INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Campus Ciudad de México

Escuela de Graduados en Ingeniería y Arquitectura

Maestría en Ciencias Computacionales

"Manejo de excepciones de Java-RMI de forma declarativa y visto como un aspecto"

TESIS PRESENTADA POR

Héctor Ernesto Cisneros Castañeda

PARA OBTENER EL GRADO DE MAESTRO
EN CIENCIAS COMPUTACIONALES

ASESOR
Dr. Bárbaro Jorge Ferro Castro
DIRECTOR DE TESIS

México D.F., agosto del 2005

Resumen

Se describe un método de diseño y una herramienta que lo soporta que ayuda al entendimiento y la utilización del paradigma de la programación orientada a aspectos y la programación declarativa para el manejo de excepciones en un sistema distribuido con Java RMI.

El lenguaje a utilizar para la programación orientada a aspectos es AspectJ, que es una extensión de Java.

Se pretende que la herramienta ayude a poder establecer, de manera fácil y clara la forma en que los elementos del sistema cooperan entre sí, esto último visto con el paradigma de la orientación a aspectos; y así conseguir la separación de conceptos (cada cosa esté en su sitio) y minimizar las dependencias entre ellos (reducción del acoplamiento entre los elementos); lo que conduce a tener código menos enmarañado, más natural y más reducido; mayor facilidad para razonar sobre los conceptos; más facilidad para depurar y hacer modificaciones en el código; código reutilizable, que se puede acoplar y desacoplar cuando sea necesario.

Contenido

Lista de Tablas

Lista de Figuras

CAPÍTULO 1

1. Introducción

1.1 Antecedentes

Para tomar la decisión de qué línea de investigación y desarrollo tendría esta tesis se consideraron varios puntos de interés, entre ellos la participación en la Cátedra de Sistemas Distribuidos Colaborativos para la Enseñanza, cátedra que se desarrolla en conjunto por profesores del ITESM Campus Ciudad de México y del Campus Cuernavaca. Al recopilar información de la situación existente de los sistemas distribuidos colaborativos se determinó que era muy importante:

- Establecer las tendencias actuales en el uso y aprovechamiento de las aplicaciones distribuidas colaborativas.
- Dirigir esfuerzos hacia el aprovechamiento de los sistemas distribuidos colaborativos en la enseñanza:
    - Desde un enfoque integral.
    - Con la ayuda de nuevas tecnologías.

En este contexto, y atendiendo a trabajos anteriores que se han realizado por profesores y estudiantes que colaboran en esta cátedra, la investigación que se aborda en esta tesis es la relacionada con el diseño de aplicaciones distribuidas usando la infraestructura de Java RMI, en particular la del tratamiento de las excepciones que se generan en un entorno de esta naturaleza.

Para abordar este tema, se parte de la hipótesis que la programación orientada a aspectos [1] y la programación declarativa [33] contribuyen a lograr diseños eficientes de calidad para el manejo de excepciones en programas, y en particular en las partes que componen un sistema distribuido.

1.2 Definición del problema

Aunque la investigación científica y tecnológica ha tenido un crecimiento importante, no se puede afirmar que los avances tecnológicos hayan generado una transformación en los

sistemas de enseñanza [20]. Algunas instituciones de educación se crearon con el propósito primordial de buscar soluciones a los problemas que afectan el ámbito de la educación, pero tal propósito no se ha llevado a cabo, principalmente por la falta de compromiso por parte de la administración educativa que facilite la incorporación del sector educativo al mundo de los avances científicos y tecnológicos; que es el objetivo principal del ***trabajo cooperativo soportado en computadora (CSCW, por sus siglas en ingles)*** y de la enseñanza a partir de una visión conceptual de la educación y la influencia de las nuevas tecnologías [20].

En lo que respecta a la evolución de la ingeniería de software, la cual es constante, ha surgido una nueva forma de descomponer los sistemas llamada **orientación a aspectos (OA)** [1], en la que para el desarrollo de software se consideran no sólo el diseño y la implementación de la funcionalidad, sino también conceptos tales como la sincronización, la distribución, el manejo de errores, la optimización del manejo de memoria y seguridad, entre otros, que brindan sus servicios a los componentes funcionales.

La **programación orientada a aspectos (AOP, por sus siglas en ingles)** es una metodología que intenta soportar la separación de competencia de los diferentes aspectos o conceptos teniendo mecanismos para abstraerlos y componerlos para formar todo el sistema de forma eficiente y fácil de entender [1]. En el Apéndice A se da una visión general de la AOP.

En lo que respecta a la **programación declarativa**, podemos mencionar inicialmente que está formada por dos estilos de programación, que son: la programación lógica y la programación funcional; en las que el programa es una simple especificación formal del problema a resolver. Es importante tener en mente que la programación *imperativa* (programación tradicional) se basa en la resolución de problemas mediante algoritmos, en donde se especifica el modo concreto de resolver un problema (en otras palabras nos interesa **el *cómo***), mientras que la programación declarativa se basa en lo que se quiere resolver, en otras palabras en **el *qué* y no en el *cómo*** [17].

El problema que se plantea en esta tesis es el manejo de excepciones remotas en Java (RMI) visto como un aspecto y definiéndolo de forma declarativa para evitar código *mezclado* y *código diseminado* [15], entendiendo por estos dos términos lo siguiente:

- El ***Código Mezclado*** se presenta cuando en un mismo módulo de un sistema de software conviven simultáneamente más de un requerimiento. Esto hace que en el modelo existan elementos de implementación de más de un requerimiento.
- El ***Código Diseminado*** se produce cuando un requerimiento está esparcido sobre varios módulos. Por lo tanto, la implementación de dicho requerimiento también queda diseminada sobre esos módulos.

1.3 Objetivos

1.3.1 Objetivo principal

El objetivo principal de esta investigación es resolver el problema que se menciona con anterioridad y que a continuación se delimita:

Proponer un método de diseño que utilice la programación orientada a aspectos y la programación declarativa para el diseño del manejo de excepciones en un sistema distribuido basado en Java RMI, y facilite el entendimiento de las técnicas mencionadas para la solución de otros problemas. El método se ilustra con una herramienta prototipo que permite aplicar aspectos y atributos para diseñar el manejo de excepciones en RMI.

La aplicación permitirá manejar las excepciones de Java RMI de forma declarativa y considerando a las excepciones como aspectos. En el Apéndice C se presenta una tabla con las diferentes excepciones RMI y una breve descripción de todas ellas.

El lenguaje a utilizar para la programación orientada a aspectos es AspectJ [6], del cual se da una reseña en el Apéndice B.

La arquitectura base se muestra en la siguiente figura:

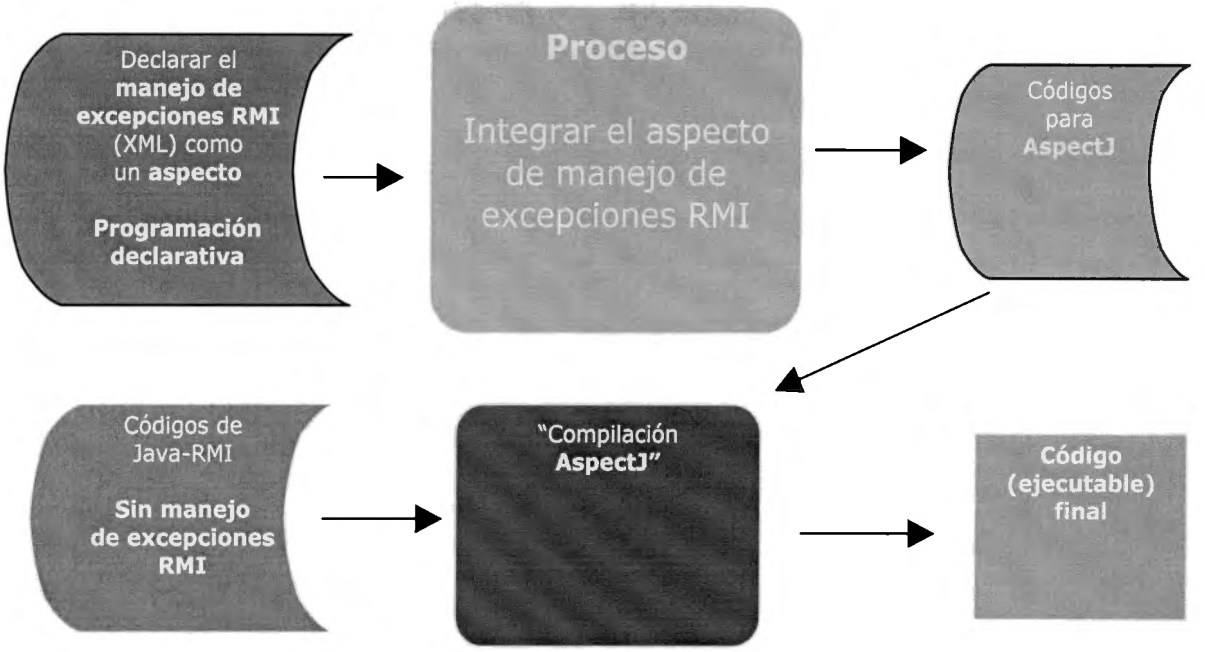

Figura 1.1 Arquitectura base

La Figura 1.1 resalta dos de los problemas principales para el diseño y construcción de programas; el de la separación de conceptos y el de la minimización de dependencias.

Al separar los conceptos se consigue que cada cosa esté en su sitio (que cada decisión se tome en un lugar concreto) y al minimizar las dependencias, se logra tener acoplamiento débil entre los distintos elementos. Los cuales son atributos de calidad importantes que entre otras cosas logran:

- Mayor facilidad para razonar sobre los conceptos, ya que están separados y con dependencia mínima.
- Más facilidad para depurar y hacer modificaciones en el código.
- Mínimo impacto sobre los módulos del sistema como resultado de modificaciones en algún concepto.
- Alta reutilización de código que se puede acoplar y desacoplar cuando sea necesario.

El manejo de excepciones de forma declarativa es un mecanismo que permite ahorrar tiempo a los programadores, durante el desarrollo y mantenimiento; por lo que debe hacerse un esfuerzo serio para aprovechar el manejo de excepciones de esta forma.

1.3.2 Objetivos secundarios

- Brindar un medio para conocer de una manera sencilla el paradigma de la programación orientada a aspectos.
- Brindar una serie de ejemplos para el estudiante que se inicia en el ámbito de la programación orientada a aspectos, en particular con la utilización de AspectJ.
- Dar un primer acercamiento al estudiante a lo que es la programación declarativa.
- Apoyar al mejor entendimiento del manejo de excepciones en Java y en particular a las excepciones de Java RMI.
- Ayudar al desarrollo de aplicaciones con Java RMI.

1.4 Justificación

Al resolver el problema mencionado, se tendrá una herramienta para el desarrollo de sistemas de software, y no sólo se podrá utilizar con fines educativos, sino también en el mundo de la arquitectura de software con la utilización de la orientación a aspectos y la programación declarativa.

1.5 Hipótesis

En base a los diferentes modelos de vistas que se utilizan en la arquitectura de software, y con ayuda de una vista orientada a aspectos se puede obtener un método de diseño auxiliado por una herramienta que permita, de forma declarativa, indicar la manera en que se deben manejar las excepciones remotas de Java, especificando las excepciones como aspectos separados de la lógica de la aplicación objetivo.

1.6 Metodología

La investigación se desarrolló primeramente considerando que en el aprendizaje colaborativo las redes de computadoras proporcionan un mecanismo para el desarrollo del aprendizaje cooperativo entre estudiantes geográficamente dispersos y hacen posible el diseño de programas educacionales para trabajar conjuntamente [13]. Se determinó la

forma en que los aspectos ayudan en el desarrollo de software, desde la fase de diseño de las aplicaciones, en lo cual incluimos el entender los conceptos que se manejan en la programación orientada a aspectos.

Posteriormente el enfoque se dió específicamente hacia AspectJ (ver apéndice B), para lo cual es importante diferenciar dos partes: la especificación del lenguaje y su implementación. La primera, define el lenguaje en el que se escribe el código; en AspectJ la funcionalidad de los conceptos encapsulados se escribe en Java estándar, y las extensiones del lenguaje se usan para definir dónde y cómo han de entrelazarse con el código de nuestra aplicación. La segunda es la implementación del lenguaje, provista de herramientas para compilar, depurar e integrar AspectJ con conocidos entornos de desarrollo integrados (*"integrated development environment"*, IDE) [6] como JBuilder, Eclipse y Emacs.

Por último, reuniendo todos los puntos investigados y estudiados se desarrolló la herramienta que permite el manejo de excepciones RMI de forma declarativa y tratadas como aspectos dentro del sistema. Para probar la herramienta se utilizaron sistemas distribuidos colaborativos desarrollados en materias impartidas en el campus Ciudad de México, de nivel licenciatura y de maestría.

En el punto 6.3, Conclusiones de la investigación, se mencionan varias ventajas que se observaron con el uso del paradigma de la programación orientada a aspectos y de la programación declarativa:

- En relación al código: menos enmarañado y más natural; lo que facilita a razonar sobre los conceptos.
- Al tiempo de desarrollo y mantenimiento: permite ahorrar tiempo dado que las modificaciones tienen un impacto mínimo en los sistemas en general.

CAPÍTULO 2

2. Marco Teórico

A continuación se describe una serie de conceptos que se están relacionados en la investigación y desarrollo de la herramienta propuesta; estos conceptos van desde las tecnologías de programación, los sistemas de enseñanza, arquitectura de software, tecnologías de la información, fallas en el software, patones de diseño y programación orientada a aspectos.

2.1 Evolución de las tecnologías de programación

La evolución de la ingeniería de software y en particular de las tecnologías de programación [15], se ha dado gracias a un principio fundamental para resolver problemas, el de dividir el problema en sub-problemas (es decir divide y vencerás) conocida como **descomposición funcional**. Después de la descomposición funcional, se dio la **programación orientada a objetos (OOP, por sus siglas en ingles)**, que ha permitido desarrollar sistemas muy complejos [4]. Este modelo de objetos se ajusta mejor a los problemas del dominio real que la descomposición funcional. La OOP facilita la integración de nuevos datos aunque quedan las funciones esparcidas por todo el código, lo que da como desventaja, que en ocasiones para integrar una nueva funcionalidad, eventualmente hay que modificar varios objetos [15]; debido ha que en este paradigma la concepción de las funciones o servicios se realiza mediante la interacción de objetos, en otras palabras, a través de la OOP se logra un diseño y una implementación que satisface la funcionalidad básica, y con una calidad aceptable; sin embargo, existen conceptos que no pueden encapsularse dentro de una unidad funcional, debido a que atraviesan todo el sistema, o varias partes de él. Algunos de estos conceptos son: sincronización, manejo de memoria, distribución, chequeo de errores, seguridad o redes, entre otros.

La necesidad de desarrollar software de calidad ha incrementado el uso de la orientación a objetos, buscando aumentar la reusabilidad y facilitar un mejor mantenimiento, incrementando la productividad en el desarrollo y el soporte a cambios en los requerimientos [15]. Pero el paradigma de orientación a objetos presenta algunas

limitaciones como se ha mencionado en el párrafo anterior. Algunas de estas limitaciones pueden ser compensadas por el uso de patrones y la forma en que se implementen los métodos.

Por oro lado, extensiones al paradigma orientado a objetos [4], como la programación orientada a aspectos, composición de filtros, programación orientada a sujeto y la programación adaptativa, son nuevas técnicas de programación que tratan de resolver las limitaciones de la orientación a objetos.

Para el desarrollo de software se consideran no sólo el diseño y la implementación de la funcionalidad, se consideran también aspectos tales como la sincronización, la distribución, el manejo de errores, la optimización del manejo de memoria y seguridad, entre otros. Son servicios esparcidos horizontalmente usados por los componentes funcionales.

Ni la programación orientada a objetos, ni la descomposición funcional son suficientes para atacar de manera eficiente este tipo de "problemas" (estos aspectos [3]); es decir estas técnicas no soportan la separación de competencias para aspectos ortogonales a la funcionalidad básica del sistema, lo que impacta de manera negativa en la calidad del software.

Una metodología que intenta soportar la separación de competencia de los diferentes aspectos es la **programación orientada a aspectos (AOP)**; que trata de separar los componentes y los aspectos, teniendo mecanismos para abstraerlos y componerlos para formar todo el sistema [2].

2.2 Arquitectura de software

Las nuevas tecnologías están cambiando todos los ámbitos de nuestras vidas afectando la forma en que hacemos las cosas: trabajar, divertirnos, relacionarnos, aprender y sutilmente nuestra forma de pensar [24].

Para poder hablar de una vista arquitectónica del manejo de excepciones como un aspecto, a continuación se presentan los conceptos relacionados a las nuevas tecnologías y a la arquitectura de software.

2.2.1 Tecnologías de la información

Las tecnologías de la información y la comunicación han desempeñado un papel fundamental en la configuración de la sociedad y la cultura, sólo percibimos la tecnología cuando falla o por algún motivo desaparece (huelga de transporte, corte de suministro eléctrico, etc.), o cuando es suficientemente nueva, ya que generalmente los cambios generan incertidumbre y ponen en peligro intereses creados [11].

2.2.1.1 Tendencias de la educación en la sociedad de las tecnologías de la información

Generalmente el enfoque que se le da al tema de las nuevas tecnologías de la información y la educación se refiere a aspectos didácticos considerándola como un medio más entre los tantos recursos con los que puede contar el docente y no se mira los cambios que se producen en el mundo para el que se está educando a niños y jóvenes [11]. Por otra parte las posibilidades que se abren paso gracias a las nuevas tecnologías de la información se pueden materializar dependiendo de decisiones políticas y de compromisos institucionales más que de los avances tecnológicos o de los medios disponibles, es decir, se podría generar una reingeniería del proceso educativo.

2.2.1.2 Evolución de las nuevas tecnologías de la información y la comunicación

La idea principal a tener en mente es que los cambios tecnológicos han dado lugar a cambios radicales en la organización, en las prácticas y formas de organización social y en la propia condición humana, esencialmente en la subjetividad y la formación de la identidad.

Las aplicaciones analógicas rápidamente migran a la digitalización y adquieren capacidades interactivas emisor-receptor para el procesamiento y manipulación de la información y aparecen nuevos tipos de materiales: multimedia, hipermedia, simulaciones, documentos dinámicos, etc. [23]. Los satélites de comunicación y las redes terrestres de alta capacidad permiten enviar y recibir información desde cualquier lugar de la tierra. Siendo así las cosas, con frecuencia olvidamos que la tecnología no sólo tiene

implicaciones sociales, sino que también es producto de las condiciones sociales y económicas de una época y país, por eso el contexto histórico permite explicar su éxito o fracaso frente a otras tecnologías y las condiciones de su generalización, es decir que sólo se puede comprender un cambio dentro del contexto de la estructura social en la que se produce.

Entre los principales desafíos que se deben enfrentar por la comunidad de la propiedad intelectual en el siglo XXI se encuentran los relacionados con el progreso constante de las tecnologías digitales, el auge del intercambio electrónico de información y la brecha tecnológica entre los países desarrollados y los países en desarrollo [25]. En Internet se está dando una explosión de contenidos comerciales, los protocolos de comunicación se están generando gracias a las instituciones que financian e impulsan la investigación, nos encontramos en un período en el que el uso comercial de las redes informáticas está produciendo la investigación en aspectos que anteriormente eran poco relevantes como la seguridad en las transacciones electrónicas, todos estos avances tecnológicos tienen lugar dentro de un marco socioeconómico determinado que posibilita su desarrollo en centros de investigación y universidades, además de la transferencia a la sociedad y su aplicación a la producción.

2.2.1.3 Las nuevas tecnologías de la información y la comunicación

Las nuevas tecnologías de la información y la comunicación se pueden definir como el conjunto de procesos y productos derivados de las nuevas herramientas de hardware y software, soportes de la información y canales de comunicación, relacionados con el almacenamiento, procesamiento, transmisión y digitalización de la información. Dentro de las características distintivas de las nuevas tecnologías de la información están la inmaterialidad, interactividad, instantaneidad, innovación, elevados parámetros de calidad de imagen y sonido, digitalización, mayor influencia sobre los procesos que sobre los productos, automatización interconexión y diversidad. *Los medios de comunicación demandan "acceso universal" a las tradicionales y nuevas tecnologías de comunicación* [26].

El paradigma de las nuevas tecnologías de comunicación lo constituyen las redes informáticas, las computadoras aisladas ofrecen cantidad de posibilidades, pero conectadas

entre sí, incrementan su funcionalidad en varios órdenes de magnitud. Las redes no sólo sirven para la comunicación entre los diferentes sitios que contienen la información almacenada en soportes físicos en cualquier formato digital, sino como herramienta para acceder a la información, a recursos y a servicios prestados por computadoras remotas, como sistema de publicación y difusión de información y como medio de información entre seres humanos [26]. El ejemplo por excelencia lo constituye Internet que es una maqueta a escala de la futura infraestructura de las comunicaciones que integrará sistemas de los que hoy se dispone (televisión, teléfono, etc.), ampliando sus posibilidades a sistemas que hoy se utilizan experimentalmente (como por ejemplo videoconferencia), y algunos que apenas imaginamos.

Es habitual la confusión entre información y conocimiento, el conocimiento implica información interiorizada adecuadamente integrada en las estructuras cognitivas, es personal e intransferible [27]. Jesse H. Shera dijo en 1960 que la información "*es el insumo del conocimiento*", que siempre es recibida en el proceso de la comunicación a través de los sentidos, independientemente del número de artefactos que intervengan entre transmisor y receptor. Afirmó también que es el nombre colectivo que se da a una parte de la suma total de aquello que puede ser conocido [28].

También es conocimiento, aquello que sabe un individuo, un grupo o una cultura. Todo lo que un ser ha aprendido o asimilado, sean valores, hechos o informaciones; aquello percibido u organizado de acuerdo a conceptos, imágenes o relaciones que el hombre ha sabido dominar. Que el conocimiento existe tan pronto como una persona lo ha adquirido por medio del descubrimiento, la información, la invención o cualquier otro medio (información aprendida). Y, que confundir a la información con el conocimiento equivale al simple error de tomar los medios por los fines, de creer que es cualitativo lo que sólo es cuantitativo, y de pensar que tener una cosa equivale a serla. Asimismo, que la información alcanza valor, cuando se hace un uso correcto de ella, y cuando ésta se convierte en conocimiento [28].

No se puede transmitir conocimiento, sólo información que puede ser convertida o no en conocimiento por el receptor en función de diversos factores previos de cada individuo. La educación está llamada a dar respuesta a estos problemas. Otra consecuencia de la ampliación de la capacidad para almacenar, codificar, procesar y transmitir todo tipo

de información es la transmisión de dos condiciones fundamentales en la comunicación: espacio y tiempo.

Las nuevas tecnologías de la comunicación han desmaterializado y englobado la información, al situarla en el ciberespacio es decir el paso de una cultura basada en el átomo a una cultura basada en el bit.

Internet puede soportar modelos tradicionales de educación a distancia, pero están surgiendo nuevos entornos de enseñanza/aprendizaje basados no sólo en formas de comunicación en tiempo real (vídeo conferencia), sino en técnicas didácticas de aprendizaje cooperativo y colaborativo, sustentadas por la capacidad interactiva de la comunicación mediada por la computadora. Estos entornos rompen la unidad tiempo, espacio y actividad de la enseñanza presencial creando aulas virtuales es decir espacio para la actividad docente, soportada por las facilidades de un sistema de comunicación mediada por una computadora [29].

El desarrollo de las nuevas tecnologías de la información y su utilización en el proceso educativo, requiere del soporte que proporciona el aprendizaje colaborativo, para optimizar su intervención y generar verdaderos ambientes de aprendizaje que promuevan el desarrollo integral de los aprendices y sus múltiples capacidades; el aprendizaje asistido por computadora [29].

Las tecnologías no sólo se van a incorporar a la información como contenidos a aprender o como destrezas a adquirir sino que se utilizarán como medio de comunicación al servicio de la formación como entornos a través de los cuales tendrán lugar procesos de enseñanza/aprendizaje.

Las redes no sólo servirán como vehículo para hacer llegar a los estudiantes materiales de auto-estudio, sino para crear un entorno fluido y multididáctico de comunicación entre profesores y alumnos, y tal vez entre los mismos alumnos (aprendizaje colaborativo). Lo necesario ahora es que las infraestructuras de comunicaciones permitan que se realice de modo generalizado.

Surge entonces la idea de meta-Universidad [11] que presta servicios educativos orientados al control de calidad capaces de ofrecer certificaciones agregando módulos de formación de muchas fuentes diferentes universidades, además proporcionando información a los estudiantes sobre distintas posibilidades de formación a distancia o mixta

(presencial/distancia), de calidad contrastada, autentificando las transacciones entre estudiantes y proveedores de formación manteniendo un registro de la formación adquirida a fin de que puedan lograrse las certificaciones a través de la universidad o de los organismos especializados.

Los entornos de enseñanza/aprendizaje exigen nuevos roles en profesores y alumnos. La misión del profesor en entornos ricos en información es la de facilitador, guía y consejero sobre fuentes apropiadas de información, la de creador de hábitos y destrezas en la búsqueda, selección y tratamiento de la información.

Los estudiantes por su parte deben adoptar un papel mucho más importante en su formación no sólo como receptores pasivos sino como agentes activos en la búsqueda, selección, procesamiento y asimilación de la información.

La digitalización y los nuevos aportes tecnológicos están dando lugar a nuevas formas de almacenar y presentar la información. Los tutoriales multimedia, las bases de datos en línea, las bibliotecas electrónicas, los hipertextos distribuidos, etc., son nuevas maneras de presentar y acceder al conocimiento que en determinados contextos superan las formas tradicionales de la explicación oral, la pizarra los apuntes y el manual.

Las herramientas de autor permitirán que los profesores además de los materiales comerciales desarrollen sus propios materiales adaptados al contexto de los estudiantes. Ha aparecido también un mercado de materiales formativos en soportes tecnológicos que han dado lugar a un nuevo concepto el "*edutenimiento*" que es un híbrido entre educación y entretenimiento accesibles a través de Internet (previo pago de su importe) [12].

El primer paso a seguir en la integración de las nuevas tecnologías a la educación es hacer lo mismo que se ha hecho antes utilizando las nuevas herramientas. Las redes informáticas ofrecen una perspectiva muy diferente de la computadora individual, permiten la comunicación entre personas rompiendo las barreras de espacio, tiempo, estatus, e identidad, pero su mayor potencial reside no sólo en lo que aportan a los métodos de enseñanza/aprendizaje, como en el hecho de que están transformando radicalmente lo que les rodea, es decir, generando una reingeniería del proceso de enseñanza/aprendizaje.

El término "nuevas tecnologías", que se ha mencionado en este documento, se denota también por algunos autores como "**tecnologías avanzadas**" [20] y se refiere a algo que está en continuo cambio y que por su misma novedad no se mantiene con el tiempo, y

la característica más sobresaliente de estas tecnologías es la inmaterialidad, que se debe entender desde su doble perspectiva: su materia prima es la información y la posibilidad de construir mensajes sin referentes externos.

Una ventaja importante de la creación en nuestro campo educativo es la posibilidad de simular fenómenos sobre los que los alumnos puedan trabajar sin riesgos, observar los elementos significativos de cada actividad proceso-fenómeno, descomponer un producto en sus partes y formar criterios propios.

Esta innovación trae consigo problemas adicionales como el de la poca capacidad que tiene la sociedad en general y la escuela en particular, de absorber las tecnologías que se vayan generando. Aunque las nuevas tecnologías se presentan como independientes, tienen altas posibilidades de estar interconectadas [20]. El análisis de las nuevas tecnologías se hace respecto a aspectos básicos: potencialidades, capacidades y posibilidades para transmitir la información y sus efectos socioculturales y políticos. De este modo el papel de las nuevas tecnologías se sitúa en tres direcciones:

- Modificación en la elaboración y distribución de los medios de comunicación.
- Creación de nuevas posibilidades de expresión.
- Desarrollo de nuevas extensiones de la información, acercándose a una idea global que albergue marcos multiculturales y transculturales.

Las nuevas tecnologías requieren un nuevo tipo de aprendiz [20], más preocupado por el proceso que por el producto, preparado para la toma de decisiones y la elección de su ruta de aprendizaje, que reclama una nueva configuración del proceso didáctico y metodológico. Estas nuevas tecnologías aportan un nuevo reto al sistema educativo al pasar de un modelo unidireccional de formación a modelos más abiertos y flexibles donde la información se sitúa en grandes bases de datos que tienden a ser compartidas por distintos alumnos. Por otra parte generan nuevas posibilidades de comunicación alumno-medio-alumno o dicho en otros términos, la interacción entre estudiantes de diferentes contextos culturales y físicos se produce gracias a un medio que hace de elemento intermedio, por ejemplo, el correo electrónico.

2.2.2 Aplicación de técnicas avanzadas de diseño

La evolución tan rápida de las nuevas tecnologías, como por ejemplo, Java o .Net, propicia un problema muy curioso: Invertimos más tiempo aprendiendo técnicas relacionadas con el modo de hacer las mismas cosas (comunicaciones, acceso a datos, formateo, etc.) que en refinar nuestras técnicas de estimación, dirección de proyectos, análisis y diseño.

