reutilización.

- *Permite utilizar un mismo sistema en diferentes middleware*. La idea de AOP es generar versiones de un sistema incluyendo aspectos. La característica de usar un mismo sistema con diferentes middlewares es que se puede conseguir varias versiones del sistema. Sin embargo, esto implica tener varias instancias del sistema (ServerSide) ejecutándose, en vez de uno solo, que afectaría o anularía el control de la concurrencia. En cambio, existe otro patrón DAP que puede solucionar esto fácilmente.

De todas maneras las ventajas superan con creces los pequeños inconvenientes que pueden surgir, por lo que se considera que el uso de PaDA para implementar sistemas distribuidos es realmente ventajoso frente a las técnicas de programación existentes actualmente. Con PaDA se pueden obtener todas las ventajas que proporciona la programación orientada a aspectos y aplicarla fácilmente a la distribución.

6.1.5. Implementación.

La implementación del patrón PaDA esta compuesto de cuatro grandes pasos :

- *Identificar los módulos.* Servidor y Cliente, mantienen la comunicación distribuida entre ellos, debe de estar claro que procesos realizan estas funciones.

- *Escribir el aspecto Server-Side*. El aspecto Server-Side es responsable de cambiar el código del módulo server usando la API especifica de distribución, implementándolo accesible a respuestas de llamadas remotas. Este aspecto tiene que cambiar también los módulos que usan como parámetros o valores de retorno del tipo server, depende de la API de distribución. Server-Side maneja las invocaciones a métodos remotos del lado del servidor. Por ejemplo creando el objeto remoto inicial y registrándolo en el servidor de nombres.

- *Escribir el aspecto del Client-Side*. El aspecto Client-Side es responsable de interceptar las llamadas locales originales efectuadas por el cliente y redireccionarlas como llamadas remotas efectuadas al módulo remoto(servidor). En definitiva maneja las invocaciones a métodos remotos del lado del cliente. Por ejemplo adquiriendo la referencia inicial de objeto remoto.

- *Escribir el aspecto manejador de excepciones*. El aspecto manejador de excepciones es responsable de tratar con una nueva excepción añadida por la definición de los aspectos las excepciones locales que deberían ser remotas. Estas nuevas excepciones envuelven la excepción original dentro de una excepción no chequeada, lanzándola sin cambiar la firma del código fuente del sistema original. Por eso, el aspecto manejador de excepciones debe proporcionar también el manejo necesario en las clases del Client-Side.

Caso de Estudio que Combina Distribución con Aspectos y RMI

Para asimilar todos los conceptos vistos hasta ahora y apreciar mejor las ventajas que puede aportar la programación orientada a aspectos (POA) en el tratamiento de aplicaciones distribuidas, se propone, a lo largo de esta sección un caso de estudio basado en una aplicación inicialmente local a la cual se le quiere imponer la propiedad de distribución hasta obtener una aplicación Cliente/Servidor distribuida.

Se tomará como ejemplo una aplicación bancaria programada en Java. En su inicio la aplicación funciona de forma local proporcionando su servicio en una misma máquina (ver *Figura 39*), pero con el tiempo debido a necesidades propias del sistema donde de ejecuta, se debe suministrar los mismos servicios anteriores pero ahora de forma remota a clientes remotos.

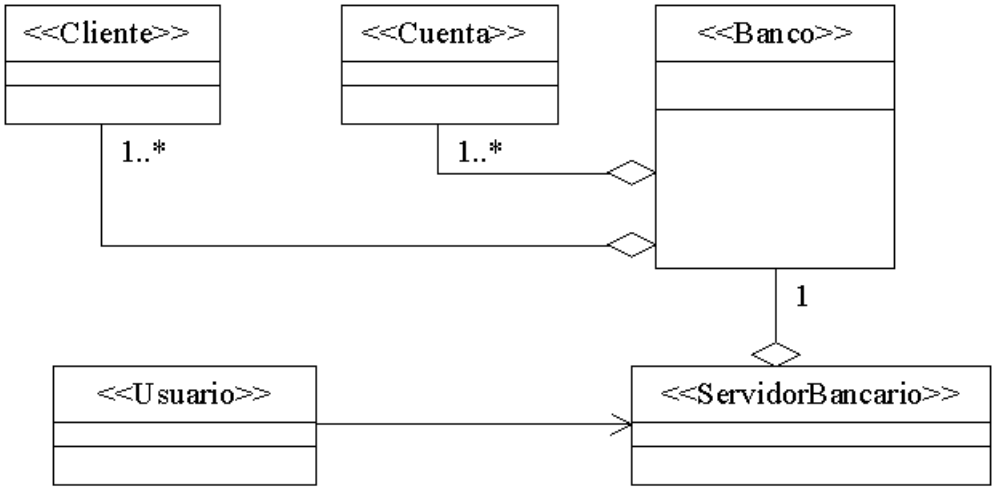


Figura 39: Diagrama de clases UML de la aplicación bancaria local.

A continuación se muestra el código Java correspondiente a una aplicación bancaria local[1]. En principio esta aplicación no tiene restricciones de distribución, la comunicación entre Usuario y ServidorBancario se realiza de forma local y en una misma máquina.

ClienteImpl

```
 1: import java.util.Hashtable;
 2: import java.io.*;
 3:
 4:public class ClienteIm
 5: {
 6:   private Hashtable clientes;
 7:   private Hashtable cuentas;
 8:
 9:   //constructor de ClienteImpl
10:   public ClienteIm()
11:   {
12:         inicialice();
13:   }
14:   //método que devuelve la cuenta con el número numCuenta
15:   public CuentaImpl getCuenta(String numCuenta)
16:   {
17:         CuentaImpl cu=((CuentaImpl)cuentas.get(numCuenta));
18:         return cu;
19:   }
20:   //método que duvuelve el cliente con nombre nomCliente
21:   public ClienteImpl getCliente(String nomCliente)
22:   {
23:         ClienteImpl cl=((□lienteImpl)clientes.get(nomCliente));
24:         return cl;
25:   }
26://método que inicializa a ClienteIm
27:   public void inicialice()
28:   {
29:         cuentas=new Hashtable(10);
30:         clientes=new Hashtable(10);
31:   //crea los clientes que pertenecen al banco
32:         ClienteImpl clienteGilito=new □lienteImpl(this, "Gilito");
33:         ClienteImpl clienteMilloncete=new
                              □lienteImpl(this,"Milloncete");
34:   //los almacena en una Hashtable para una mejor localización
35:         clientes.put("Milloncete", clienteMilloncete);
36:         clientes.put("Gilito", clienteGilito);
37:   //crea las cuentas correspondientes de cada cliente
38:         CuentaImpl cuentaM =new
                              CuentaImpl(this,clienteMilloncete,"6782");
39:         CuentaImpl cuentaG =new
                              CuentaImpl(this,clienteGilito,"2454");
40:   //opera sobre dichas cuentas y las almacena en otra Hashtable
41:         ((CuentaImpl)cuentaG).depositar(600);
42:         cuentas.put("2454",cuentaG);
43:
44:         ((CuentaImpl)cuentaM).depositar(700);
45:         cuentas.put("6782",cuentaM);
46:   }
47: }
```

Código 20: BancoImpl con Java sin restricciones de distribución.

CuentaImpl

```
 1: import java.io.*;
 2: public class CuentaImpl
 3: {
 4:   private BancoImpl banco;
 5:   private ClienteImpl cliente;
 6:   private String numCuenta;
 7:   private long saldo=0;
 8:   //constructor de CuentaImpl
 9:   public CuentaImpl(BancoImpl banco, ClienteImpl cliente,
                                                 String numCuenta)
10:   {
11:         this.banco=banco;
12:         this.cliente=cliente;
13:         this.numCuenta=numCuenta;
14:   }
15:   //método que deposita una "cantidad" de euros en la cuenta
16:   public String depositar (long cantidad)
17:   {
18:         saldo=saldo+cantidad;
19:         System.out.println("Depositado:"+cantidad+"euros en la cuenta
                          "+numCuenta);
20:         return("Depositado "+cantidad+
                          "euros en la cuenta "+numCuenta);
21:   }
22:   //método que devuelve el banco en la que se encuentra la cuenta
23:   public BancoImpl getBanco()
24:   {
25:         return banco;
26:   }
27:   //método que devuelve el número de la cuenta
28:   public String getNum()
29:   {
30:         return numCuenta;
31:   }
32:   //método que devuelve el cliente al que pertenece la cuenta
33:   public ClienteImpl getCliente()
34:   {
35:         return cliente;
36:   }
37:   //método que devuelve el saldo que contiene la cuenta
38:   public long getSaldo()
39:   {
40:         System.out.println("El saldo de la cuenta: "+numCuenta+
                                                 " es de "+saldo);
41:         return saldo;
42:   }
43:   //método que saca una "cantidad" de euros de la cuenta
44:   public long sacarDinero(long cantidad) throws NoHayDineroException
45:   {
46:         if (cantidad>saldo){
47:               throw new NoHayDineroException();
48:         }
49:         saldo=saldo-cantidad;
50:         System.out.println("Se ha sacado "+cantidad+
                          "euros de la cuenta "+numCuenta);
51:         return cantidad;
52:   }
53: }
```

Código 21: CuentaImpl con Java sin restricciones de distribución.

ClienteImpl

```
1: public class ClienteImpl
 2:{
 3:   private BancoImpl banco;
 4:   private String nombre;
 5:
 6:   //constructor de ClienteImpl
 7:   public ClienteImpl(BancoImpl banco,String nombre)
 8:   {
 9:         this.banco=banco;
10:         this.nombre=nombre;
11:   }
12:   //método que devuelve el banco donde se encuentra el cliente
13:   public BancoImpl getBanco()
14:   {
15:         return banco;
16:   }
17:   //método que devuelve el nombre del cliente
18:   public String getNombre()
19:   {
20:         return nombre;
22:   }
21: }
```

Código 22: ClienteImpl con Java sin restricciones de distribución.

NoHayDineroException

```
 1: import java.io.*;
 2:
 3: public class NoHayDineroException extends Exception
 4: {
 5://constructor de NoHayDineroException
 6:   public NoHayDineroException()
 7:   {
 8:         System.out.println("No hay dinero suficiente .");
 9:   }
10: }
```

Código 23: NoHayDineroException con Java sin restricciones de distribución.

Usuario

```
 1: public class Usuario
 2: {
 3:   BancoImpl banco;
 4:   String numCuenta;
 5:
 6:   //constructor de la clase Usuario
 7:   public Usuario(BancoImpl banco,String numCuenta)
 8:   {
 9:         System.out.println("-------------------------------------");
10:         System.out.println("    Creado nuevo Usuario:");
11:         System.out.println("-------------------------------------");
12:         this.banco=banco;
13:         this.numCuenta=numCuenta;
14:         inicialize();
15:   }
16:   //método que inicializa al Usuario
17:   public void inicialize()
18:   {
19:         CuentaImpl cuentaMia;
20:         String depositando;
21:         try
22:         {
23:   //obtenemos la cuenta con numero "numCuenta" a través del banco
24:         cuentaMia=banco.getCuenta(numCuenta);
25:   //realizamo las operaciones sobre dicha cuenta
26:               depositando=cuentaMia.depositar(25);
27:               System.out.println(depositando);
28:               String c=cuentaMia.getNum();
29:               long saldo=cuentaMia.getSaldo();
30:               System.out.println("El saldo de la cuenta: "+c+
                                          " es de "+saldo);
31:               saldo=cuentaMia.sacarDinero(10);
32:               System.out.println("Se ha sacado "+saldo+
                                          "euros de la cuenta "+c);
33:               saldo=cuentaMia.getSaldo();
34:               System.out.println("Se ha sacado "+saldo+
                                          "euros de la cuenta "+c);
35:         }
36:         catch(NoHayDineroException nhde)
37:         {
38:               System.out.println("no hay dinero para sacar");
39:         }
40:         catch(Exception ex)
41:         {
42:               System.out.println("exception");
43:         }
44:   }
45: }
```

Código 24: Usuario con Java sin restricciones de distribución.

ServidorBancario

```
 1: import java.io.*;
 2:
 3: public class ServidorBancario
 4: {
 5:   BancoImpl banco;
 6:
 7:   //Constructor de ServidorBancario
 8:   public ServidorBancario()
 9:   {
10:
11:         System.out.println("-------------------------------------");
12:         System.out.println("    Creado nuevo banco:");
13:         System.out.println("-------------------------------------");
14:
15:         banco=new BancoImpl();
16:   }
17:   //método que devuele el objeto banco
18:   public BancoImpl getBanco()
19:   {
20:         return banco;
21:   }
22: }
```

Codigo 25: ServidorBancario con Java sin restricciones de distribución.

Principal

```
 1: public class Principal
 2:{
 3:   //método main para poder ser ejecutable
 4:   public static void main(String args[])
 5:   {
 6:         BancoImpl banco;
 7:         ServidorBancario server=new ServidorBancario();
 8:         banco=server.banco;
 9:         Usuario cliente1=new Usuario(banco,"6782");
10:         Usuario cliente2=new Usuario(banco,"2454");
11:
12:   }
13: }
```

Código 26: Principal con Java sin restricciones de distribución.

El objetivo final que se persigue es distribuir la aplicación original (local) que no posee restricciones de distribución alguna y que la comunicación entre las clases Usuario y ServidorBancario sea distribuida. De manera que un Usuario se pueda ejecutar en una máquina y el ServidorBancario se pueda ejecutar en otra máquina distinta, simulando así una aplicación bancaria real.

Teniendo claro qué es lo que se quiere hacer y cómo debe de comportarse la aplicación final, podemos representar un diagrama de clases UML [3] del comportamiento de la aplicación distribuida que se quiere conseguir finalmente (ver *Figura 40*). Para esto las partes básicas del sistema deben ser identificadas, se tiene que tener claro cual es el papel de cada objeto y como interactúa con el resto y como se realizaran sus comunicaciones. En el diagrama UML debe quedar plasmado el mecanismo de llamadas utilizado en cada invocación a objetos, que antes se habrá analizado en profundidad.

Ante esta situación se pueden adoptar diversas soluciones. Por un lado si tenemos en cuenta que se está trabajando en un entorno de programación Java, la primera solución en adoptar será la propuesta de Java para implementar aplicaciones distribuidas, es decir aplicar Java RMI.

Por otro lado, según lo visto en apartados anteriores, otra propuesta que se podrá utilizar será aplicar un patrón de distribución con aspectos: aplicar PaDA para distribuir la aplicación bancaria. Así que la herramienta de trabajo deberá ser AspectJ ya que la aplicación original viene dada en un lenguaje base Java.

Tanto para una versión como para la otra, las fases de desarrollo del software son idénticas: ambas versiones deben conseguir un mismo resultado y ambas parten de la misma aplicación original. Se pueden dividir estas fases en las siguientes:

- Análisis: Se debe dejar bien claro cuál es la lógica de negocio con la que se va a trabajar de acuerdo con el dominio del problema. Detectar cuáles son los objetos principales y cuáles son los secundarios y de qué modo se comunican entre ellos, una manera fácil de lograrlo es simular una ejecución y anotar las invocaciones de los objetos. No se realizará un modelo perfecto sino un punto de partida. Con esta información se creará un diagrama UML del comportamiento de la aplicación (ver *Figura 39*), no debe de ser muy detallado en cuanto métodos y atributos, lo interesante es que quede claro la comunicación que existe entre ellos y los servicios básicos que deben ofrecer.

- Análisis de la Reutilización: Se debe comprobar si hay algún objeto de la aplicación original que se pueda reutilizar o si hay alguna porción de código reutilizable. Se debe intentar reutilizar al máximo la aplicación original. Un buen análisis de reutilización supone un ahorro considerable tanto de tiempo como de coste.

- Diseño: En el diseño cuenta mucho la tecnología que se va a usar. En esta fase el modelo idealizado del análisis empieza a tomar forma. Se tendrá que realizar un diseño a nivel de sistema, donde el sistema final resultante en ambas versiones tiene que cumplir unos requisitos, su comportamiento final debe coincidir con el comportamiento mostrado en la *Figura 40*. Y profundizando, se deberá realizar un comportamiento a nivel de objeto, diseño completamente distintos según las versiones. Como es lógico se deberá realizar un diseño a nivel particular de objetos consecuente con el diseño de sistema que se quiere conseguir.

- Iteración: Dependiendo de la tecnología que se use la iteración será de una manera o de otra, pero siempre dando como resultado la aplicación que se muestra en el diagrama UML de la *Figura 40.*

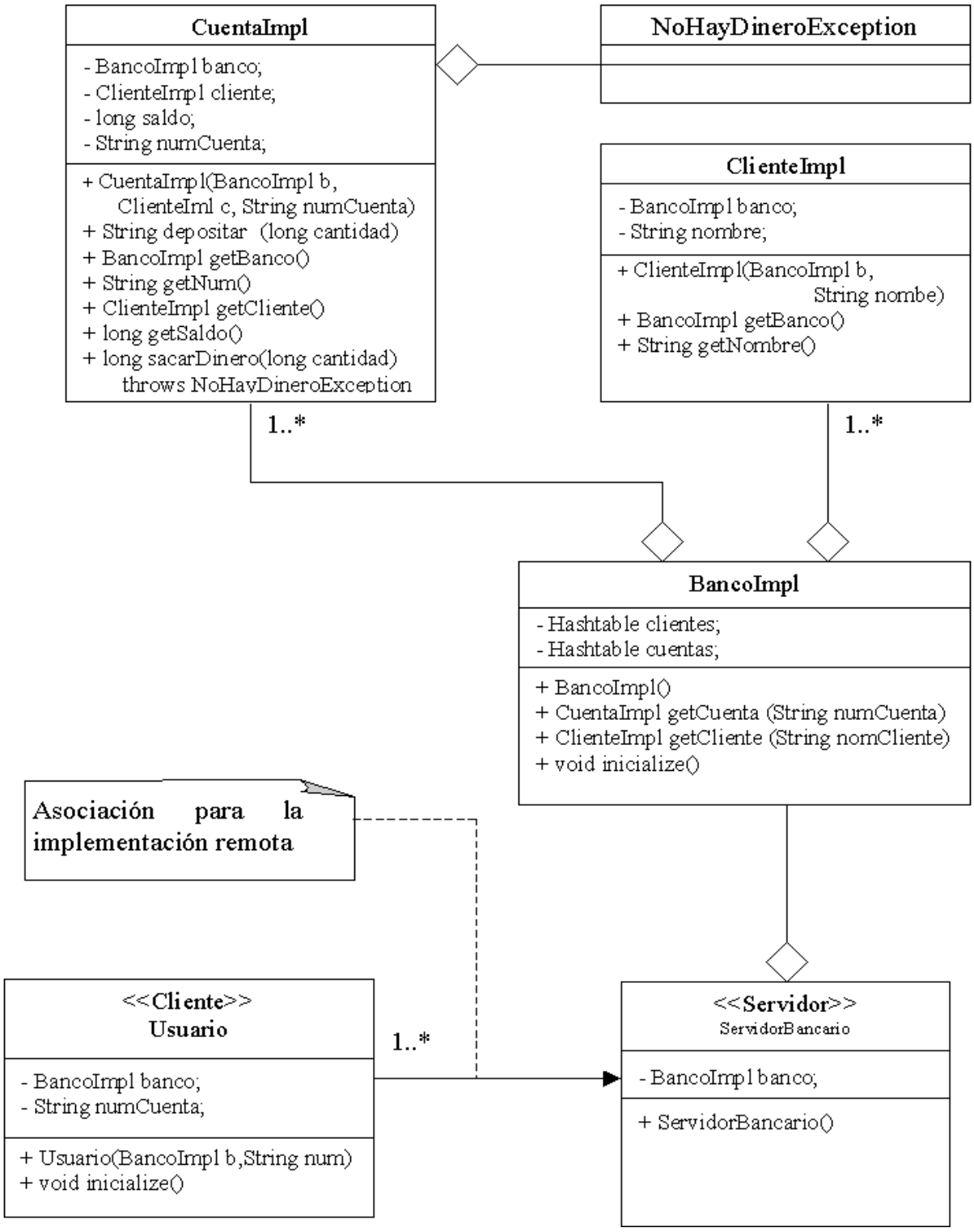


Figura 40: Diagrama de clases UML de la aplicación bancaria final.

7.1 Versión de Distribución con RMI sin POA.

Para implementar distribución al sistema sin utilizar programación orientada a aspectos(POA) se utilizará RMI que es la propuesta de Java para aplicar distribución. A continuación se describe el proceso de desarrollo del análisis que cumple los requisitos en el dominio del problema.

- Análisis:

  Para el desarrollo de la aplicación bancaria distribuida mediante RMI la lógica de negocio será la propia aplicación original. Según la propuesta de distribución, la comunicación entre Usuario y ServidorBancario ha de ser completamente remota. Usuario y ServidorBancario serán los objetos ejecutables del sistema y por ello los principales. Si se tiene en mente una aplicación Cliente/Servidor, Usuario deberá hacer de Cliente del sistema y ServidorBancario de Servidor. Siguiendo con este orden aparece un objeto secundario, BancoImpl, este objeto se encontrará registrado en el servidor de nombres para estar accesible al Usuario. Los otros objetos, CuentaImpl y ClienteImpl, se encuentran dentro de BancoImpl por lo que accediendo a BancoImpl se accede a estos objetos (ver *Figura 41*).

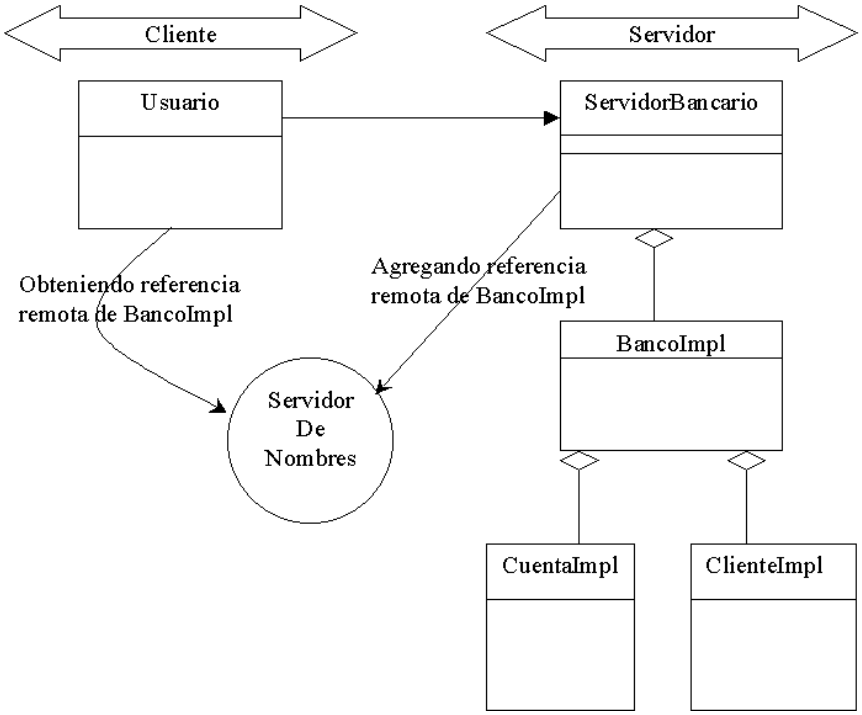


Figura 41: Diagrama resultado del Análisis.

- Análisis de la Reutilización:

  Como se dijo en el análisis anterior, la aplicación bancaria original es en sí la lógica de negocio de la nueva aplicación distribuida ya que la funcionalidad básica no se ve afectada por la distribución. Por lo que se tendrá que utilizar la aplicación original como punto de partida de la nueva aplicación distribuida. No podemos hablar de reutilización porque es una modificación sobre un código existente. En verdad aunque se está hablando de una aplicación original y una nueva aplicación final distribuida, solamente existe una aplicación que modificaremos obteniendo como resultado de estas modificaciones una aplicación distribuida en lugar de una aplicación local, es un proceso de metamorfosis, antes de este proceso la aplicación era local y después distribuido.

- Diseño:

  La tecnología que se usará para esta versión es RMI exclusivamente. Se deberá separar la parte del cliente con la parte del servidor, convertir los objetos en remotos teniendo en cuenta que BancoImpl se exportara automáticamente mientras que CuentaImpl y ClienteImpl se exportará implícitamente. Los objetos remotos deben implementar una interfaz remota, que se ha de crear, y todos los métodos pertenecientes a dicha interfaz deben poder lanzar la excepción RemoteException.

  Se tendrán que seguir los pasos de diseño para una aplicación distribuida con RMI vistos en el apartado 5.

- Iteración:

  A continuación se puede observar como el código necesario para la distribución de la aplicación afecta casi a la totalidad de las clases y queda dispersado entre ellas, para una mejor visualización encerraremos este código responsable de implementar distribución con RMI dentro de unos rectángulos coloreados.

  Una de las peculiaridades de RMI es que trabaja mediante interfaces remotas. Toda la comunicación entre los objetos se hace a través de estas interfaces, no se puede realizar de otra manera la distribución RMI si no es mediante interfaces. De esta manera cuando antes se definía una referencia a un objeto (ejem: ***BancoImpl*** banco=new BancoImpl(...)), ahora si la referencia se quiere hacer de un objeto remoto se hará a su interfaz remota (ejem: ***I_Banco*** banco=new BancoImpl(...)).

I_Banco

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Banco extends Remote
5: {
6:   public I_Cuenta getCuenta(String nomCliente)throws RemoteException;
7:
8:   public I_Cliente getCliente(String numCuenta)
                                                  throws RemoteException;
9: }
```

Código 27: Código de la interfaz I_Banco.

I_Cliente

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Cliente extends Remote
5: {
6:  public I_Banco getBanco()throws RemoteException;
7:
8:  public String getNombre()throws RemoteException;
9: }
```

Códigо28: Código de la interfaz I_Cliente.

I_Cuenta

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Cuenta extends Remote
5: {
6:   public I_Banco getBanco() throws RemoteException;
7:
8:   public I_Cliente getCliente() throws RemoteException;
9:
10:  public String depositar(long cantidad) throws RemoteException;
11:
12:  public long getSaldo() throws RemoteException;
13:
14:  public String getNum()throws RemoteException;
15:
16:  public long sacarDinero (long cantidad)
                        throws NoHayDineroException, RemoteException;
17: }
```

Código 29: Código de la Interfaz I_Cuenta.

Lo primero que se debe añadir al código son las interfaces de los objetos que se comportarán de forma remota. Para éste caso se debe añadir una interfaz remota para BancoImpl, ClienteImpl y CuentaImpl, las interfaces serán respectivamente I_Banco, I_Cliente e I_Cuenta, los métodos declarados en estas interfaces deben poder lanzar la excepción RemoteException a parte de las excepciones que lanzaban los métodos anteriormente.

NoHayDineroException

```
 1: import java.io.*;
 2:
 3: public class NoHayDineroException extends Exception
 4: {
 5://constructor de NoHayDineroException
 6:   public NoHayDineroException()
 7:   {
 8:         System.out.println("No hay dinero suficiente .");
 9:   }
10: }
```

Código 30: Código de la excepción NoHayDineroException con Java.

BancoImpl

```
 1: import java.util.Hashtable;

 2: import java.rmi.server.UnicastRemoteObject;
 3: import java.rmi.RemoteException;

 4: import java.io.*;
 5:
 6: public class BancoImpl extends UnicastRemoteObject implements I_Banco
 7: {
 8:   private Hashtable clientes;
 9:   private Hashtable cuentas;
10:
11:   //constructor de BancoImpl

12:   public BancoImpl() throws RemoteException
13:   {
14:         inicialice();
15:   }
16:   //método que devuelve la cuenta con número "numCuenta"
17:   public I_Cuenta  getCuenta(String numCuenta)

                                        throws RemoteException

18:   {

19:         I_Cuenta  cu=((CuentaImpl)cuentas.get(numCuenta));

20:         return cu;
21:   }
```

```
22: //método qeu duvuelve el cliente con nombre nomCliente

23:   public I_Cliente  getCliente(String nomCliente)

                                          throws RemoteException
24:   {
25:         I_Cliente  cl=((ClienteImpl)clientes.get(nomCliente));

26:         return cl;
27:   }
28: ://método que inicializa a BancoImpl

29:   public void inicialice() throws RemoteException
30:   {
31:         cuentas=new Hashtable(10);
32:         clientes=new Hashtable(10);
33: ://método que inicializa a BancoImpl

34:         I_Cliente clienteGilito=new ClienteImpl(this, "Gilito");

35:   //exportación explicita del objeto ClienteImpl clienteGilito

36:         UnicastRemoteObject.exportObject(clienteGilito);

37:         clientes.put("Gilito", clienteGilito);


38:         I_Cliente clienteMilloncete=new ClienteImpl(this,
                                              "Milloncete");

39:   //exportación explicita del objeto ClienteImpl clienteMilloncete

40:         UnicastRemoteObject.exportObject(clienteMilloncete);

41:         clientes.put("Milloncete", clienteMilloncete);

42:         I_Cuenta  cuentaG =new CuentaImpl(this,clienteGilito,"2454");

43:         ((CuentaImpl)cuentaG).depositar(600);
44:   //exportación explicita del objeto CuentaImpl cuentaG

45:         UnicastRemoteObject.exportObject(cuentaG);

46:         cuentas.put("2454",cuentaG);

47:         I_Cuenta  cuentaM =new CuentaImpl(this,
                                    clienteMilloncete,"6782");

48:         ((CuentaImpl)cuentaM).depositar(700);
49: //exportación explicita del objeto CuentaImpl cuentaM

50:         UnicastRemoteObject.exportObject(cuentaM);

51:         cuentas.put("6782",cuentaM);
52:   }
53: }
```

Código31: BancoImpl con Java RMI con restricciones de distribución.

BancoImpl es el objeto remoto por excelencia de nuestra aplicación. Este objeto deberá exportarse y registrarse en el servidor de nombres para estar accesible para el resto de Usuarios que lo requieran.

La exportación de este objeto remoto se hará de forma automática, por esto este objeto debe extender de UnicastRemoteObject

Por definición un objeto remoto debe implementar una interfaz remota, BancoImpl implementara la interfaz remota I_Banco que fue crea.

```
 6: public class BancoImpl extends UnicastRemoteObject implements I_Banco
```

BancoImpl guarda una hashtable para los clientes y otra para las cuentas de dichos clientes, es dentro de BancoImpl donde se crean los clientes y las cuentas y es BancoImpl quien proporciona a los Usuarios las referencias a ellos.

```
 8:   private Hashtable clientes;
 9:   private Hashtable cuentas;

31:         cuentas=new Hashtable(10);
32:         clientes=new Hashtable(10);
```

Dentro de estas hashtables se encontrarán las cuentas y los clientes que el BancoImpl creará.

```
34:         I_Cliente clienteGilito=new ClienteImpl(this, "Gilito");

42:         I_Cuenta  cuentaG =new CuentaImpl(this,clienteGilito,"2454");
```

Los objetos CuentaImpl y ClienteImpl son objetos remotos y como tal deben de ser exportados, pero estos de forma implícita ya que no heredan de UnicastRemoteObjectc.

```
36:         UnicastRemoteObject.exportObject(clienteGilito);

40:         UnicastRemoteObject.exportObject(clienteMilloncete);
```

CuentaImpl

```
 1: import java.rmi.RemoteException;
 2: import java.io.*;
 3:
 4: public class CuentaImpl  implements I_Cuenta
 5: {
 6:   private I_Banco banco;
 7:   private I_Cliente  cliente;
 8:   private long saldo;
 9:   private String numCuenta;
10:
11: //constructor de CuentaImpl
12:   public CuentaImpl(I_Banco banco, I_Cliente cliente,
                                                 String numCuenta)
```

```
13:    {
14:          this.banco=banco;
15:          this.cliente=cliente;
16:          this.numCuenta=numCuenta;
17:          this.saldo=0;
18:    }
19:    //método que deposita una "cantidad" de euros en la cuenta

20:    public String depositar (long cantidad) throws RemoteException
21:    {
22:          saldo=saldo+cantidad;
23:          System.out.println("Depositado:"+cantidad+
                                    "euros en la cuenta "+numCuenta);
24:          return("Depositado "+cantidad+"euros en la cuenta "
                                                          +numCuenta);
25:    }
26: //método que devuelve el banco en la que se encuentra la cuenta

27:    public I_Banco  getBanco()throws RemoteException
28:    {
29:          return banco;
30:    }
31:    //método que devuelve el número de la cuenta

32:    public String getNum() throws RemoteException
33:    {
34:          return numCuenta;
35:    }
36:    //método que devuelve el cliente al que pertenece la cuenta

37:    public I_Cliente getCliente()  throws RemoteException
38:    {
39:          return cliente;
40:    }
41:    //método que devuelve el saldo que contiene la cuenta

42:    public long getSaldo() throws RemoteException
43:    {
44:          System.out.println("El saldo de la cuenta: "+numCuenta+
                                                    " es de "+saldo);
45:          return saldo;
46:    }
47:    //método que saca una "cantidad" de euros de la cuenta
48:    public long sacarDinero(long cantidad) throws NoHayDineroException,

                                          RemoteException
49:    {
50:          if (cantidad>saldo)
51:          {
52:                throw new NoHayDineroException();
53:          }
54:          saldo=saldo-cantidad;
55:    System.out.println("Se ha sacado "+cantidad+"euros de la cuenta
                                "+numCuenta);
56:          return cantidad;
57:    }
58: }
```

Código 32: CuentaImpl con Java RMI con restricciones de distribución.

CuentaImpl es otro objeto remoto y por definición todos los objetos remotos deben implementar una interfaz remota, la interfaz remota creada para este objeto es I_Cuenta.

A diferencia del objeto remoto BancoImpl, CuentaImpl no debe extender de UnicastRemoteObjectc, por lo que deberá ser exportado implícitamente por BancoImpl.

Tampoco es necesario registrar este objeto en el servidor de nombres, Usuario obtendrá su referencia a través de BancoImpl.