Es un poco triste ver en el día a día que la gente empieza a utilizar técnicas de Java 1.5 (porque está ahora de moda) y no saben que es el polimorfismo, tienen conocimiento teóricos mínimos de UML (de lo que recuerdan en la universidad, en el mejor de los casos) y un desconocimiento total (o casi) de las técnicas avanzadas de diseño (como patrones de diseño, de los cuales hablaremos con más detalle en la siguiente sección).

Esto provoca que, nos lancemos a escribir código demasiado deprisa y que tengamos que parchearlo constantemente, quedando al final una "aplicación" inmantenible que nos llena de insatisfacción y continuos problemas y errores.

Hasta que todo el mundo no sea consciente que construir un programa es como construir una casa y que hay que invertir un gran esfuerzo en diseñar antes de empezar a escribir "el problema será siempre el mismo".

2.2.2.1 Patrones de diseño, en la implementación automatizada de componentes de código

Al utilizar patrones de código [14], la plantilla de un patrón, representando una decisión de diseño, puede contribuir a la implementación de un conjunto de procedimientos, módulos u objetos en distintos contextos, esto es similar a la manera en que la programación orientada a aspectos se relaciona con otros mecanismos.

La generación del código fuente, análogo a la programación orientada en aspectos, se logra mediante la aplicación de plantillas a distintos contextos; las plantillas tienen distintos puntos de variación en que se colocan valores concretos u otras plantillas.

Un patrón de código es similar al concepto de asuntos (concerns) utilizados en la programación basada en aspectos, ambos se basan en la idea de que los sistemas computacionales son mejor programados al especificar de manera separada los varios

asuntos de un sistema, y entonces utilizar un mecanismo para entrelazarlos en un programa coherente.

Los patrones pueden ser funcionales o no funcionales como el manejo de transacciones o manejo de excepciones.

Como se muestra en la Figura 2.1, algunos de los candidatos a ser patrones de código encontrados en distintas porciones de código fuente, fueron examinados para reconocer semejanzas y puntos de variación. A partir de las semejanzas se formaron las plantillas que representan patrones de código.

Figura 2.1 Porciones de código candidatos a ser patones

En algunas ocasiones los patrones se presentan como elementos de otros patrones (intermezclados), de manera que un texto se repite no solamente en distintas partes del sistema, sino que además son comunes dentro de otros fragmentos que son también patrones de código. En la Figura 2.2 se esquematiza la manera en que un patrón se descompone en otros dos sub-patrones.

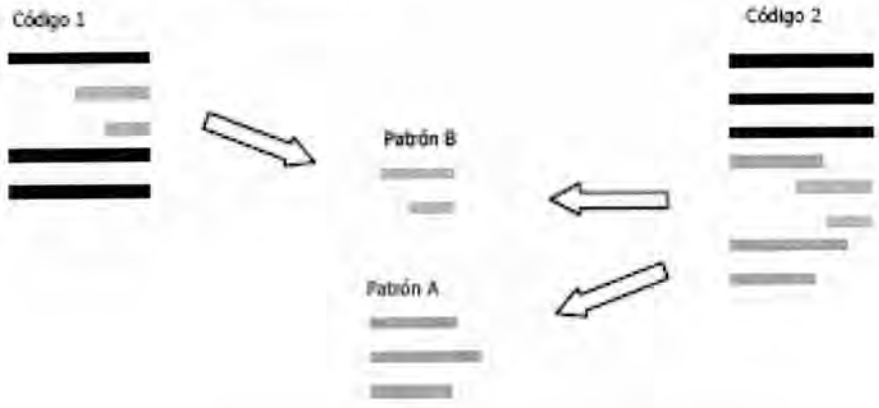

Figura 2.2 Descomposición de un patrón en sub-patrones

Patrones de código aparecen repetidamente en los métodos de una misma clase y en clases con propósitos similares. Al ser clasificadas dentro de la capa a la que pertenecen, pueden mostrarse como se observa en la siguiente tabla:

Función del patrón	Capa	Elemento
Manejo de excepciones	Todas las capas	
Notificación de cambio	Negocio	Acceso a propiedades
Registro de cumplimiento de reglas de negocio		
Añadir, eliminar e iterar sobre grupos de objetos de tipo definido		Colecciones / Iteradores de objetos
Acceso a base de datos	Acceso a datos	Varios

Tabla 2.1 Clasificación de la función del patrón en base a la capa a la que pertenecen

2.3 Fallas en el software

Muchos trabajos en la programación estructurada, tratan de proveer métodos y herramientas para el diseño correcto de software y evitar fallas. Muchos esfuerzos [7], en particular para el diseño de lenguajes de programación que no permiten a los diseñadores de software estructurar sistemas complejos en cuanto a módulos con abstracciones de los datos. Asociadas con esta abstracción, las excepciones especifican cómo se debe responder a una posible falla en tiempo de ejecución sin violar las propiedades invariantes. Los mecanismos del lenguaje para el manejo de excepciones, pueden ayudar a los diseñadores a implementar correctamente las especificaciones para las excepciones.

Fallas de diseño pueden llevar a la detección de excepciones no especificadas, lo que se conoce como detección de fallas. Para tratar tales excepciones que no son previstas, se ha propuesto la técnica de manejo de excepción por *default* antes llamada "called backward recovery"; la cual es una técnica complementaria al diseño de fallas y la combinación sirve para obtener sistemas muy confiables.

En lo que respecta a errores, ocurren generalmente al tiempo de estarse ejecutando un programa, por ejemplo una división entre cero. Muchos lenguajes de programación, entre ellos Java, tienen un mecanismo muy completo para el manejo de errores y

excepciones. La diferencia entre una excepción y un error es que una excepción si es atrapada por el programa y se puede recuperar de ella para continuar con su ejecución; mientras que un error, aunque sea atrapado, el programa se suspenderá y terminará su ejecución. Es por esta razón que es conveniente y necesario atrapar errores o excepciones, así el programador podrá incluir algún mecanismo de corrección de errores o al menos de avisarle al usuario en lugar de que sea el servidor quien le avise.

En Java las excepciones son objetos que se crean cuando una condición anormal ocurre. Existen dos tipos de excepciones:

- "Checked". Señalan una condición anormal que el cliente debe manejar. Todas las excepciones "checked" deben ser capturadas y manejadas (verificadas), el compilador y la JVM (Java Virtual Machine) lo verifican.
- "Unchecked". Son el resultado de una lógica incorrecta o de errores en la programación, excepciones no verificadas.

En el capítulo 4 se describe a detalle el manejo de excepciones en Java.

2.4 Aspectos

Podemos dar una definición inicial de aspecto, diciendo que es un modulo potencialmente capaz de encapsular software y puede "cruzar" varias tareas (en el capítulo 3, se ven a detalle los conceptos de la programación orientada a aspectos). Y un sub-aspecto se da cuando es parte de una tarea que es usada por otros aspectos.

Un concepto importante dentro de la orientación a aspectos son los puntos de unión, como ya se menciono el capítulo 3 se dedica en particular a la orientación a aspectos, pero a continuación introducimos brevemente el concepto.

2.4.1 Puntos de unión

Un punto de unión ("Join Point") es un punto bien definido en la ejecución de un programa. Para algunos autores, el conjunto posible de puntos de unión incluye todas las localizaciones en el código de un componente o aplicación, es decir, podríamos definir

puntos de unión a nivel de sentencia, de este modo podríamos describir que ocurre en cualquier parte del código.

En base a las investigaciones realizadas para esta tesis, en el ámbito del paradigma orientado a aspectos y el desarrollo de aplicaciones distribuidas, podemos argumentar que los puntos de unión más extendidos o que con mayor frecuencia se utilizarán en el desarrollo de sistemas distribuidos responden a: invocaciones a métodos, manejo de excepciones y acceso a atributos.

La justificación de estos puntos de unión se basa en el funcionamiento de los sistemas basados en componentes, donde los componentes se comunican mediante el paso de mensajes (mediante la invocación a métodos), el lanzamiento de una excepción desde un componente, y que debe ser tratada por otro componente, modificaciones de atributos de un componente, etc.

2.4.2 Vista arquitectónica de aspectos

Aunque el uso de aspectos ha ganado aceptación, el diseño incremental de aspectos se ha abandonado, esto es, usar desde el diseño la orientación a aspectos y que ese diseño nos brinde un código reutilizable.

La forma de cooperar entre los aspectos debe ser declarada explícitamente, claro, desde la etapa de diseño. Se ha propuesto en [8] el uso de una vista arquitectónica, una extensión de UML (para tener mayor flexibilidad) y un nuevo diagrama para mostrar la combinación de aspectos para diferentes tareas.

Ya que el diseño orientado a aspectos nos permite acercarnos a las relaciones entre aspectos, podemos limitar el alcance de que cada uno de ellos y usar reglas de composición que implementan esas relaciones. Los métodos actuales apoyan la modularización, sin promover el diseño incremental de aspectos.

Un nuevo problema es el traslape entre aspectos, pues no se llega a ver con los lenguajes convencionales y tiene una gran complejidad en composición y mantenimiento de aspectos.

Conceptualmente, el modelo de arquitectura de aspectos nos da una vista, en la arquitectura de software, es decir brinda una perspectiva de la orientación a aspectos en la arquitectura de software. Para ello, los aspectos se ven como simples bloques de

construcción, en los cuales los aspectos complejos pueden sub-dividirse en sub-aspectos y cada competencia del sistema en un módulo.

La vista de aspectos, soporta remodularización, es decir, ayuda al mantenimiento al poder cambiar aspectos sin cambiar la funcionalidad del sistema. Enfatiza la distinción de tareas y cosas de interés conceptual que deben ser tratadas por uno o más aspectos.

La orientación a aspectos promueve la modularidad, apoyando el diseño de aspectos por composición, lo que nos permite clasificar esos módulos en:

- Complejos y adaptados en el tiempo, por cambios en los requerimientos (diseño incremental).
- Colecciones de aspectos (interaccionan y cooperan) con relaciones explícitas.

Con los sub-aspectos surge lo que podemos llamar el traslape de aspectos, y principalmente es por tres causas:

- Sub-aspectos con intereses propios, pueden tener importancia durante el ciclo de vida.
- Justifica que se de una re-modularización.
- Puede ser aislado en un sub-aspecto.

En la arquitectura de aspectos, un aspecto es un mapeo que describe un posible cruce (entre tareas) incremental de un diseño existente, ese aspecto puede ser usado esencialmente de forma genérica, con el patrón "*composite*", es decir puede ser instanciado varias veces.

Por otra parte, la composición de aspectos [8], se basa en el principio de super-imposición, por ejemplo, teniendo una operación asimétrica, un aspecto "A" es aplicado sobre otro "B", es decir, (A/B), entonces:

- A/B = B/A cuando no hay vinculación (no hay dependencia).
- A/B <> B/A hay vinculación (hay dependencia). Un elemento de A está vinculado a un elemento de B, A tiene que ser aplicado sobre B (B después que A), es decir, B depende de A (se tiene una relación de dependencia).

2.4.3 Aplicaciones implementadas con AOP

La AOP persigue implementar una aplicación de forma eficiente y fácil de entender. Sigue el paradigma de la orientación a objetos, soportando por tanto la descomposición

orientada a objetos, así como la procedural y la descomposición funcional; es decir, como puede utilizarse con los diferentes estilos de programación, no se considera una extensión de la orientación a objetos. Como una primera definición para diferenciar aspectos de componentes [2]:

> "Los aspectos tienden a no ser unidades de la descomposición funcional del sistema, sino a ser propiedades que afectan la ejecución de los componentes en una manera sistemática".

Como ya se mencionó, con el uso de la programación orientada a aspectos se evita el código *mezclado* y el c*ódigo diseminado* [15]:

- **El *Código Mezclado:*** en un mismo módulo de un sistema, conviven más de un requerimiento. En el modelo existen elementos de implementación de más de un requerimiento.
- **El *Código Diseminado:*** un requerimiento está esparcido sobre varios módulos. La implementación de dicho requerimiento también queda diseminada sobre esos módulos.

La combinación de estos síntomas afecta tanto al diseño como a la implementación de software [15]. La implementación simultánea de varios conceptos tiene dos efectos negativos:

- El primero es una baja correspondencia entre un concepto y su implementación.
- El segundo es una menor productividad de los desarrolladores, ya que se distraen del concepto principal.

Por otro lado, la implementación de varios conceptos en un mismo módulo lleva a un código poco reutilizable, de baja calidad, y propenso a errores. Esto se debe a que alguno de los tantos conceptos puede ser subestimado.

Por último, la evolución es más difícil debido a la insuficiente modularización. Los futuros cambios en un requerimiento implican revisar y modificar cada uno de los módulos donde esté presente ese requerimiento.

AOP, nos brinda mecanismos adecuados para abstraer y encapsular conceptos que no forman parte de la funcionalidad básica de los sistemas:

- Depuración.
- Sincronización.
- Distribución.
- Seguridad
- Administración de memoria.
- Manejo de excepciones.

2.4.3.1 Ventajas de AOP

R. Laddad [2], enumera algunas de las principales ventajas de la AOP. En principio, ayuda eficazmente a superar los problemas causados por el *Código Mezclado* y *Código Diseminado*. En cuanto a la modularización, la AOP logra separar cada concepto con mínimo acoplamiento, resultando en implementaciones modularizadas aún en la presencia de conceptos que se entrecruzan. Esto lleva a un código más limpio, menos duplicado, más fácil de entender y de mantener. A su vez la separación de conceptos permite agregar nuevos aspectos, modificar y/o remover aspectos existentes fácilmente, lo que permite una mejor evolución, y mayor reusabilidad. Durante el diseño, el diseñador se ve beneficiado ya que puede retrasar las decisiones sobre requerimientos actuales o futuros, debido a que permite, luego, implementarlos separadamente e incluirlos automáticamente en el sistema.

Esto último resuelve el dilema del arquitecto sobre cuántos recursos invertir en la etapa de diseño, cuando la cuestión planteada es "demasiado diseño", ayudando a no tener afectaciones ni en el diseño ni en la implementación del software [15], y obteniendo como resultado una alta correspondencia entre un concepto (diseño) y su implementación, y una mayor productividad de los desarrolladores, ya que no se *distraen* del concepto principal.

El código que implementa la funcionalidad básica es mucho más limpio y entendible. Brinda una mayor facilidad para realizar cambios y modificaciones y su desarrollo se vuelve mucho más intuitivo y natural.

2.4.3.2 Desventajas de AOP

Guezzi et.al. [3] mencionan tres desventajas. La primera remarca posibles choques entre el código funcional, expresado en el lenguaje base, y el código de aspectos expresado en los

lenguajes de aspectos. Usualmente, estos choques nacen de la necesidad de violar el encapsulamiento para implementar los diferentes aspectos, sabiendo de antemano el riesgo potencial que se corre al utilizar estas prácticas. La segunda es la posibilidad de choques entre los aspectos. El ejemplo clásico es tener dos aspectos que trabajan perfectamente por separado, pero al aplicarlos conjuntamente resultan en un comportamiento anormal. La tercera, trata con posibles choques entre el código de aspectos y los mecanismos del lenguaje. Uno de los ejemplos más conocidos de este problema es la *anomalía de herencia.* Dentro del contexto de la AOP, el término puede ser usado para indicar la dificultad de heredar el código de un aspecto en presencia de la herencia.

Resumiendo, podemos mencionar que las desventajas surgen del hecho de que la AOP está en una etapa de consolidación y de aprendizaje por la mayoría de la gente que la empieza a utilizar, lo que puede provocar:

- Choques entre el código funcional, expresado en el lenguaje base y los lenguajes de aspectos.
- Choques entre los aspectos.
- Choques entre el código de aspectos y los mecanismos del lenguaje.

CAPÍTULO 3

3. Orientación a aspectos

En las siguientes secciones se presentan los principales conceptos que se manejan dentro del paradigma de la orientación a aspectos.

3.1 Programación orientada a aspectos

El concepto fue introducido por Gregor Kiczales y su grupo, el equipo de Demeter [19] estaba centrado en la programación adaptativa (Adaptive Programming AP, por sus siglas en ingles) una instancia temprana de la programación orientada a objetos. En 1995 se publicó la primera definición del concepto de aspecto, por el grupo Demeter [19]:

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

La relación entre la AOP y la AP surge de la Ley de Demeter [15]: "Solo conversa con tus amigos inmediatos". Esta ley inventada en 1987 en la Northeastern University y popularizada en libros de Booch, Budd, Coleman, Larman, Page-Jones, Rumbaugh, entre otros, es una simple regla de estilo en el diseño de sistemas orientados a objetos.

Para poder escribir código respetando la Ley de Demeter debe observarse que los conceptos que se entrecruzan entre varias clases deberían y tendrían que ser claramente encapsulados. Esto resultaría en una clara separación de los conceptos de comportamiento y de aquellos conceptos de la funcionalidad básica.

Los principales objetivos de la programación orientada a objetos [19] son el de separar los conceptos y minimizar las dependencias entre ellos. Con separar los conceptos se consigue que cada cosa esté en su sitio (que cada decisión se tome en un lugar concreto) y con minimizar las dependencias, se consigue disminuir el acoplamiento entre los distintos elementos.

Al perseguir estos objetivos se tienen varias ventajas, como son:

- Código menos enmarañado, más natural y más reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y con dependencia mínima.
- Más facilidad para depurar y hacer modificaciones en el código.
- Impacto mínimo de un conjunto de modificaciones en la definición de un concepto.
- Código reutilizable con facilidades de acoplamiento y desacoplamiento cuando sea necesario.

La idea central que persigue la AOP es permitir que un programa sea construido describiendo cada concepto separadamente. El soporte para este nuevo paradigma se logra a través de una clase especial de lenguajes [15], llamados lenguajes orientados a aspectos (LOA), los cuales brindan mecanismos y constructores para capturar aquellos elementos que se diseminan por todo el sistema. A estos elementos se les da el nombre de aspectos. Una definición para tales lenguajes sería: Los LOA son aquellos lenguajes que permiten separar la definición de la funcionalidad pura de la definición de los diferentes aspectos. Los LOA deben satisfacer varias propiedades deseables, entre ellas:

- Cada aspecto debe ser claramente identificable.
- Cada aspecto debe auto-contenerse.
- Los aspectos deben ser fácilmente intercambiables.
- Los aspectos no deben interferir entre ellos.
- Los aspectos no deben interferir con los mecanismos usados para definir y evolucionar la funcionalidad, como la herencia.

Las propiedades anteriores son las deseables, pero en ocasiones no es posible lograr satisfacer todas ellas, por ejemplo cuando se tienen sub-aspectos, los aspectos que comparten esos sub-aspectos no son auto-contenidos.

3.2 Un aspecto

La definición de aspecto ha ido evolucionando, desde la primera definición mencionada en los párrafos anteriores, hasta la que se maneja actualmente: Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen desde la etapa de diseño hasta la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa es una unidad modular que aparece en otras unidades modulares del programa (G. Kiczales [19]).

Una definición más informal: los aspectos son la unidad básica de la AOP, y son las partes de una aplicación que describen las cuestiones claves relacionadas a la funcionalidad esencial o el rendimiento, pueden verse como los elementos que se diseminan por todo el código y que son difíciles de describir localmente con respecto a otros componentes [15].

3.3 Componente y aspecto

Se puede diferenciar entre un componente y un aspecto viendo el primero como aquella propiedad que se puede encapsular claramente en una clase mientras que un aspecto no se puede encapsular [1].

Los aspectos no suelen ser unidades de descomposición funcional del sistema, sino propiedades que afectan al rendimiento de los componentes que brindan las funcionalidades del sistema. Algunos ejemplos de aspectos son: los patrones de acceso a memoria, la sincronización de procesos concurrentes, el registro de sucesos, el manejo de errores, etc. En el punto 5.2 se describe con detalle la idea de que los aspectos afectan el rendimiento de los componentes que brindan las funcionalidades básicas del sistema.

La idea es tener un programa formado por un conjunto de aspectos, más un modelo de objetos. Con el modelo de objetos se tiene la funcionalidad básica, mientras que con los aspectos se tienen las características de rendimiento y otras no relacionadas con la funcionalidad básica del programa. Y en general este tipo de programas es más compacto y modular, teniendo cada aspecto su propia competencia.

Con lo mencionado anteriormente, podemos diferenciar los aspectos de los demás integrantes del sistema: al momento de implementar una propiedad, tenemos que la misma tomará una de las dos siguientes formas [15]:

1. Un componente: si puede encapsularse claramente dentro de un procedimiento generalizado. Un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo.
2. Un aspecto: sino puede encapsularse claramente en un procedimiento generalizado. Los aspectos tienden a ser propiedades que afectan el desempeño y la funcionalidad de los componentes en forma sistemática.

A la luz de estos términos, podemos enunciar la meta principal de la AOP: brindar un contexto al programador que permita separar claramente componentes y aspectos, separando componentes entre sí, aspectos entre sí, y aspectos de componentes, a través de mecanismos que hagan posible abstraerlos y componerlos para producir el sistema completo. Tenemos entonces una importante y clara diferencia respecto de los lenguajes de procedimiento generalizado (LPG), donde todos los esfuerzos se concentran únicamente en los componentes, dejando de lado los aspectos [15]. Una vez diferenciados los aspectos de los componentes, estamos en condiciones de **definir a un aspecto como un concepto que no es posible encapsularlo claramente, y que resulta diseminado por todo el código.**

3.4 Fundamentos

Englobando los conceptos mencionados, tenemos que con las clases se implementa la funcionalidad básica o principal de la aplicación [15]; mientras que con los aspectos se tienen los servicios horizontales, tales como, la persistencia, el manejo de errores, la sincronización de procesos, la comunicación de procesos, etc.

Los aspectos se escriben mediante lenguajes de descripción de aspectos [2], por lo que los lenguajes orientados a aspectos, definen una nueva unidad de programación para encapsular las funcionalidades que cruzan todo el código aspectos). Para que los aspectos y componentes de las aplicaciones se puedan entremezclar se deben tener puntos en común, los cuales se conocen como puntos de enlace, que son una clase especial de interfaz; el

encargado de realizar el proceso de mezclarlos se conoce como tejedor ("weaver") que mezcla los diferentes mecanismos de abstracción y composición.

3.5 Definiciones para la programación orientada a aspectos

Se necesita contar con:

- Lenguaje para definir funcionalidad básica (lenguaje base), el cual puede ser un lenguaje de propósito general.
- Lenguaje de aspectos, define la forma de los aspectos.
- Tejedor, combina los lenguajes, el proceso de mezcla puede realizarse en tiempo de compilación o en tiempo de ejecución.

3.6 Tejedor

Los aspectos nos describen apéndices al comportamiento de los objetos [15]. Hacen referencia a las clases de los objetos y definen en qué puntos se han de colocar estos apéndices (puntos de enlace, que pueden ser métodos o asignaciones de variables). Las clases y aspectos pueden mezclarse de dos formas: estática o dinámicamente.

Enlazado estático. Implica modificar el código fuente de una clase insertando los apéndices al comportamiento (sentencias) en esos puntos de enlace. La principal ventaja es que se evita que el nivel de abstracción derive en un impacto negativo en el rendimiento. Como desventaja, es difícil identificar los aspectos en el código, lo cual para remplazar aspectos de forma dinámica en tiempo de ejecución da un problema de eficiencia. Un ejemplo de este tipo de tejedor es AspectJ [18], que es el que se usamos para el desarrollo del prototipo.

Enlazado dinámico. Una precondición para que se pueda realizar un enlazado dinámico es que los aspectos existan de forma explícita (tanto en compilación como en ejecución), para lo cual los aspectos y las estructuras enlazadas se modelan como objetos, y el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica. Un ejemplo de este tipo de tejedor es JPAL [15].

3.7 Lenguajes orientados a aspectos

A continuación se describen brevemente algunos lenguajes orientados a aspectos disponibles actualmente. En el apéndice B se ve con mayor profundidad el lenguaje orientado a aspectos AspectJ, ya que hemos seleccionado este lenguaje por ser la herramienta con mayor usabilidad, futuro, popularidad y desarrollo.

3.7.1 JPAL

La principal característica de este lenguaje es que los puntos de enlace son especificados independientemente del lenguaje base. Estos puntos de enlace independientes reciben el nombre de *Junction Point* (JP). De ahí el nombre JPAL que significa *Junction Point Aspect Language* [15]. El tejedor JPAL genera un esquema del tejedor 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 (JP) disparándose la ejecución de las acciones (código del aspecto correspondiente).

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 de aspectos, lo cual hace posible modificar en tiempos de ejecución los programas de aspectos [15]. Sin embargo, esta solución no es lo suficientemente poderosa como para agregar o reemplazar programas de aspectos en ejecución. Para tal efecto se agrega al Esquema del Tejedor una entidad llamada Administrador de Programas de Aspectos (APA), el cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una biblioteca dinámica que almacena los aspectos y permite dinámicamente agregar, quitar o modificar aspectos, y mandar mensajes a dichos aspectos. 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.

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 [15].

Resumiendo, JPAL tiene la particularidad de separar los puntos de enlace, 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 [15]. Su principal aplicación es para la implementación de sistemas distribuidos.

3.7.2 D

D [15] 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 Lopes [30].

La sincronización es un ítem 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 ítem 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.

Con respecto a la herencia, los módulos de aspectos se relacionan con la herencia de clases a través de simples reglas: Sea C una clase y A el módulo de aspecto asociado a C, se verifican las siguientes propiedades:

- Visibilidad de campos: Los elementos heredados desde ancestros de C son visibles a A.
- No hay propagación de efectos: A no afecta a ninguna superclase de C.
- Redefinición de semántica: A redefine completamente cualquier módulo de aspecto del mismo tipo definido en una superclase de C. No hay ninguna relación entre A y los módulos de aspectos de las superclases de C.
- Herencia de coordinación: A afecta todas las subclases de C que no tienen asociado un módulo de aspecto del mismo tipo. A no afecta los nuevos métodos definidos en una subclase de C. Si un método m declarado en C es redefinido por una subclase de C sin asociarle otro módulo de aspecto del mismo tipo entonces A también es el aspecto para ese método.

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.

3.7.2.1 COOL

COOL (*COOrdination aspect Language* [15]) es un lenguaje de aspectos de dominio específico para tratar con la exclusión mutua de hilos, sincronización, suspensión y reactivación de hilos.

Un programa COOL consiste de un conjunto de módulos coordinadores. Los módulos coordinadores, o directamente coordinadores, se asocian con las clases a través

del nombre. Un mismo coordinador puede coordinar a más de una clase. La unidad mínima de sincronización es el método. La declaración de un coordinador describe la estrategia de coordinación. Los coordinadores no son clases: utilizan un lenguaje diferente, no pueden ser instanciados y sirven para un propósito muy específico. Los coordinadores se asocian automáticamente con las instancias de clases que coordinan en tiempo de instanciación, y durante el tiempo de vida de los objetos esta relación tiene un protocolo bien definido.

Los coordinadores tienen conocimiento de las clases que coordinan para definir la mejor estrategia de coordinación posible. Sin embargo, las clases no tienen conocimiento de los aspectos, es decir que dentro de una clase no es posible nombrar a un coordinador.

La asociación entre los objetos y los coordinadores es uno-a-uno por defecto y recibe el nombre de coordinación "per object". Sin embargo, un coordinador también puede asociarse con todos los objetos de una o más clases y recibe el nombre de coordinación "per class". Las estrategias de sincronización se expresan de forma declarativa.

El cuerpo de un coordinador contiene declaración de variables de condición y ordinarias, un conjunto de métodos auto-exclusivos, varios conjuntos de métodos mutuamente exclusivos y manejadores de métodos. Los métodos nombrados en el cuerpo deben ser métodos válidos de las clases coordinadas.

Un coordinador asociado a una clase C tiene la siguiente visibilidad:

- Todos los métodos públicos, protegidos y privados de C.
- Todos los métodos no privados de las superclases de C.
- Todas las variables privadas, protegidas y públicas de C.
- Todas las variables no privadas de las superclases de C.

3.7.2.2 RIDL

RIDL (*Remote Interaction and Data transfers aspect Language* [15]) es un lenguaje de aspectos de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución.

Un programa RIDL consiste de un conjunto de módulos de portales. Los módulos de portales o directamente portales se asocian con las clases por el nombre. Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada

a él, y puede asociarse como máximo a una clase. La unidad mínima de interacción remota es el método.

La declaración de un portal identifica una clase cuyas instancias pueden invocarse desde espacios remotos. Dichas instancias se llaman objetos remotos. La declaración de un portal identifica qué métodos de una clase serán exportados sobre la red. En el portal estos métodos se llaman operaciones remotas. Para cada una de estas operaciones se describe qué objetos remotos esperan y qué datos enviarán a los llamadores.

Los portales no son clases: utilizan un lenguaje diferente, no pueden ser instanciados y sirven para un propósito muy específico. Tampoco son tipos en el sentido estricto de la palabra. Un portal se asocia automáticamente con una instancia de la clase en el momento que una referencia a esa instancia se exporta fuera del espacio donde la instancia fue creada. Durante el tiempo de vida de la instancia esta relación se mantiene mediante un protocolo bien definido.

Un portal asociado a una clase C tiene la siguiente visibilidad:

- Todos los métodos públicos, protegidos y privados de C a excepción de los métodos estáticos.
- Todos los métodos no privados de las superclases de C a excepción de los métodos estáticos.
- Todas las variables privadas, protegidas y públicas de todas las clases de una aplicación D.

Este último punto establece una dependencia explícita entre los portales y las relaciones estructurales completas de las clases. Esta dependencia expone la necesidad de controlar la transferencia de datos entre los distintos espacios de ejecución.