```
4: public class CuentaImpl implements I_Cuenta
```

Este objeto permite al Usuario depositar una cantidad de euros en la cuenta, permite consultar el saldo de la cuenta, permite sacar una cantidad de euros de la cuenta y también permite saber a que cliente pertenece la cuenta y en que banco se encuentra la cuenta.

ClienteImpl

```
 1: import java.rmi.RemoteException;

 2: public class ClienteImpl implements I_Cliente
 3: {

 4:    private I_Banco  banco;

 5:    private String nombre;
 6: //constructor de ClienteImpl

 7:    public ClienteImpl( I_Banco  banco,String nombre)
 8:    {
 9:          this.banco=banco;
10:          this.nombre=nombre;
11:    }
12:    //método que devuelve el banco donde se encuentra el cliente

13:    public I_Banco  getBanco() throws RemoteException
14:    {
15:          return banco;
16:    }
17:    //método que devuelve el nombre del cliente

18:    public String getNombre() throws RemoteException
19:    {
20:          return nombre;
21:    }
22: }
```

Código 33: ClienteImpl con Java RMI con restricciones de distribución.

ClienteImpl es otro objeto remoto y por definición todos los objetos remotos deben implementar una interfaz remota, la interfaz remota creada para este objeto es I_Cliente.

Al igual que CuentaImpl, ClienteImpl no debe extender de UnicastRemoteObject, por lo que deberá ser exportado implícitamente por BancoImpl.

Tampoco es necesario registrar este objeto en el servidor de nombres, Usuario obtendrá su referencia a través de BancoImpl.

Este objeto permite al Usuario saber el banco al que pertenece el cliente y saber el nombre de este cliente.

ServidorBancario

```
 1: import java.rmi.Naming;
 2: import java.rmi.RemoteException;
 3: import java.rmi.server.UnicastRemoteObject;

 4: import java.io.*;
 5:
 6: public class ServidorBancario
 7: {
 8:   I_Banco banco;
 9:
10:   //Constructor ServidorBancario
11:   public ServidorBancario()
12:   {
13:         System.out.println("--------------------------------------");
14:         System.out.println("    Creado nuevo banco:");
15:         System.out.println("--------------------------------------");
16:
 17:        try
18:        {
19:              banco=new BancoImpl();
20:        }

21:        catch (RemoteException remoteException)
22:        {

23:              System.err.println("Fallo durante la exportación : " +
                                        remoteException);
25:        }
26:
27:        try
28:        {
29:              Naming.rebind("Banco",banco);
30:              System.out.println(Naming.lookup("//localhost/Banco"));
31:        }
32:

33:        catch (RemoteException remoteException)
34:        {

35:              System.err.println("Fallo del registro del nombre: "+
                                        remoteException);
36:        }
```

```
42:           catch (java.rmi.NotBoundException notbound) {}

44:    }
45:
46:    public static void main(String args[])
47:    {
48:           new ServidorBancario();
49:    }

50: }
```

Código 34: ServidorBancario con Java RMI con restricciones de distribución.

El ServidorBancario es el servidor de la aplicación, es el encargado poner en funcionamiento la aplicación, el ServidorBancario crea el BancoImpl, y es el encargado de registrar el BancoImpl que ha creado en un servidor de nombres. Es este objeto el que comparte la comunicación remota con el Usuario.

Y por ultimo el Usuario. El Usuario recoge una referencia remota del BancoImpl de nombres, pero para ello debe saber donde se encuentra, el servidor de nombres debe estar en la misma máquina que el servidor de manera que debemos pasarle al Usuario la dirección IP de la máquina donde se encuentra el ServidorBancario.

Una vez que el servidor ya dispone de la referencia al BancoImpl puede realizar su actividad, realizando peticiones a BancoImpl y a los objetos que contiene.

Usuario se ejecutara en una máquina diferente a la de ServidorBancario y por ello ha de poderse ejecutar independientemente de que el ServidorBancario se haya ejecutado con anterioridad, para esto se ha tenido que añadir al código original un método main , donde se recoge la dirección IP a través de la entrada estándar.

Usuario

```
 1: import java.rmi.*;
 2:
 3: public class Usuario
 4: {
 5:   I_Banco banco;
 6:
 7: //constructor de la clase Usuario

 9:    public Usuario( String host)
10:   {
11:         try
12:         {
13:               System.setSecurityManager(new RMISecurityManager());
14:               banco = (I_Banco)Naming.lookup("rmi://"
                                                  +host+"/Banco");
15:         }
16:         catch (RemoteException remoteException)
17:         {
18:               System.err.println("Excepcion remota: " +
                                                  remoteException);
19:         }
```

```
20:         catch(Exception e)
21:         {
22:               System.err.println("Exception");
23:         }
24:         inicialize();
25:   }

26:   //método que inicializa al Usuario
27:   public void inicialize()
28:   {
29:         I_Cuenta cuentaMia;

30:         String depositando;
31:
32:         try
33:         {
34:               cuentaMia=banco.getCuenta("6782");
35:               depositando=cuentaMia.depositar(25);
36:               System.out.println(depositando);
37:               String c=cuentaMia.getNum();
38:               long saldo=cuentaMia.getSaldo();
39:               System.out.println("El saldo de la cuenta: "+c+
                                       " es de "+saldo);
40:               saldo=cuentaMia.sacarDinero(10);
41:               System.out.println("Se ha sacado "+saldo+
                                       "euros de la cuenta "+c);
42:               saldo=cuentaMia.getSaldo();
43:               System.out.println("El saldo de la cuenta: "+c+
                                       " es de "+saldo);
44:         }
45:         catch(NoHayDineroException nhde)
46:         {
47:               System.out.println("no hay dinero para sacar");
48:         }
49:         catch(Exception ex)
50:         {
51:               System.out.println("exception");
52:         }
53:   }
54:
55:   public static void main(String[] args)
56:   {
57:         new Usuario(args[0]);
58:   }
59: }
```

Código 35: Usuario con Java RMI con restricciones de distribución.

Ya se tiene la aplicación distribuida final, ahora hay que compilar la aplicación y ejecutarla tal como se vio en los apartados cinco punto tres punto dos, cinco punto tres punto tres y en cinco punto tres punto cuatro.

Con esta opción hemos conseguido el propósito fijado, se ha conseguido una aplicación bancaria con las mismas características que la original pero a demás distribuida.

Como vimos en el apartado cinco, RMI es una tecnología orientada a objetos que consigue distribuir de una manera lógica los diferentes objetos de la aplicación, RMI se convierte así en una alternativa muy viable para realizar la distribución de la aplicación, proporcionando una serie de particularidades propias de RMI destacables:

- No se tiene que tener en cuenta el protocolo sobre el que se trabaja, ya que RMI permite abstraer las interfaces de comunicación a llamadas locales.

- Los objetos distribuidos se ejecutan en JVM remotas, pero el usuario no percibe el cambio con la aplicación local, la distribución se realiza de forma transparente al usuario.

- Dado que RMI es una solución cien por cien Java para objetos remotos, proporciona todas las ventajas de las capacidades de Java, puede ser utilizado sobre diversas plataformas, desde mainframes a cajas UNIX hasta máquinas Windows que soporten dispositivos mientras haya una implementación de JVM para esa plataforma.

Pero no se puede olvidar que RMI es una tecnología orientada a objetos y que como tal no realiza una separación entre los requisitos funcionales y los requisitos operacionales, es decir, sufre el problema conocido como falta de separación de aspectos (separations of concerns).

Este problema causa siempre una codificación poco clara, donde el código que atiende a los intereses de la distribución(código seleccionado por los recuadros coloreados) se entrelazan con el código funcional de las clases, quedando a demás esparcido por la inmensidad del código de la aplicación sin estar localizado en una sola parte o módulo del código.

Esto supone una serie de dificultades para el mantenimiento del sistema, una mayor probabilidad de fallos durante su ciclo de creación, dificultades durante su ciclo de vida, una menor reutilización, menor modularidad y sobre todo condena a la nueva aplicación a una mínima escalabilidad ya que ante futuras necesidades el código de partida que se deberá cambiar estará confuso, complicando y retardando la inserción de nuevos requisitos operacionales al sistema.

7.2 Versión de Distribución con POA+RMI=PaDA

La otra propuesta para distribuir la aplicación bancaria es implementar la aplicación bancaria con aspectos, en AspectJ utilizando el patrón de distribución con aspectos PaDA visto en el apartado 6.1.

Esta propuesta intentará solventar los problemas que existentes con RMI, utilizaremos una distribución basada en RMI y la implementaremos a través de aspectos.

Al igual que en la versión de RMI, se seguirán una serie de fases para el desarrollo de la aplicación distribuida:

- Análisis:

  Para el desarrollo de la aplicación bancaria distribuida mediante PaDA la lógica de negocio será, al igual que en la versión RMI, la propia aplicación original. Según la propuesta de distribución, la comunicación entre Usuario y ServidorBancario ha de ser completamente remota. Usuario y ServidorBancario serán los objetos ejecutables del sistema y por ello los principales. Si tenemos en mente una aplicación Cliente/Servidor, Usuario deberá hacer de Cliente del sistema y ServidorBancario de Servidor. Siguiendo con este orden aparece un objeto secundario, BancoImpl, este objeto se encontrará registrado en el servidor de nombres para estar accesible al Usuario. Los otros objetos, CuentaImpl y ClienteImpl, se encuentran dentro del BancoImpl, por lo que accediendo a BancoImpl se accede a estos objetos (Ver figura).

- Análisis de la Reutilización:

  A diferencia del análisis de reutilización de la versión anterior, en la versión con distribución mediante el patrón PaDA, todo es reutilizable.

  La lógica de negocio no se verá afectada en ningún caso, el código de distribución ya no estará esparcido por el resto de la aplicación sino que se encontrara centralizado en unos módulos especiales(aspectos) que son en si reutilizables para otras aplicaciones. Se podrá reutilizar estos aspectos de otros ya existentes, el hecho de estar aplicando un patrón ya aumenta un grado de reutilización.

  Muy importante, en este caso el código de la versión original no se ve modificado, sólo incrementa su comportamiento inicial a distribuido de forma transparente, por lo que futuros cambios serán más fácil de llevar a cabo.

- Diseño:

  La tecnología que se va a usar en esta versión es POA, junto a Java. Se deberá separar la parte del cliente de la parte del servidor, convertir los objetos en remotos teniendo en cuenta que BancoImpl se exportará automáticamente mientras que CuentaImpl y ClienteImpl se exportarán implícitamente. Los objetos remotos deben implementar una interfaz remota, que se ha de crear, y todos los métodos pertenecientes a dicha interfaz deben poder lanzar la excepción RemoteException.

Para la realización de esta aplicación se dividirá el diseño en cuatro partes:

- Diseño de la lógica de negocio: Esta parte ya esta realizada ya que para aplicar la distribución con POA no hace falta modificar la aplicación base.

- Diseño del aspecto ServerSide: El aspecto Server-Side es responsable de cambiar el código del módulo server usando la API especifica de distribución , implementándolo accesible a respuestas de llamadas remotas. Este aspecto tiene que cambiar también los módulos que usan como parámetros o valores de retorno del tipo server, depende de la API de distribución. Server-Side maneja las invocaciones a métodos remotos del lado del servidor, por ejemplo, creando el objeto remoto inicial y registrándolo en el servidor de nombres.

- Diseño del aspecto ClientSide: El aspecto Client-Side es responsable de interceptar las llamadas locales originales efectuadas por el cliente y redireccionarlas como llamadas remotas efectuadas al módulo remoto(servidor). En definitiva maneja las invocaciones a métodos remotos del lado del cliente, por ejemplo, adquiriendo la referencia inicial de objeto remoto.

- Diseño del aspecto ExceptionHandler: El aspecto manejador de excepciones ExceptionHandler es responsable de manejar con una nueva excepción añadida por la definición de los aspectos las excepciones locales que deberán lanzarse remotamente. Estas excepciones creadas envuelven a una excepción no chequeada, lanzándolas sin cambiar la firma del código fuente del sistema original. Por eso, el aspecto manejador de excepciones debe proporcionar también el manejo necesario en las clases del Client-Side.

- Iteración:

El patrón que se va a utilizar para implementar la distribución en nuestra aplicación va a ser el patrón PaDA. Como se vio anteriormente en el apartado 6.

Con este patrón conseguiremos que la aplicación implemente distribución característica de RMI, pero realizando todos los cambios en los aspectos creados para tal fin.

Antes de escribir cualquier aspecto se deberá escribir el código de las interfaces remotas que los objetos, cuyo comportamiento se quiere que sea remoto, deberán implementar por definición. Como se analizo, los objetos remotos serán BancoImpl, CuentaImpl y ClienteImpl, para continuar con una analogía con RMI las interfaces creadas para tales objetos serán I_Banco, I_Cuenta e I_Cliente. Son idénticas a las interfaces que se crearon en la versión de RMI.

I_Banco

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Banco extends Remote
5: {
6:   public I_Cuenta getCuenta(String nomCliente)throws RemoteException;
7:
8:   public I_Cliente getCliente(String numCuenta)
                                                  throws RemoteException;
9: }
```

Código36: Código de la interfaz I_Banco versión AOP.

I_Cliente

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Cliente extends Remote
5: {
6:  public I_Banco getBanco()throws RemoteException;
7:
8:  public String getNombre()throws RemoteException;
9: }
```

Código37: Código de la interfaz I_Cliente versión AOP.

I_Cuenta

```
1: import java.rmi.Remote;
2: import java.rmi.RemoteException;
3:
4: public interface I_Cuenta extends Remote
5: {
6:   public I_Banco getBanco() throws RemoteException;
7:
8:   public I_Cliente getCliente() throws RemoteException;
9:
10:  public String depositar(long cantidad) throws RemoteException;
11:
12:  public long getSaldo() throws RemoteException;
13:
14:  public String getNum()throws RemoteException;
15:
16:  public long sacarDinero (long cantidad)
                        throws NoHayDineroException, RemoteException;
17: }
```

Código 38: Código de la Interfaz I_Cuenta versión AOP.

- El código de la **funcionalidad básica** no se modifica en nada, queda exactamente igual que en la aplicación local, de hecho es el código de la aplicación local original sin restricciones de distribución.

- **Aspecto ServerSide**: Este aspecto se encarga de la parte del servidor del sistema.

La distribución que se va a realizar con estos aspectos será una distribución RMI, para ello se deben importar los paquetes específicos del API de RMI:

```
1: import java.rmi.*;
2: import java.rmi.server.*;
```

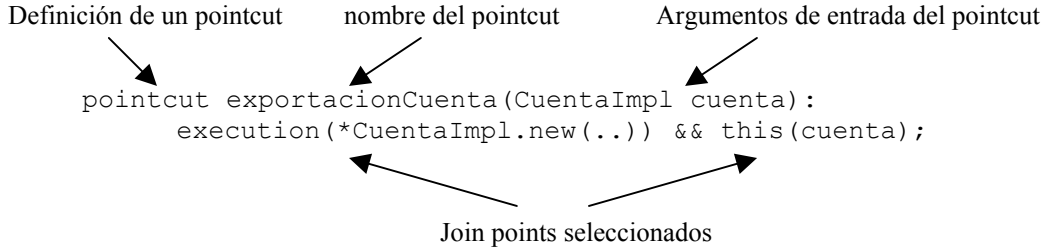
BancoImpl debe de extender de UnicastRemoteObject para que en su creación se exporte automáticamente. Para esto usamos una introducción que modifica la jerarquía de herencias de BancoImpl para que extienda de UnicastRemoteObject.

```
7: declare parents: BancoImpl extends UnicastRemoteObject;
```

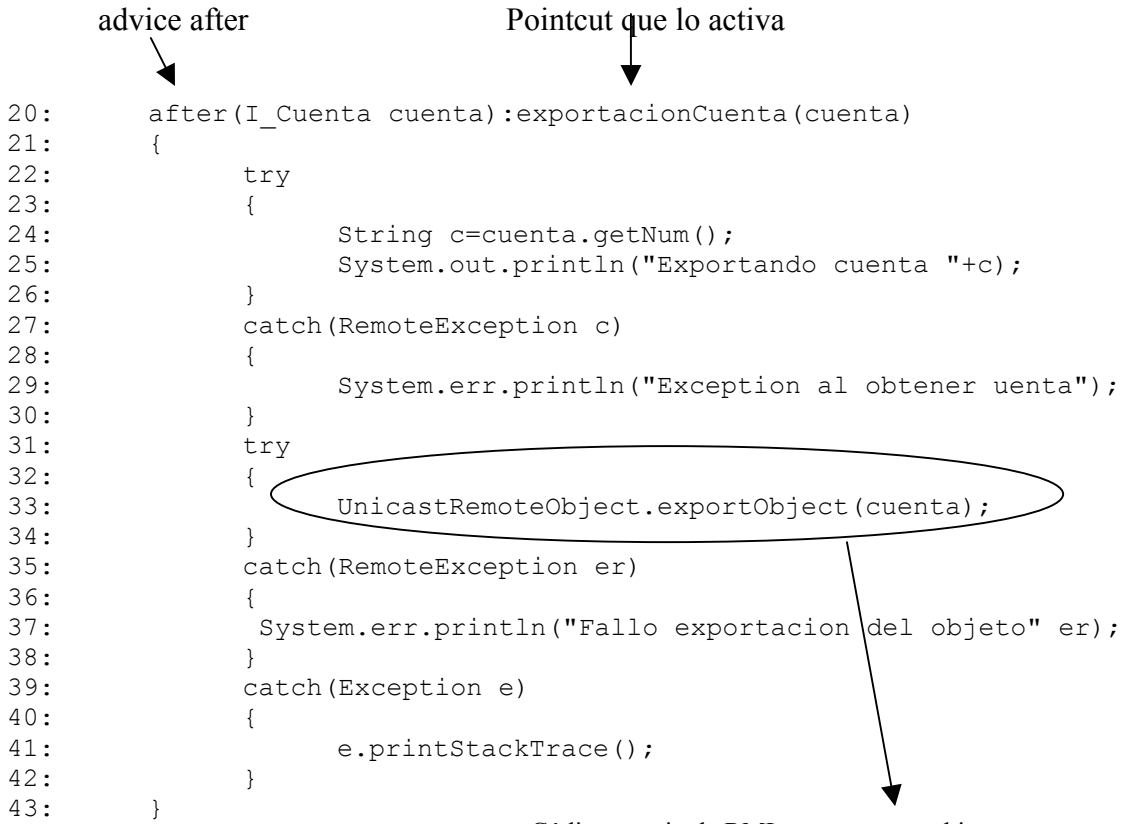
Por definición un objeto remoto debe implementar una interfaz remota, anteriormente se crearon unas interfaces especificas para cada uno de los objetos remotos que se desean implementar, con estas sentencias se añade una nuevas implementaciones a los objetos.