3.7.3 ASPECTC

AspectC [15] 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 toman 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: varref (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 [15], 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.

3.7.4 ASPECTS

AspectS [15] 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*. 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, enviadores o clases enviadoras.

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 paso 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 meta-programación. "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.

3.7.5 ASPECTC++

AspectC++ [15] 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 enlaces 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 enlaces. 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.

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++.

Si bien el manejo de aspectos en AspectC++ es similar al manejo que proporciona AspectJ, introduce modificaciones importantes dentro del modelo de puntos de enlace, permitiendo puntos de enlace sobre clases, objetos, y sobre el flujo de control. Esto resulta en un diseño del lenguaje más coherente con el paradigma de aspectos.

3.7.6 MALAJ

Malaj [15] es un sistema que soporta la programación orientada a aspectos. Define constructores lingüísticos separados para cada aspecto de dominio específico, donde el código de los aspectos tiene una visibilidad limitada del código funcional, reduciendo los posibles conflictos con las características lingüísticas tradicionales y también, con el principio de encapsulación.

Malaj es un lenguaje orientado a aspectos de dominio específico, concentrándose en dos aspectos: sincronización y relocalización. Puede verse como un sucesor de los lenguajes COOL y RIDL por su filosofía, enfatizando la necesidad de restringir la visibilidad de los aspectos, y reglas claras de composición con los constructores tradicionales. Para cada aspecto, provee un constructor lingüístico distinto, limitando así la visibilidad del aspecto sobre el módulo funcional asociado a él. Esto último se logra al estudiar cuidadosamente la relación entre el código funcional y un aspecto dado.

El lenguaje base de *Malaj* es una versión restringida de Java, donde se han removido los servicios que proveen los aspectos mencionados. Los servicios removidos son: la palabra clave "synchronized", y los métodos "wait", "notify" y "notifyAll".

Para el aspecto de sincronización *Malaj* provee el constructor guardián. Cada guardián es una unidad distinta con su propio nombre, y se asocia con una clase en particular (esto es, "vigila" esa clase) y expresa la sincronización de un conjunto relacionado de métodos de esa clase, es decir, que el guardián de una clase V representa básicamente el conjunto de métodos sincronizados de V. Los guardianes no pueden acceder a los elementos privados de la clase que vigilan, y el acceso a los atributos públicos y protegidos se limita a un acceso de sólo lectura. El comportamiento adicional en un guardián para un método m de la clase asociada se especifica introduciendo código que se

ejecutará antes o después de m, a través de las cláusulas "before" y "after". Una última característica de los guardianes es que pueden heredarse. Para reducir el problema de anomalía de herencia [15] las cláusulas "before" y "after" en una clase pueden referirse a las cláusulas "before" y "after" de su clase padre a través de la sentencia super.

El aspecto de relocalización involucra el movimiento de objetos entre sitios en un ambiente de redes. Este tipo de relación es claramente dinámico. Para este aspecto *Malaj* provee el constructor "relocator", que llamaremos relocador. Un relocador será una unidad diferente con su propio nombre, y se asocia con una clase en particular.

Las acciones de relocalización pueden ejecutarse antes o después de la ejecución de un método. Para modelar este comportamiento, el relocador brinda cláusulas "before" y "after", que permiten la especificación deseada. También la visibilidad del relocator es limitada. Como el constructor guardián, un relocador puede heredarse, y las cláusulas "before" y "after" en una clase pueden referirse a las cláusulas "before" y "after" de su clase padre a través de la sentencia super, y así reducir el impacto de la anomalía de herencia [15].

Como conclusión *Malaj* provee una solución intermedia entre flexibilidad y poder, además ayuda al entendimiento y facilidad de cambio. No permite describir cualquier aspecto, pero sí captura el comportamiento de dos conceptos relacionados con el código funcional. El próximo paso en *Malaj* es extenderlo para abarcar otros aspectos.

3.7.7 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* [15] 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 biblioteca. 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.

3.7.8 AspectJ

AspectJ [18] es una extensión de Java orientada a aspectos y de propósito general con un nuevo constructor llamado aspecto. Los aspectos cortan las clases, las interfaces y otros aspectos. Mejoran la separación de competencias haciendo posible localizar de forma limpia los conceptos de diseño del corte. Un aspecto en AspectJ, es una clase con la particularidad que puede contener unos constructores de corte.

Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos.

Los aspectos son constructores que trabajan al cortar de forma transversal la modularidad de las clases de forma limpia y cuidadosamente diseñada. Por lo tanto, un aspecto puede afectar a la implementación de un número de métodos en un número de clases, lo que permite capturar la estructura de corte modular de este tipo de conceptos de forma limpia.

Un aspecto en AspectJ contiene:

- Cortes. (*"pointcuts"*) capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos. Se utilizan para definir el código de los aspectos utilizando avisos.
- Introducciones. (*"introduction"*) para introducir elementos completamente nuevos en las clases dadas. Entre estos elementos podemos añadir:
    - Un método
    - Un constructor
    - Un atributo
    - Varios de los elementos anteriores a la vez
    - Varios de los elementos anteriores en varias clases

- Avisos (*"advices"*) definen partes de la implementación del aspecto que se ejecutan en puntos bien definidos. El cuerpo de un aviso puede ser añadido en distintos puntos de código:
    - "Before". Se ejecuta justo antes de las acciones asociadas
    - "Alter". Se ejecuta justo después de las acciones asociadas.
    - "Catch". Se ejecuta cuando durante la ejecución de las acciones asociadas se lanza una excepción.
    - "Finally". Se ejecuta justo después de las acciones asociadas; incluso aunque haya producido una excepción.
    - "Around". Atrapa la ejecución de los métodos designados por el evento.

3.8 Lenguajes de aspectos de dominio específico vs lenguajes de propósito general

Los lenguajes de dominio específico soportan uno o más de los sistemas de aspectos que se han mencionado (distribución, manejo de errores, sincronización, etc.). No pueden soportar otros aspectos distintos. Tienen un nivel de abstracción mayor al lenguaje base. Normalmente imponen restricciones en la utilización del lenguaje base (para garantizar que los conceptos del dominio de los aspectos se programen con el lenguaje de aspectos, para no producir un conflicto).

Por su parte los de propósito general se diseñan para ser utilizados con cualquier clase de aspecto sin imponer restricciones al lenguaje base. Tienen un severo inconveniente y es que permiten la separación de código pero no garantizan la separación de funcionalidades.

Desde el punto de vista práctico, siempre le es más fácil a los programadores el aprender un lenguaje de propósito general, que el tener que estudiar varios lenguajes distintos de propósito específico.

3.8.1 Tabla comparativa de las herramientas

En la siguiente tabla se encuentran resumidas las principales características de las herramientas orientadas a aspectos descriptas en las secciones anteriores:

Leng. OA	Leng. Base	Tejido	Propósito	Características salientes
JPAL	Independiente	Dinámico	General	Meta-Tejedor. Independiente del leng. Base. Totalmente dinámico.
D	Cualquier lenguaje OO	Estático	Específico	Usado en la programación distribuida. Provee interacción entre aspectos y herencia.
COOL	Cualquier lenguaje OO	Estático	Específico	Describe la sincronización de hilos concurrentes. Visibilidad limitada del aspecto
RIDL	Cualquier lenguaje OO	Estático	Específico	Modulariza la interacción remota. Visibilidad limitada del aspecto.
AspectC	C	Estático	General	Usado en la implementación orientada a aspectos de sistemas operativos
AspectS	Squeak	Dinámico	General	Basado en el paradigma de paso de mensajes. Todo aspecto es un objeto.
AspectC++	C++	Estático	General	Aspectos con extensiones del concepto de clases. Descripción más natural de los puntos de enlace.
MALAJ	Java	Dinámico	Específico	Modulariza los aspectos de sincronización y relocalización. Su objetivo es eliminar los conflictos entre AOP y OOP.
HyperJ	Java	Dinámico	General	Basado en "hyperslices". Permite la composición de aspectos. Ambiente visual de trabajo.
AspectJ	Java	Estático	General	Los aspectos cortan clases, interfaces y otros aspectos. Mejora la separación de competencias, localizando de forma limpia los conceptos de diseño de corte

Tabla 3.1 Comparación entre herramientas orientadas a aspectos

Como lo refleja la tabla 3.1, las tres decisiones de diseño: lenguaje base, tejido y propósito, son totalmente independientes entre sí para los LOA. Esto es, que todas las combinaciones entre las decisiones son teóricamente posibles. Aunque existe la tendencia a implementar el comportamiento del tejedor según la naturaleza del lenguaje base: si el

lenguaje base es estático entonces el tejedor es estático y si el lenguaje base es dinámico entonces el tejedor es dinámico.

3.9 Aspectos en el diseño (uso de UML)

Cuando las aplicaciones necesitan un alto grado de adaptabilidad o que se requiere que se puedan reutilizar, se debe considerar desde el diseño la orientación a aspectos. Los trabajos sugeridos proponen utilizar UML como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar el diseño funcional de los objetos separando del diseño no funcional del mismo, o lo que es lo mismo, representar la funcionalidad básica separada de los otros aspectos.

Las ventajas que tiene el capturar los aspectos ya desde la fase de diseño son:

- Facilita la creación de documentación y el aprendizaje. El tener los aspectos como constructores de diseño permite que los desarrolladores los reconozcan en los primeros estados del proceso de desarrollo, teniendo así una visión de más alto nivel y ayudando así a que los diseñadores de aspectos y los principiantes puedan aprender y documentar los modelos de aspectos de un modo más intuitivo, pudiendo incluso utilizar herramientas CASE para tener representado el modelo de forma visual.
- Facilita la reutilización de los aspectos. La facilidad de documentación y aprendizaje influye en la reutilización de la información de los aspectos. Al saber como se diseña y como afecta a otras clases, es más fácil ver como se pueden utilizar de otra forma, lo que incrementaría la reutilización de los aspectos.

3.10 Comparaciones entre OOP y AOP

Estudios realizados en [15], muestran como se consiguen beneficios entre implementaciones realizadas con la programación orientada a objetos y las implementaciones resultantes después de realizar las modificaciones y trabajar con programación orientada a aspectos, entre ellos podemos mencionar que al utilizar aspectos la ventaja más destacable es que el código que implementa la funcionalidad básica es mucho más limpio y entendible. Otra ventaja importante es una mayor facilidad para

realizar cambios y modificaciones, como también acciones para remover o agregar comportamiento. En cuanto al desarrollo, el mismo se vuelve mucho más intuitivo y natural, dada la naturaleza declarativa de los *puntos de enlace*, y al comportamiento adicional que permite definir AspectJ. En [15] a fin de cuantificar los beneficios de la AOP, y AspectJ se definen y se aplican métricas orientadas a aspectos. Las cuales incluyen los conceptos relacionados al paradigma de aspectos, reflejando de una manera fiel y precisa el desempeño de la AOP.

- En primer lugar se define la **métrica *Beneficio AOP.*** Mide cuál es el beneficio de introducir aspectos.
  - Define una relación de esfuerzo de desarrollo entre una implementación sin considerar aspectos, y otra donde se aplica el paradigma.
  - Está definida en función de dos factores claves de un proyecto:
    - el tamaño del producto, y
    - el esfuerzo de desarrollo.
- Considera el tamaño de una implementación sin y con aspectos, y pondera estos valores con un factor de desarrollo. Para esto último, se tomó en cuenta el factor de desarrollo propuesto por Rich Rice [5]. Este factor establece que para un sistema pequeño el tiempo de desarrollo promedio en Java es de 10 horas/programador, mientras que en AspectJ es de 2 horas/programador.

A fin de considerar el tamaño de ambas implementaciones, se evalúa el tamaño físico de los archivos de las clases y del aspecto, medidos en Kb. La métrica se define como un cociente entre los valores aplicados a ambas implementaciones. Un valor de resultado mayor que uno significa un impacto positivo de la utilización de aspectos. La definición de la métrica es:

$$Beneficio\ POA = \frac{Tamaño\_sin\_Aspectos * Factor\_de\_Desarrollo\_Java}{Tamaño\_con\_Aspectos * Factor\_de\_Desarrollo\_AspectJ}$$

donde

$$Tamaño\_Con\_Aspectos = Tamaño\_Funcionalidad\_Básica + Tamaño\_Aspecto$$

Figura 3.1 Fórmula beneficio_AOP

Para aplicar está al ejemplo mostrado en [15], se considera el tamaño físico de las clases, las cuales se especifican en la siguiente tabla:

Clases	Con aspectos	Sin aspectos
Tftpd	6,14 Kb	10,1 Kb
TftpServerConnection	5,09 Kb	6,35 Kb
TftpWriteConnection	8,49 Kb	12,1 Kb
Aspect logging	8,30 Kb	-

Tabla 3.2 Clasificación de la función del patrón en base a la capa a la que pertenecen

Luego:

$$Tamaño\_Sin\_Aspectos = (10,1 + 6,35 + 12,1)Kb = 28,55\text{Kb}$$

$$\begin{aligned} Tamaño\_Con\_Aspectos &= Tamaño\_Funcionalidad\_Básica + Tamaño\_Aspecto \\ &= (6,14 + 5,09 + 8,49)Kb + 8,30Kb \\ &= 19,72Kb + 8,30Kb = 28,02\text{Kb} \end{aligned}$$

Reemplazando estos valores en la fórmula de la métrica se obtiene:

$$Beneficio\_POA = \frac{28,55Kb * 10hp}{28,02Kb * 2hp} = \frac{285,5}{56,04} = 5,09$$

Como ya se ha mencionado, la utilización de aspectos produce un código de funcionalidad básica más "puro", debido a que el comportamiento de los conceptos entrecruzados queda encapsulado en los aspectos. Para definir el porcentaje de código que se reduce al introducir aspectos, se define la **segunda métrica, llamada *Limpieza*.** La misma, mide el porcentaje de código que se depura de la funcionalidad básica al introducir aspectos. Utiliza el tamaño de las clases, medido en Kb. La definición es como se detalla a continuación:

$$Limpieza = \frac{(Tamaño\_Sin\_Aspectos - Tamaño\_Funcionalidad\_Básica) * 100}{Tamaño\_Sin\_Aspectos}$$

Figura 3.2 Fórmula de la limpieza

Aplicando esta métrica en el caso de estudio, resulta el siguiente valor:

$$Limpieza = \frac{(28,55 - 19,72)Kb * 100}{28,55Kb} = 30.93$$

Por último se compara el tamaño de las clases que implementan las funcionalidades básicas de las dos alternativas, expresando en líneas de código (LDC):

Clases	Con aspectos	Sin aspectos
Tftpd	199 ldc	318 ldc
TftpServerConnection	176 LDC	212 LDC
TftpWriteConnection	339 LDC	471 LDC
Tamaño total	714 LDC	1001 LDC

Tabla 3.3 Comparación del tamaño de las clases en líneas de código (LDC)

Con respecto a métricas para el desarrollo orientado a aspectos, existen escasas publicaciones sobre el tema. Como se menciona en [15], Zhao propone métricas específicas para cuantificar el flujo de control en software orientado a aspectos. Las mismas se pueden utilizar para medir la complejidad de un programa orientado a aspectos. Pero a partir de los resultados obtenidos en las métricas presentadas podemos concluir que la utilización de aspectos fue ampliamente exitosa. Primero, la métrica *Beneficio AOP* arrojó el valor 5,09 que al ser mayor que 1 se traduce en un impacto positivo. Segundo, el valor 30,93 calculado por la métrica *Limpieza* indica que el código original se ha limpiado casi en un 31%. Es importante notar que la cantidad de líneas de código de la funcionalidad básica se redujo en 300 líneas de código, que representa el 30% del total, confirmando el valor obtenido en la métrica de *Limpieza*.

En relación a las pruebas efectuadas en el trabajo de investigación de esta tesis, en el capítulo de conclusiones se dan las observaciones correspondientes.

CAPÍTULO 4

4. Excepciones en Java

Los lenguajes de programación (casi todos [33]) brindan mecanismos para el manejo de excepciones, que permiten definir acciones a realizar en caso de producirse una situación anómala (excepción) durante la ejecución del programa. La acción a realizar suele consistir en una acción correctiva, o la generación de un informe acerca del error producido. Ejemplos de lenguajes con manejo o gestión de excepciones son [33]: Java, .Net, C++, Objective-C, Ada, Eiffel, y Ocaml.

Los lenguajes con manejo de excepciones incorporan en sus bibliotecas la capacidad de detectar y notificar errores. Cuando un error es detectado se siguen estas acciones:

1. Se interrumpe la ejecución del código en curso.
2. Se crea un objeto excepción que contiene información del problema. Esta acción es conocida como "lanzar una excepción".
3. Si hay un manejador de excepciones en el contexto actual se le transfiere el control. En caso contrario, pasa la referencia del objeto excepción al contexto anterior en la pila de llamadas.
4. Sino hay ningún manejador capaz de gestionar la excepción, el hilo que la generó es terminado.

Es importante usar excepciones para que el código de tratamiento o manejo del error esté separado del resto del programa. Esto aumenta la legibilidad y permite centrarse en cada tipo de código. Un mismo manejador puede gestionar las excepciones de varios ámbitos anidados. Esto reduce la cantidad de código necesaria para gestionar los errores.

El mecanismo de Java de manejo de excepciones ayuda a que las aplicaciones sean más robustas y deja, de una manera elegante, responder cuando las cosas no van como se esperaba [33]. Queremos dar especial atención a las diferencias entre ejecutar el manejo de excepciones "programáticamente" y usando la programación declarativa.

4.1 Manejo de excepciones en Java

Antes de introducirnos en la forma en que se manejan las excepciones en Java, se debe tener en mente de qué realmente ocurre cuando un método lanza una excepción. Una comprensión de los procesos que ocurren en la JVM cuando una excepción ocurre ayudará a entender acerca de la importancia de lanzar excepciones, y que sean las excepciones correctas, considerando que hay una sobre-carga adicional para la JVM para manejar una excepción [34].

En Java las excepciones son objetos que se crean cuando una condición anormal ocurre, a menudo referenciada como una condición de excepción, que se presenta durante la ejecución de una aplicación. Cuando una aplicación Java lanza una excepción, crea un objeto de alguna clase descendiente de java.lang.Throwable. La clase Throwable tiene dos subclases directas: java.lang.Error y java.lang.Exception. La figura 4.1 muestra un árbol con la jerarquía parcial para la clase Throwable.

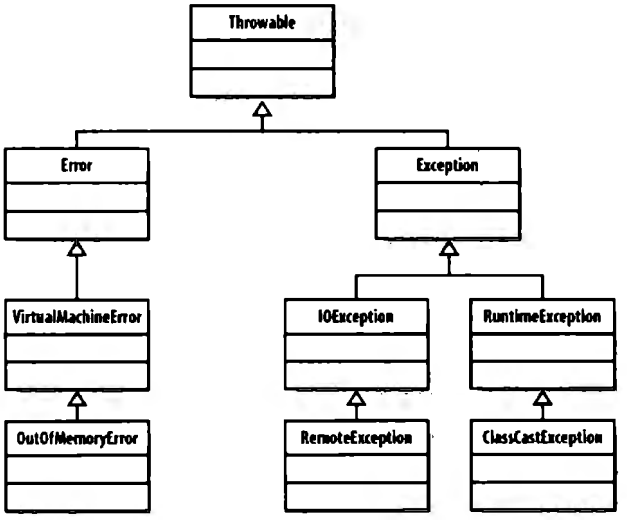

Figura 4.1 Diagrama parcial de la herencia de la clase Throwable

El espacio no permite mostrar toda la descendencia de la clase Throwable, hay más de 100 subclases directas e indirectas en los paquetes de Java. Normalmente, miembros de Exception se utilizan para indicar que condiciones anormales usualmente se puede manejar por la aplicación [34]. La otra rama de Throwable, la clase Error y sus descendientes, se reserva para problemas más serios que ocurren durante el ciclo de vida de una aplicación. Por ejemplo, sino hay más memoria disponible para una aplicación, un OutOfMemoryError

ocurrirá, y no hay nada que un cliente puede hacer acerca de eso. Por eso, los clientes generalmente no se preocupan por manipulación de las subclases de Error. En la mayoría de los casos, es la JVM misma que lanza esas instancias de Error o de sus subclases.

4.2 El método de invocación al "catch"

La JVM usa un método de invocación al "catch", también referida como una llamada "catch", para dejar huella de la sucesión de métodos invocados en cada hilo [34]. La pila tiene información local acerca de cada método que ha sido llamado y contiene la manera de retroceder o regresar al método main original de la aplicación. Cuando cada nuevo método es invocado, un nuevo *"stack frame"* es puesto en el tope de la pila, y el nuevo método se empieza a ejecutar. El estado local de cada método es también salvado dentro de cada *"stack frame"*. La figura 4.2 ilustra un ejemplo de Java *"call catch"*.

Cuando un método Java se completa normalmente, la JVM hace pop del método presente en la pila y continúa el proceso del método previo [34]. Cuando una condición de excepción ocurre, la JVM debe hallar un manejador (*"handler"*) satisfactorio de la excepción.

Figura 4.2 Ejemplo de el método de invocación al "catch"

Lo primero que se comprueba es ver si el método presente atrapa la excepción o una de sus excepciones padre. Si es así, la ejecución continuará en esa cláusula "catch". Si el

método presente no provee una cláusula de la captura para manejar la excepción, la JVM hace popping a los métodos que se tienen en la pila hasta que se halle un manejador para la excepción o una de sus excepciones padre. Eventualmente, si se hace pop a toda la pila se retrocede al método main () y si todavía no se halla un manejador para la excepción, el hilo terminará. Si ese hilo es el hilo principal y no hay ningún otro hilo (*non-deamon*) corriendo, la aplicación terminará. Si la JVM halla un manejador de la excepción a lo largo del camino (pops), ese método *"frame"* iniciará por encima de la pila y la ejecución continuará de allí.

Es importante saber cómo la JVM maneja excepciones porque hay bastante oculto debajo de una capucha cuando las excepciones ocurren en las aplicaciones. Puede ser mucho trabajo para la JVM localizar un manejador de la excepción para una excepción particular, sobre todo si se localiza lejos (abajo) del "catch". Es muy importante que se provean manejadores suficientes de las excepciones en los niveles apropiados. Si se permite que las excepciones se lancen, es probable detener su aplicación.

4.3 Excepciones "checked" y excepciones "unchecked"

En Java las excepciones son objetos que se crean cuando una condición anormal ocurre, existen dos tipos de excepciones [34]:

- "Checked". Señalan una condición anormal que el cliente debe manejar. Todas las excepciones "checked" deben ser capturadas y manejadas (verificadas) o se declaran en la cláusula de los "throws" seguidos por la firma del método. Es por eso por lo que se llama "checked". El compilador y la JVM verificarán que todas las excepciones que pueden ocurrir en un método se manejen.
- "Unchecked". Son el resultado de una lógica incorrecta o de errores en la programación. El compilador y la JVM no cuidan de si se ignoran excepciones "unchecked" (no verificadas), porque éstas son excepciones que el cliente usualmente no puede manejar. Excepciones "unchecked", tal como java.lang.ClassCastException, es típicamente el resultado de una lógica incorrecta o de errores en la programación.

La determinación de si una excepción es "checked" o "unchecked" (verificada o no verificada) se basa absolutamente en su localidad en la herencia de la excepción. Todas las clases que son descendientes de la clase java.lang.Exception, excepto las clases de RuntimeException, son excepciones "checked" o verificadas; el compilador asegurará que están o se manejan por el método o se enlistan en la cláusula de los *"throws"*.

RuntimeException y sus descendientes son excepciones "unchecked" o no verificadas, y el compilador no se quejará acerca de éstos, de no ser enlistadas en una cláusula de los *"throws"* del método o sean manejados por un bloque *"try/catch"*. Es por eso que son referenciadas como "unchecked" o no verificadas.

4.4 Manejo de excepciones programáticamente vs declarativamente

El manejo de excepciones de forma declarativa se logra vía el expresar la política del manejo de la aplicación [16], incluyendo qué excepciones son lanzadas y como deben ser manejadas, en un archivo texto (típicamente se usa XML) que es completamente externo al código de la aplicación. Esta mejora hace fácil modificar la lógica de procesamiento de las excepciones sin recompilar el código.

El manejo de excepción de forma programática es lo opuesto. Es el método tradicional, implica escribir en la aplicación especifica, la codificación de los métodos internos con el manejo de las excepciones, en lugar de simplemente modificar un archivo externo de configuración.

4.5 Buenas y malas prácticas

4.5.1 Buenas prácticas

Buenas prácticas sobre el tratamiento de excepciones [33]:

- Cada capa debe lanzar una excepción al nivel de abstracción que le corresponda. Por ejemplo, la capa de persistencia debe lanzar PersistenciaException y no SQLException. Observe que sino usamos excepciones genéricas, estamos acoplando una capa a los detalles de implementación de la otra.

- Sino hay recuperación posible de una excepción no se debe obligar a recuperarla. En este caso se debe permitir su propagación hasta que llegue como error al usuario, o alguna capa pueda realizar una acción correctiva.
- Considere envolver excepciones "checked" (verificadas) con excepciones "unchecked" (no verificadas). Para lo que se recomienda consultar "checked" vs "unchecked" acerca de cuando debemos usar una u otra.
- No hacer log si se va a relanzar la excepción. Si el sistema de manejo de excepciones de la aplicación está bien definido, será más productivo y claro dejar que lo haga por nosotros.
- No capturar excepciones y dejarlas sin tratar. Si se silencia el error, se obliga a pasar el depurador o *debugger* para descubrirlo.
- Nunca perder los *"stacktraces"* de las excepciones que se envuelvan. El *"stacktrace"* es información esencial para descubrir el problema. Hay entornos especiales (RMI, EJB, aplicaciones web) que pueden requerir medidas específicas.
- Permitir que las excepciones se propaguen entre capas.
- Declarar las RuntimeExceptions en el *"throws"* del método. Es más legible dejar explícito su uso, y ayuda a las herramientas que manipulan la clase para exportarla como servicio, aplicar AOP, generar código, etc.
- Evitar que los *"threads"* (hilos) terminen sin avisar debido a excepciones. Usa Thread.UncaughtExceptionHandler.
- Usa una excepción adecuada de la API o define una propia. Cada excepción debe reflejar un tipo de error. Al crear excepciones personalizadas podemos añadirles información acerca del contexto del error.
- Evitar que el bloque *"finally"* falle. Cuando se produce una excepción en el "try", y luego otra en el *"finally"*, es esta última la que se propaga. La anterior queda oculta.
- Evitar usar excepciones para control de flujo. La creación del objeto excepción es costosa porque se crea con una copia de todas las llamadas de la pila.
- Considerar el uso de gestores de excepción personalizados. *"Frameworks"* como *"struts"* permiten activar *"handlers"* determinados para cada excepción.

- Hacer configurable el tratamiento de excepciones. El tratamiento puede variar para cada contexto. Por ejemplo, es adecuado que una aplicación web adjunte un *"stacktrace"* en la propia página si el usuario pertenece a los grupos *"users"*, *"developers"*.

4.5.2 Malas prácticas

Como malas prácticas podríamos describir "en negativo" cada buena práctica, e ilustrar con fragmentos de código algunas de ellas [33]:

- No hacer log cada vez que se capture una excepción:

```
catch (Exception e) {
        e.printStackTrace();
        throw new WrappingException(e);
}
```

  Este código se ve en sistemas que no tienen un manejador global que permite capturar y anotar cualquier excepción antes de que llegue al cliente. "Struts" por ejemplo, tiene la clase ExceptionHandler.

- No han definido sus objetos excepción de modo que transmitan la información necesaria.
- Realizan una acción de recuperación del error y desean dejar constancia del fallo. Este es el único motivo legítimo.
- No pierda el *"stacktrace"*:

```
catch (Exception e) {
        e.printStackTrace();
        throw new WrapperException(e.getMessage());
}
```

  A partir de JDK 1.4 se pasa siempre la excepción original en el constructor de la nueva excepción. En JDK 1.3 se usa NestableException de commons-lang.

- No silencie las excepciones:

```
catch (Exception e) {
        e.printStackTrace();
}
```

  El código anterior hace log a System.err, pero el sistema es inestable y continua ejecutándose.

- No muestre mensajes inútiles:

Se ha producido un error. Consulta con el administrador

O al menos no hacerlo cuando la aplicación esta en desarrollo, o el usuario es un desarrollador. No es difícil discriminar, y se evitará mucho tiempo y dinero, hablando del costo de desarrollo.

- Evite el uso de *"throws Exception"*. Considerando el siguiente ejemplo:

```
Connection c = ...;
Statement s = null;
ResultSet rs = null;
try {
        // código JDBC
}
finally {
        if (rs != null) {
                try {
                        rs.close();
                }
                catch (Exception e) { ... }
        }
        if (s != null) {
                try {
                        s.close();
                }
                catch (Exception e) { ... }
        }
        try {
                c.close();
        }
        catch (Exception e) { ... }
}
```

Obviamente no es cuestión de escribir *"throws"* con docenas de excepciones (observar que en el ejemplo solo se ha usado *Exception*, pero hay muchas más). Tampoco vamos a complicar (más) el código relanzando en cada *"catch"*. Es perfectamente válido usar *"throws Exception"*, y relanzar luego a PersistenciaException.

Lo que no debe hacerse es no relanzar como excepción genérica: todos los métodos implicados acabarán con un *"throws Exception"*. Sabremos que algo falló, pero no se tendrá ni idea del que.