```
11:      declare parents: BancoImpl implements I_Banco;
12:      declare parents: CuentaImpl implements I_Cuenta;
13:      declare parents: ClienteImpl implements I_Cliente;
```

Se crearán dos puntos de corte (pointcut) uno en la línea 16 y otro en la línea 18 del código de ServerSide, uno por cada objeto remoto que no es exportado automáticamente, que seleccionen la creación de un nuevo objeto remoto, pero solo de aquellos que no son exportados automáticamente, estos pointcuts serán utilizados para exportar implícitamente estos objetos.



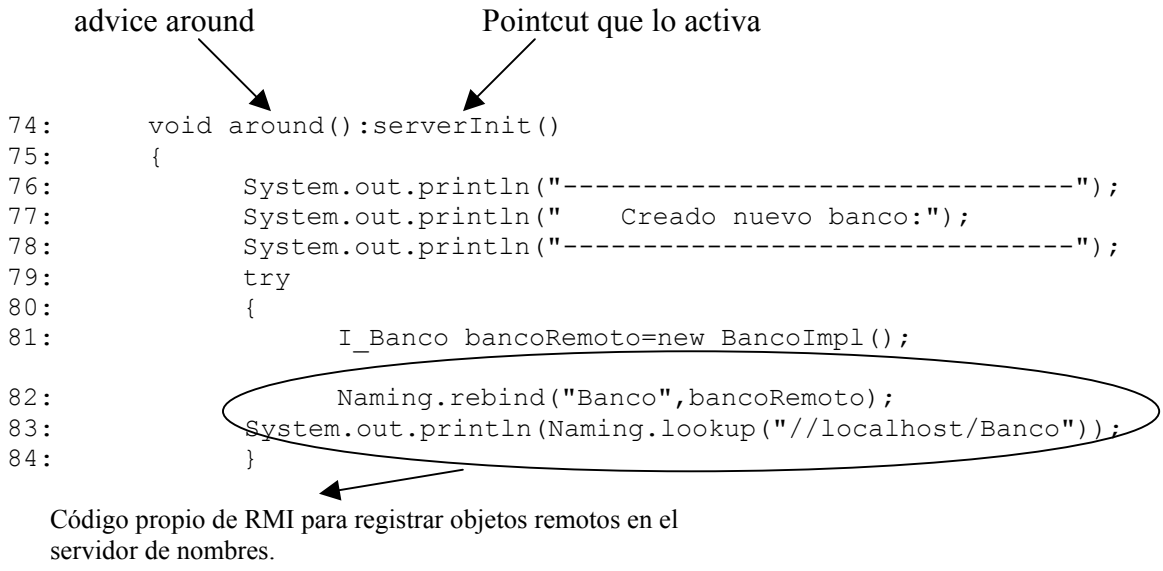
Se crearán dos avisos (advices) desde la línea 20 a la 43 y otro desde la línea 46 a la 69, uno por cada objeto remoto que no es exportado automáticamente. Como los pointcut creados seleccionan la creación de los objetos remotos y la exportación ha de hacerse una vez que dichos objetos existen, los advices son after, para que el cuerpo de dicho advice se ejecute “después” de la creación de los objetos.



Creamos otro pointcut para capturar la creación del ServidorBancario, es decir del servidor de la aplicación.

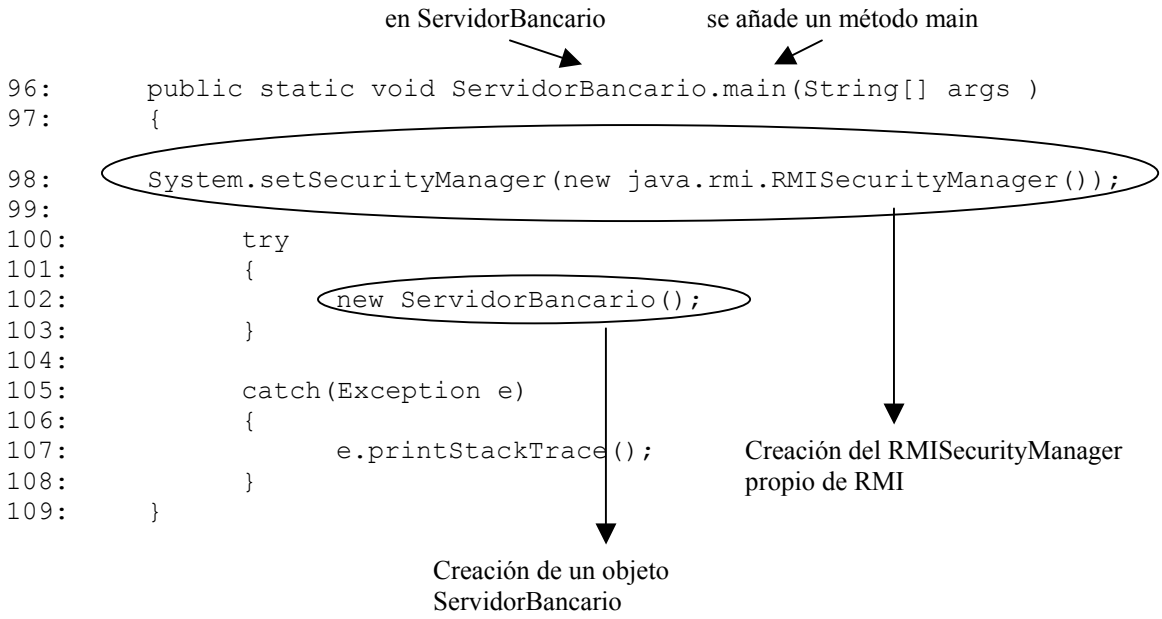
```
71:     pointcut serverInit(): execution(*ServidorBancario.new(..));
```

Capturamos la ejecución del constructor de ServidorBancario, en su lugar se ejecutara el código del advice around que lo captura. Este advice sustituye el constructor original, y en su lugar creara un objeto BancoImpl remoto, y lo registrara en el servidor de nombres que se encuentra en la misma máquina donde se ejecuta ServidorBancario.



```
85:             catch(RemoteException er)
86:             {
87:             System.err.println("Fallo del registro de nombres");
88:             }
89:             catch(Exception e)
90:             {
91:                   e.printStackTrace();
92:             }
93:       }
```

Y por último para que ServidorBancario sea ejecutable independientemente, debe de tener un método main. Desde el aspecto ServerSide se añade el nuevo método main a ServidorBancario, en este método ServidorBancario creare un SecurityManager de RMI(sólo permite conexiones con permisos) y crea un objeto ServidorBancario.



Véase como queda el código al completo del aspecto ServerSide.

ServerSide

```
 1: import java.rmi.*;
 2: import java.rmi.server.*;
 3:
 4: aspect ServerSide
 5: {
 6:  //Para que BancoImpl se exporte automaticamente
 7:  declare parents: BancoImpl extends UnicastRemoteObject;
 8:
 9: //Modificar la estructura de los objetos para que implementen una
10: //interfaz remota, convirtiéndolos asi en objetos remotos.
11:  declare parents: BancoImpl implements I_Banco;
12:  declare parents: CuentaImpl implements I_Cuenta;
13:  declare parents: ClienteImpl implements I_Cliente;
```

```
14:
15:  //Capturar la ejecucion de CuentaImpl
16:  pointcut exportacionCuenta(CuentaImpl cuenta):
                     execution(*CuentaImpl.new(..)) && this(cuenta);
17:  //Capturar la ejecución de ClienteImpl
18:  pointcut exportacionCliente(ClienteImplcliente):
                     execution(*ClienteImpl.new(..)) && this(cliente);

19:  //exportar implícitamente los objetos cuenta
20:  after(I_Cuenta cuenta):exportacionCuenta(cuenta)
21:  {
22:        try
23:        {
24:              String c=cuenta.getNum();
25:              System.out.println("Exportando cuenta "+c);
26:        }
27:        catch(RemoteException c)
28:        {
29:              System.err.println("Exception al obtener Cuenta");
30:        }
31:        try
32:        {
33:              UnicastRemoteObject.exportObject(cuenta);
34:        }
35:        catch(RemoteException er)
36:        {
37:              System.err.println("Fallo exportacion del objeto" +er);
38:        }
39:        catch(Exception e)
40:        {
41:              e.printStackTrace();
42:        }
43:  }
44:
45:  //exportar explícitamente los objetos cliente
46:  after(I_Cliente cliente):exportacionCliente(cliente)
47:  {
48:        try
49:        {
50:              String cl=cliente.getNombre();
51:              System.out.println("Exportando cliente "+cl);
52:        }
53:        catch(RemoteException cl)
54:        {
55:              System.err.println("Exception al obtener el Cliente");
56:        }
57:        try
58:        {
59:              UnicastRemoteObject.exportObject(cliente);
60:        }
61:        catch(RemoteException er)
62:        {
63:              System.err.println("Fallo exportacion del objeto" +er);
64:        }
65:        catch(Exception e)
66:        {
67:              e.printStackTrace();
68:        }
69:  }
70:  //Capturamos la ejecución de ServidorBancario
71:  pointcut serverInit(): execution(*ServidorBancario.new(..));
```

```
 72:  //Capturamos la ejecución del constructor del ServidorBancario
 73:  //Registramos el objeto banco en el servidor de nombres
 74:  void around():serverInit()
 75:  {
 76:        System.out.println("-------------------------------------");
 77:        System.out.println("    Creado nuevo banco:");
 78:        System.out.println("-------------------------------------");
 79:        try
 80:        {
 81:              I_Banco bancoRemoto=new BancoImpl();
 82:              Naming.rebind("Banco",bancoRemoto);
 83:              System.out.println(Naming.lookup("//localhost/Banco"));
 84:        }
 85:        catch(RemoteException er)
 86:        {
 87:              System.err.println("Fallo del registro de nombres");
 88:        }
 89:        catch(Exception e)
 90:        {
 91:              e.printStackTrace();
 92:        }
 93:  }
 94:
 95:  //Creamos el metodo main del servidor
 96:  public static void ServidorBancario.main(String[] args )
 97:  {
 98:        System.setSecurityManager(new java.rmi.RMISecurityManager());
 99:
100:        try
101:        {
102:              new ServidorBancario();
103:        }
104:
105:        catch(Exception e)
106:        {
107:              e.printStackTrace();
108:        }
109:  }
110: }
```

Código 39: Aspecto ServerSide para implementar restricción de distribución en la parte del servidor.

- **Aspecto ClientSide**: Este aspecto se encarga de la parte del cliente del sistema.

La distribución que se va a realizar con estos aspectos será una distribución RMI, para ello se deben importar los paquetes específicos del API de RMI:

```
1: import java.rmi.*;
2: import java.rmi.server.*;
```

Crear una referencia al objeto remotoBancoImpl, esta referencia se utilizara para acceder al objeto BancoImpl que el servidor registro en el servidor de nombres. Esta referencia será utilizada por el aspecto ClientSide solamente.

```
7: private I_Banco bancoRemoto;
```

Característica propia de RMI al referenciar con interfaces.

Crear una referencia a un String que llamaremos dirección dentro del Usuario (proceso cliente del sistema), esto se hace para guardar la dirección de la máquina donde se ejecuta el servidor.

```
8:       String Usuario.direccion;
```

Crear un nuevo constructor en el Usuario que reciba como argumento de entrada la dirección de la máquina donde esta el servidor del sistema(ServidorBancario).

en Usuario se añade un método constructor

```
11:      public Usuario.new(String host)
12:      {
13:           direccion=host;

14:           System.out.println("--------------------------------");
15:           System.out.println("    Creado nuevo Usuario:");
16:           System.out.println("--------------------------------");
17:      }
```

Se crea un punto de corte(pointcut) que seleccione la ejecución del Usuario

```
19:      pointcut usuarioInit():execution(*Usuario.new(..));
```

Se crea un advice after que recibe como argumentos de entrada un String que se corresponde con la dirección ip del servidor, y el propio usuario sobre el que se esta trabajado. Después de que Usuario haya sido creado, obtendrá la referencia al objeto remoto BancoImpl e inicializara la actividad del Usuario.

```
22:      after(String host,Usuario u):usuarioInit() && args(host)
                                      && this(u)
23:      {
24:           try
25:           {
26:           bancoRemoto = (I_Banco)Naming.lookup
                                      ("rmi://"+host+"/Banco");

27:                u.banco=bancoRemoto;
28:           }
29:           catch(Exception e)
30:           {
31:                e.printStackTrace();
32:           }
33:                u.inicialize();
34:      }
```

Sentencia característica de RMI para obtener la referencia remota a través del servidor de nombres

Inicializa la actividad de Usuario

Se crea un pointcut que selecciona las llamadas locales realizadas por Usuario a los métodos de BancoImpl.

```
37:      pointcut llamadasLocales():
                    within(Usuario) && call(* BancoImpl.*(..));
```

Se crearán dos advices around para capturar la ejecución de las llamadas locales y redireccionarlas al objeto remoto. Se creará un advice por cada uno de los métodos de la interfaz remota que implementa BancoImpl, en este caso una advice para el método getCuenta(línea 39-51 del código ClientSide) y otro para el método getCliente(línea 52-64 del código ClientSide).

Estos advices capturan la llamada de los métodos correspondientes que se producen de Usuario a BancoImpl con los argumentos necesarios y con la posibilidad que se pueda lanzar una excepción RemoteException.

El advice se encargara de realizar la llamada al objeto remoto cuya referencia es bancoRemoto obtenida del servidor de nombres, y devuelve el resultado de dicha llamada a la invocación local, de manera que el resultado final es una llamada remota transparente.

```
39:      CuentaImpl around(String nomCliente) throws RemoteException:
                  llamadasLocales()
                  && call(CuentaImpl getCuenta(String nomCliente)
                  && args(String nomCliente)
40:      {
41:            I_Cuenta respuesta=null;
42:            try
43:            {
44:                  respuesta=bancoRemoto.getCuenta(nomCliente);
45:            }
46:            catch(RemoteException re)
47:            {
48:                        re.printStackTrace();
49:            }
50:                  return respuesta;
51:            }
```

La invocación se redirecciona al objeto remto

Y por último para que ClientSide sea ejecutable independientemente, debe de tener un método main. Desde el aspecto ClientSide se añade el nuevo método main que recibe por la entrada estándar la dirección ip del servidor.