4.6 Un resumen del procesamiento de excepciones

En Java, cuando surge una condición excepcional, se crea un objeto que representa la excepción y se envía al método que provocó esta excepción. Este método puede elegir manejar la excepción él mismo o pasarla. Pero en algún punto, la excepción es capturada y procesada.

Las excepciones pueden ser generadas por el intérprete de Java o pueden ser generadas por el propio código. Las excepciones generadas por Java están relacionadas con errores que violan las reglas del lenguaje Java o las restricciones del entorno de ejecución de Java. Las excepciones generadas manualmente se suelen utilizar para informar de algún error al método llamante.

El manejo de excepciones en Java se realiza mediante cinco palabras clave: *"try, catch, throw, throws y finally"*. El funcionamiento básico es el siguiente: las sentencias del programa que se quieren controlar se incluyen en un bloque *"try"*; si se produce una excepción dentro de un bloque *"try"*, entonces esta excepción es lanzada. El programa puede capturar esta excepción, utilizando la sentencia *"catch"*, y tratarla como desee. Si se quiere generar una excepción manualmente, se utiliza la palabra clave *"throw"*. Cualquier excepción que se lanza fuera de un método debe ser especificada como tal utilizando la sentencia *"throws"*. Cualquier código que se quiera ejecutar obligatoriamente antes de que termine un método, debe introducirse en un bloque *"finally"*.

CAPÍTULO 5

5. Diseño de la solución

5.1 Problema

El problema que se plantea en esta tesis es el manejo de excepciones remotas en Java (RMI) visto como un aspecto y definiéndolo de forma declarativa para evitar *código mezclado* y *código diseminado* (ver definiciones en el punto 1.2). Su objetivo, como se ha mencionado con anterioridad es:

Proponer un método de diseño que utilice la programación orientada a aspectos y la programación declarativa para el diseño del manejo de excepciones en un sistema distribuido basado en Java RMI, y facilite el entendimiento de las técnicas mencionadas para la solución de otros problemas. El método se ilustra con una herramienta prototipo que permite aplicar aspectos y atributos para diseñar el manejo de excepciones en RMI.

La herramienta, que soporta el método de diseño mencionado permite, mediante AOP, el diseñar el manejo de las excepciones de Java RMI de forma declarativa considerándolas como aspectos. Esta combinación ayuda al entendimiento y utilización del paradigma de la programación orientada a aspectos y la programación declarativa, puesto que permite establecer, de manera fácil y clara la forma en que los elementos del sistema cooperan entre sí, logrando con ello la separación de conceptos (cada cosa esté en su sitio) y también minimizar las dependencias entre ellos (reducción del acoplamiento entre los elementos); con lo que, como ya se ha mencionado en los capítulos previos, se logra tener código menos enmarañado, más natural y más reducido, lo que facilita tanto el poder razonar sobre los conceptos así como depurar y hacer modificaciones en el código.

5.1.1 Consideraciones

En base a lo descrito en el punto anterior y retomando lo expuesto en el capítulo 1, donde se menciona la importancia que tienen las aplicaciones distribuidas colaborativas y el aprovechamiento de las mismas, se necesita contar con herramientas y mecanismos que ayuden al buen y rápido entendimiento de las tecnologías ya mencionadas en este documento.

Además se consideró que un punto de suma importancia en todos los sistemas, para su buen desempeño, es el manejo de excepciones, ya que el lanzar una excepción es la forma en que se informa que algo anormal ha ocurrido durante el procesamiento de un método; por lo que se brinda una herramienta que ayuda no sólo al desarrollo de sistemas, sino que apoya a dichos sistemas desde sus fases de análisis y diseño para poder establecer de manera fácil y clara la forma en que los elementos del sistema cooperan entre sí. Esto último visto con el paradigma de la orientación a aspectos, es decir, se consigue separar los conceptos y minimizar las dependencias entre ellos (ver punto 1.3.1); lo que nos da las ventajas vistas en la sección 1.3.1. La arquitectura base de la herramienta se muestra en la figura 1.1.

La aplicación genera el código necesario para el manejo de excepciones RMI, esto claro, para sistemas que utilicen RMI para la comunicación remota. En un archivo XML, que se describe a detalle en el apartado 5.3.1, se declara la forma en que se quiere manejar las excepciones RMI. La aplicación genera cuatro archivos con los cuales se realiza el manejo de excepciones tratadas como aspectos.

Adicionalmente la aplicación puede generar dos archivos más, uno para manejar como un aspecto el registro de las excepciones que se lanzan y el otro para tener un aspecto que hace el seguimiento de los posibles puntos de corte dentro de una aplicación. Se incluyen estos dos aspectos adicionales con fines didácticos, para un mejor entendimiento de ver los aspectos dentro de un sistema, pero debe tenerse en cuenta que al generarlos puede suceder que la aplicación se vea afectada en cuestión de rendimiento por lo que se recomienda entonces generarlos inicialmente para fines de comprensión pero posteriormente evitarlo, lo cual se declara también en el archivo XML.

Por último, después de haber generado el aspecto de manejo de excepciones, y los aspectos adicionales si se requiere o desea, solo basta con compilar la aplicación a la cual

se le quiere agregar el manejo de excepciones pero ya con el compilador AspectJ; claro considerando el código que maneja la funcionalidad básica del sistema y el código generado por la herramienta presentada.

Esquemáticamente podemos decir que la entrada a la herramienta es el archivo de configuración (archivo XML); el proceso dará como salida código para AspectJ que junto que con el código de la aplicación en cuestión (código con funcionalidad básica) se podrá compilar con AspectJ y así tener código ejecutable con el manejo de excepciones deseado. Lo anterior se muestra esquemáticamente en la figura 5.1.

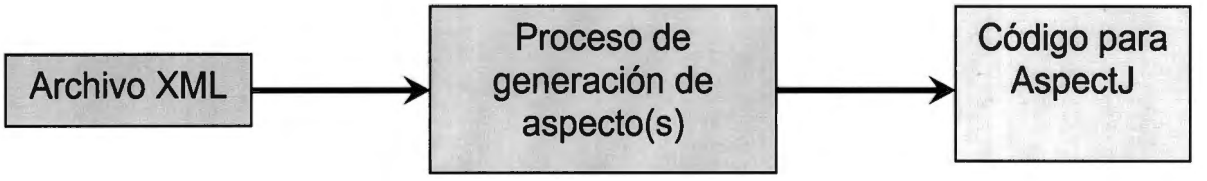

Figura 5.1 Esquema de entrada y salida de la herramienta

5.1.2 Pasos del método

Antes de describir a profundidad la herramienta desarrollada y como introducción al método propuesto, en este punto, se dan los pasos necesarios para poder generar el manejo de excepciones que se quiera para cualquier aplicación basada en Java RMI (ver el objetivo principal de la tesis, punto 1.3.1).

Los pasos del método propuesto son:

1. Tener una lista de las diferentes excepciones de Java RMI a contemplar en el manejo de excepciones que se generará con la herramienta desarrollada (en el Apéndice D, se tiene una lista de todas las excepciones de Java RMI con una breve descripción).
2. Definir los puntos de corte que se van a considerar (ver punto 5.3.1 Lectura del archivo XML). Para cada punto de corte que se establecerá en el archivo XML:
    a. Declarar el o los join points. Puntos en el código base donde se quiere que se agregue el manejo de excepciones, en específico son los métodos o funciones que se quieren considerar para los bloques try/catch que se generarán. Un join point contiene:
        i. Tipo de retorno del método o función.

ii. Nombre de la clase que contiene el método o función.
iii. Nombre del método o función.
iv. Argumentos del método o función.

Para más detalle del formato del archivo XML ver el punto 5.3.1 y para información de la sintaxis de AspectJ el Apéndice B.

b. Definir las excepciones que son particulares para ese punto de corte y definirlas como excepciones internas (punto 5.3.1), de forma general ésta definición consta de:
   i. Datos: nombre y descripción.
   ii. Para el manejo de la excepción se puede manejar alguna de las combinaciones de las opciones siguientes:
      a. Ejecutar. Consta del nombre del método a ejecutar, tipo de retorno y sus parámetros necesarios.
      b. Configurar. Se declara si hace reintentos, indicando cuántos, y si se quiere que espere entre cada intento (tiempo en milisegundos), además se le puede indicar qué excepción lanzar, es decir, se puede cambiar la excepción original y/o se puede indicar que se encapsule la excepción en alguna otra excepción (que debe ser tipo RuntimeException para que el compilador permita la compilación correspondiente).
c. Para excepciones en donde su manejo es similar en más de un punto de corte, se pueden definir como referencias a excepciones, es decir, dentro del punto de corte sólo se pondrá para la excepción correspondiente el campo referencia. Y después de la definición de los puntos de corte se tendrán las definiciones de las referencias a excepciones. Es importante mencionar que para todas las referencias a excepciones se valida su existencia al terminal de leer el archivo XML. De forma general la definición de una referencia a excepción consta de:
   i. Nombre de la referencia.
   ii. Datos de la excepción: nombre y descripción.

iii. Y al igual que el punto 2.b.ii se puede definir su configuración y/o algún método a ejecutar.

d. Para aquellas excepciones que el manejo deseado es el mismo, se pueden definir grupos de excepciones, para las cuales se deben de hacer la referencia en el o los puntos de corte que se requiera, es decir, un grupo de excepciones puede ser compartido por los puntos de corte. Es importante mencionar que para todas las referencias a grupos de excepciones se valida su existencia al terminal de leer el archivo XML. De forma general la definición de una referencia a un grupo de excepciones consta de:
   i. Datos del grupo: nombre y descripción.
   ii. Excepciones. Conjunto de excepciones, para las cuales la información requerida (datos) es: nombre y descripción.
   iii. Y su configuración, como se menciona en el punto 2.b.ii de esta lista.

e. Para el caso de grupos de excepciones que correspondan a un mismo paquete de Java RMI (java.rmi, java.rmi.activation y java.rmi.server) se puede optar por definir una referencia a paquete de excepciones y además puede ser compartido por varios puntos de corte. Es importante mencionar que para todas las referencias a paquetes de excepciones se valida su existencia al terminal de leer el archivo XML. De forma general la definición de una referencia a paquete de excepciones consta de:
   i. Nombre de la referencia al paquete.
   ii. Datos: nombre del paquete Java RMI que se quiere considerar y su descripción.
   iii. Su configuración, como se menciona en el punto 2.b.ii de esta lista.

Todas las excepciones que se definan, ya sea como excepciones internas para un punto de corte, como referencia a excepciones, como referencia a grupos de excepciones y como referencia a paquetes de Java RMI se validan al leer el

archivo XML en base al archivo de configuración excepciones_validas_RMI.xml y su esquema correspondiente, ver el punto 5.3.2.1.

3. Una vez que se ha declarado el manejo deseado en el archivo XML, ejecutar la herramienta que se brinda, para más detalle ver el punto 5.3.4. Los parámetros necesarios y su descripción se tienen en el punto 5.3.4.1. Así como la descripción del log que se genera en el punto 5.3.3.
4. Con el código generado y el código base de la aplicación, se necesita compilar con AspectJ para obtener el código ejecutable final, ver el punto 5.3.4.2.

5.2 Diseño de alto nivel

Para el diseño de la solución se considera que los aspectos distribuidos implementan básicamente acceso remoto a sistemas usando RMI y al manejo de excepciones; por lo que podemos dividir (sin perder generalidad) los sistemas distribuidos en: el servidor, el cliente y el manejo de excepciones. Con base a esta estructura propuesta, el diseño de alto nivel se muestra con el diagrama de la figura 5.2; en donde se marca como los aspectos que comparten tanto el cliente como el servidor se pueden englobar en lo que se ha llamado "*Módulo de aspectos comunes*", en el cual se podrán tener aspectos como por ejemplo: manejo de bitácoras, integridad de la base de datos o manejo de excepciones, entre otros.

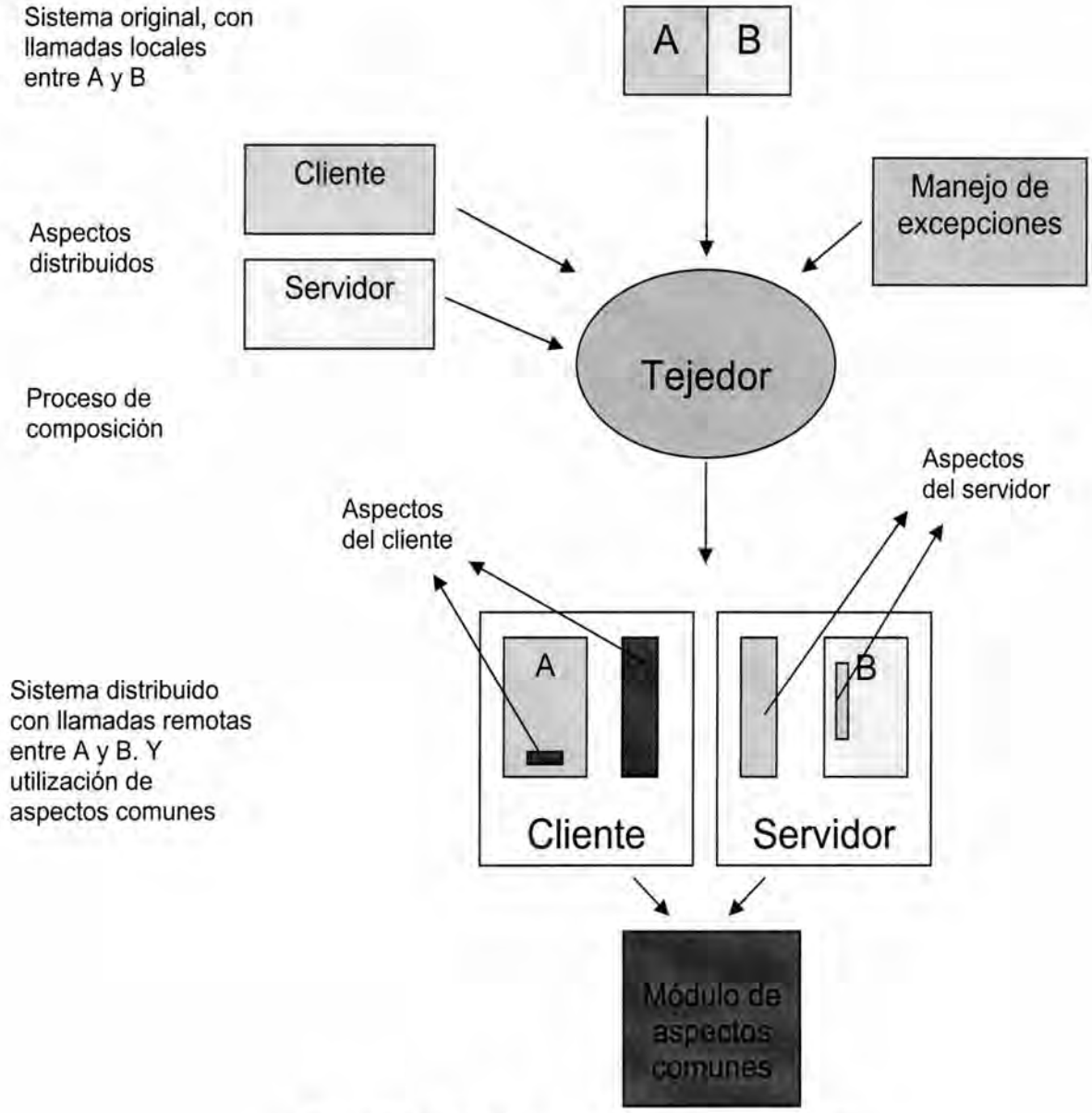

Figura 5.2 Diseño de alto nivel de la solución propuesta

Note que los aspectos que se consideran viables para el módulo de aspectos comunes, son un cruce de asuntos natural, usualmente implementado con código repetido y que no pertenece a la funcionalidad básica o principal del sistema en cuestión.

En la figura 5.3 se presenta un ejemplo de los aspectos que podrían formar parte del módulo mencionado, entre ellos se tiene claro, el manejo de excepciones; en la figura se muestra el cruce de esos aspectos con posibles clases que implementan la funcionalidad básica y conforman al servidor y/o al cliente.

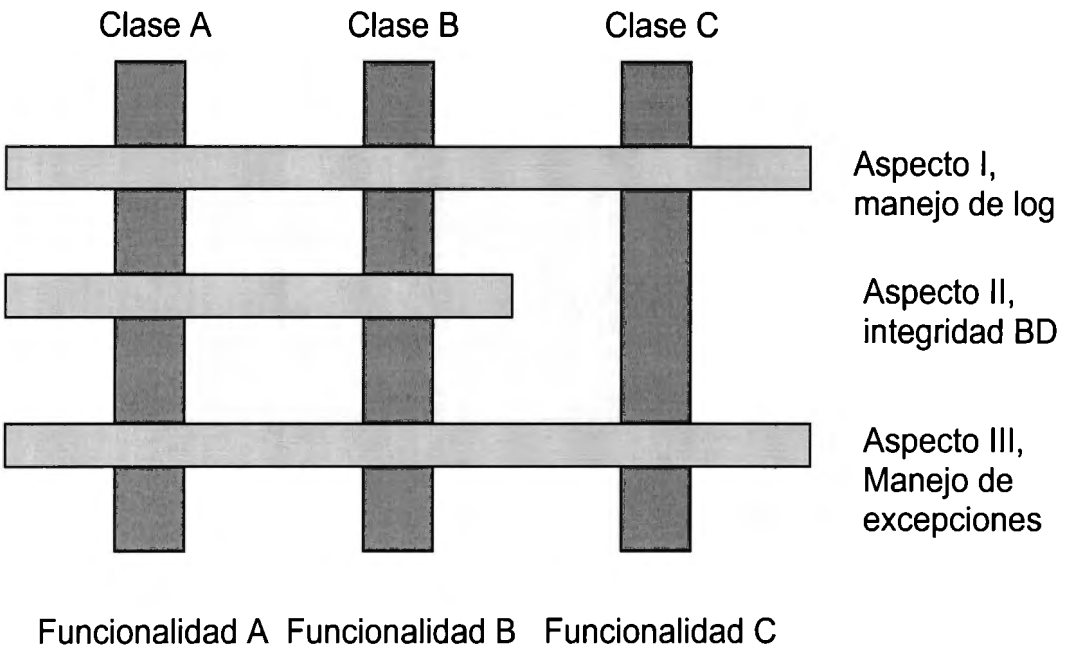

Figura 5.3 Funcionalidad básica y cruce de crosscutting concern's (aspectos)

Como se puede ver con el diseño mostrado, se busca que la herramienta permita manejar de forma "separada" a la aplicación lo relacionado al manejo de excepciones RMI, es decir, separar la parte de manejo de excepciones de la funcionalidad básica de la aplicación.

En base a la arquitectura base mostrada en la figura 1.1, al esquema de entrada y salida (figura 5.1) y al diseño de alto nivel que se tiene en la figura 5.2, la solución planteada considera varias partes o componentes del diseño, las cuales a continuación se numeran y se describen:

1. Códigos de Java-RMI. Código Java, que contendrá la funcionalidad básica de la aplicación distribuida basada en RMI, a la cual se le quiere agregar el manejo de excepciones.
2. Archivo XML. Archivo en el que se declara la forma para implementar el manejo de excepciones de la aplicación distribuida.
3. Proceso de generación. Se genera código para ser compilado con AspectJ, en base a la información del archivo XML utilizando el patrón de diseño para el manejo de excepciones. Este patrón se describe más adelante.

4. Aspecto(s) generados. Los aspectos diseñan el manejo de excepciones correspondiente.

Este diseño se deriva de la experiencia, no sólo de investigar, sino de implementar varios aspectos y el análisis relacionado al manejo de excepciones remotas en los sistemas distribuidos.

5.2.1 Patrón para el manejo de excepciones

En [6] se presenta un patrón llamado "***the exception introduction pattern***", el cual se utilizó como base para el desarrollo de la herramienta propuesta. A continuación se dan algunas descripciones importantes para la incorporación del patrón a la herramienta.

Cuando se usan aspectos para introducir cruce de asuntos en un sistema, a menudo es necesario lanzar una excepción "checked" o verificada (como lo es una excepción de tipo RMI) que no forma parte de la lista original de excepciones declaradas para ese o esos puntos de unión.

En nuestro caso se desea mantener intacto el código para aplicaciones existentes y para las aplicaciones nuevas no tener que definir todas las excepciones RMI que pueden ser lanzadas. Con este patrón podemos agregar las opciones de manejo de excepciones para el cruce de asuntos de una forma consistente y sistemática.

Como se menciona en el apéndice B, AspectJ no proporciona ningún aviso para declarar que posiblemente se va a lanzar una excepción "checked" o verificada (una excepción que extiende directamente de Exception) a menos que el aviso del punto de corte tenga declarado que se lanzará esa excepción. De modo que en muchas ocasiones, el tejer los asuntos dentro de un sistema requiere que un aspecto trate con nuevas excepciones "checked" o verificadas.

Debido a que el aviso no puede declarar que posiblemente lanzará esas excepciones, se necesita otra forma de tratar con ellas. Además, ciertos tipos de cruce de asuntos, como es el caso de la recuperación de errores y el manejo de transacciones, requieren que los aspectos capturen las excepciones de todos los tipos que se lancen por los puntos de unión en bloques "catch".

De modo que, ¿qué debe hacer el aspecto con la excepción después de ejecutar la lógica del asunto específico en el bloque "*catch*"? Este es un problema complejo donde los aspectos son reutilizables y no saben acerca de las excepciones específicas del negocio.

Si consideramos que el aspecto que maneja las excepciones de RMI introduce un cruce de asuntos y que las excepciones RMI son del tipo "checked" o verificadas, entonces sería necesario que dichas excepciones se consideraran en cada uno de los "*throws*" de los métodos remotos correspondientes y en cada "*catch*" de las llamadas a esos métodos. Para evitar esto, lo que hacemos es capturar la excepción original; ejecutar alguna lógica (si se desea), y lanzar entonces una excepción del tipo *runtime* que envuelva a la excepción "checked" (o verificada).

Necesitamos declarar una excepción *runtime* propia que contenga una excepción "checked" (o verificada) como la causa. Esta excepción puede ser usada como argumento del constructor de la clase particular de excepción.

El aspecto a crear capturará todas las excepciones que pueden ser lanzadas por los métodos que capturan los puntos de corte. Es decir, necesitamos un "aviso durante" que procede con la captura de la operación en un bloque *"try/catch"*. Cuando el bloque *"catch"* es ejecutado, se ejecuta la lógica específica del manejador de excepciones y lanza una nueva excepción *runtime* propia que envuelve a la excepción capturada.

El aspecto deberá encapsular la nueva excepción en el lado del servidor y manejarla en el lado del cliente, manteniendo el desempeño esperado y usando los mensajes necesarios para la interfaz correspondiente.

Para poder tener el manejo de la excepción en el cliente, tenemos que enfrentar el reto de preservar la especificación de los requerimientos de la excepción en los métodos que son afectados por la implementación del asunto en cuestión, mientras se permite que el aspecto ejecute su lógica capturando todos los tipos de excepciones.

Agregaremos otro aspecto al sistema, aspecto PreserveRemoteException, para restaurar la excepción "checked" (verificada) lanzada originalmente. La figura 5.5 muestra la implementación del aspecto que restaura la excepción original cuando es capturada una excepción *runtime* propia, con lo que los métodos que llamen a los métodos remotos generadores de la excepción, toman la excepción que saben manejar o están preparados para hacerlo y no necesitarán modificar su lógica de proceso de excepciones.

Se agrega al patrón base el obtener la excepción original dentro del "aviso durante" (ver la función QuitaWrapper de la clase ExHanAspAbstract), con lo que ahí se puede manejar la excepción original y llevar acabo el manejo correspondiente. Si después de ello se desea enviar una excepción encapsulada, dependiendo el manejo declarado, se puede encapsular con la excepción *runtime* propia, o inclusive se le puede decir que encapsule una excepción de otro tipo, y después de ello el aspecto PreserveRemoteException trabajará normalmente según el patrón.

5.2.1.1 Resumen del patrón para el manejo de excepciones

El patrón simplemente sugiere que se capture la excepción original, se ejecute cualquier lógica, y se lance una excepción *runtime* propia que envuelve la excepción "checked" (verificada). Necesitamos declarar una excepción *runtime* propia del asunto específico que contiene una excepción "checked" (verificada) como la causa (que se le puede pasar al constructor).

Mientras lanzamos una excepción *runtime* propia, propagamos la excepción a un nivel alto a los llamadores, esto causará un problema a los llamadores, que no conocen de los aspectos presentes en el sistema, esto es, no están preparados para capturar la excepción "unchecked" (no verificada).

El patrón maneja de una forma simple y sistemática la necesidad de usar una excepción *runtime* propia del asunto, en lugar de usar excepciones "checked" (verificadas).

El patrón también maneja el problema que se origina cuando se quiere implementar un cruce de asuntos, para rápidamente capturar todos los tipos de excepciones, incluyendo las del negocio.

Como modificación al patrón se agregó el tomar la excepción original, que es la causa de la excepción propia capturada, desde el "aviso durante", con lo que se puede realizar el manejo correspondiente.

Si se genera una nueva excepción o por la declaración del archivo XML, se puede relanzar esa excepción o una nueva; esto claro utilizando nuevamente el encapsularla con la excepción *runtime* propia declarada para estos fines.

5.2.2 Las 4 clases, para el manejo de excepciones

Como se ha descrito, para cuando se tiene que manejar excepciones específicas de un asunto en la implementación de cruce de asuntos, se presenta una forma simple y sistemática que usa excepción *runtime* específica en lugar de usar excepciones "checked" (o verificadas).

En los aspectos ExHanAspAbstract y ExHanAspect, que son donde realmente se tiene el manejo que se declara en el archivo XML, se van agregando, al momento de ir creándolos, comentarios que indican de una forma clara lo que se realiza y a lo que corresponde del archivo XML; dichos comentarios inician con los caracteres // *** , ver la figura 5.5.

A continuación se describen los tres aspectos, uno de ellos abstracto, así como la excepción de tipo *runtime* que se crean con la herramienta para con ellos realizar el manejo de excepciones según se declare en el archivo XML correspondiente.

5.2.2.1 La clase ExHanAspRuntimeException

Es la clase en la cual se define la excepción propia, que extiende de RuntimeException, y que se usará para envolver la excepción original, la cual se le pasa como argumento al constructor.

Esta clase siempre se genera igual, es decir, no depende de lo declarado en el archivo XML; en la figura 5.4 se tiene la definición de la clase.

```
public class ExHanAspRuntimeException extends RuntimeException {
    public ExHanAspRuntimeException(Throwable cause) {
        super(cause);
    } // constructor
} // class
```

Figura 5.4 Clase ExHanAspRuntimeException

5.2.2.2 La clase PreserveRemoteException

Es la clase con la cual se quita la envoltura a la excepción, para que el cliente vea la excepción original. Formalmente es la declaración de un nuevo aspecto[1] al cual se le da

[1] En el Apéndice B se describe la sintaxis de aspect.

precedencia sobre el aspecto que tiene el manejo de excepciones y define un "advice after throwing", ver Apéndice B para más detalle.

El quitar la envoltura lo hace para cada excepción del tipo RMI valida, la lista de excepciones RMI que se consideran válidas se tiene en el archivo XML llamado excepciones_validas_RMI.xml, ver punto 5.3.2.1.

En la figura 5.5 se tiene un ejemplo de este archivo. Nótese que solo se muestran tres avisos después (*"afters"*), en realidad el archivo contendrá el número de avisos según el número de excepciones que se tengan declaradas en el archivo XML mencionado[2].

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.activation.*;

public aspect PreserveRemoteException {
    declare precedence: PreserveRemoteException , ExHanAspect;

    after() throwing(ExHanAspRuntimeException ex) throws AccessException
    : call(* *.*(..) throws AccessException) {
        Throwable cause = ex.getCause();
        if (cause instanceof AccessException) {
            System.out.println("**** Preserve " + cause);
            throw (AccessException)cause;
        }
        throw ex;
    }

    after() throwing(ExHanAspRuntimeException ex) throws AlreadyBoundException
    : call(* *.*(..) throws AlreadyBoundException) {
        Throwable cause = ex.getCause();
        if (cause instanceof AlreadyBoundException) {
            System.out.println("**** Preserve " + cause);
            throw (AlreadyBoundException)cause;
        }
        throw ex;
    }

    after() throwing(ExHanAspRuntimeException ex) throws ActivateFailedException
    : call(* *.*(..) throws ActivateFailedException) {
        Throwable cause = ex.getCause();
        if (cause instanceof ActivateFailedException) {
            System.out.println("**** Preserve " + cause);
            throw (ActivateFailedException)cause;
        }
        throw ex;
    }

}  // aspect
```

Figura 5.5 Clase PreserveRemoteException

[2] En el Apéndice E se presenta el archivo con las 26 definiciones.

5.2.2.3 La clase ExHanAspect

Es la clase en la cual se definen los diferentes puntos de corte donde se quiere agregar el aspecto de manejo de excepciones. Este archivo cambia dependiendo de lo que se declare en el archivo XML. A manera de ejemplo, en la figura 5.6 se tiene una definición de la clase, en la cual se observa que la clase extiende de ExHanAspAbstract, que se describe en el siguiente apartado; el ejemplo muestra dos puntos de corte[3]:

- puntoCorte_1. Se declara para todas las llamadas al método Pba_throw de la clase MiInterfazRemoto sin importar el tipo de retorno ni los parámetros del método, así como a las llamadas al método Pba_throw_3 de la clase MiInterfazRemoto, sin importar el tipo de retorno pero solo para cuando se le da como parámetro un int.
- puntoCorte_2. Se declara para las llamadas al método Pba_throw_2 de la clase MiInterfazRemoto donde el tipo de retorno sea un *void* y los parámetros del método correspondan a un *String* y a un *boolean*.

```
import java.rmi.RemoteException;

public aspect ExHanAspect extends ExHanAspAbstract {

    pointcut puntoCorte_1() :
        // *** descripcion del punto de corte 1
        call(* MiInterfazRemoto.Pba_throw(..))
        || call(* MiInterfazRemoto.Pba_throw_3(int));

    pointcut puntoCorte_2() :
        // *** descripcion del punto de corte 2
        call(void MiInterfazRemoto.Pba_throw_2(String, boolean));

}  // aspect
```

Figura 5.6 Clase ExHanAspect

Como se puede ver en la figura 5.6, se incluyen comentarios (con el formato // ***), que en este caso sólo se tienen para la descripción de los puntos de corte declarados. Esta información se toma del XML.

[3] Definidos en el archivo XML

5.2.2.4 La clase ExHanAspAbstract

Es la clase en la cual se definen para los diferentes puntos de corte, su "aviso durante" (*"advice around"*), en el cual se hace el manejo de excepciones que corresponde a lo declarado en el archivo XML de configuración.

El listado[4] de ejemplo se muestra en el Apéndice E, y se puede ver que se incluyen comentarios, para todos los elementos y atributos declarados en el XML, con lo que se ayuda a comprender mejor los archivos generados.

Para un entendimiento rápido y correcto del mismo, se recomienda ver el punto 5.3.1, para detectar su correspondencia con la información declarada en el XML.

5.2.3 Los dos aspectos adicionales

Como se mencionó al inicio de este capítulo, la aplicación permite que además de los cuatro archivos necesarios para el manejo de excepciones se puedan generar dos aspectos más, cada uno de ellos en una clase independiente.

El primero de ellos en un aspecto que permite tener el registro de las excepciones generadas y el segundo permite hacer un seguimiento de los puntos de corte de una aplicación.

Ambos facilitan el entendimiento de ver los aspectos dentro de un sistema, pero el generarlos puede hacer que la aplicación se vea muy afectada en cuestión de rendimiento por lo que se recomienda entonces generarlos inicialmente para fines de comprensión pero posteriormente evitarlo[5].

Los dos aspectos se muestran en las figuras 5.7 y 5.8 respectivamente.

[4] Este archivo cambia dependiendo lo que se declare en el archivo XML de configuración.
[5] Lo cual se declara en el archivo XML de configuración.

```
import java.util.logging.*;
import org.aspectj.lang.*;

public aspect AspLogExc {
    Logger _logger = Logger.getLogger("exceptions");

    AspLogExc() {
        _logger.setLevel(Level.ALL);
    }

    pointcut exceptionLogMethods() : call(* *.*(..)) && !within(AspLogExc);

    after() throwing(Throwable ex) : exceptionLogMethods() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        if (_logger.isLoggable(Level.WARNING)) {
            //Signature sig = thisJoinPointStaticPart.getSignature();
            _logger.logp(Level.WARNING,
                    sig.getDeclaringType().getName(),
                    sig.getName(),"AspLogExc Exception logger aspect", ex);
        } else  {
            // solo la imprimo
        System.err.println("AspLogExc Exception logger aspect ["
                + sig.getDeclaringType().getName() + "."
                + sig.getName() + "]");
        ex.printStackTrace(System.err);
        }  // else
    }  // after

}  // aspect
```

Figura 5.7 Aspecto AspLogExc

```
public aspect AspSegJoinPoints {

    private int _callDepth = -1;

    pointcut tracePoints() : call (* *.*.*(..)) && !within(AspSegJoinPoints);

    before() : tracePoints() {
        _callDepth++;
        print("Before", thisJoinPoint);
    }  // before

    after() : tracePoints() {
        print("After", thisJoinPoint);
        _callDepth--;
    } // after