```
67:      public static void Usuario.main(String[] args)
68:      {
69:            new Usuario(args[0]);
70:      }
```

Véase como queda el código al completo del aspecto ServerSide.

ClientSide

```
 1: import java.rmi.*;
 2: import java.rmi.server.*;
 3:import java.io.*;
 4:
 5: aspect ClientSide
 6: {
 7:   private I_Banco bancoRemoto;
 8:   String Usuario.direccion;
 9:
10:   //Creamos un nuevo constructor para el usuario
11:   public Usuario.new(String host)
12:   {
13:         direccion=host;
14:         System.out.println("--------------------------------------");
15:         System.out.println("    Creado nuevo Usuario:");
16:         System.out.println("--------------------------------------");
17:   }
18:
19:   pointcut usuarioInit():execution(*Usuario.new(..));
20:
21:   //obtenemos la referencia remota de banco
22:   after(String host,Usuario u):usuarioInit() && args(host) && this(u)
23:   {
24:         try
25:         {
26:               bancoRemoto = (I_Banco)Naming.lookup
                                              ("rmi://"+host+"/Banco");
27:               u.banco=bancoRemoto;
28:               u.inicialize();
29:         }
30:         catch(Exception e)
31:         {
32:               e.printStackTrace();
33:         }
34:   }
35:
36:   //capturar las llamadas locales que deben ser redireccionadas
37:   pointcut llamadasLocales():
                           within(Usuario) && call(* BancoImpl.*(..));

38:   //redireccionar las llamadas locales al objeto remoto
39:   CuentaImpl around(String nomCliente) throws RemoteException:
                llamadasLocales()
                && call(CuentaImpl getCuenta(String nomCliente)
                && args(String nomCliente)
40:   {
41:         I_Cuenta respuesta=null;
42:         try
43:         {
44:               respuesta=bancoRemoto.getCuenta(nomCliente);
45:         }
46:         catch(RemoteException re)
47:         {
48:                     re.printStackTrace();
49:         }
50:               return respuesta;
```

```
51:         }

52:   ClienteImpl around(String numCuenta) throws RemoteException:
                  llamadasLocales()
                  && call(ClienteImpl getCliente(String numCuenta)
                  && args(String numCuenta)
53:   {
54:         I_Cliente respuesta=null;
55:         try
56:         {
57:               respuesta=bancoRemoto.getCliente(numCuenta);
58:         }
59:         catch(RemoteException re)
60:         {
61:                     re.printStackTrace();
62:         }
63:               return respuesta;
64:   }
65:
66:   //Creamos el metodo main del Usuario
67:   public static void Usuario.main(String[] args)
68:   {
69:         new Usuario(args[0]);
70:   }
71: }
```

Código 40: Aspecto ClientSide para implementar restricción de distribución en la parte del cliente.

- **Aspecto ExceptionHandler:** Este aspecto afecta tanto a la parte cliente del sistema como la parte servidor. Controla las excepciones locales para que puedan ser tratadas remotamente

La distribución que se va a realizar con estos aspectos será una distribución RMI, para ello se deben importar los paquetes específicos del API de RMI:

```
1: import java.rmi.*;
2: import java.rmi.server.*;
```

Se crea un pointcut que seleccione la excepción local NoHayDineroException, en concreto el bloque catch del manejador de excepciones.

```
11:      pointcut p():handler(NoHayDineroException);
```

Se declaran todas las excepciones que sean seleccionadas por el pointcut anterior como excepciones no chequeadas.

```
13:      declare soft:RuntimeException:p();
```

Se crea otro pointcut para tratar el resto de excepciones remotas que puedan ocurrir durante la invocación al objeto remoto BancoImpl.

```
16:      pointcut excepcion() :execution(*BancoImpl.*(..));
```

Se crea un advice after que en el caso de que durante la ejecución de cualquier método de BancoImpl se produzca una excepción entonces lanza desde ExceptionHandler una excepción no chequeada con el mensaje "Fallo remoto".

```
17:      after() throwing (SoftException ex): excepcion()
18:      {
19:            System.err.println("Fallo remoto");
20:      }
```

Veamos como queda el código al completo del aspecto ExceptionHandler.

ExceptionHandler

```
 1: import java.rmi.*;
 2: import java.io.*;
 3: import java.rmi.server.*;
 4:
 5: //tiene que capturar las excepciones que antes se realizaban de forma
 6: // local y tratarlas para poder lanzarlas remotamente
 7:
 8: aspect ExceptionHandler
 9:{
10:   //captura la excepcion local
11:   pointcut p():handler(NoHayDineroException);
12:   //envuelve la excepción local dentro de una excepcion no chequeada
13:   declare soft:RuntimeException:p();
14:
15:   //Trata el resto de excepciones que pueden ocurrir remotamente
16:   pointcut excepcion() :execution(*BancoImpl.*(..));
17:   after() throwing (SoftException ex): excepcion()
18:   {
19:         System.err.println("Fallo remoto");
20:   }
21: }
```

Código 41: Aspecto ExceptionHandler para implementar restricción de distribución en la aplicación bancaria.

Ya se tiene todo el código necesario para implementar la aplicación distribuida con aspectos, por un lado el código de la funcionalidad básica y por otro lado el de distribución. Se realizará el proceso de entrelazado(weaving) para obtener la aplicación final distribuida, este proceso entrelazara y compilara el código generando los ficheros .class de nuestra aplicación, para entrelazar y compilar los ficheros fuente usamos el tejedor(Weaver) de AspectJ *ajc*. En este proceso deben participar todas las clases y aspectos asociados de la aplicación, si suponemos que se encuentran dentro de un mismo directorio llamado AplicacionBancaria que cuelga del directorio raíz(C:\), el comando a ejecutar sería el siguiente:

ajc C:\AplicaciónBancaria.java*

De esta manera todos los archivos .java (clases y aspectos) serán entrelazados y compilados obteniendo los archivos .class necesarios.

Como los aspectos han implementado distribución propia de RMI es necesario generar los Stubs y Skeleleton de los objetos ahora remotos(BancoImpl, CuentaImpl, ClienteImpl), para esto se dispone del comando *rmic* de RMI.

rmic BancoImpl
rmic CuentaImpl
rmic ClienteImpl

Ahora la aplicación distribuida recibe el mismo tratamiento que una aplicación distribuida mediante RMI. Se deberán ejecutar los procesos clientes y el proceso servidor, en el servidor deberá estar activo el registro de nombres rmiregistry y al igual que en RMI debe existir un fichero .java.policy que contenga los permisos necesarios para que la comunicación remota se posible.

Este ejemplo es un ejemplo significativo de distribución y se eligió para demostrar algunas de las características de la programación orientada a aspectos y sobre todo de la distribución con la programación orientada a aspectos, aunque revela sólo una pequeña parte de las verdaderas capacidades de esta tecnología.

Se puede observar que la información que solamente interesa para la distribución se ha pasado al código de los aspectos, y como la distribución es implementada de forma transparente a la aplicación bancaria, ejecutando el código fuente del sistema independientemente de la API.

Con esta versión la aplicación queda más clara al aplicar la estrategia de distribución usando la programación orientada a aspectos. El sistema obtiene una alta modularidad.

Como el código de distribución esta completamente separado del código base del sistema, los futuros cambios del sistema son más simples y no causan ningún impacto en el código base, uno de estos cambios puede ser un cambio en la API de comunicaciones o cambio de aspectos existentes para corregir errores.

Conclusión y Líneas Futuras

Actualmente, la implementación de distribución en sistemas tiene una alta demanda. La continua evolución a la que se ven sometidos los distintos sistemas desarrollados hace necesaria la implantación de un modelo de desarrollo que permita una fácil adaptabilidad y reutilización de los mismos, sin mencionar la importancia de un mínimo periodo de tiempo para ello.

En este proyecto se ha enfocado el desarrollo de aplicaciones distribuidas desde la perspectiva del paradigma de la Programación Orientada a Aspectos. Concretamente se ha tenido en cuenta el enfoque propuesto por el patrón PaDA, que propone la implementación de las especificaciones necesarias de la aplicación mediante aspectos, con el fin de componerlos posteriormente junto al comportamiento funcional, para dar lugar a una aplicación distribuida RMI.

Este enfoque ha permitido comprobar que efectivamente la programación orientada a aspectos es una buena aproximación para el desarrollo de aplicaciones distribuidas ya que una aplicación de estas características, cuando se trataba con la programación convencional (programación orientada a objetos), tenía una gran complejidad, una mínima adaptabilidad y mínima reutilización.

Los aspectos son módulos que integran una aplicación orientada a aspectos. Dicha modularidad permite la introducción de requisitos adicionales sin comprometer la claridad, adaptabilidad, mantenimiento y reutilización del código.

La adaptabilidad es un área donde la programación orientada a aspectos sobresale, sin embargo según en qué casos de uso será mayor o menor. En el caso de un cambio pequeño y no revolucionario el trabajo requerido será casi por igual en las dos versiones. Si la aplicación original estaba preparada desde su creación inicial para una serie de cambios, esta claro que estas consideraciones anticipadas pueden suponer una ventaja que hay que considerar. Ante un cambio rotundo o de gran envergadura, POA es superior con diferencia, ya que con la programación tradicional el sistema entero tendría que ser rehecho.

Pero si tal es la facilidad de agregar, la misma es para eliminar, en el caso de que una aplicación tratada con programación orientada a aspectos decida cambiar su comportamiento eliminando ciertos requisitos, no hace falta rehacer de nuevo la aplicación, simplemente eliminar los aspectos que implementaban dicho requisito. Las aplicaciones se vuelven más flexibles.

Al estar el código de distribución centrado en unos módulos diferenciados, el código esta más claro, menos enredado, y en el número de líneas de código de aplicación disminuyen, ya que el código repetitivo es eliminado y concentrado, reduciendo el coste de mantenimiento del código. Por la disminución del código esta claro que disminuye el mantenimiento pero sobre todo por la claridad y modularidad del código.

La programación orientada a aspectos es sin duda el futuro de la programación, no sólo soluciona las desventajas de la programación convencional, sino que deja abierta una puerta con infinidad de soluciones aún por descubrir.

Con AspectJ como uno de los lenguajes más fuertes en el momento, se acumulan las ventajas de la orientación a objetos en Java y la orientación a aspectos.

La programación orientada a aspectos es aún muy joven, sería realmente interesante desarrollar una comunidad de usuarios reales para poder estudiar los efectos prácticos.

En cuanto a la tecnología, sin duda debe evolucionar y madurar, la línea de los patrones es muy interesante, ya que consiguen generalizar los problemas y dar soluciones comunes sin necesidad de que el usuario profundice demasiado.

El grupo de trabajo de Xerox Palo Alto Research Center, tiene pensado continuar sus investigaciones centradas en AspectJ. Intentar mejorar el tejedor (Weaver) para evitar la necesidad de requerir el acceso a todo el código fuente del sistema, intentar poder realizar una recompilación completa desde cualquier parte del programa siempre que el programa de usuario cambie

En cuanto a los últimos trabajos en Ingeniería del Software se puede concluir que la Orientación a Aspectos se está extendiendo a las fases tempranas del desarrollo de SW, no sólo al diseño sino a la fase de captura de requisitos donde la identificación de aspectos en los casos de uso de UML constituyen un frente abierto.

Anexo A: Pasos Genéricos para Implementar Distribución Usando PaDA.

1. Crear las interfaces que los objetos deben implementar para poder comportarse de forma remota

1.1 El futuro objeto remoto deberá implementar una interfaz remota que cumpla las siguientes especificaciones.

1.1.1 La interfaz remota debe importar los paquetes necesarios de la API especifica de RMI.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

1.1.2 Hacer que la interfaz extienda de Remote.

```
public [NombreInterfaz] extends Remote.
```

1.1.3 Hacer que los métodos definidos en la interfaz puedan lanzar la excepción RemoteException.

```
[modificador][Type return][nombreMétodo]
            ([parametrosEntrada]) throws RemoteException;
```

1.1.4 Ejemplo genérico .

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface [NombreInterfaz1] extends Remote
{
      public [Type return1] [nombreMetodo1]
            ([ParámetrosEntrada]) throws RemoteException;

      public [Type return2] [nombreMetodo2]
            ([ParámetrosEntrada]) throws RemoteException;
}
```

2. Crear el aspecto ServerSide.

2.1 El aspecto deberá importar los paquetes de la API especifica de RMI que se necesitan.

```
import java.rmi.*;
import java.rmi.server.*;
```

2.2 Declarar el aspecto.

```
public aspect ServerSide
```

2.3 Cambiar la herencia de aquellos objetos que se pretende que sean remotos y además se exporten automáticamente.

Para esto se extenderá los futuros objetos remotos de UnicastRemoteObject.

```
declare parents: [NombreClaseRemota] extends
                                    UnicastRemoteObject;
```

2.4 Obligar a los futuros objetos remotos que implementen una interfaz remota(creada en el apartado 1).

```
declare parents: [NombreClaseRemota] implements
                                    [InterfazRemota];
```

2.5 Crear un punto de corte(pointcut) que seleccione la ejecución de un nuevo objeto remoto.

```
pointcut [nombrePointcut]([NombreClaseRemota] [remotin]):
  execution (*[NombreClaseRemota].new(..)) && this([remotin]);
```

Esto se realizara para todos los objetos que se han definido anteriormente como remotos pero que deben exportarse explícitamente.

2.6 Exportar los objetos que no se exportan automáticamente.(en el caso de ser necesario)

Es decir aquellos que implementan la interfaz remota pero que no extienden de UnicastRemoteObject.

```
after([InterfazRemota objetoRemoto]):
                                nombrePointcut(objetoRemoto)
{
   try
   {
         UnicastRemoteObject.exportObject(objetoRemoto);
   }
   catch(RemoteException er)
   {
         System.err.println("Fallo de exportación"  +er);
   }
```

```
}
```

2.7 Capturar la ejecución del proceso que actúa como servidor.

```
pointcut [nombrePointcutServidor]():
                    execution(*[ProcesoServidor].new(..));
```

2.8 Crear, en el proceso servidor, los objetos remotos que necesiten ser registrados en el servidor de nombres, es decir los objetos que tienen que estar accesibles para los procesos cliente.

```
void around():[NombrePointcutServidor]()
{
   try
   {
         [InterfazRemota objetoRemoto]=[ClaseRemota].new ();
         Naming.rebind("[NombreDeRegistro]",objetoRemoto);
         Naming.lookup("//localhost/NombreDeRegistro");
   }
   catch(RemoteException er)
   {
         System.err.println("Fallo de registro" +er);
   }
   catch(Exception e)
   {
         e.printStackTrace();
   }
}
```

2.9 Crear el método main del proceso servidor para que sea ejecutable.

```
public static void [ProcesoServidor].main(String[] args )
{
   System.setSecurityManager( new
                           java.rmi.RMISecurityManager());

   try
   {
         new [ProcesoServidor]();
   }
   catch(Exception e)
   {
         e.printStackTrace();
   }
}
```

2.10. Ejemplo genérico

```
import java.rmi.*;
import java.rmi.server.*;

public aspect ServerSide
{
//Sólo para aquellas que deben exportar automáticamente.
   declare parents: [ClaseRemota1E] extends UnicastRemoteObject;
   declare parents: [ClaseRemota2E] extends UnicastRemoteObject;
                                  .
                                  .
                                  .
```

```
declare parents: [ClaseRemotaNE] extends
                                        UnicastRemoteObject;

//Sólo para aquellas que no deben exportar automáticamente.
pointcut [nombrePointcut1]([ClaseRemota1 r1]): execution
         (*[ClaseRemota1].new(..)) && this(r1);

pointcut [nombrePointcut2]([ClaseRemota2 r2]): execution
         (*[ClaseRemota2].new(..)) && this(r2);
                                    .
                                    .
                                    .
pointcut [nombrePointcutN]([ClaseRemotaN rn]): execution
         (*[ClaseRemotaN].new(..)) && this(rn);

after([NombreInterfaz1 r1]): [nombrePointcut1](r1)
{
   try
   {
         UnicastRemoteObject.exportObject(r1);
   }
   catch(RemoteException er)
   {
         System.err.println("Fallo de exportacion " +er);
   }
}

after([NombreInterfaz2 r2]): [nombrePointcut2](r2)
{
   try
   {
         UnicastRemoteObject.exportObject(r2);
   }
   catch(RemoteException er)
   {
         System.err.println("Fallo de exportación " +er);
   }
}
                                    .
                                    .
                                    .
after([NombreInterfazN rn]): [nombrePointcutN](rn)
{
   try
   {
         UnicastRemoteObject.exportObject(rn);
   }
   catch(RemoteException er)
   {
         System.err.println("Fallo de exportacion " +er);
   }
}

pointcut [PointcutServidor](): execution
                                (*[ProcesoServidor].new(..));
```

```
      //Solo se registraran los objetos que deban ser accesibles
      //remotamente por el cliente
          void around():[PointcutServidor]()
          {
             try
             {
                   [NombreInterfaz1E r1e]=[ClaseRemota1E].new ();
                   Naming.rebind("[NombreDeRegistro1]",r1e);
                   Naming.lookup("//localhost/NombreDeRegistro1");
             }
             catch(RemoteException er)
             {
                   System.err.println("Fallo de registro" +er);
             }
             catch(Exception e)
             {
                   e.printStackTrace();
             }
             try
             {
                   [NombreInterfaz2E r2e]=[ClaseRemota2E].new ();
                   Naming.rebind("[NombreDeRegistro2]",r2e);
                   Naming.lookup("//localhost/NombreDeRegistro2");
             }
             catch(RemoteException er)
             {
                   System.err.println("Fallo de registro" +er);
             }
             catch(Exception e){
                   e.printStackTrace();
             }
                                  .
                                  .
                                  .
             try
             {
                   [NombreInterfazNE rne]=[ClaseRemotaNE].new ();
                   Naming.rebind("[NombreDeRegistroN]",rne);
                   Naming.lookup("//localhost/NombreDeRegistroN");
             }
             catch(RemoteException er){
                   System.err.println("Fallo de registro" +er);
             }
             catch(Exception e){
                   e.printStackTrace();
             }
       }

       public static void [ProcesoServidor].main(String[] args )
       {
             System.setSecurityManager(new java.rmi.RMISecurityManager());
             try
             {
                   new [ProcesoServidor]();
             }
             catch(Exception e){
                   e.printStackTrace();
             }
       }
}
```

3. Crear el aspecto ClientSide.

3.1 Importar los paquetes de la API especifica de RMI que necesitamos.

```
import java.rmi.*;
import java.rmi.server.*;
```

3.2 Declarar el aspecto.

```
public aspect ClientSide
```

3.3 Crear una referencia a los objetos remotos a los que queremos acceder a través del servidor de nombres.

```
private [NombreInterfaz1E objetoRemoto1];
private [NombreInterfaz2E objetoRemoto2];
            .
            .
            .
private [NombreInterfazNE objetoRemotoN];
```

3.4Crear una referencia del tipo String dentro del proceso que actúa como cliente.

Esto se hace para guardar la dirección de la máquina donde se ejecuta el servidor.

```
String [ProcesoCliente].direccion;
```

3.5 Crear un nuevo constructor en el cliente que reciba como argumento de entrada la dirección de la máquina donde esta el servidor.

```
public [ProcesoCliente].new(String host)
{
      direccion=host;
}
```

Si queremos podemos completar el constructor para que muestre por pantalla un mensaje o cualquier otra cosa que se considere oportuna.

3.6 Captura la ejecución del proceso que actúa como cliente.

```
pointcut [nombrePointcutCliente](): execution
                              (*[ProcesoCliente].new(..));
```

3.7 Obtener la referencia remota localizada en el servidor de nombres.

```
after(String host,[ProcesoCliente c]) :
        [nombrePointcutCliente]() && args(host) && this(c)
{
      try
      {
            objetoRemoto = ([InterfazRemota])Naming.lookup
                  ("rmi://"+host+"/[NombreDeRegistro]");
            c.[referenciaLocal]= objetoRemoto;
      }
      catch(Exception e)
      {
            e.printStackTrace();
      }
//Continuamos la ejecución normal del cliente, normalmente se
//corresponde con la invocación al método inicialize() del
//constructor.
      c.[inicialize]();
}
```

3.8 Capturar las llamadas realizadas de forma local para poder redireccionarlas al objeto remoto.

```
 pointcut llamadasLocales(): within([ProcesoCliente]) &&
        call(* [ObjetoRemoto].*(..));

[TyperReturn] around(parametrosEntrada]) throws RemoteException:
      llamadasLocales()
      && call([TypeReturn][nombreMetodo]([parametrosEntrada])
      && args([parametrosEntrada])
{
      TypeReturn respuesta=null;
      try
      {
           respuesta=objetoRemoto.[nombreMetodo]
                                    (parametrosEntrada);
      }
      catch(RemoteException re)
      {
            re.printStackTrace();
      }
      return respuesta
}
```

3.9 Crear el método main del proceso cliente para que sea ejecutable.

```
//Es necesario pasarle por la entrada estándar la dirección //de
la máquina donde se encuentra el servidor.

public static void [ProcesoCliente].main(String[] args)
{
   new [ProcesoCliente](args[0]);
}
```

3.10 Ejemplo genérico.

```
   import java.rmi.*;
   import java.rmi.server.*;

   public aspect ClientSide
   {
//Solo para los objetos que se encuentran registrados en el
//servidor de nombres
      private [NombreInterfaz1E objetoRemoto1];
      private [NombreInterfaz2E objetoRemoto2];
                  .
                  .
                  .
      private [NombreInterfazNE objetoRemotoN];

      String [ProcesoCliente].direccion;
//Nuevo constructor para el cliente
      public [ProcesoCliente].new(String host)
      {
            direccion=host;
      }
      pointcut [nombrePointcutCliente]() :execution
                                    (*[ProcesoCliente].new(..));
      after(String host,[ProcesoCliente c]) :
            [nombrePointcutCliente]() && args(host) && this(c)
      {
           try{
               objetoRemoto1 = ([InterfazRemota1]) Naming.lookup
                       ("rmi://"+host+"/[NombreDeRegistro1]");
               c.[referenciaLocal1]= objetoRemoto1;
            }
            catch(Exception e){
               e.printStackTrace();
            }
           try{
               objetoRemoto2 = ([InterfazRemota2])Naming.lookup
                       ("rmi://"+host+"/[NombreDeRegistro2]");
               c.[referenciaLocal2]= objetoRemoto2;
            }
            catch(Exception e){
                  e.printStackTrace();
            }
                              .
                              .
                              .
            try{
               objetoRemotoN = ([InterfazRemotaN])Naming.lookup
                       ("rmi://"+host+"/[NombreDeRegistroN]");
               c.[referenciaLocalN]= objetoRemotoN;
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
//Continuamos la ejecución normal del cliente, normalmente se
corresponde con la invocación al método inicialize().
            c.[inicialize]();
      }
   pointcut llamadasLocales1(): within([ProcesoCliente]) && call(*
           [ObjetoRemoto1].*(..));
```

```
   pointcut llamadasLocales2(): within([ProcesoCliente]) && call(*
          [ObjetoRemoto2].*(..));
                                  .
                                  .
                                  .
   pointcut llamadasLocalesN(): within([ProcesoCliente]) && call(*
          [ObjetoRemotoN].*(..));

  [TyperReturn1]around(parametrosEntrada1])throws RemoteException :
      llamadasLocales1()
      && call([TypeReturn1][nombreMetodo1]([parametrosEntrada1])
      && args([parametrosEntrada1])
  {
        TypeReturn1 respuesta=null;
        try{
            respuesta=
              objetoRemoto1.[nombreMetodo1](parametrosEntrada1);
        }
        catch(RemoteException re){
            re.printStackTrace();
        }
        return respuesta;
  }
  [TyperReturn2]around(parametrosEntrada2]) throws RemoteException:
      llamadasLocales2()
      && call([TypeReturn2][nombreMetodo2]([parametrosEntrada2])
      && args([parametrosEntrada2])
      {
        TypeReturn2 respuesta=null;
        try{
            respuesta=
              objetoRemoto2.[nombreMetodo2](parametrosEntrada2);
        }
        catch(RemoteException re){
                  re.printStackTrace();
        }
        return respuesta;
  }
                                  .
                                  .
                                  .
[TyperReturnN] around(parametrosEntradaN])throws RemoteException :
      llamadasLocalesN()
      &&call([TypeReturnN][nombreMetodoN]([parametrosEntradaN])
      && args([parametrosEntradaN])
{
      TypeReturnN respuesta=null;
      try{
      respuesta=
              objetoRemotoN.[nombreMetodoN](parametrosEntradaN);
      }
            catch(RemoteException re){
                  re.printStackTrace();
            }
            return respuesta;
      }
   public static void [ProcesoCliente].main(String[] args)
   {
      new Usuario(args[0]);
   }
}
```

4. Crear el aspecto ExceptionHandler.

4.1 Importar los paquetes de la API especifica de RMI que necesitamos.

```
import java.rmi.*;
import java.rmi.server.*;
```

4.2 Declarar el aspecto.

```
public aspect ExceptionHandler
```

4.3 Capturar el manejador de excepciones locales que deben ser lanzadas remotamente.

```
pointcut [NombrePointcutExcepcion]():handler ([ExcepcionLocal]);
```

4.4 Encapsular dichas excepciones locales dentro de una excepción no chequeada.

```
declare soft:RemoteException:[NombrePointcutException]();
```

4.5 Seleccionar el objeto remoto que se encuentra registrado en el servidor de nombres y relanzar la excepción original encapsulada dentro de una excepción no chequeada.

```
pointcut excepcion() :execution(* [ObjetoRemoto].*(..));

after() throwing (SoftException ex): execution()
{
      System.err.println("Fallo remoto");
}
```

4.2 Ejemplo genérico.

```
import java.rmi.*;
import java.rmi.server.*;

public aspect ExceptionHandler
{
   pointcut [NombrePointcutExcepcion1]():handler
                                  ([ExcepcionLocal1]);

   pointcut [NombrePointcutExcepcion2]():handler
                                  ([ExcepcionLocal2]);
                                .
                                .
                                .
   pointcut [NombrePointcutExcepcionN]():handler
                                  ([ExcepcionLocaN]);

  declare soft:RemoteException:[NombrePointcutException1]();
  declare soft:RemoteException:[NombrePointcutException2]();
                                .
                                .
```

```
                                .
        declare soft:RemoteException:[NombrePointcutExceptionN]();

        pointcut excepcion() :execution1(* [ObjetoRemoto1].*(..));
        pointcut excepcion() :execution2(* [ObjetoRemoto2].*(..));
                                .
                                .
                                .
        pointcut excepcion() :executionN(* [ObjetoRemotoN].*(..));

        after() throwing (SoftException ex): execution1(){
             System.err.println("Fallo remoto");
        }

        after() throwing (SoftException ex): execution2(){
             System.err.println("Fallo remoto");
        }
                                .
                                .
                                .
        after() throwing (SoftException ex): executionN(){
             System.err.println("Fallo remoto");
        }
}
```

Ejemplo de aplicación resultante al aplicar PaDA a la aplicación HolaMundo original

Interfaz HolaMundoRemoto

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public HolaMundoRemoto extends Remote
{
      public String objHola() throws
RemoteException;
}
```

1

HolaMundoO

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

2.1

2.4

```
public class HolaMundoO extends UnicastRemoteObject
                        implements HolaMundoRemoto
```

2.5

```
{
      public HolaMundoO()
      {
            super();
      }
      public String objHola() throws RemoteException
      {
            return("Hola Mundo");
      }
}
```

4.5

HolaMundoC

```
import java.io.*;
import java.rmi.*:
import java.rmi.server.*;
```

3.1

```
public class HolaMundoC
{
      private HolaMundoO hm;

      private HolaMundoRemoto remoto;
```

3.3

```
      private String dirección;
```

3.4

```
public HolaMundoC(HolaMundoO hm)
{
      this.hm=hm;
      inicialize();
}
```

3.5

```
public HolaMundoC(String host)
{
dirección=host;

      try{
      remoto=(HolaMundoRemoto)Namming.lookup
                        ("rmi://"+host+"/ObjetoRemoto");
}

cathc(RemoteException re)
{
      re.printStackTrace();
}
inicialize();

}
```

3.7

```
public void inicialice()
{
      //System.out.println(hm.objHola());
      System.out.println(remoto.objHola());
}
```

```
public static void main(String[] args)
{
new HolaMundoC(args[0]);
}

}
```

3.9

HolaMundoS

```
import java.rmi.*;
import java.rmi.server.*;
```

2.1

```
public class HolaMundoS
{
      HolaMundoO holaO;

      public HolaMundoS()
      {
            try{
                  HolaMundoRemoto remoto =new HolaMundoO();
                  HolaO=remoto;
                  Naming.rebind("ObjetoRemoto",remoto)
                  Naming.lookup("//localhost/ObjetoRemoto");
            }
            catch(RemoteException re)
            {
            re.printStackTrace();
            }
      }

      public HolaMundoO getO
      {
            return holaO;
      }

      public static void main(String[] args)
      {
      new HolaMundoS(args[0]);
      }
}
```

2.8

2.9

```
import java.io.*;
public class Principal{
      public static void main(String[] args)    {
            HolaMundoS hms = new HolaMundoS();
            HolaMundoO hm=hms.getO();
            HolaMundoC hmc =new HolaMundoC(hm);
            System.exit(0);
      }
}
```

Diagrama de clases de la aplicación original HolaMundo

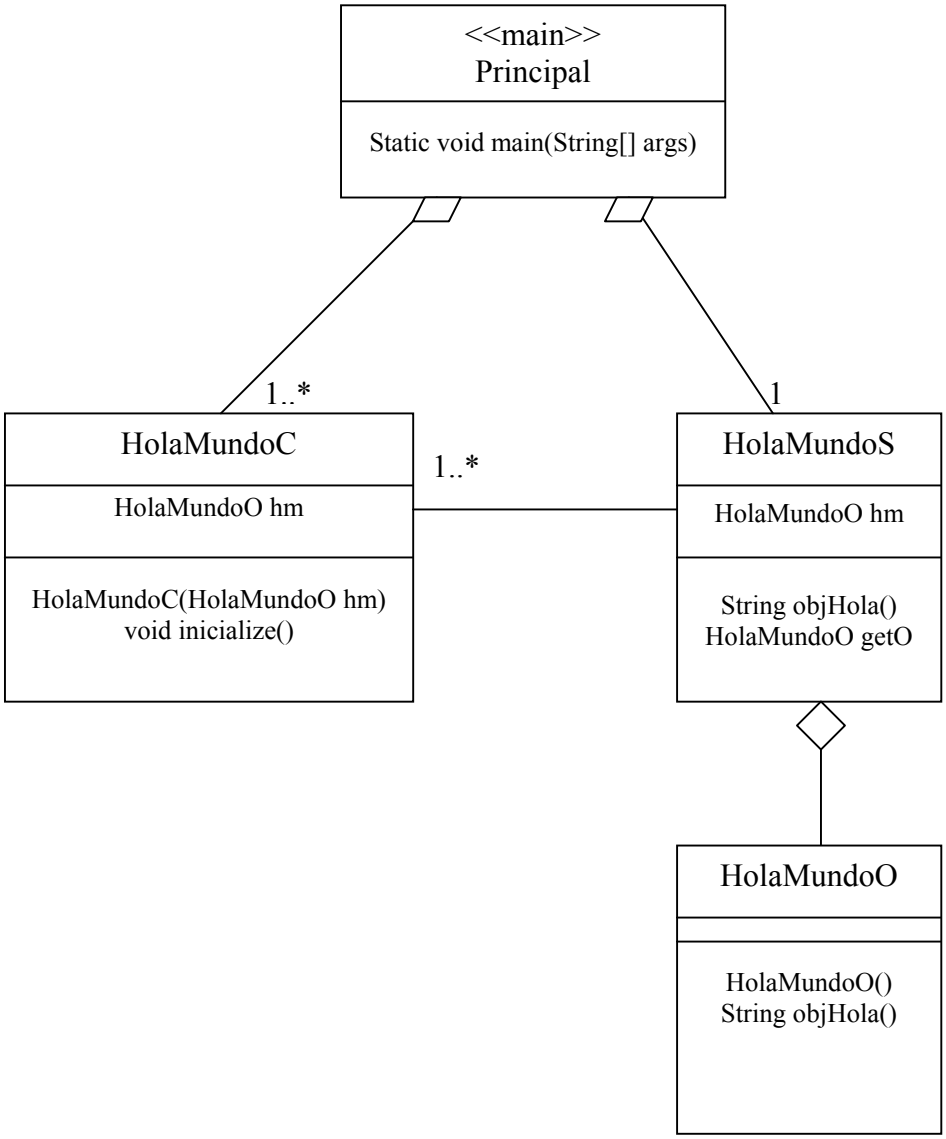


Figura 42: Diagrama de clases de la aplicación original HolaMundo.

Diagrama de clases de la aplicación después de aplicar PaDA

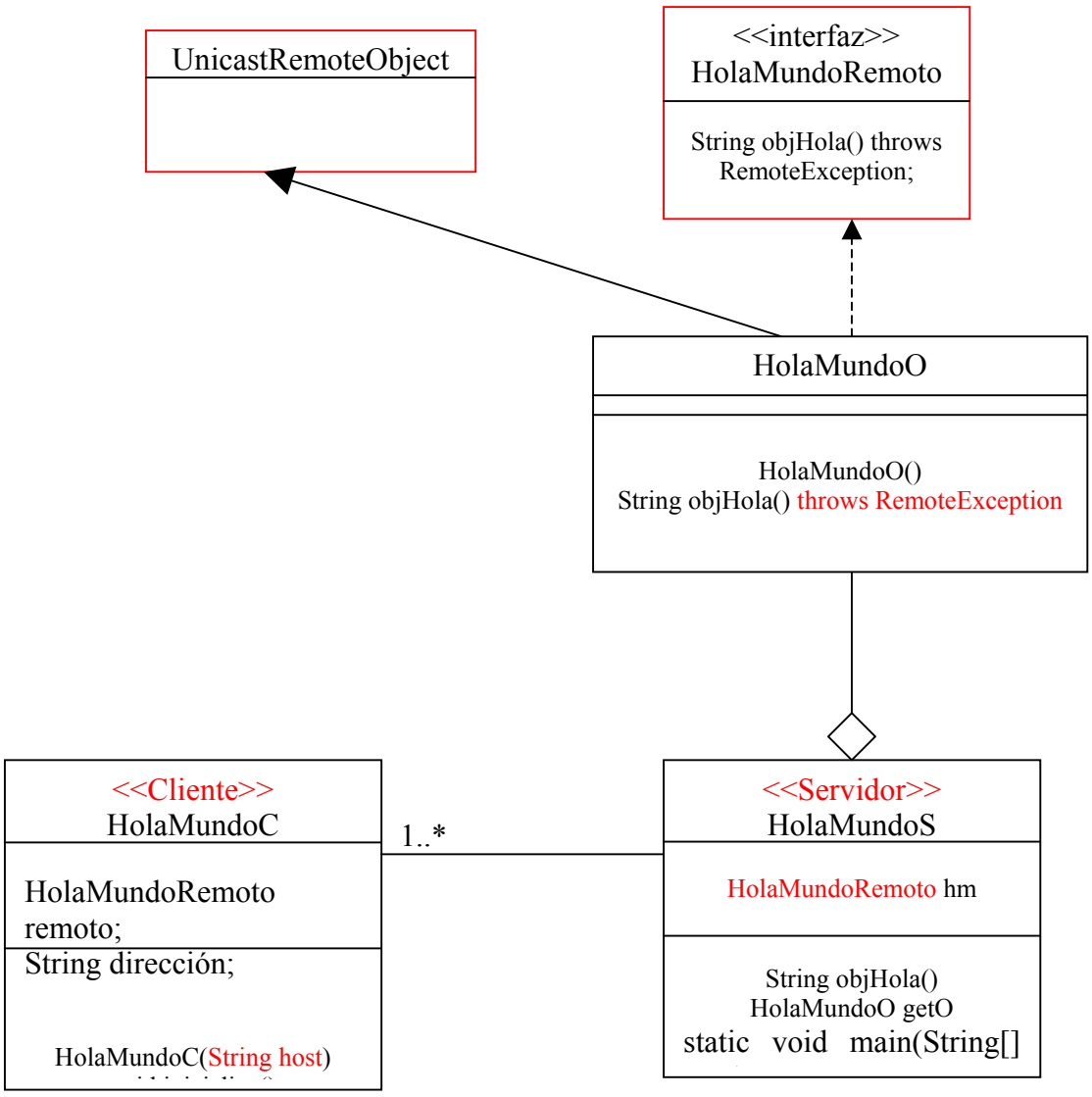


Figura 43: Diagrama de clases después de aplicar PaDA.

Código fuente de la aplicación HolaMundo original.

HolaMundoO