    private void print(String prefix, Object message) {
        for(int i = 0, spaces = _callDepth * 2; i < spaces; i++) {
            System.out.print(" ");
        }  // for
        System.out.println(prefix + ": " + message);
    }// print
}// aspect
```

Figura 5.8 Aspecto AspSegJoinPoints

5.3 Aplicación desarrollada

La aplicación la podemos dividir en dos partes, para una forma más fácil de explicarla y manejarla:

- La primera es la lectura de la configuración deseada, lo cual se realiza mediante la lectura de un archivo XML, utilizando un esquema para sus validaciones y Xerces para leerlo; creando una estructura de objetos con la información correspondiente.
- La segunda parte, es la creación de los archivos necesarios para tener el manejo de excepciones como un aspecto dentro de la aplicación correspondiente.

La aplicación se encuentra en el directorio Man_Exc_Asp_XML, la estructura de subdirectorios se muestra en la figura 5.9.

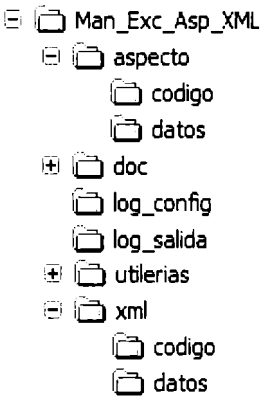

Figura 5.9 Estructura de subdirectorios de la aplicación

En el subdirectorio "*aspecto*", dentro de "*codigo*", se tienen el código fuente de las clases necesarias para la generación de los aspectos y los archivos .class. En "*datos*" se tiene un archivo XML y su esquema, que son para la lista de excepciones RMI válidas, se recomienda no modificar estos archivos a menos que se tenga una nueva excepción RMI.

En el subdirectorio "*doc*", se encuentra la documentación generada con Javadoc, es decir, la información de las clases y paquetes que conforman la aplicación en formato HTML.

En los subdirectorio *"log_config"* y *"log_salida"*, se tienen los archivos con las configuraciones posibles para el archivo de log (archivo de salida tipo texto y tipo XML), que se maneja con log4j, y el archivo con el log generado[6].

En el subdirectorio "*utilerias*", se tienen en código fuente, las clases y archivos necesarios para diferentes tareas dentro de la aplicación, como por ejemplo la interfaz donde se declaran las constantes del sistema y los códigos de errores posibles que se pueden generar[7].

Por último, en el subdirectorio "*XML*", se tiene la misma estructura que en aspecto, es decir un directorio con los fuentes y los archivos .class (subdirectorio "*codigo*") y uno con los datos (subdirectorio "*datos*") que es donde se encuentra el archivo XML donde se debe declarar el manejo de excepciones que se quiere generar; en el apartado siguiente se describe este proceso.

5.3.1 Lectura del archivo XML

El archivo que contiene la configuración es *excepciones.xml*, se encuentra dentro del subdirectorio *"datos"* del directorio *"XML"*, y el archivo con el esquema para validarlo es *esquema.xsd*, que se encuentra en el mismo subdirectorio, a continuación se describen estos dos archivos y las validaciones más importantes.

Para el archivo *excepciones.XML*, del cual se tiene un ejemplo en el Apéndice E, la estructura del cuerpo del documento es:

- Elemento ManejoExcepciones, en el cual se tiene una secuencia de elementos, los cuales describen los diferentes pointCuts, las referencias a excepciones, las referencias a grupos, las referencias a paquetes y el *default*.

  Como atributos de ManejoExcepciones se tienen: nombre (para fines informativos), ruta_salida donde se escribirán los archivos generados, genera_aspecto_logging (bandera con la que se indica si se quiere generar el aspecto de log de excepciones) y genera_aspecto_seguimiento_PC (bandera con

[6] Lo relacionado al log se describe a detalle en la sección 5.3.3.
[7] En el punto 5.3.2.2 se describe esta estructura.

la que de indica si se quiere generar el aspecto de seguimiento de puntos de corte).

- Los poinCuts contienen una serie de elementos que son: datos (nombre y descripción), clausula (del tipoClausula, que se describe a continuación), excepciones[8], referencias a grupos de excepciones y referencias a paquetes de excepciones[9].

  Como atributo se tiene "advice" (el cual es validado que sea el *string* "*call*", esto por la forma que se necesita declarar el punto de corte para el patrón utilizado como base en el manejo de excepciones, ver el Apéndice E).

- Tipo tipoClausula, contiene una serie de "joinPoints", que básicamente es una serie de elementos "joinPoint"; para los cuales tenemos dos formas de definirlos:
  - La primera, es definir cada parámetro, dentro de una secuencia de elementos parámetro (con los campos: nombre, tipo, valor, etc.).
  - O definir un *String* en el cual se tienen definidos los parámetros necesarios para poder definir la cláusula.

  Como atributos de "joinPoint" tenemos: retorno (tipo de retorno del método), clase (donde se encuentra el método), metodo (método a ejecutar).

- Para las excepciones, se pueden definir de dos formas:
  - Definición de excepción interna, es decir, solo es vista por el pointCut donde se declara y contendrá los elementos:
    - tipoDatos[10], obligatorio.
    - tipoConfigura[11], opcional.
    - tipoEjecutar[12], opcional.
  - Definir una referencia a excepción, la cual se encontrará declara fuera del los pointCuts y será vista por todos los puntos de corte, es decir, se

[8] Las cuales pueden ser declaradas en este punto (solo para el pointcut, o como tipoReferenciaExcepcion con lo cual pueden ser utilizadas por otros pointCuts.
[9] Estas dos últimas, se describen en tipoReferencia.
[10] Se describe más adelante, en esta misma sección.
[11] Se describe más adelante, en esta misma sección.
[12] Se describe en esta misma sección.

podrá utilizar desde cualquier pointCut que haga la referencia correspondiente, ver tipoReferenciaExcepción.

- El tipoConfigura contiene los atributos: intentar (con el que se indica cuantas veces se quiere que se haga un reintento después de haber cachado la excepción correspondiente, es decir para hacer un while) su declaración es opcional, delay (con el que se indica cuanto tiempo esperar entre reintento y reintento, es opcional), lanzar (se indica que excepción lanzar, en lugar de la original, también es opcional) y encapsular (con lo que se le indica si se desea encapsular o no la excepción original, es opcional su declaración).
- El tipoEjecutar, es para indicarle que se quiere ejecutar otro método, se le tiene que definir su secuencia de parámetros (opcionales).

  Tiene como atributos: método (nombre del método a ejecutar, debe incluir el nombre de la clase, es decir la sintaxis es clase.metodo) obligatorio y retorno (tipo de retorno que se espera del método ejecutado) obligatorio.
- El tipoReferenciaExcepcion contiene lo necesario para saber que hacer para la excepción correspondiente, sus elementos son: referencia (que contiene el nombre con el cual se hace la referencia desde algún o algunos pointCuts) obligatorio, datos (nombre y descripción) obligatorio, su configuración (indicando si se quiere hacer reintentos y/o delay, si se lanza otro tipo de excepción, o si se quiere hacer un encapsulamiento) opcional y ejecutar (información del método a ejecutar) opcional.
- El tipoReferenciaGrupo, es muy similar al tipoReferenciaExcepción, pues sólo tiene de más el conjunto de excepciones que conforman el grupo y se declaran dentro del elemento excepciones y cada excepción se define con un elemento datos, es decir, con su nombre y su descripción.
- El tipoReferenciasPaquetes, al igual que el tipoReferenciaGrupo, contiene la información del tipoReferenciaExcepcion, pero en lugar de un conjunto de excepciones se tiene en el campo de nombre del elemento datos el nombre del paquete de excepciones RMI, es decir, son validos los paquetes: Java.rmi, Java.rmi.activation y Java.rmi.server.

Las validaciones que se realizan con el esquema son:

- Para ManejoExcepciones:
    - Los elementos excepción, grupo y paquete pueden o no existir, sin límite de ocurrencias.
    - Para pointCut debe existir al menos una ocurrencia.
    - El elemento default es obligatorio y sólo una ocurrencia.
    - Sus atributos son obligatorios.
- Para tipoPointCut:
    - Los elementos datos y cláusula son obligatorios y solo una ocurrencia.
    - Los elementos excepción, grupo_ref y paquete_ref son opcionales y si limite de ocurrencias.
    - Su único atributo, "advice", es obligatorio y esta restringido al "*String cal*".
- Para tipoClausula:
    - Se conforma del elemento joinPoints, que es obligatorio y sólo una ocurrencia.
- Para tipoJoinPoints:
    - Secuencia de elementos joinPoint, al menos una ocurrencia y sin límite en las mismas.
- Para tipoJoinPoint:
    - Se tiene la opción de definir el grupo de losParametros o un elemento parametro_str del tipoReferencia.
    - Sus atributos son obligatorios.
- Para el grupo losParametros:
    - Es una secuencia de elementos tipoParametros, es obligatorio y sólo una ocurrencia.
- Para tipoDatos:
    - Contiene dos atributos obligatorios y de tipo *String*, que son nombre y descripción.
- Para tipoReferencia:
    - Contiene un sólo atributo *String* que es obligatorio.

- Para tipoReferenciaExcepcion:
    - Se tiene la opción de definir el grupo de campos o un elemento referencia del tipoReferencia.
- Para el grupo campos:
    - Es una secuencia de elementos: tipoDatos (obligatorio y sólo una ocurrencia), tipoConfigura (opcional y máximo una ocurrencia) y tipoEjecutar (opcional y máximo una ocurrencia).
- Para tipoExcepcion:
    - Secuencia de elementos: tipoReferencia (obligatoria y máximo una ocurrencia), tipoDatos (obligatorio y máximo una ocurrencia), tipoConfigura (opcional y máximo una ocurrencia) y tipoEjecutar (opcional y máximo una ocurrencia).
- Para tipoConfigura:
    - Tiene cuatro atributos obligatorios de tipo *String*.
- Para tipoEjecuta:
    - Puede contener un elemento tipoParametros.
    - Tiene dos atributos obligatorios de tipo *String*.
- Para tipoParametros:
    - Secuencia de elementos: tipoParametro (obligatoria y sin máximo en número de ocurrencias.
- Para tipoParametro:
    - Contiene cuatro atributos, de los cuales dos son obligatorios y dos opcionales.
- Para tipoGrupo:
    - Secuencia de elementos: tipoDatos (obligatoria y máximo una ocurrencia), tipoConfigura (opcional y máximo una ocurrencia), tipoExcepcionesGrupo (obligatoria y máximo una ocurrencia) y tipoEjecutar (opcional y máximo una ocurrencia).
- Para tipoExcepcionGrupo:
    - Secuencia de elementos tipoDatos, obligatoria y sin máximo de ocurrencias.

- Para tipoPaquete:
    - Secuencia de elementos: tipoReferencia (obligatoria y máximo una ocurrencia), tipoDatos (obligatorio y máximo una ocurrencia), tipoConfigura (opcional y máximo una ocurrencia) y tipoEjecutar (opcional y máximo una ocurrencia).