```
public class HolaMundoO
{
	public HolaMundoO()
	{
		super();
	}
	public String objHola()
	{
		return("Hola Mundo");
	}
}
```

HolaMundoC

```
import java.io.*;
public class HolaMundoC
{
	private HolaMundoO hm;
	public HolaMundoC(HolaMundoO hm)
	{
		this.hm=hm;
		inicialize();
	}
	public void inicialize()
	{
		System.out.println(hm.objHola());
	}
}
```

HolaMundoS

```
public class HolaMundoS
{	private HolaMundoO holaO;

	public HolaMundoS()
	{
		HolaMundoO hmi=new HolaMundoO();
	}
	public HolaMundoO getO
	{
		return holaO;
	}
}
```

Principal

```
public class Principal{
	public static void main(String[] args)
	{
		HolaMundoS hms = new HolaMundoS();
		HolaMundoO hm=hms.getO();
		HolaMundoC hmc =new HolaMundoC(hm);
	}
```

```
}
```

Código fuente de los aspectos

Interfaz remota HolaMundoRemoto

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public HolaMundoRemoto extends Remote
{
      public String objHola() throws RemoteException;
}
```

Aspecto ServerSide

```
import java.rmi.*;
import java.rmi.server.*;

public aspect ServerSide
{

      declare parents: HolaMundoO extends UnicastRemoteObject;

      declare parents: HolaMundoO implements HolaMundoRemoto;

      pointcut initServer(): execution(*HolaMundoS.new(..));

      void around():initServer()
      {
            try
            {
                  HolaMundoRemoto mundoRemoto=HolaMundoO.new ();
                  Naming.rebind("MundoRemoto",mundoRemoto);
                  Naming.lookup("//localhost/MundoRemoto");
            }
            catch(RemoteException er)
            {
                  System.err.println("Fallo de registro " +er);
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
      }
      public static void HolaMundoS.main(String[] args )
      {
            System.setSecurityManager(new java.rmi.RMISecurityManager());
            try
            {
                  new HolaMundoS();
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
      }
```

Aspecto ClientSide

```
import java.rmi.*;
import java.rmi.server.*;


public aspect ClientSide
{
      private HolaMundoRemoto holaRemoto;

      String HolaMundoC.direccion;

      public HolaMundoC.new(String host)
      {
            direccion=host;
      }

      pointcut initClient(): execution(*HolaMundoC.new(..));

      after(String host,HolaMundoC c]):initClient() && args(host) &&
      this(c)
      {
            try
            {
                  holaRemoto = (HolaMuncoRemoto)Naming.lookup
                              ("rmi://"+host+"/MundoRemoto]");
                  c.hmo= holaRemoto;
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
     //Continuamos la ejecución normal del cliente, normalmente se
     //corresponde con la invocación al método inicialize() del
     //constructor.
            c.inicialize();
      }

      pointcut llamadasLocales():
                  within(HolaMundoC) && call(* HolaMundoO.*(..));

      String around() : llamadasLocales() && call(String objHola()
      {
      String respuesta=null;
      try
            {
            respuesta=holaRemoto.objHola();
      }
            catch(RemoteException re)
      {
                  re.printStackTrace();
            }

      return respuesta;
      }
      public static void HolaMundoC.main(String[] args)
      {
            new HolaMundoC(args[0]);
      }
}
```

Aspecto ExceptionHandler

```
import java.rmi.*;
import java.rmi.server.*;

public aspect ExceptionHandler
{
	//No lanza excepciones locales

	pointcut excepcion() :execution(* HolaMundoO.*(..));

	after() throwing (SoftException ex): excepcion(){
			System.err.println("Fallo remoto");
	}
}
```

Diagrama de clases de los aspectos

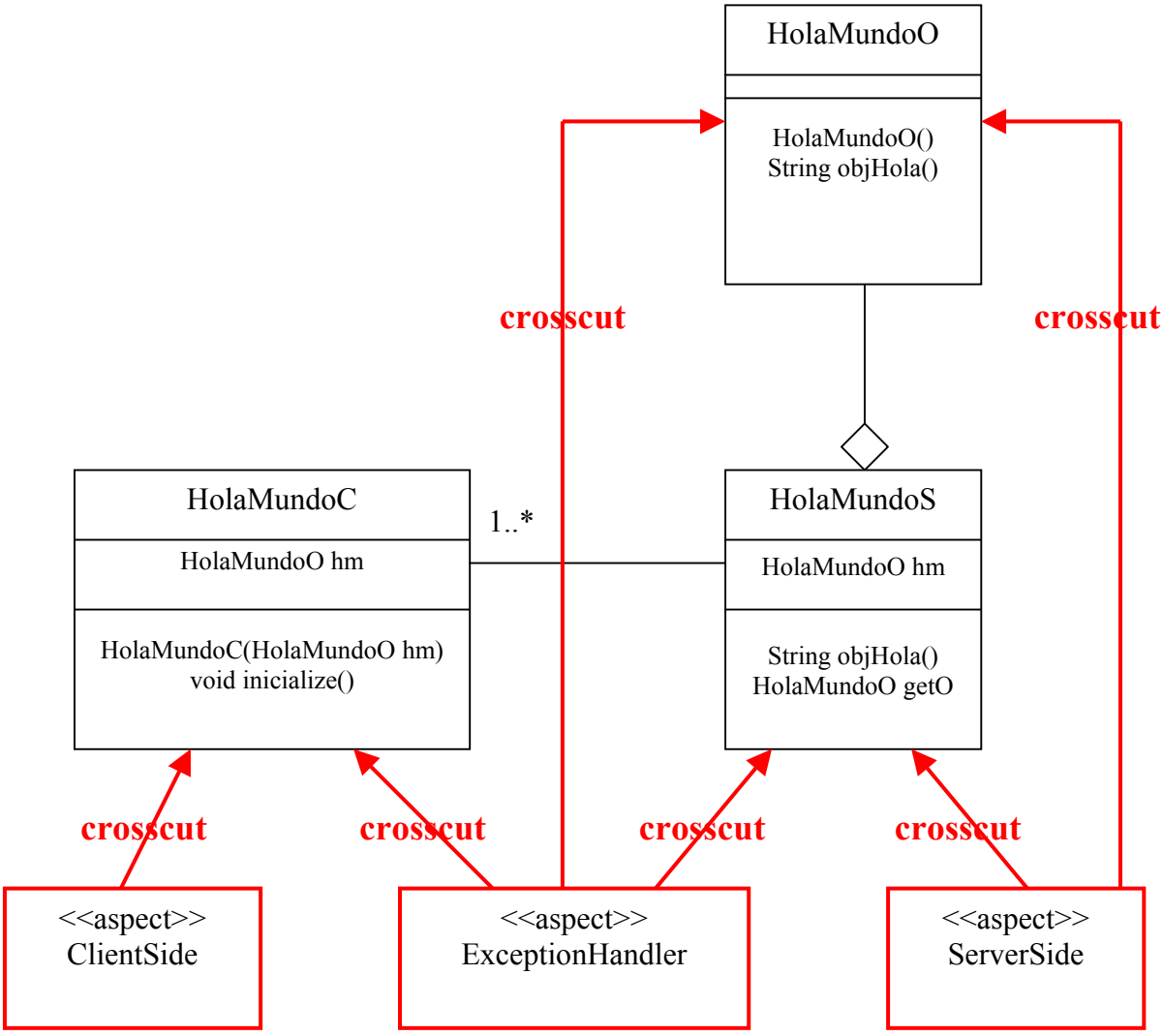


Figura 44: Diagrama de clases de los aspectos.

Anexo B: Requerimientos del Sistema y Herramientas Auxiliares

El entorno de trabajo de este proyecto esta basado en herramientas relacionadas con el mundo de Java, con un par de excepciones. El siguiente software fue utilizado durante la ejecución del proyecto:

- El proyecto ha sido realizado bajo un sistema operativo Windows 98, no siendo imprescindible para la ejecución de tal, se podría haber optado por cualquier otro sistema operativo ya sea cualquier versión de Windows como del sistema operativo Linux. Pero la elección del resto de herramientas utilizadas se pueden ver influenciada por el sistema operativo elegido.

- Una plataforma Java2, J2SDK 1.4.0_01 y la herramienta Kawa IDE pro.

- RMI y todas sus herramientas(como rmiregistry y el compilador rmic) viene incorporado en la plataforma Java2

- AspectJ 1.0.6, esta aplicación se puede encontrar en http://eclipse.org/aspectj/ es de libre distribución.

- Para poder ver los documentos pdf se necesita tener instalada la aplicación Adobe Acrobat Reader, esta aplicación se puede encontrar en la siguiente url: http://www.adobe.es/products/acrobat/readstep.html

- Para poder imprimir un documento en formato pdf es necesario tener instalado la impresora Acrobat PDFWriter, esta aplicación se puede encontrar en la siguiente url: http://www.e-mision.net/crazyhouse/secciones/codigo.asp?i=DOW35 

- http://www.adobe.es/products/acrobat/readstep.html

Anexo C: Otros Lenguajes Orientados a Aspectos

A continuación se describirá con más detalle algunos de los lenguajes orientados a aspectos disponibles actualmente, que fueron mencionados en el apartado 2.2.2 . Excluiremos de este anexo el lenguaje AspectJ ya que este lenguaje ha sido estudiado en profundidad a lo largo del proyecto.

- **JPAL:**

  La principal característica de este lenguaje es que los puntos de unión(join point) son especificados independientemente del lenguaje base. Estos puntos de enlace independientes reciben el nombre de Junction Point (JP). De aquí el nombre de este lenguaje JPAL que significa Junction Point Aspect Language, esto es, Lenguaje de Aspectos basados en Junction Point.

  El uso de programas escritos en JPAL para describir nuevos lenguajes de aspectos facilita para ese lenguaje el desarrollo de tejedores(weavers). De hecho, el weaver de JPAL genera un esquema de aspectos llamado "Esquema del Tejedor". Este esquema tiene un mecanismo que automáticamente conecta el código base con los programas de aspectos en puntos de control llamados "acciones".

  El código que agrega el Esquema del Tejedor invoca, cuando es alcanzado en ejecución, las acciones correspondientes para permitir la ejecución de los programas de aspectos. Esto permite una vinculación dinámica con los programas orientados a aspectos, lo cual hace posible modificar en tiempos de ejecución los programas orientados a aspectos. Sin embargo, esta solución no es lo suficientemente poderosa como para agregar o reemplazar programas orientados a aspectos en ejecución. Por esto causa se agrega al Esquema del Tejedor una entidad llamada Administrador de Programas de Aspectos (APA), la cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una librería dinámica que almacena los aspectos y permite agregar, quitar o modificar aspectos, y mandar mensajes a dichos aspectos de una manera dinámica.

  La comunicación entre el Administrador y el Esquema del Tejedor se logra a través de un Protocolo de Comunicación entre Procesos que permite registrar aspectos dinámicamente.

  La siguiente figura resume la arquitectura JPAL:

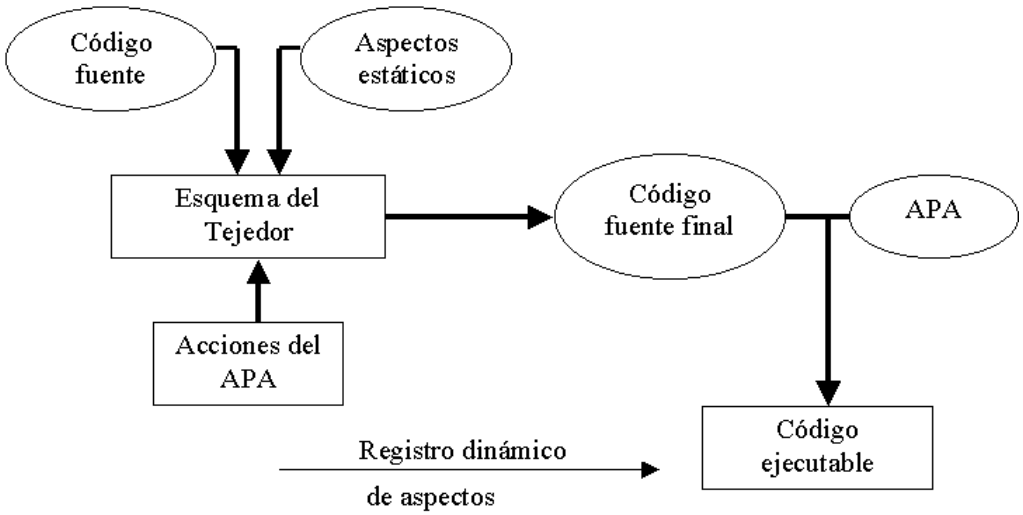


Figura 45: Arquitectura JPAL.

El costo de ejecución de una arquitectura basada en JPAL depende en la implementación del Administrador de Programas de Aspectos que tiene mayor eficiencia en ambientes interpretados.

Resumiendo, JPAL , un descendiente de AspectJ, tiene la particularidad de separar los puntos de unión, que son independientes del lenguaje base, de sus acciones asociadas que dependen de decisiones de implementación. Esta separación permite generar un Esquema de Tejedor para cualquier lenguaje de aspectos. Este esquema ofrece un puente entre el control de la ejecución y la ejecución de la acción. Tomando ventaja de esta redirección se obtiene un modelo de arquitectura para el manejo dinámico de aspectos.

Su principal aplicación es para la implementación de sistemas distribuidos.

- **D**

D es un ambiente de lenguajes de aspectos para la programación distribuida. Se llama ambiente de lenguajes, en vez de solamente lenguaje, porque consiste en realidad de dos lenguajes: COOL, para controlar la sincronización de hilos(threads), y RIDL, para programar la interacción entre componentes remotos. Estos dos lenguajes se diseñaron de manera independiente de un lenguaje componente. Sin embargo establecen un número de condiciones sobre el lenguaje componente.

El diseño de D es semi-independiente del lenguaje componente, ya que D impone requerimientos sobre el lenguaje componente que satisfacen naturalmente los lenguajes orientados a objetos. Luego el lenguaje componente puede ser cualquiera mientras sea orientado a objetos. Por lo tanto, en teoría podría ser implementado con C++, Smalltalk, CLOS, Java o Eiffel. Cristina Lopes, principal diseñadora del lenguaje, implementó D sobre Java llamando al lenguaje resultante DJ.

La sincronización es un requisito que D captura como un aspecto separado. Por lo tanto las clases no pueden tener código para el control de concurrencia. Un programa puede tener varios hilos concurrentes. La estrategia de sincronización por defecto, sin la intervención de COOL, es que no hay estrategia: en la presencia de múltiples hilos todos los métodos de todos los objetos pueden ejecutarse concurrentemente.

La integración remota es el otro requisito que D captura como un aspecto separado. Luego las clases tampoco pueden tener código para la comunicación remota. Un programa sin la intervención de RIDL no es un programa distribuido, la estrategia de comunicación por defecto es que no hay ninguna estrategia de comunicación, entonces un programa es distribuido sólo si utiliza RIDL.

Los módulos de aspectos pueden acceder a los componentes. Este acceso sin embargo está especificado por un protocolo el cual forma parte del concepto de aspecto. Este protocolo puede ser diferente para cada aspecto debido a que diferentes aspectos influyen de maneras diferentes sobre los componentes. En caso de COOL y RIDL los derechos de acceso son idénticos, se les permite inspeccionar al objeto pero no invocar sus métodos o modificar su estado.

No es posible para los módulos de aspectos referir a otro módulo de aspecto, como consecuencia no es posible establecer ninguna relación entre los módulos de aspectos, incluyendo a la herencia.

- **ASPECTC**

AspectC es un simple lenguaje de aspectos de propósito general que extiende C, es un subconjunto de AspectJ sin ningún soporte para la programación orientada a objetos o módulos explícitos.

El código de aspectos, conocido como aviso, interactúa con la funcionalidad básica en los límites de una llamada a una función, y puede ejecutarse antes, después, o durante dicha llamada. Los elementos centrales del lenguaje tienen como objetivo señalar llamadas de funciones particulares, acceder a los parámetros de dichas llamadas, y adherir avisos a ellas.

Los cortes en AspectC tomas las siguientes formas:

- Llamadas a una función: *call(f(arg))*, captura todas las llamadas a la función f con un argumento.
- Durante el flujo de control: *cflow(cualquier corte)*, captura el contexto de ejecución dinámico del corte.
- Referencias a una variable: v*arref(nombre_variable)*, captura las referencias a la variable nombre_variable.

- Todos los cortes se pueden describir utilizando expresiones lógicas, aumentando la expresividad del lenguaje: el operador “y”(&&), el operador “o” (||), y el operador de negación(!). Un ejemplo sería: call(f(arg)) || call(h(arg)), con lo cual se captura las llamadas a la función f o las llamadas a la función g.

Como el lenguaje C es de naturaleza estática, el tejedor de AspectC es estático.

Es interesante contar con esta herramienta de aspectos basada en C, debido a que C es utilizado en numerosas aplicaciones, en especial, aquellas donde la eficiencia es primordial, como por ejemplo, la implementación de sistemas operativos. Al existir también conceptos entrecruzados en la implementación de sistemas operativos, AspectC pasa a ser la herramienta ideal para tal desarrollo. Un trabajo a mencionar sería el proyecto *a-kernel*, cuya meta es determinar si la programación orientada a aspectos puede optimizar la modularidad de los sistemas operativos, y reducir así la complejidad y fragilidad asociada con la implementación de los mismos.

- **ASPECTS**

AspectS extiende el ambiente Squeak/Smalltalk para permitir un sistema de desarrollo orientado a aspectos. Squeak es una implementación abierta y portable de Smalltalk-80 cuya máquina virtual está completamente escrita en Smalltalk, para mayor referencia visitar su página [38]. Principalmente, AspectS está basado en dos proyectos anteriores: AspectJ de Xerox Parc y el MethodWrappers de John Brant, que es un mecanismo poderoso para agregar comportamiento a un método compilado en Squeak.

AspectS, un lenguaje de aspectos de propósito general, utiliza el modelo de lenguaje de AspectJ y ayuda a descubrir la relación que hay entre los aspectos y los ambientes dinámicos. Soporta programación en un metanivel, manejando el fenómeno de *Código Mezclado* a través de módulos de aspectos relacionados. Está implementado en Squeak sin cambiar la sintaxis, ni la máquina virtual.

En este lenguaje los aspectos se implementan a través de clases y sus instancias actúan como un objeto, respetando el principio de uniformidad. Un aspecto puede contener un conjunto de receptores, transmisores o clases transmisoras. Estos objetos se agregan o se remueven por el cliente y serán usados por el proceso de tejer en ejecución para determinar si el comportamiento debe activarse o no.

En Squeak, la interacción de los objetos está basada en el paradigma de pasaje de mensajes. Luego, en AspectS los puntos de enlace se implementan a través de envíos de mensajes. Los avisos asocian fragmentos de código de un aspecto con cortes y sus respectivos puntos de enlace, como los objetivos del tejedor para ubicar estos fragmentos en el sistema. Para representar esos fragmentos de código se utilizan bloques(instancias de la clase BlockContext).

Los tipos de avisos definibles en AspectS son:

- Antes y después de la invocación a un método (*AsBeforeAfterAdvice*).
- Para manejo de excepciones(*AsHandlerAdvice*).
- Durante la invocación de un método(*AsAroundAdvice*).

Un calificador de avisos (*AsAdviceQualifier*) es usado para controlar la selección del aviso apropiado. Es similar al concepto de designadores de cortes de AspectJ.

Utiliza un tejedor dinámico que transforma el sistema base de acuerdo a lo especificado en los aspectos. El código tejido se basa en el MethodWrapper y la

metaprogramación. MethodWrapper es un mecanismo que permite introducir código que es ejecutado antes, después o durante la ejecución de un método.

El proceso de tejer sucede cada vez que una instancia de aspectos es instalada. Para revertir los efectos de un aspecto al sistema, el aspecto debe ser desinstalado. A este proceso se lo conoce como "destejer", del inglés unweaving. El tejido de AspectS es completamente dinámico ya que ocurre en ejecución.

- **ASPECTC++**

AspectC++ es un lenguaje de aspectos de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. En este lenguaje los puntos de enlace son puntos en el código componente donde los aspectos pueden interferir. Los puntos de unión son capaces de referir a código, tipos, objetos, y flujos de control.

Las expresiones de corte son utilizadas para identificar un conjunto de puntos de unión. Se componen a partir de los designadores de corte y un conjunto de operadores algebraicos. La declaración de los avisos es utilizada para especificar código que debe ejecutarse en los puntos de enlace determinados por la expresión de corte.

La información del contexto del punto de enlace puede exponerse mediante cortes con argumentos y expresiones que contienen identificadores en vez de nombres de tipos, todas las veces que se necesite.

Diferentes tipos de aviso pueden ser declarados, permitiendo que el aspecto introduzca comportamiento en diferentes momentos: el aviso después (after advice), el aviso antes (before advice) y el aviso durante (around advice).

Los aspectos en AspectC++ implementan en forma modular los conceptos entrecruzados y son extensiones del concepto de clase en C++. Además de atributos y métodos, los aspectos pueden contener declaraciones de avisos. Los aspectos pueden derivarse de clases y aspectos, pero no es posible derivar una clase de un aspecto.

La arquitectura del compilador de AspectC++ sigue las siguientes pautas:

Primero el código fuente de AspectC++ es analizado léxica, sintáctica y semánticamente. Luego se ingresa a la etapa de planificación, donde las expresiones de corte son evaluadas y se calculan los conjuntos de puntos de enlace. Un plan para el tejedor es creado conteniendo los puntos de enlace y las operaciones a realizar en estos puntos. El tejedor es ahora responsable de transformar el plan en comandos de manipulación concretos basados en el árbol sintáctico de C++ generado por el analizador PUMA. A continuación el manipulador realiza los cambios necesarios en el código. La salida del manipulador es código fuente en C++, con el código de aspectos "tejido" dentro de él. Este código no contiene constructores del lenguaje AspectC++ y por lo tanto puede ser convertido en código ejecutable usando un compilador tradicional de C++.

- **HYPERJ**

La aproximación por Ossher y Tarr sobre la separación multidimensional de conceptos (MDSOC en inglés) es llamada hyperspaces, y como soporte se construyó la herramienta HyperJ en Java.

Para analizar con mayor profundidad HyperJ es necesario introducir primero cierta terminología relativa a MDSOC :

- Un *espacio de concepto* concentra todas las unidades, es decir todos los constructores sintácticos del lenguaje, en un cuerpo de software, como una librería. Organiza las unidades en ese cuerpo de software para separar todos los conceptos importantes, describe las interrelaciones entre los conceptos e indica cómo los componentes del software y el resto del sistema pueden construirse a partir de las unidades que especifican los conceptos.

- En HyperJ un *hiperespacio* (hyperspace) es un *espacio de concepto* especialmente estructurado para soportar la múltiple separación de conceptos. Su principal característica es que sus unidades se organizan en una matriz multidimensional donde cada eje representa una dimensión de concepto y cada punto en el eje es un concepto en esa dimensión.

- Los *hiperslices* son bloques constructores; pueden integrarse para formar un bloque constructor más grande y eventualmente un sistema completo.

- Un *hipermódulo* consiste de un conjunto de *hiperslices* y conjunto de reglas de integración, las cuales especifican cómo los *hiperslices* se relacionan entre ellos y cómo deben integrarse.

Una vez introducida la terminología se puede continuar con el análisis de HyperJ. Esta herramienta permite componer un conjunto de modelos separados, donde cada uno encapsula un concepto definiendo e implementando una jerarquía de clases apropiada para ese concepto. Generalmente los modelos se superponen y pueden o no referenciarse entre ellos. Cada modelo debe entenderse por sí solo. Cualquier modelo puede aumentar su comportamiento componiéndose con otro: HyperJ no exige una jerarquía base distinguida y no diferencia entre clases y aspectos, permitiendo así que los *hiperslices* puedan extenderse, adaptarse o integrarse mutuamente cuando se lo necesite. Esto demuestra un mayor nivel de expresividad para la descripción de aspectos en comparación con las herramientas descriptas anteriormente.

Existe una versión prototipo de HyperJ que brinda un marco visual para la creación y modificación de las relaciones de composición, permitiendo un proceso simple de prueba y error para la integración de conceptos en el sistema. El usuario comienza especificando un *hipermódulo* eligiendo un conjunto de conceptos y un conjunto tentativo de reglas de integración. HyperJ crea *hiperslices* válidos para esos conceptos y los compone basándose en las reglas que recibe. Luego el *hiperslice* resultante se muestra en pantalla, si el usuario no está conforme puede en ese momento introducir nuevas reglas o modificar las reglas existentes y así continuar este proceso de refinación hasta obtener el modelo deseado.

Anexo D: Opciones Aceptadas por el Compilador ajc.

ajc acepta las siguientes opciones (*options*):

-injars *JarList* :
Permite como bytecode de entrada cualquier archivo .class que se encuentre dentro del archivo .jar especificado. Estas clases estarán a la salida, pero posiblemente entrelazadas con los aspectos aplicados. JarList al igual que el classpath, es un argumento que contiene una lista de paths de los ficheros jar.

-aspectpath *JarList* :
Teje los aspectos binarios del fichero zip JarList en todo los archivos fuentes.

-argfile *File:*
El fichero es una lista de argumentos. Estos argumentos son insertados en el argumento *list*

-outjar *output.jar* :
Pone las clases de salida en un fichero zip output.jar.

-incremental:
Ejecución continua del compilador. Después de la compilación inicial, el compilador espera a leer una nueva línea por la entrada estándar para volver a compilar y no termina hasta que no lea una 'q' por la entrada estándar. De esta manera se asegura que solo se vuelve a compilar los elementos necesarios, haciendo mucho más rápida la segunda compilación.

-sourceroots *DirPaths* :
Busca y establece todos los ficheros fuente .java o .aj en un directorio perteneciente a DirPaths. DirPaths es un argumento que contiene un listado de paths de directorios.

-emacssym :
Genera un archivo .ajesym compatible para emacs.

-Xlint:{level}
Añade un nivel por defecto para los mensajes sobre errores del código que será generado en un crosscutting. {level} puede elegirse en: ignorar, warning o error. Esto elimina entradas en org/aspectj/weaver/XlintDefault.properties de aspectjtools.jar, pero no elimina los niveles fijados usando - la opción de Xlintfile.

-Xlintfile *PropertyFile*
Especifica las propiedades del archivo para fijar un nivel de especificación de mensajes de los crosscutting. PropertyFile es un path para Java .

-help
Muestra información sobre el uso y opciones del compilador ajc.

-version
Muestra la versión del compilador de AspectJ.

-classpath *Path*

Especifica a los usuarios donde buscar los archivos class. Path es solo un argumento que contiene una lista con las rutas de los archivos zip o directorios delimitados por la phath especifico de la plataforma.

-bootclasspath *Path*

Elimina la localización de bootclasspath de la máquina virtual con el propósito de evaluar los tipos al compilar. Path es sólo un argumento que contiene una lista de path de archivos zip o directorios delimitados por el path especifico de la plataforma.

-extdirs *Path*

Elimina la localización de los directorios de extensión de la máquina virtual con el propósito de evaluar los tipos al compilar. Path es sólo un argumento que contiene una lista de path de archivos zip o directorios delimitados por el path especifico de la plataforma.

-d *Directory*

Especifica donde colocar los archivos .class generados, si no se especifica se tomara como directorio el directorio actual de trabajo.

-target *[1.1|1.2]*

Especifica el destino donde poner el classfile. (1.1 o 1.2, por defecto es 1.1)

-1.3

Fija el nivel a 1.3 (por defecto)

-1.4

Fija el nivel a 1.4.

-source *[1.3|1.4]*

Toggle assertions (1.3 o 1.4, por defecto es 1.3 en modo-1.3 y 1.4 en modo -1.4). Cuando usa -1.3, en assert() en Java 1.4 dará un error en compilación. Cuando utiliza-1.4, debe confirmarse una clave para implementar la assertion de acuerdo con el lenguaje 1.4.

-nowarn

No emite ningún warning (equivalente a '-warn:none'). Esto no suprime los mensajes generados por una declaración de warning o Xlint.

-warn: *items*

Emite warnings para cualquier instancia delimitada por la lista de código dudoso (ejem '-warn:unusedLocals,deprecation'):

ConstructorName	método con el nombre del constructor
packageDefaultMethod	sobrescribir un método de un paquete por defecto.
deprecation	uso de tipos o elementos deprecated
maskedCatchBlocks	ocultar el bloque catch
unusedLocals	variable local no utilizada

unusedArguments	argumento de un método no utilizado
unusedImports	código del import no utilizado en el archivo
none	suprime todos los warnings de compilación

-warn:no suprime los warnings generados por una declaración o por Xlint.

-deprecation

Hace lo mismo que –warn: deprecation.

-noImportError

No emite ningún error para los imports sin resolver.

-proceedOnError

Guarda después de compilar un error, descarga las clases con métodos afectados.

-g*:[lines,vars,source]*

debug de los atributos de nivel, esto se puede hacer de tres formas:

-g	debug de toda la información ('-g:lines,vars,source')
-g:none	no hace debug de ninguna información
-g:{items}	debug de alguna o toda la información [lines, vars, source], (‘-g:lines,source’).

-preserveAllLocals

Protege todas las variables locales durante la generación del código(facilita el debugging)

-referenceInfo

Genera información de referencia.

-encoding *format*

Especifica el formato de codificación por defecto del código fuente. Especifica codificación custom para archivos básicos añadiendo como sufijo el nombre del código fuente file/fólder con ‘[encoding]’.

-verbose

Emite mensajes accediendo/procesando de las unidades de compilación.

-log *file*

Especifica el fichero log para los mensajes de compilación.

-progress

Muestra el progreso (requiere el modo -log).

-time

Muestra información sobre el tiempo transcurrido.

-noExit

No llama a System.exit(n) al final de la compilación (n=0 si no error)

-repeat *N*

Repite el proceso de compilación N periodos (típico durante el funcionamiento del análisis).

-Xnoweave

(Experimental) produce los archivos .class tejidos para crear los -injars.

-Xnoinline

(Experimental) do not inline around advice

-XincrementalFile *file*

(Experimental) Trabaja de forma incrementas, se usa con un archivo de entrada que controla el compilador. El compilador se pone en marcha cuando el archivo es modificado y se para cuando se suprime dicho archivo.

-XserializableAspects

(Experimental) Normalmente declarar un aspecto serializable es un error, esta opción quita esta restricción.

Referencias

[1] Libro: H.M Deitel, P.J. Deitel. Java how to program. Prentice Hall. 2002.

[2] Libro: William Grosso. Java RMI: designing & building distributed applications. O'Reilly & Associates. 2001.

[3] Libro: C.T. Arrington. Enterprise Java with UML. JohnWiley & Sons. 2001.

[4] Libro: Ivan Kiselev. Aspect-Oriented Programming with AspectJ.Sams.2002

[5] Reina A.M. *Visión General de la Programación Orientada a Aspectos.* Informe Técnico del Departamento de Lenguajes y Sistemas Informáticos de la Facultad de Informática y Estadística de la Universidad de Sevilla, Diciembre 2000. http://www.lsi.us.es/~informes/aopv3.pdf

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programmin*g, Xerox Palo Alto Research Center, 1997

[7] G. Kiczales, C. Lopes. *Aspect-Oriented Programming with AspectJ TM –Tutorial-*.Xerox Parc. http://www.aspectj.org.

[8] *AspectJ TM* : *User's Guide.* Xerox Parc Corporation. http://www.aspectj.org

[9] Coady, Y., G. Kiczales, and M. Feeley. *Exploring an Aspect-Oriented Approach to Operating System Code.* In: Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Minneapolis, Minnesota, USA (2000)

[10] Clemente P., Hernández J., Sánchez F . *Reutilización y adaptabilidad en componentes distribuidos: ¿es la programación orientada a aspectos una posible solución?* IScDIS'2000. 1er Taller de Trabajo en Ingeniería del Software basada en Componentes Distribuidos, Noviembre 2000. http://tdg.lsi.us.es/~sit02/res/papers/clemente.html

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J, Irwin, J, *Aspect-Oriented Programming.* In Proc. ECOOP'97 (Finland, June 1997) Springer-Verlag http://trese.cs.utwente.nl/aop-ecoop99/aop97.html

[12] Sergio Soares and Paulo Borba. *PaDA: A Pattern for Distribution Aspects.* In Second Latin American Conference on Pattern Languages Programming | SugarLoafPLoP, Itaipava, Rio de Janeiro, Brazil, August 2002. http://www.cin.ufpe.br/~scbs/artigos/soares_borba_sugarloafplop2002.pdf

[13] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications. ACM Press, November 2002. To appear. http://oopsla.acm.org/

[14] Página principal de AspectJ =http://www.aspectj.org visitada 20/11/2002

[15] Página principal de AspectJ= http://www.eclipse.com visitada 26/06/2003

[16] Página del proyecto AspectJ= http://www.parc.xerox.com/spl/projects/aop/aspectj/