- Para tipoDefault:
    - Secuencia de elementos: tipoConfigura (opcional y máximo una ocurrencia) y tipoEjecutar (opcional y máximo una ocurrencia).

5.3.1.1 Creación de estructuraXML

La lectura del archivo XML lleva a obtener una estructura de objetos con la información correspondiente y así poder generar los aspectos que contendrán el manejo de excepciones, el código fuente para esto se encuentra en el subdirectorio *"codigo"* del subdirectorio XML, las clases que ahí se tiene se muestran en la tabla 5.1, con una pequeña descripción.

LeeXML.Java	Clase principal para generar la estructura con la información del manejo de excepciones, se lee el archivo XML/datos/excepciones.XML que a la ves es validado con el archivo XML/datos/esquema.xsd por medio de xcerces
estructuraXML.Java	Clase en la cual se tiene la información para el manejo de excepciones, se crea en la clase LeeXML
pointCut.Java	Clase en la cual se tiene la información para cada pointcut declarado en el XML
excepcion.Java	Clase en la cual se tiene la información para cada excepción (referencia a excepción) declarada en el XML
grupo.Java	Clase en la cual se tiene la información para cada grupo de excepciones (referencia a grupo) declarada en el XML
paquete.Java	Clase en la cual se tiene la información para cada paquete de excepciones (referencia a paquete) declarada en el XML
Default.Java	Clase con la cual se tiene lo declarado como manejo de default para las excepciones, es decir, la configuración para esas excepciones
clausula.Java	Clase con la cual se tiene la información para formar la cláusula de cada pointCut
joinPoint.Java	Clase en la cual se tiene la información para cada jointPoint de la estructura, con los cuales se crea la cláusula del pointCut correspondiente

configura.Java	Clase con la cual se tiene la información de la configuración para una excepción, grupo, paquete o el default; indica si hay que reintentar o hacer un delay; así como si se lanza alguna excepción y/o si va encapsulada
ejecutar.Java	Clase con la cual se tiene la información del método a ejecutar para una excepción, grupo, paquete o el default; contiene el tipo de retorno del método así como sus parámetros
parametro.Java	Clase en la cual se maneja la información de los parámetros de los métodos a ejecutar y de los métodos en los jointPoint
datos.Java	Clase con la cual se le da los atributos de nombre y descripción a otros objetos que lo necesitan

Tabla 5.1 Clases para la creación de la estructura XML

A continuación se tienen los diagramas de clases de este paquete.

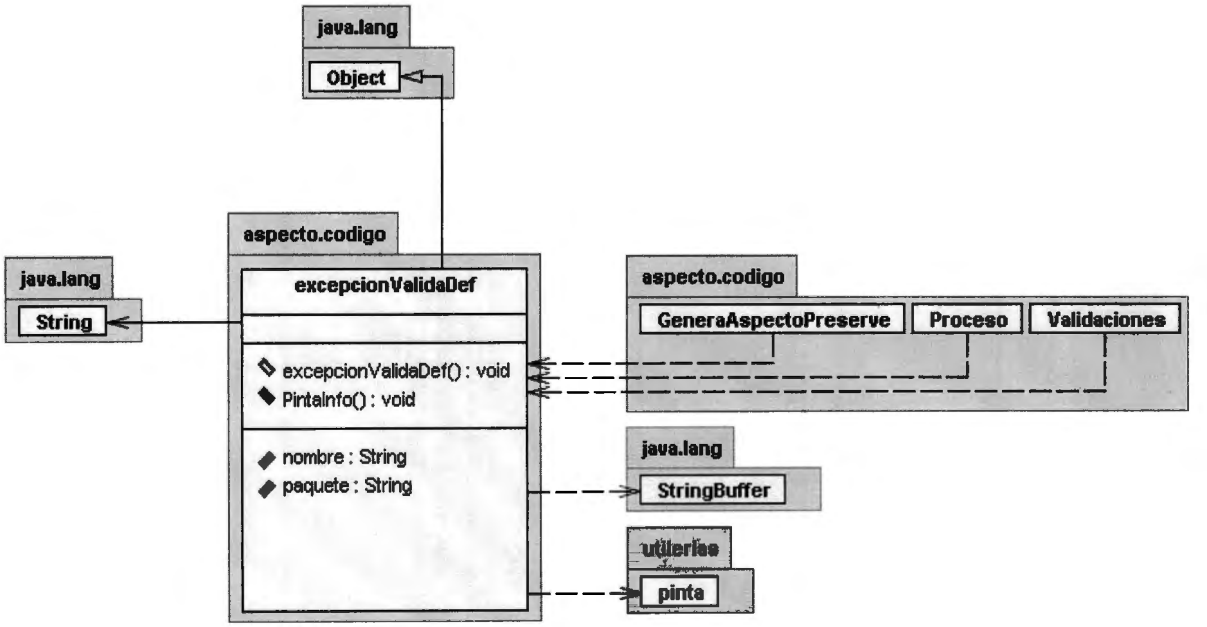

Figura 5.10 Diagrama de clase, excepcionValidaDef.Java

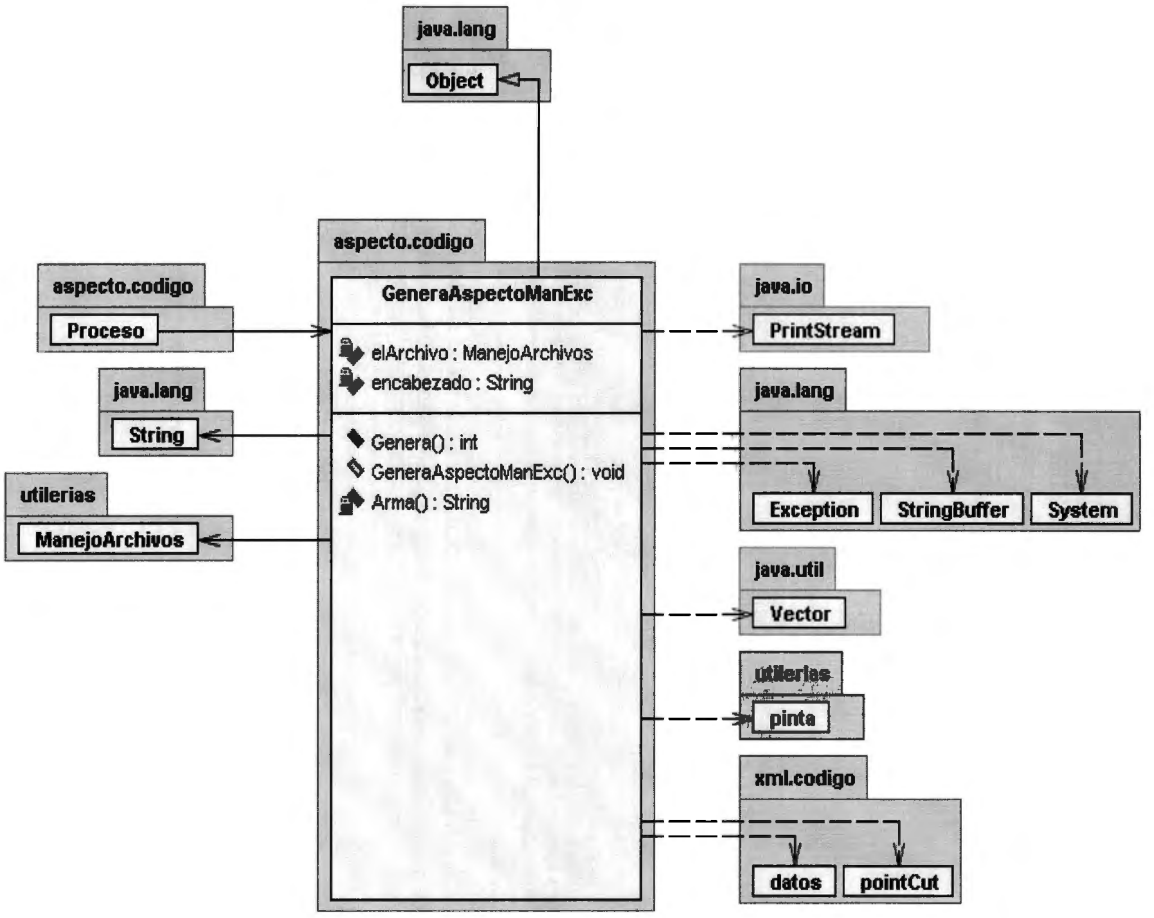

Figura 5.11 Diagrama de clase, GeneraAspectoManExc.Java

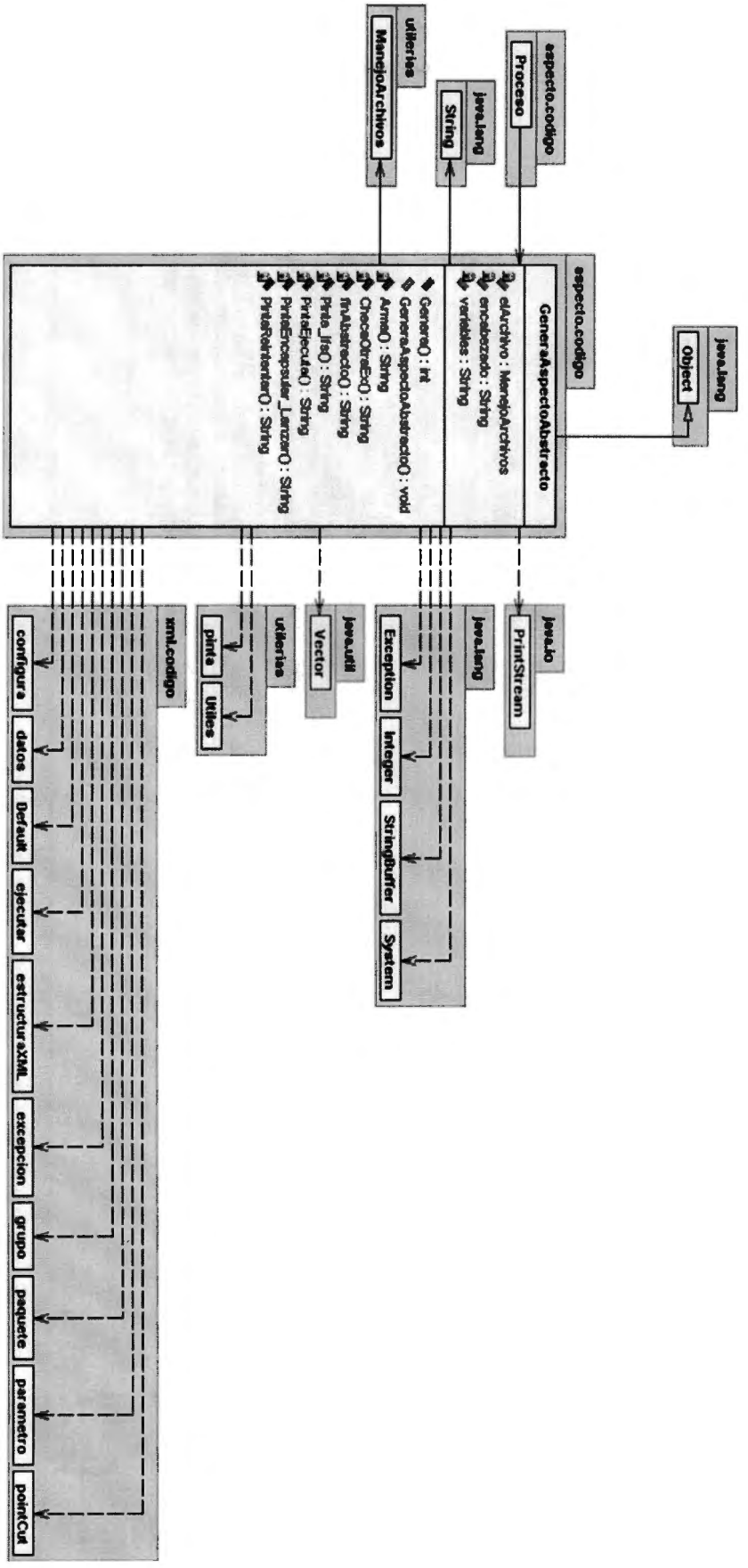

Figura 5.12 Diagrama de clase, GeneraAspectoAbstracto.Java

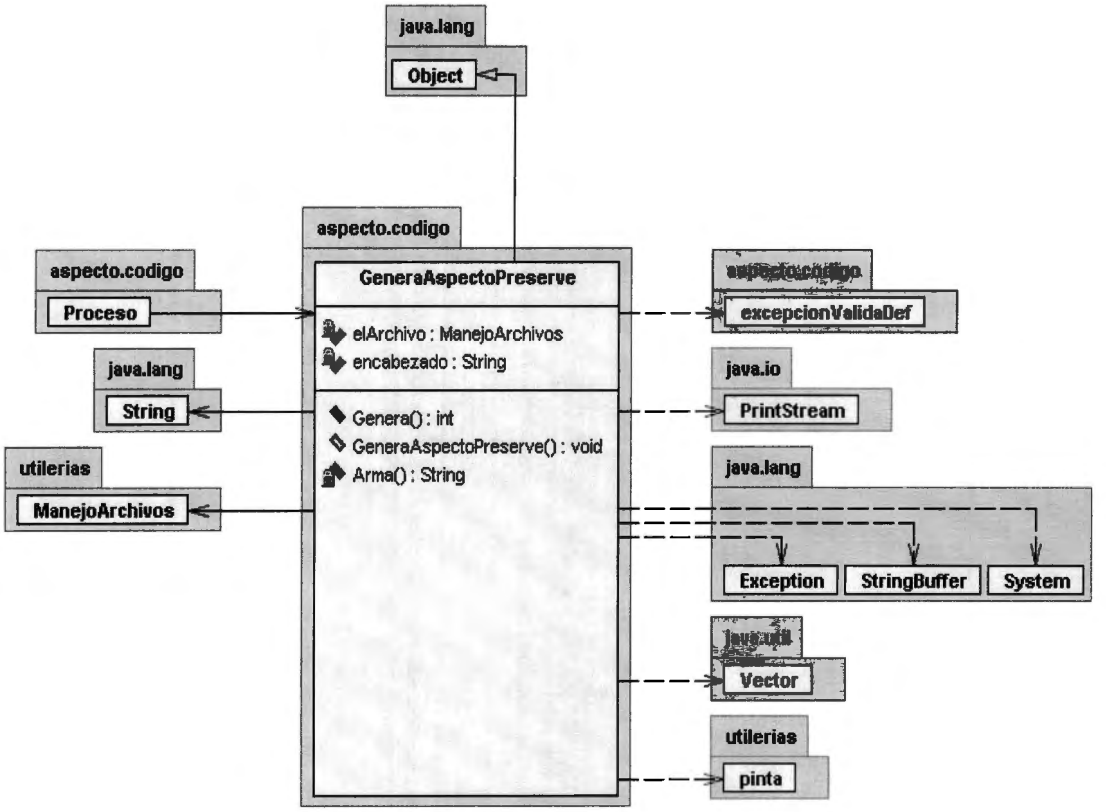

Figura 5.13 Diagrama de clase, GeneraAspectoPreserve.Java

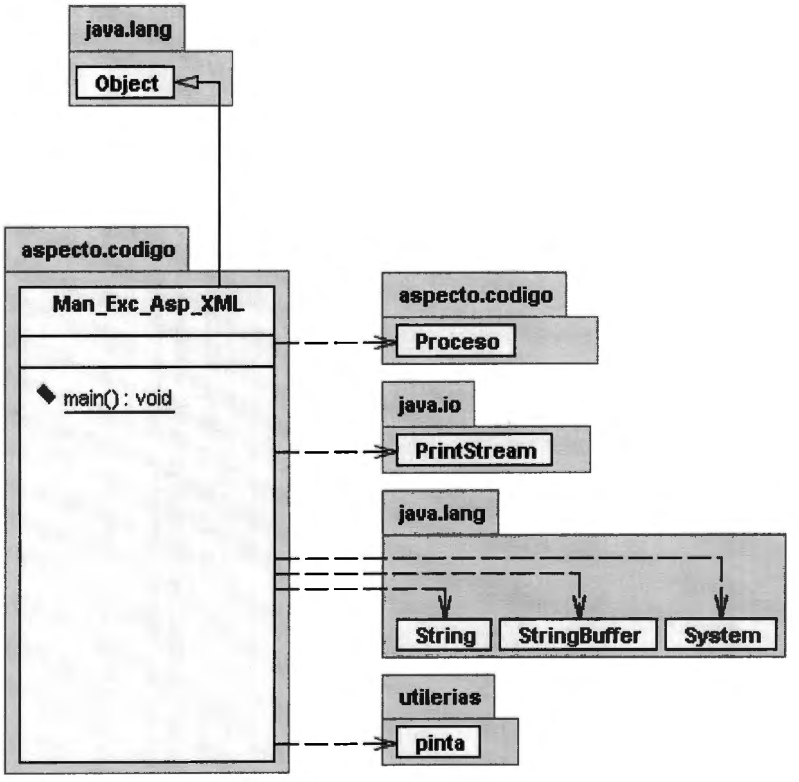

Figura 5.14 Diagrama de clase, Man_Exc_Asp_XML.Java

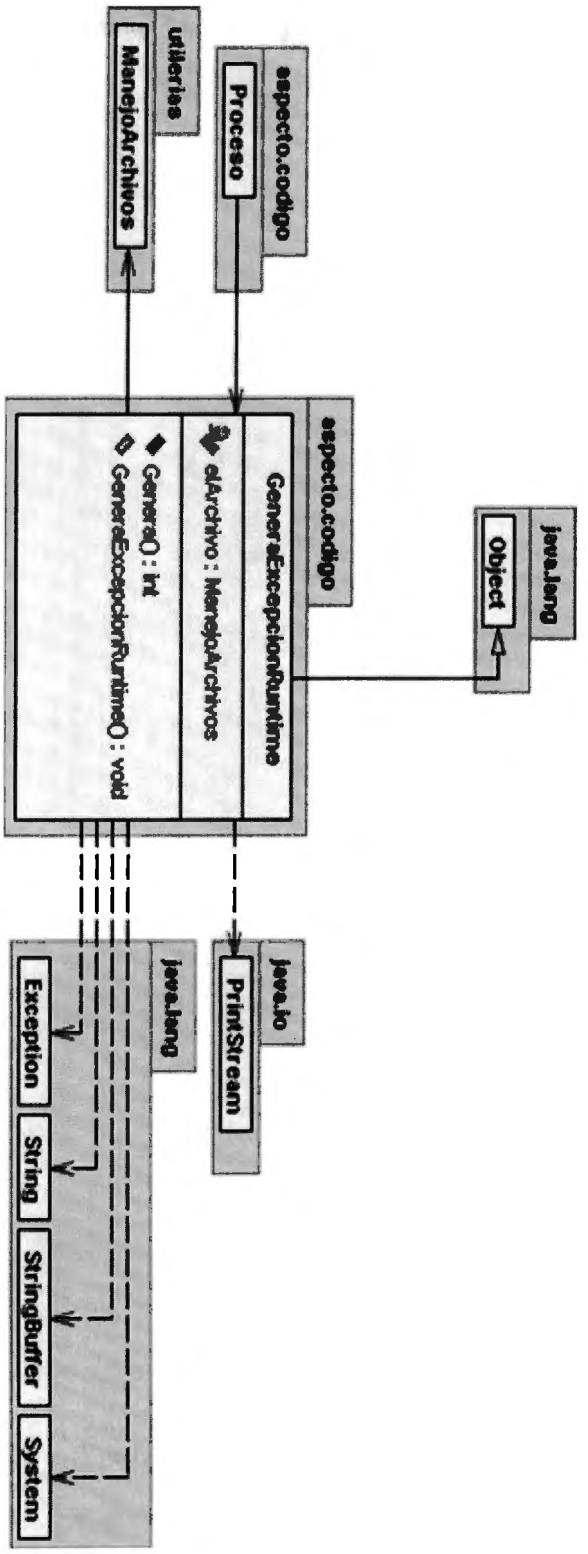

Figura 5.15 Diagrama de clase, GeneraExcepcionRuntime.Java

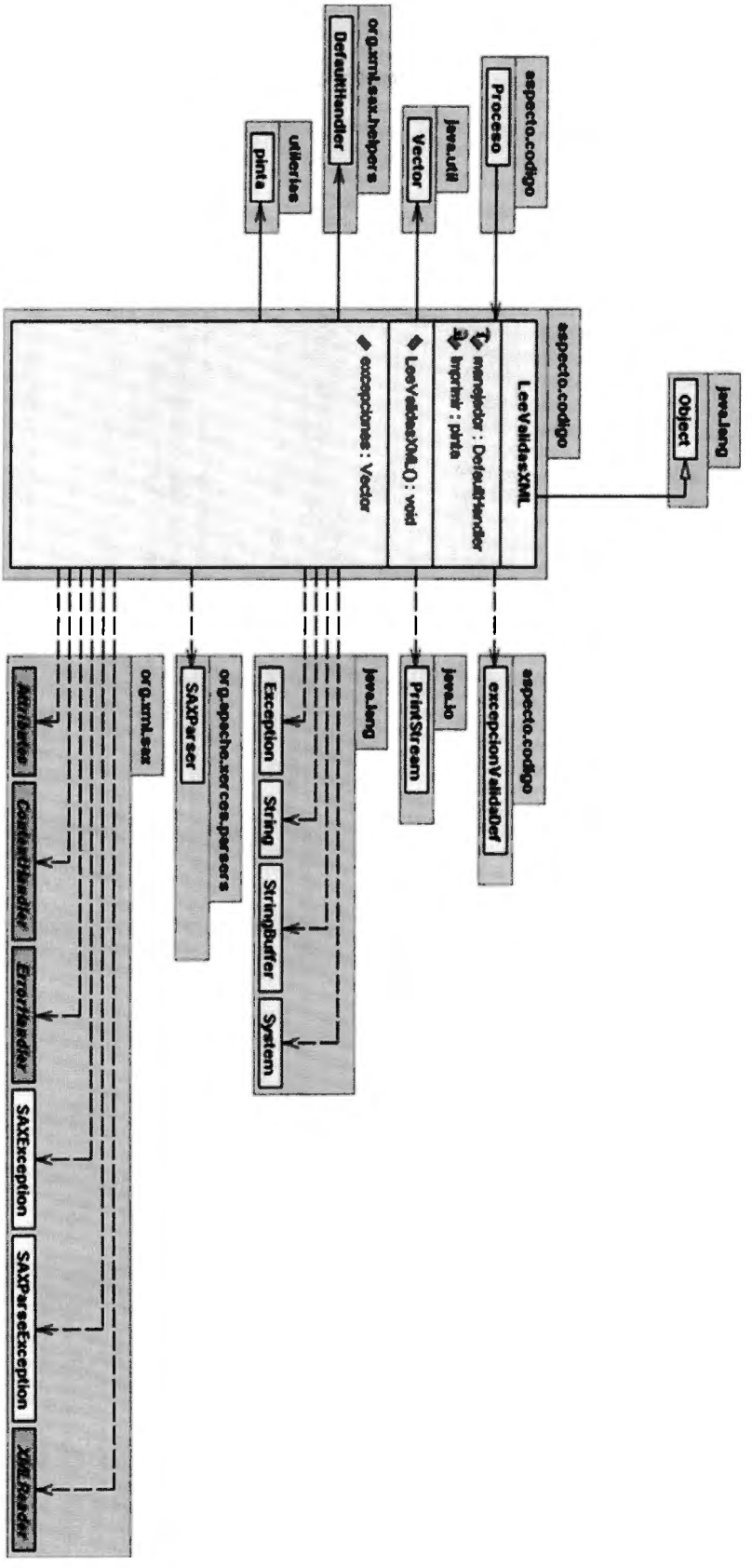

Figura 5.16 Diagrama de clase, LeeValidasXML.Java

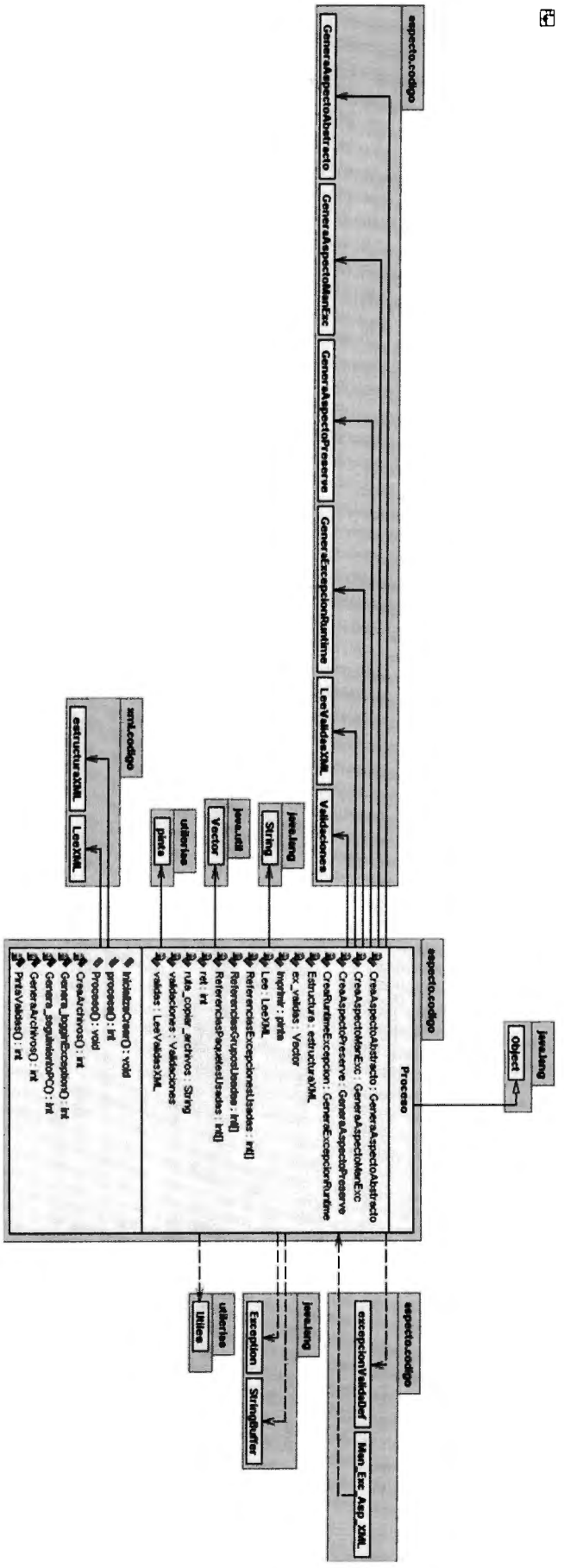

Figura 5.17 Diagrama de clase, Proceso.Java

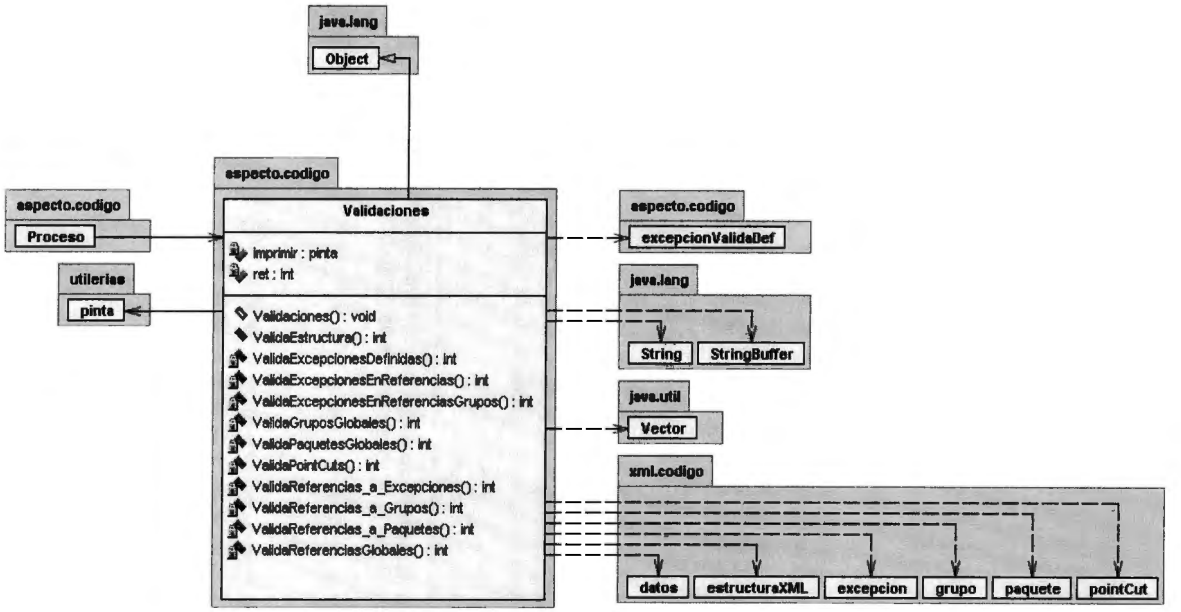

Figura 5.18 Diagrama de clase, Validaciones.Java

5.3.2 Generación de aspectos

Ya que se tiene la estructura correspondiente se procede a crear los aspectos, para el aspecto de manejo de excepciones se crean las clases que se mencionan en el punto 5.3.1 y si se define, los aspectos opcionales que se muestran en el punto 5.2.3.

5.3.2.1 Clases

El código fuente utilizado para la generación de los aspectos se encuentra en el subdirectorio *"codigo"* del subdirectorio *"aspecto"*, las clases que ahí se tiene se muestran en la tabla 5.2, con su descripción.

Man_Exc_Asp_XML.Java	Clase que contiene el main del proceso de generar los archivos para el aspecto de manejo de excepciones y de los aspectos de log de excepciones y de seguimiento de pointCuts. Todo esto en base a la información que se lee del archivo XML
Proceso.Java	Clase en la cual se tiene el proceso principal, con el cual se crean los aspectos
Validaciones.Java	Clase en la cual se realizan las validaciones necesarias desde la clase Proceso

excepcionValidaDef.Java	Clase que contiene la información de las excepciones válidas, definidas según el archivo XML: "aspecto/datos/excepciones_validas_RMI.XML" que se lee con xerces y es validado con el esquema: "aspecto/datos/esquema_validas_RMI.xsd". Ver LeeValidasXML.
LeeValidasXML.Java	Clase que permite llenar el vector con las excepciones válidas. Esto en relación al archivo XML: "aspecto/datos/excepciones_validas_RMI.XML" que se lee con xerces y es validado con el esquema: " aspecto /datos/esquema_validas_RMI.xsd"
GeneraExcepcionRuntime.Java	Clase con la que se genera el archivo correspondiente a la excepción Runtime necesaria para el manejo de excepciones
GeneraAspectoPreserve.Java	Clase con la que se genera el archivo correspondiente al Aspecto Preserve que sirve para quitar el wrapper a la excepción capturada, esto para todas las excepciones definidas, ver excepcionValidaDef
GeneraAspectoManExc.Java	Clase con la que se genera el archivo correspondiente al Aspecto para el manejo de excepciones
GeneraAspectoAbstracto.Java	Clase con la que se genera el archivo correspondiente al Aspecto abstracto para el manejo de excepciones

Tabla 5.2 Clases para la generación de los aspectos

Las figuras siguientes muestran los diagramas de clases correspondientes.

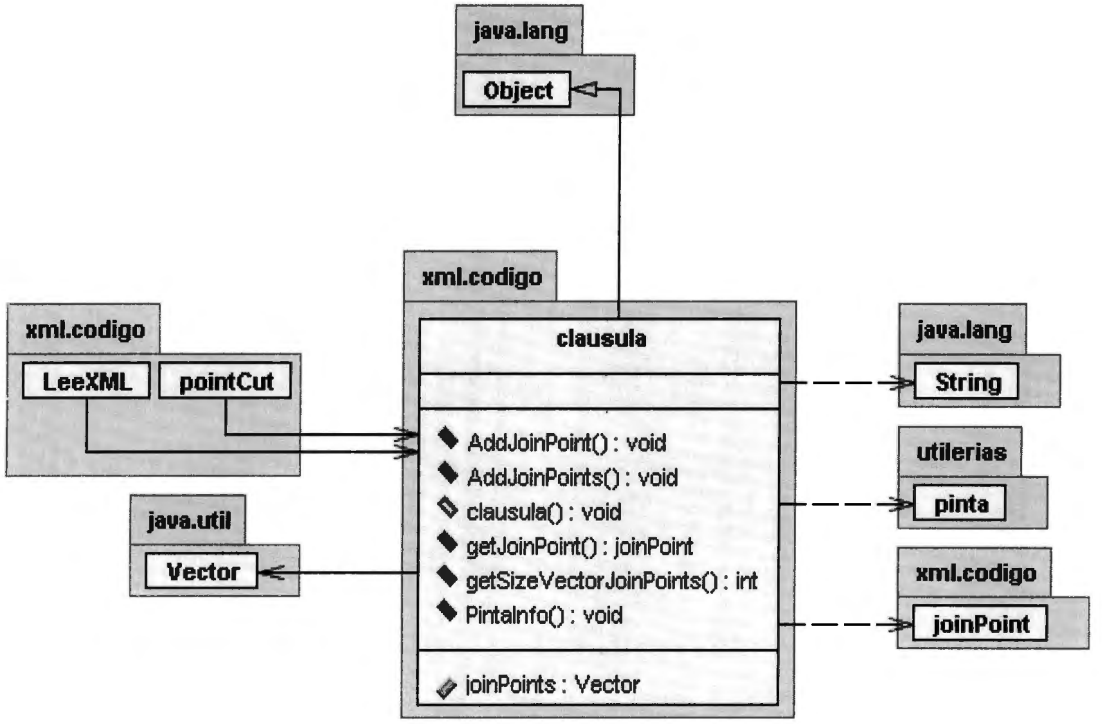

Figura 5.19 Diagrama de clase, clausula.Java

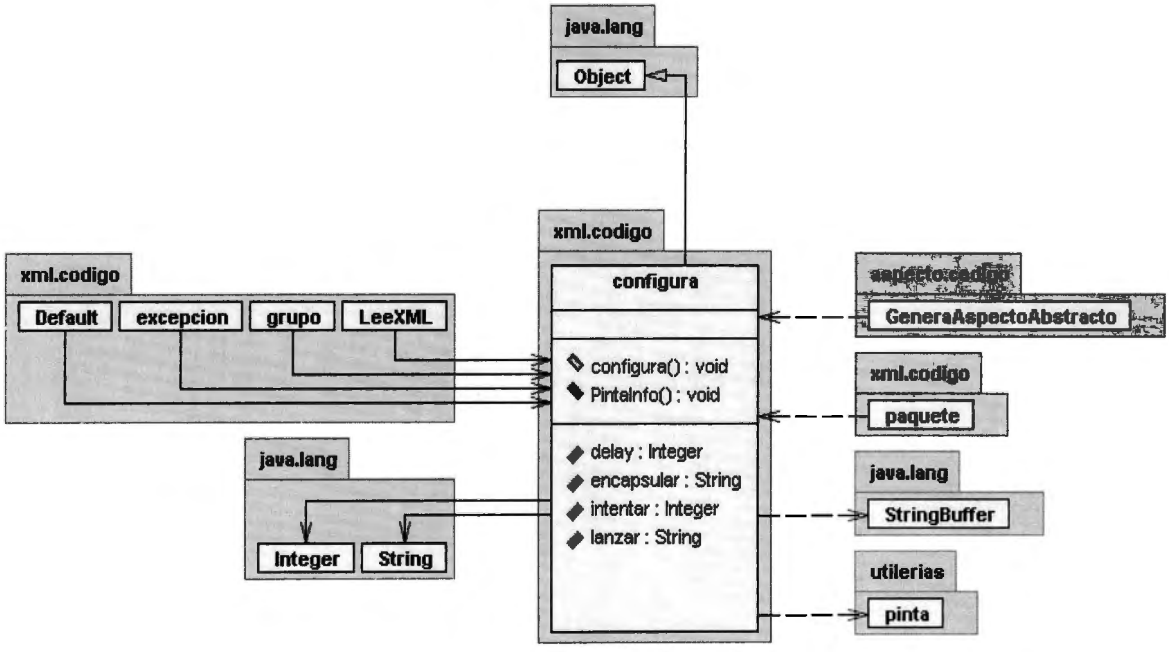

Figura 5.20 Diagrama de clase, configura.Java

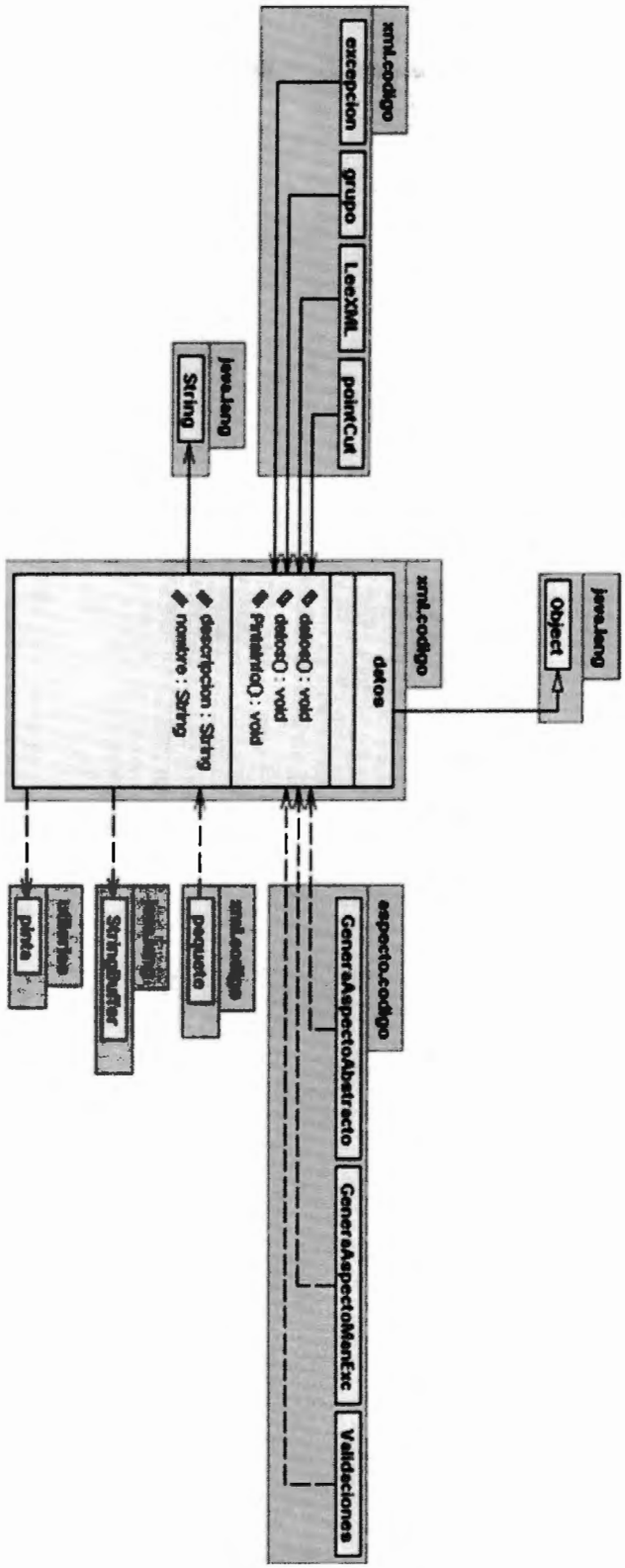

Figura 5.21 Diagrama de clase, datos.Java

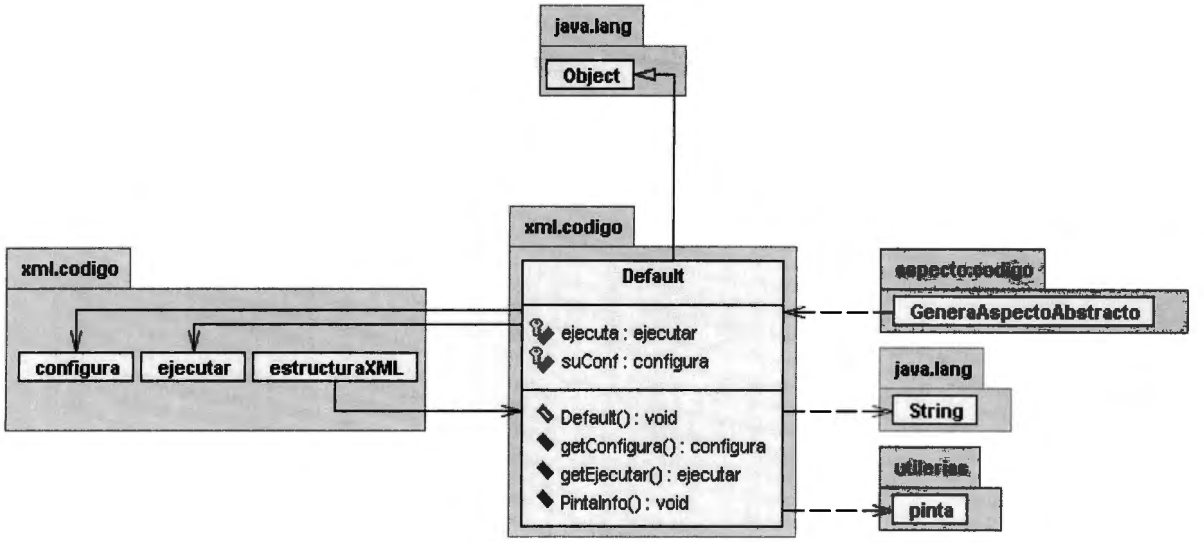

Figura 5.22 Diagrama de clase, Default.Java

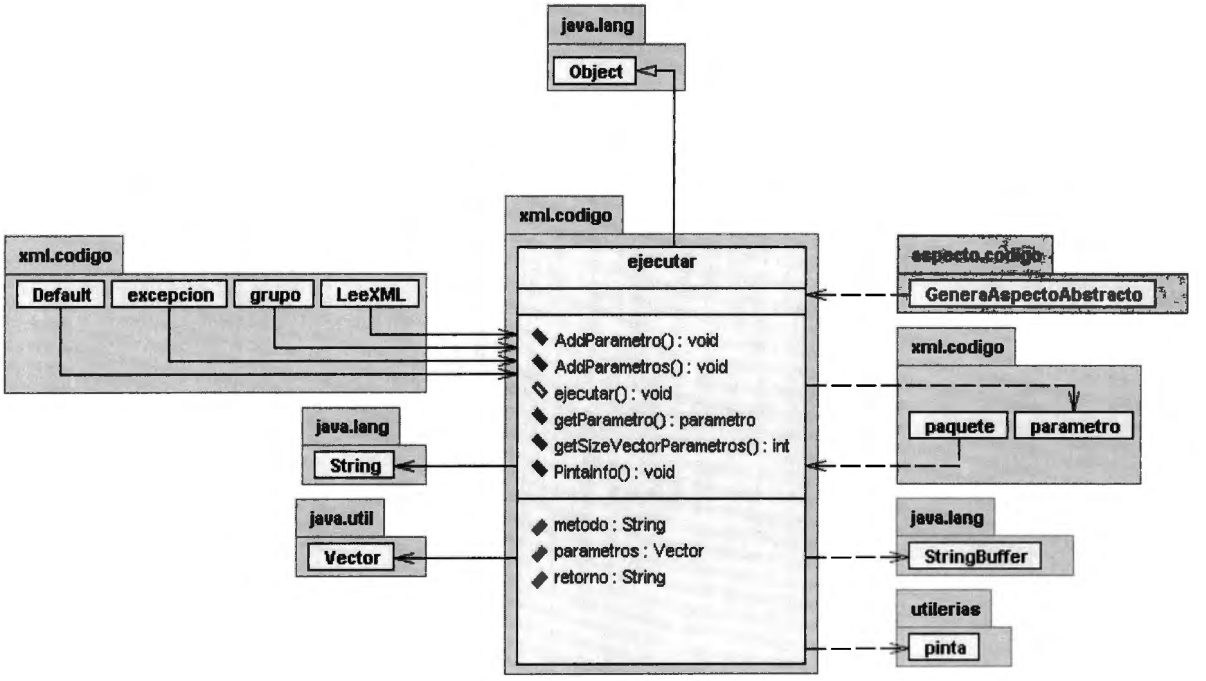

Figura 5.23 Diagrama de clase, ejecutar.Java

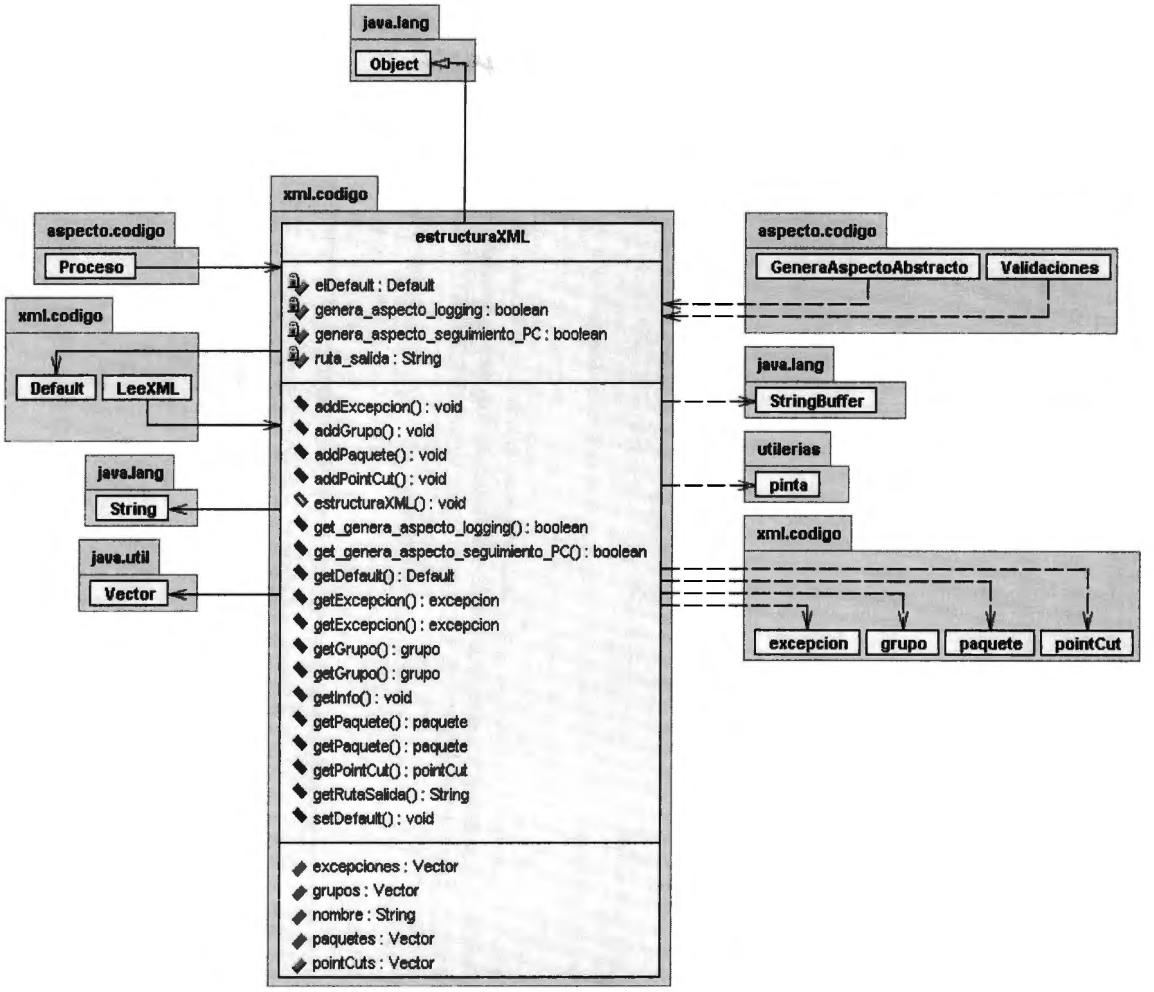

Figura 5.24 Diagrama de clase, estructuraXML.Java

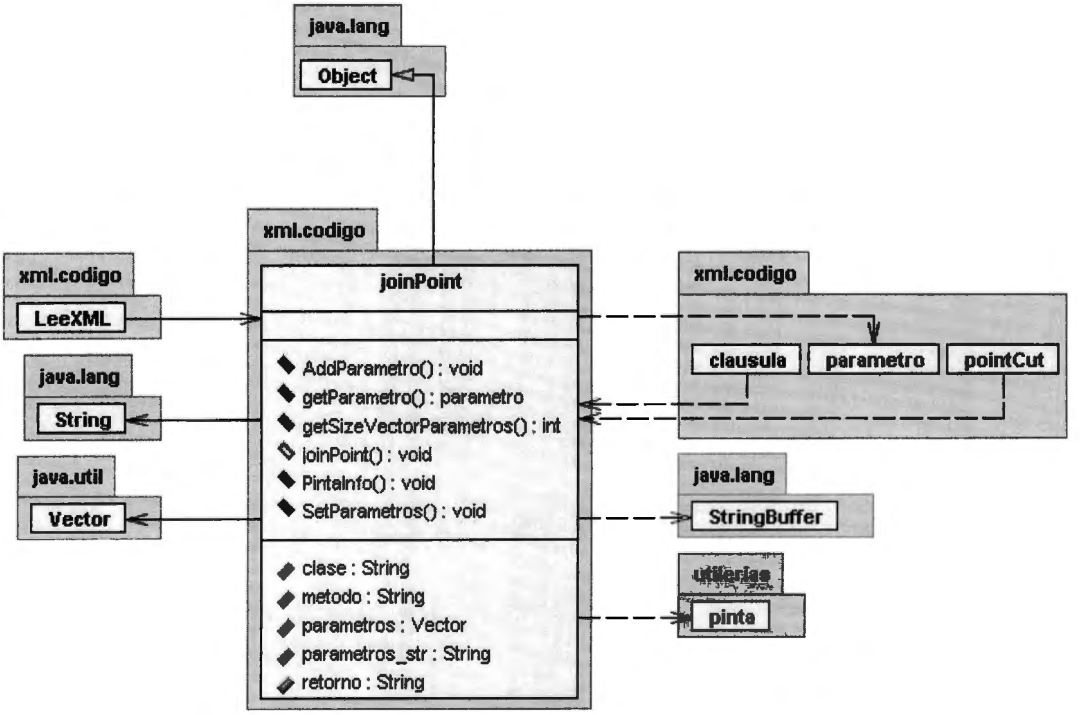

Figura 5.25 Diagrama de clase, joinPoint.Java

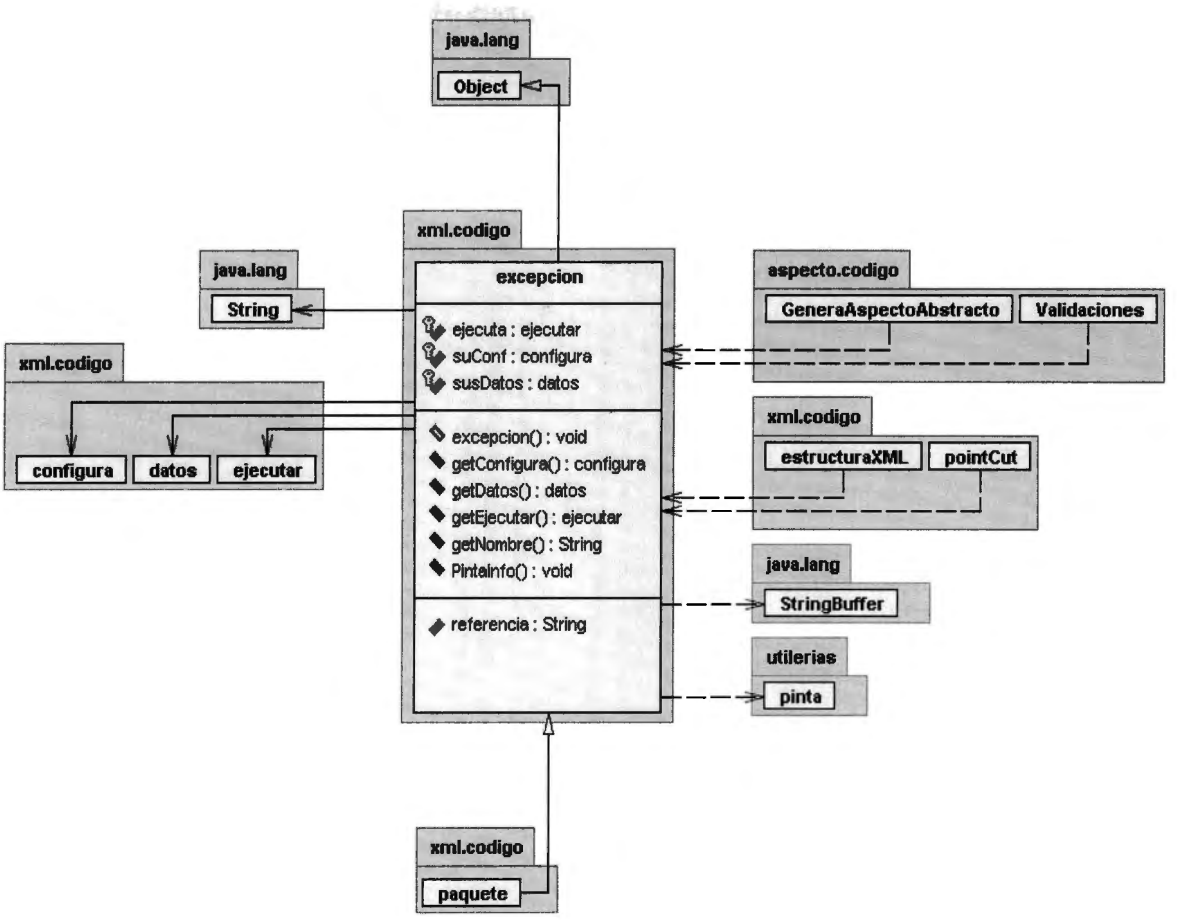

Figura 5.26 Diagrama de clase, excepcion.Java

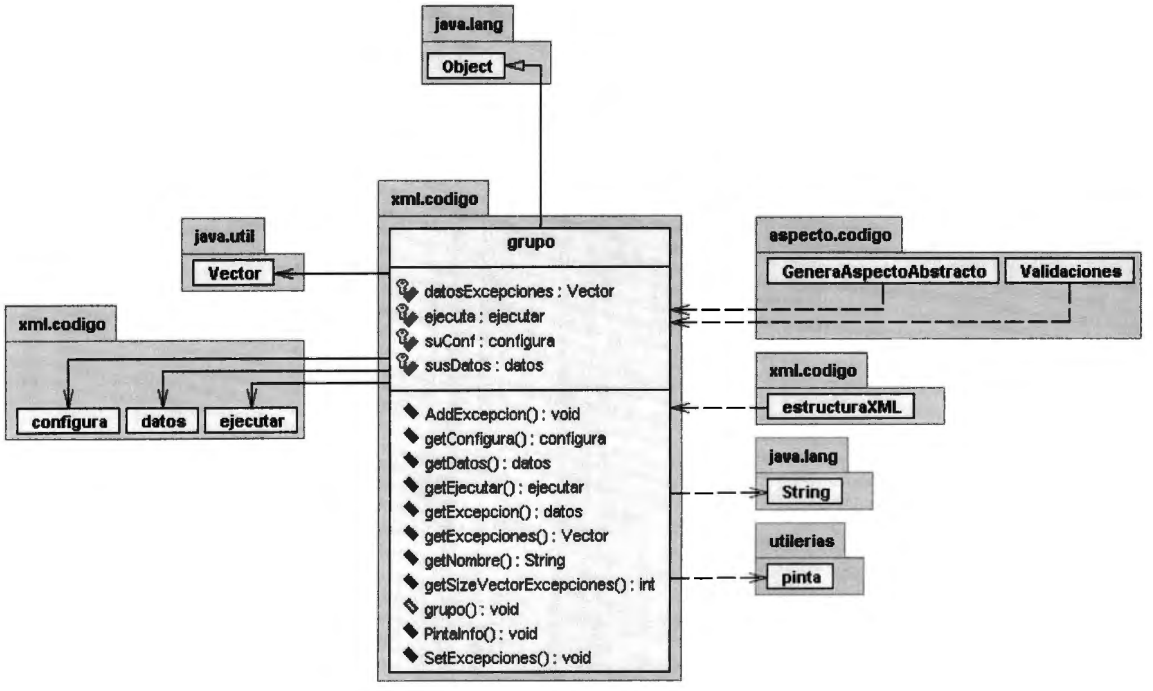

Figura 5.27 Diagrama de clase, grupo.Java

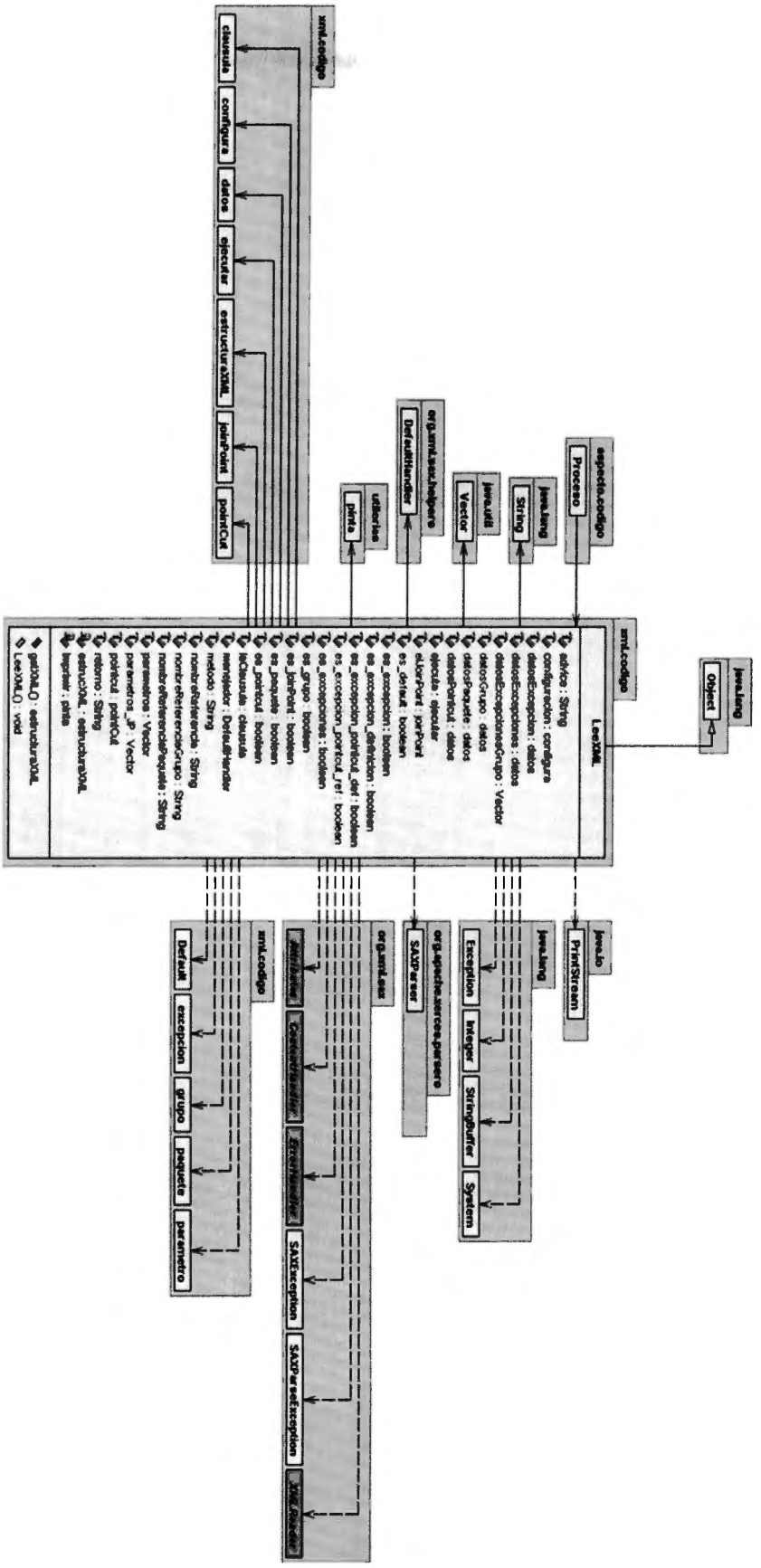

Figura 5.28 Diagrama de clase, LeeXML.Java

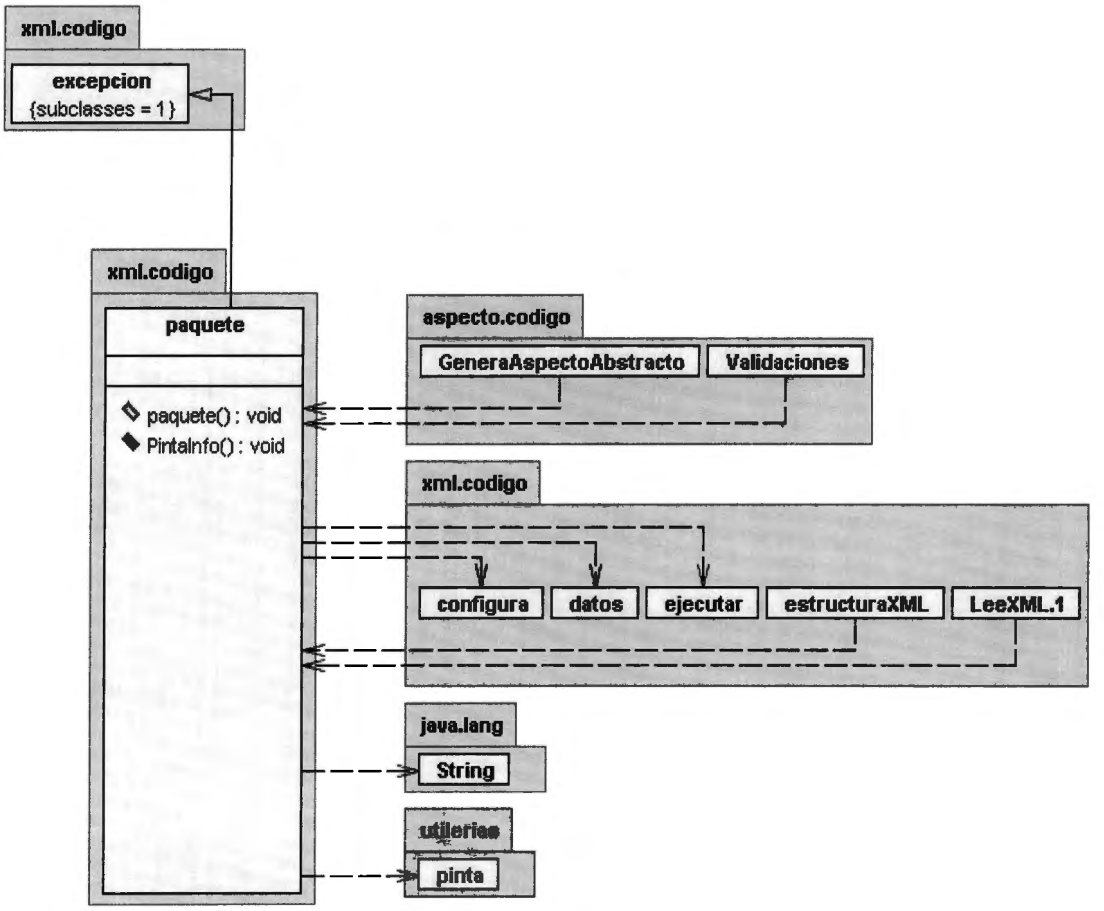

Figura 5.29 Diagrama de clase, paquete.Java

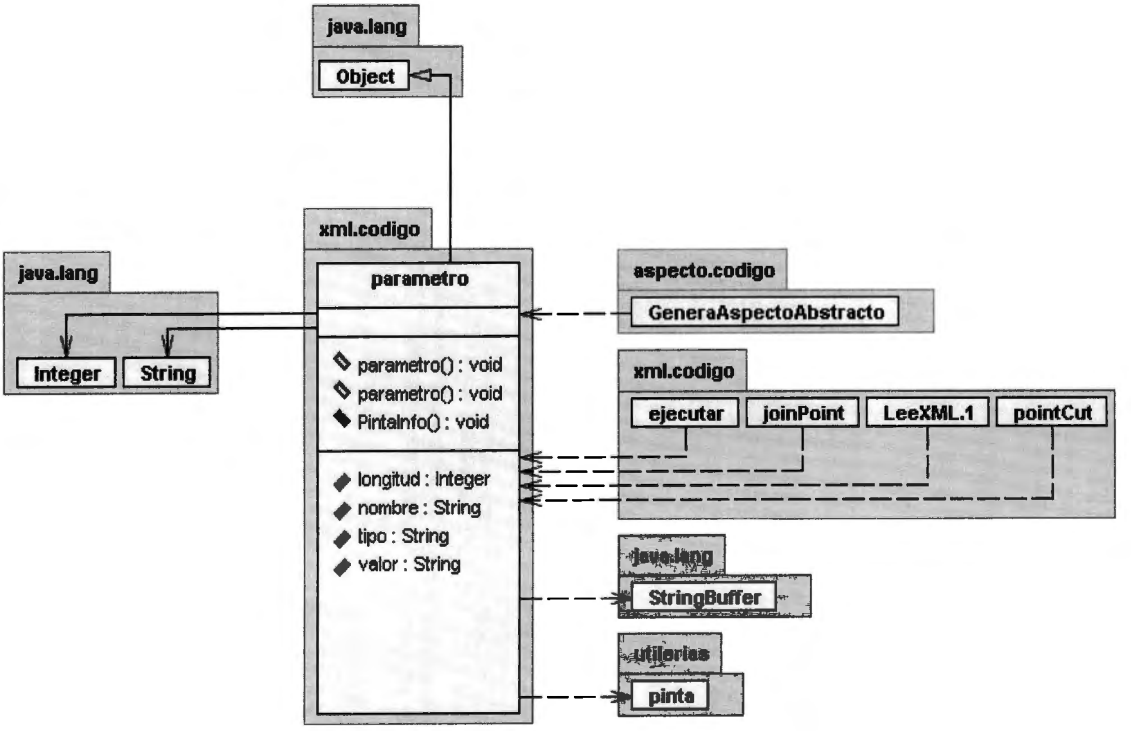

Figura 5.30 Diagrama de clase, parametro.Java

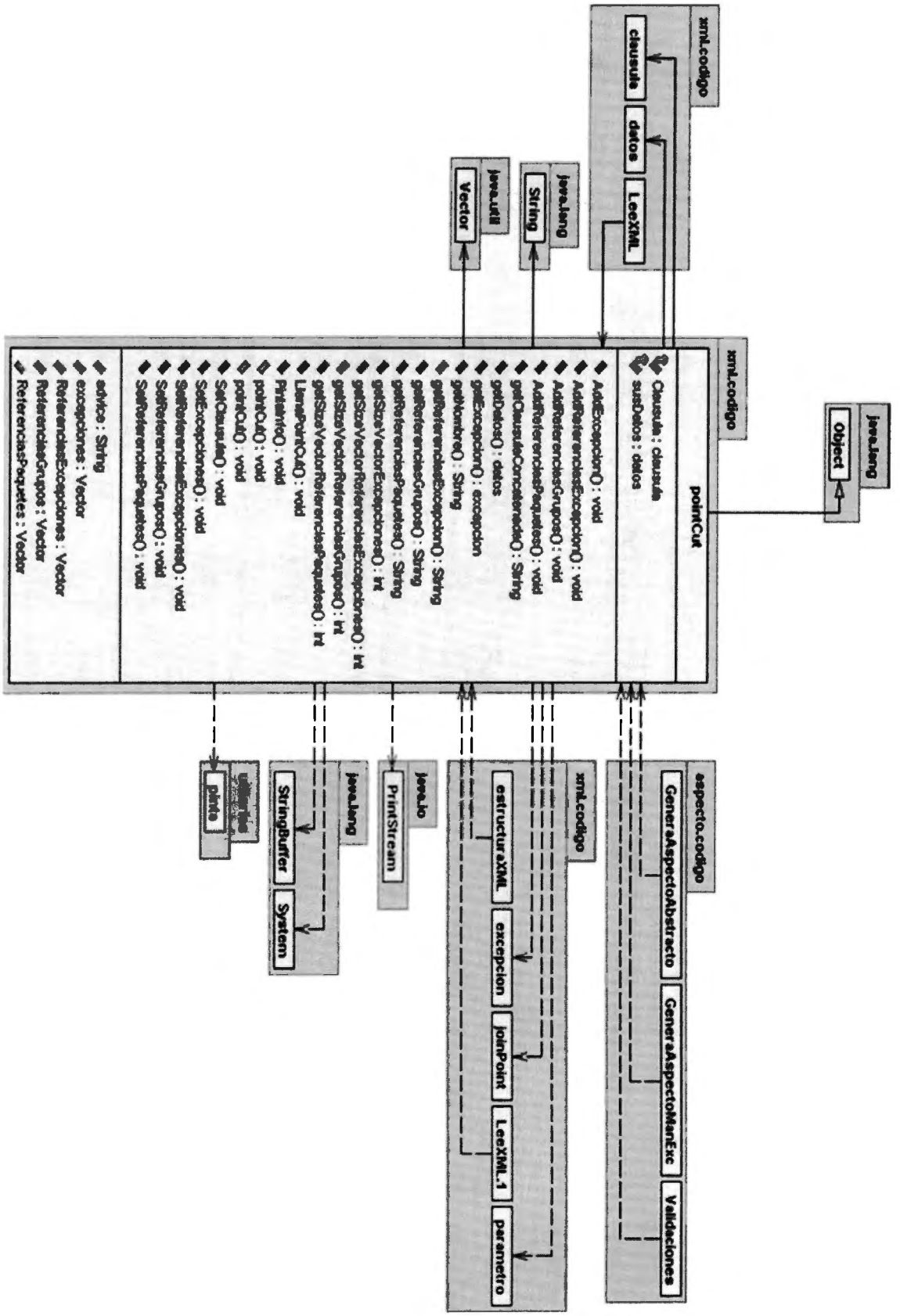

Figura 5.31 Diagrama de clase, pointCut.Java

5.3.2.2 Utilerías

Se tiene un conjunto de clases que ayudan a una serie de funciones necesarias en el proceso de la aplicación, el código correspondiente se encuentra en el subdirectorio *"utilerias"*, las clases que ahí se tiene se muestran en la tabla 5.3 con la descripción correspondiente.

ManejoArchivos.Java	Clase que se utiliza para lo relacionado a crear, guardar y abrir archivos
pinta.Java	Clase que se utiliza para el manejo del log
Utiles.Java	Clase que se utiliza para el manejo de utilerías necesarias en la aplicación
Constantes.Java	Interfaz para definir constantes que se usan en la aplicación, principalmente se trata de strings que se utilizan para la generación de los archivos
Errores.Java	Interfaz para definir los errores posibles con los que puede terminar la aplicación o que se pueden detectar en la ejecución de la misma

Tabla 5.3 Clases de utilerías de la aplicación

Además de las clases mencionadas en la tabla 5.3, se tienen tres subdirectorios, su contenido se describe en la tabla 5.4.

SUBDIRECTORIO	ARCHIVO	DESCRIPCION
aspecto	fin_abstracto.txt	Se tiene el final del aspecto ExHanAspect, pues como esta parte nunca cambia, se toma de este archivo y se pone al final de la clase correspondiente, ver GeneraAspectoManExc.Java
ASPECTO_LoggingException	AspLogExc.txt	Se tiene el aspecto AspLogExc, con el cual se maneja el log de excepciones lanzadas, como este aspecto no cambia, se copia dicho archivo, la generación de este aspecto es opcional, ver Proceso.Java
ASPECTO_SeguimientoJoinPoints	AspSegJoinPoints.txt	Se tiene el aspecto AspSegJoinPoints con el cual se maneja el seguimiento de join point para una aplicación como este aspecto no cambia, se copia dicho archivo, la generación de este aspecto es opcional, ver Proceso.Java

Tabla 5.4 Contenido del subdirectorio utilerias

Los diagramas de las clases que conforman el paquete se muestran en las figuras siguientes.

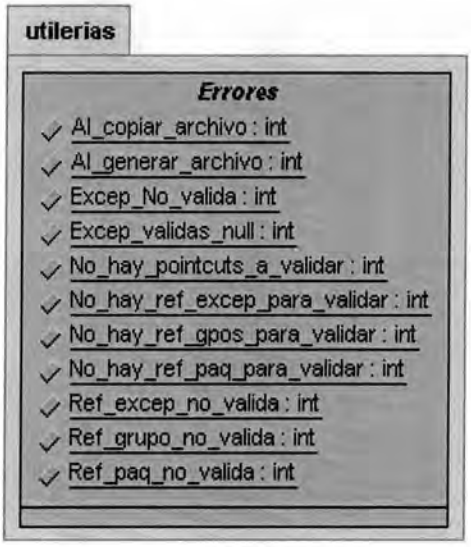

Figura 5.32 Diagrama de clase, Errores.Java

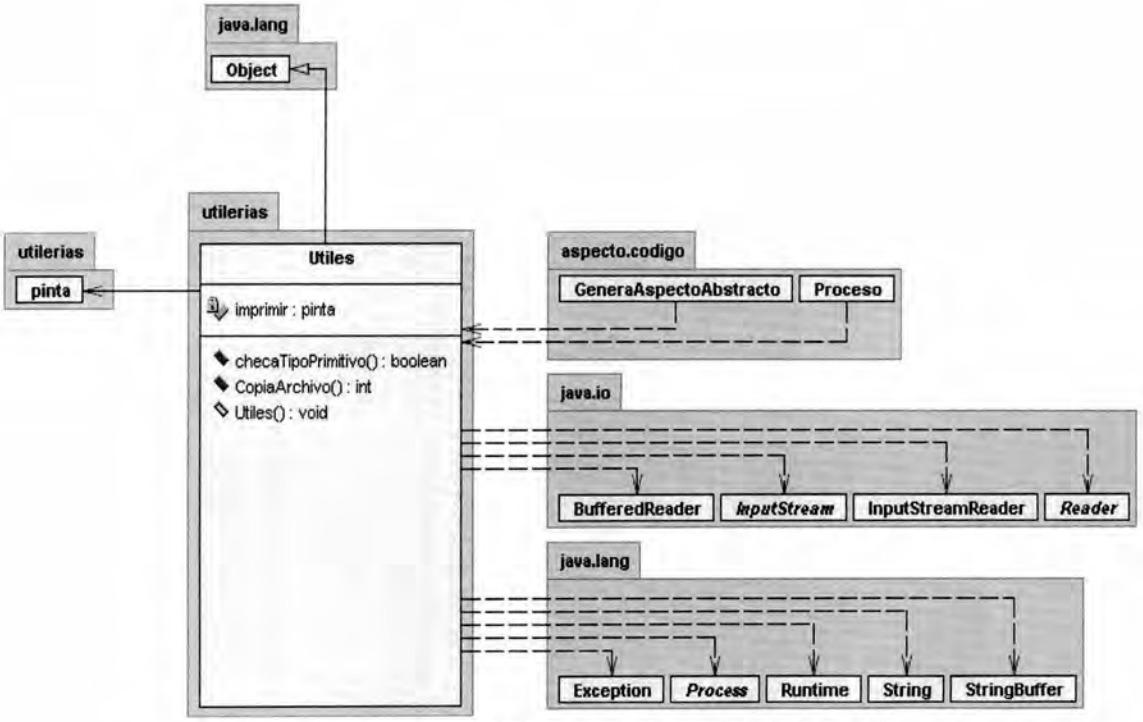

Figura 5.33 Diagrama de clase, Utiles.Java

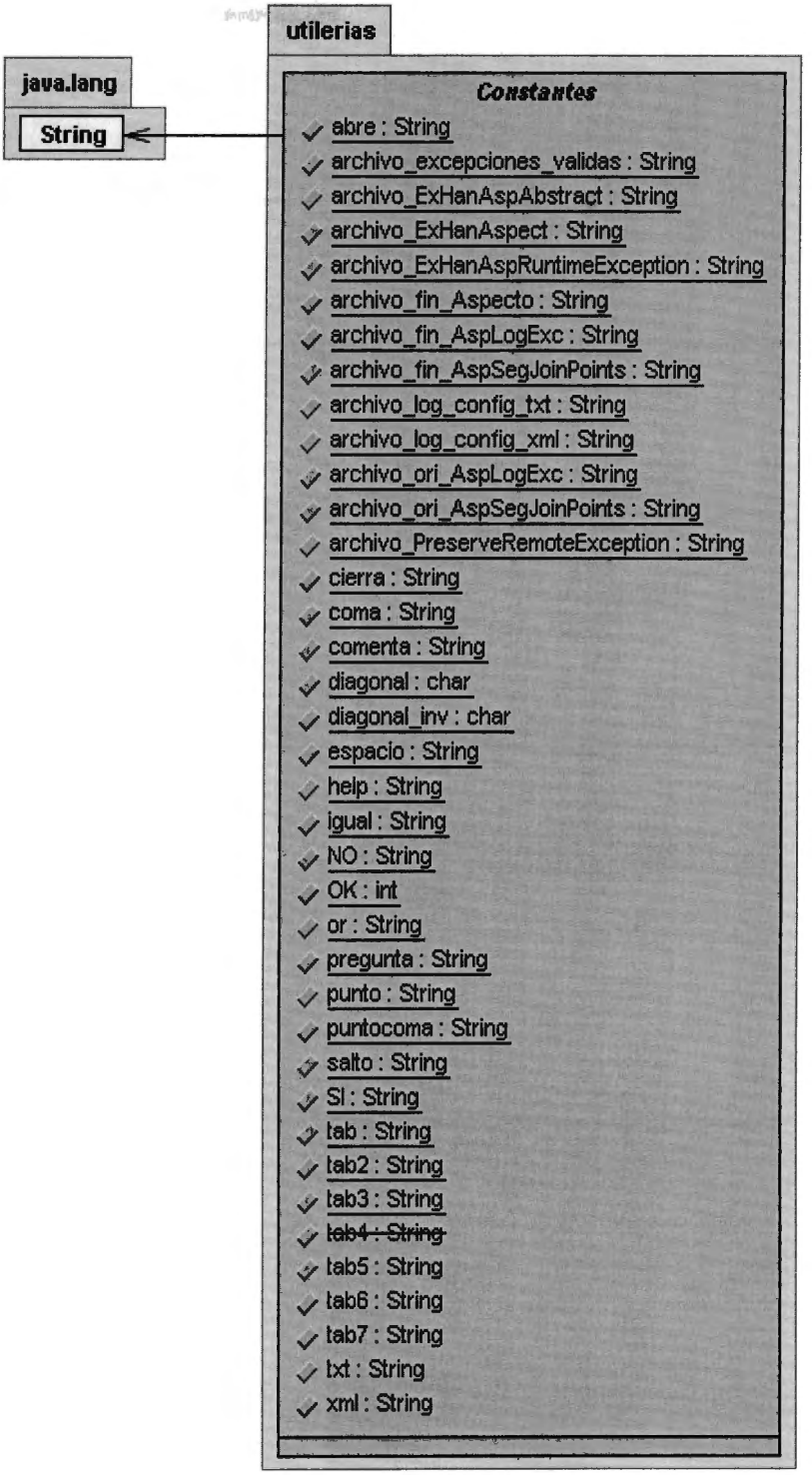

Figura 5.34 Diagrama de clase, Constantes.Java

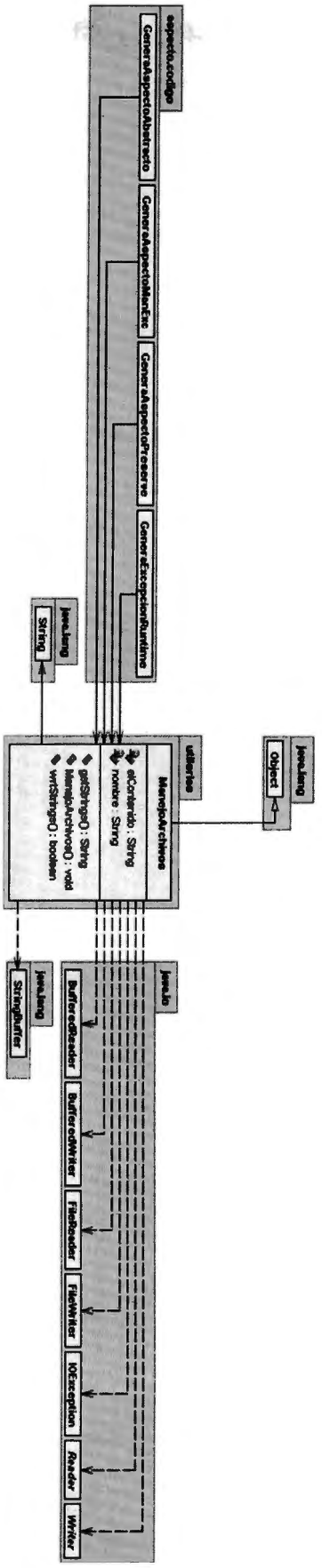

Figura 5.35 Diagrama de clase, ManejoArchivos.Java

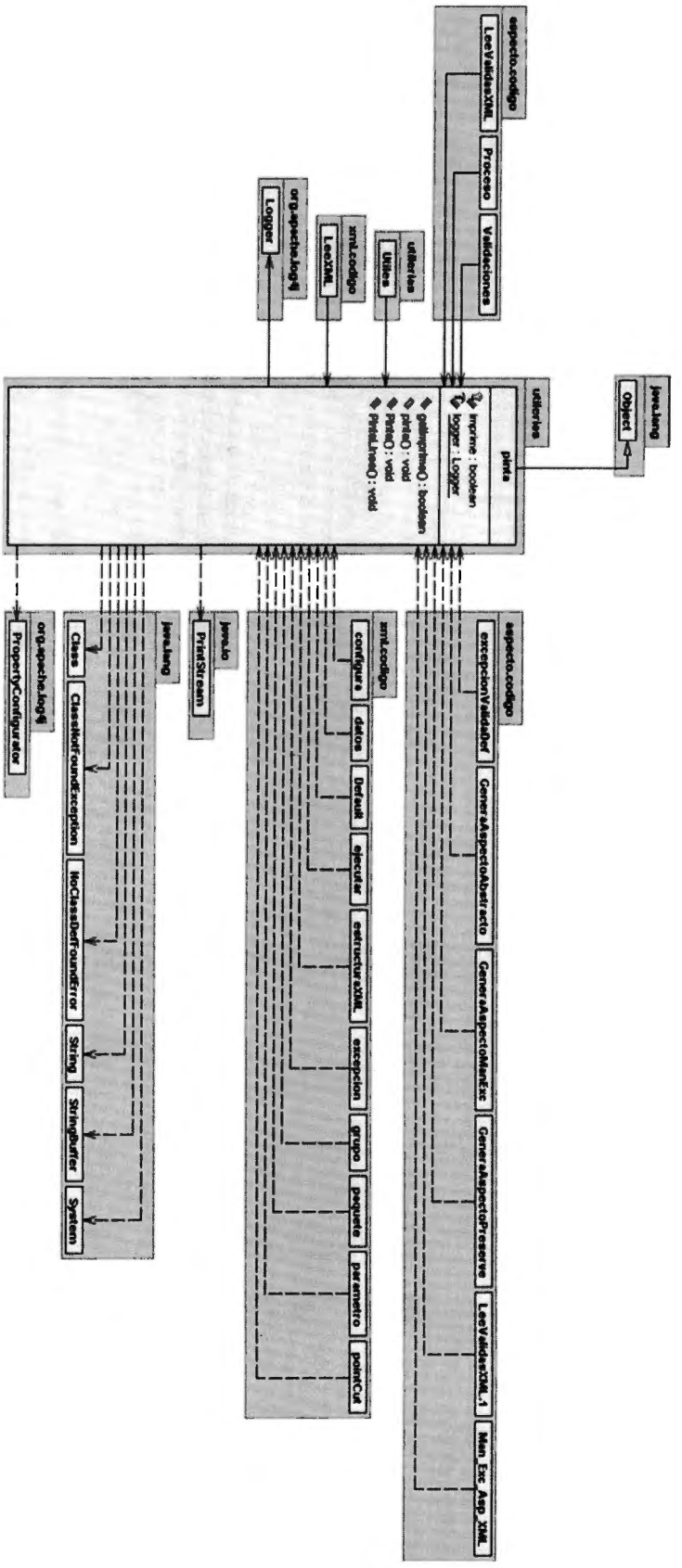

Figura 5.36 Diagrama de clase, Pinta.Java

5.3.3 Manejo del log con log4j

Basándonos que cuando se construye una aplicación Java, uno de los principales errores que se suelen cometer es una gestión mala de *logs* (por ejemplo el uso de System.out.println()), que posteriormente es difícil eliminar o filtrar de un modo sencillo; Se pensó en utilizar el estándar log4j (por su significado en ingles: *logs for Java*) donde se dispone de un mecanismo sencillo para especificar la fuente de datos, dónde y como se mostraran los mensajes y el tipo de mensaje a mostrar, para más información [32].

Se tiene un manejador del log en la clase pinta.java, el cual puede ser configurado como archivo de texto o como archivo XML, para lo cual se tienen los archivos de configuración: *log4j.properties.XML* y *log4j.properties.txt* que se encuentran el subdirectorio "log_config" y el archivo de salida es *log_ejecucion.txt*, se encuentra en el subdirectorio "log_salida".

5.3.4 Compilación y ejecución de la aplicación

Una vez que se ha declarado en el archivo *excepciones.XML* (ver 5.3.1) como se desea el manejo de excepciones RMI, solo basta ejecutar el comando:

"java aspecto.codigo.Man_Exc_Asp_XML log_pantalla formato_archivo_log"

La aplicación al terminar, despliega un código de error o el valor de 0 si todo se ejecutó correctamente y se crearon los aspectos correspondientes, los parámetros se describen en la sección siguiente.

Si se requiere compilar la aplicación se puede hacer desde la línea de comandos o ejecutando el archivo batch *comp_genera.bat* que se encuentra en el subdirectorio de la aplicación. Es importante mencionar que se necesita para la compilación exitosa:

- Tener instalado Java, el j2sdk, se recomienda contar con la versión 1.5.
- Tener instalado AspectJ, se recomienda la versión 1.2. Con lo que se necesita la variable de ambiente ASPECTJ_HOME, que corresponde al directorio donde esta instalado AspectJ.

- Para Xerces, la versión con la que se realizó la aplicación es 2.6.2. Por lo que se debe incluir en el CLASSPATH:
    - Los archivos xercesImpl.jar y XMLParserAPIs.jar, que se encuentran en el directorio donde se instalo xerces.
- El CLASSPATH debe incluir:
    - El archivo tools.jar del jdk (que se encuentra dentro del subdirectorio lib del jdk).
    - El archivo aspectjrt.jar de AspectJ (que se encuentra dentro del subdirectorio lib del AspectJ.
- El JAVA_HOME, debe de corresponder al directorio donde esta instalado el jdk.
- El PATH, debe de contener:
    - El subdirectorio bin del jdk.
    - El subdirectorio bin de AspectJ
    - El directorio de instalación del jdk

5.3.4.1 Parámetros y ayuda de la aplicación

Para poder ejecutar la aplicación se deben incluir dos parámetros, el primer parámetro (*log_pantalla*) corresponde a si se quiere o no mostrar en pantalla el log de lo que se va procesando y ejecutando (parámetro de tipo *string* con valores SI o NO) y el segundo parámetro (*formato_archivo_log*) corresponde al formato con el cual se quiere el archivo de salida del log (parámetro de tipo *string* con valores TXT o XML).

Se tienen un archivo batch que incluye los parámetros SI y TXT, el archivo es *run_genera.bat* y se encuentra en el subdirectorio de la aplicación. Los archivos de configuración se presentan en el Apendice B, Listados.

Se cuenta con la opción de que la aplicación despliegue una información de ayuda, para lo cual solo basta darle el parámetro "?" o "help", la información que se muestra se tiene en la figura 5.37.

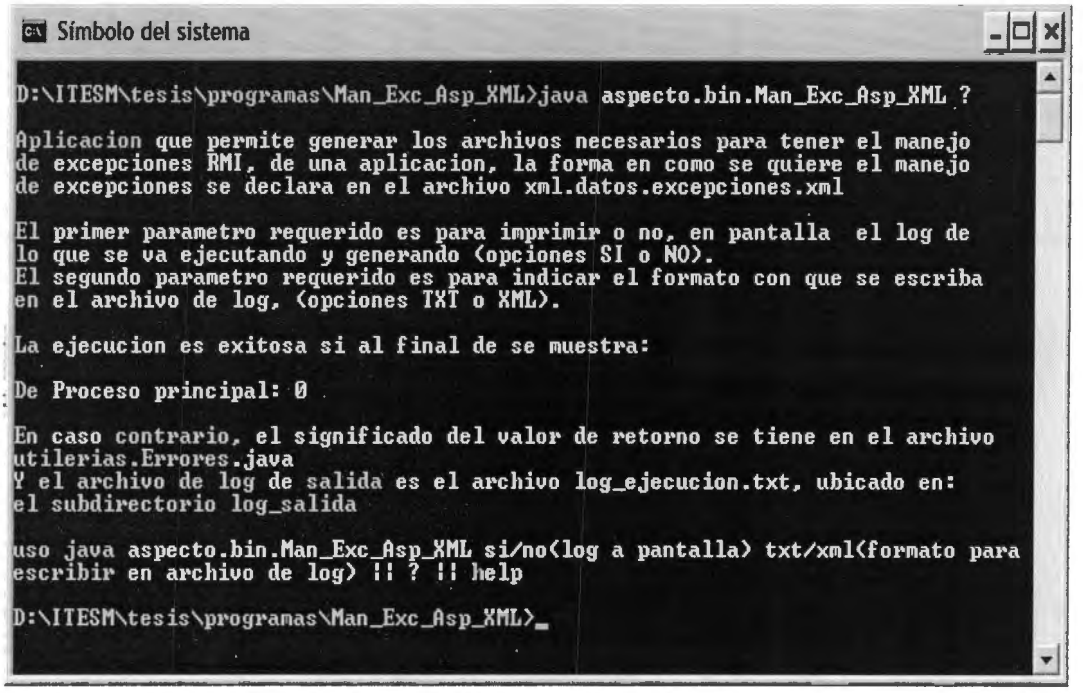

Figura 5.37 Información desplegada como ayuda para la ejecución de la aplicación

5.3.4.2 Integración del sistema con lo generado por la herramienta

Una ves que se ejecuta correctamente la aplicación, es decir, después de haber generado el aspecto de manejo de excepciones y los aspectos adicionales si se requiere o desea; solo bastara con compilar la aplicación a la cual se le quiere agregar el manejo de excepciones, pero ya con el compilador AspectJ considerando el código que maneja la funcionalidad básica del sistema y el código generado por la herramienta presentada. Con lo que se obtendrá el código ejecutable ya con el manejo de excepciones que se describió en el archivo XML de configuración de la herramienta, ver figura 5.1.

5.3.5 Prueba de la aplicación

La herramienta se probó con algunos sistemas desarrollados durante la maestría, y otros sistemas que utilizan Java RMI, de manera de ejemplo en la presente tesis se muestra una aplicación desarrollada en la materia de Ambientes de Programación Avanzada, la cual se describe en las siguientes secciones.

5.3.5.1 Sistema de prueba, pizarra colaborativa

Con fines de entender la aplicación tomada para el ejemplo, sin la necesidad de entender todo su funcionamiento, se presenta un fragmento del documento de especificación de requisitos (SRS) de la pizarra colaborativa (la documentación completa de dicha aplicación se tiene en el CD de esta tesis).

Alcance

El principal beneficio de la aplicación es ayudar a un grupo de estudiantes a interactuar en línea; para poder ir enriqueciendo con sus aportaciones el trabajo realizado por cada uno. El objetivo del proyecto es la realización de una pizarra virtual para la edición de documentos colaborativos en un esquema cliente-servidor con las siguientes características:

1. Servicio de gestión de usuarios en línea que permitirá conocer y mostrar los usuarios que estén activos, de manera centralizada por el servidor.
2. Ambiente de visualización mediante un panel donde se muestre el contenido del archivo que se trabajará de manera colaborativa y cuente con los controles necesarios para su operación.
3. Facilidades para la manipulación y almacenamiento de los archivos.
4. Mecanismo de mensajería que permita el envío de mensajes a los usuarios activos del sistema (chat).
5. Esquema seguro de sincronización para la manipulación de los recursos de la pizarra.
6. Ambiente de autenticación y administración de los usuarios.

Perspectiva del producto

Este sistema puede verse como un módulo de una aplicación que esté buscando crear un ambiente de aprendizaje colaborativo y a distancia; es decir, podría ser usado tanto en un salón de clases, como en cualquier lugar que se tenga acceso a Internet.

Funciones del producto

De manera gráfica el sistema cuenta con dos funcionalidades importantes:

a) El administrador: Se encarga de mantener el servidor funcionando y dar mantenimiento a la base de datos de los usuarios. De forma gráfica se muestra en la figura 5.38.

Figura 5.38 Administrador de la pizarra colaborativa

b) Los usuario de la pizarra colaborativa: De una manera sencilla, podemos describir el funcionamiento de la pizarra de la siguiente manera: un nuevo usuario solicita ser registrado, el servidor verifica que este usuario exista; si es así, le da acceso a la pizarra, y a partir de ese momento empieza a interactuar el servidor con ese nuevo usuario, tanto en la información de la pizarra, como la lista de usuarios conectados y los mensajes enviados por el Chat. La figura 5.39 describe el proceso mencionado:

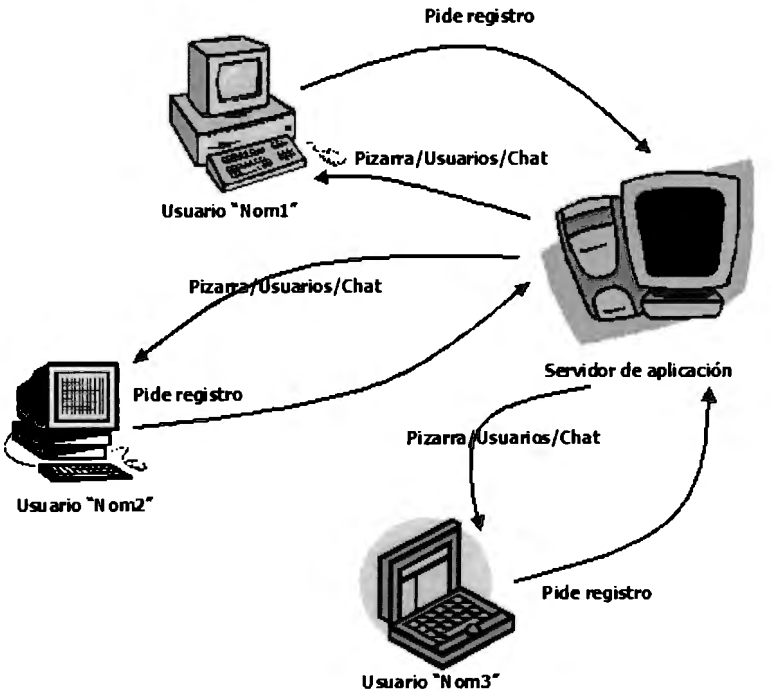

Figura 5.39 Proceso de la pizarra colaborativa

Características de los usuarios

- El administrador deberá tener experiencia en el funcionamiento de RMI (Invocation Remote Method) y de Servlets, para mantener siempre el sistema en funcionamiento.
- Los usuarios de la pizarra solo el uso básico de un navegador en Internet.

Arquitectura de la aplicación

Es una aplicación distribuida desarrollada con las siguientes tecnologías:

i. El lenguaje de desarrollo es Java, ya que día a día se está convirtiendo en un lenguaje de programación universal; es decir, ya no sólo sirve como lenguaje para programar en entornos de Internet, sino que se está utilizando cada vez más como herramienta de programación orientada a objetos, aprovechando sus características de lenguaje "multiparadigma".

ii. La administración de los usuarios es a través de Servlets, como es conocido el acceso de clientes desde Internet o intranets corporativas es una forma segura de permitir a varios usuarios acceder a datos y recursos de forma sencilla. Este tipo de acceso está basado en el uso de los estándares Hypertext Markup Language (HTML) e Hypertext Transfer Protocol (HTTP) de la Wordk Wide Web por parte de los clientes. El conjunto de API Servlet abstrae un marco de solución común para responder a peticiones HTTP.

iii. También son utilizados algunos JSP (Java Server Pages) que son una extensión del estándar Java definido sobre las extensiones de Servlets. La meta de los JSP es la creación y gestión simplificada de páginas Web dinámicas.

iv. El compartir información entre usuarios, tanto de la pizarra como los usuarios activos y el Chat, con el uso de RMI (Invocation Remote Method). Este enfoque hace un uso intensivo de las interfaces remotas. Cuando se desea crear un objeto remoto, se enmascara la implantación subyacente pasando una interfaz. Por consiguiente, cuando el cliente obtiene una referencia a un objeto remoto, lo que verdaderamente logra es una referencia a una interfaz, que resulta estar conectada a algún fragmento de código local que habla a través de la red.

De manera gráfica la arquitectura de nuestra aplicación sería la mostrada en la figura 5.40.

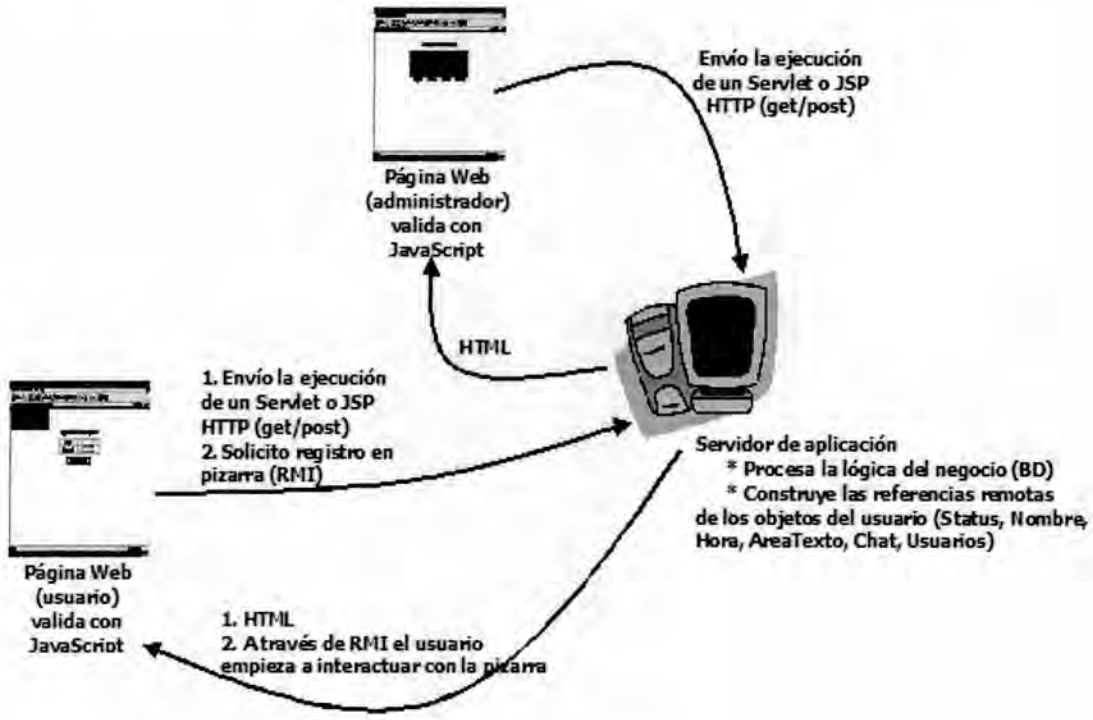

Figura 5.40 Arquitectura de la pizarra colaborativa

5.3.5.2 Resultados de la prueba

Las pruebas realizadas nos permitieron modificar de una manera fácil y rápida el comportamiento requerido para diferentes excepciones RMI, es importante mencionar que el código de la pizarra colaborativa no se modificó para las pruebas de incorporación del aspecto que maneja las excepciones RMI. Sin embargo si se modificó para generar diversas excepciones y así probar de una manera completa las diferentes opciones que se declaraban en el archivo XML.

Se destaca que al no necesitar modificar la aplicación existente, se logra separar el manejo de excepciones RMI (visto como un aspecto o un cruce de asuntos) de la funcionalidad básica del sistema, lo que permite tener código reutilizable y portable de una manera sencilla.

Además se puede cambiar la forma de manejar las excepciones RMI de una manera fácil, pues solo basta definir el nuevo comportamiento que se quiere, generar el código

correspondiente con la ejecución de la herramienta mostrada y compilar la aplicación incluyendo las clases generadas.

En el capítulo de resultados de la investigación y conclusiones se presenta más información relacionada con los resultados de las pruebas realizadas.

CAPÍTULO 6

6. Resultados de la investigación y conclusiones

En este capítulo se presentan los resultados y conclusiones obtenidas después de la investigación y desarrollo de la presente tesis. Se organizan los resultados en base a los diferentes conceptos manejados en la investigación y al final se dan las conclusiones sobre la herramienta desarrollada.

6.1 Ingeniería de software

La Ingeniería de Software ha evolucionado desde sus comienzos. Con esta evolución se introdujeron conceptos tales como el agrupamiento de instrucciones a través de procedimientos y funciones, módulos, bloques estructurados, tipos de datos abstractos, genericidad, y herencia entre otros. Estos conceptos proveen formas de abstracción y constituyen el medio para lograr una programación de alto nivel. Asimismo, los progresos más importantes se han obtenido aplicando estos conceptos junto con tres principios estrechamente relacionados entre sí: abstracción, encapsulamiento, y modularidad.

La Programación Orientada a Objetos (OOP) introdujo un avance importante forzando el encapsulamiento y la abstracción, por medio de una unidad que captura tanto funcionalidad como comportamiento y estructura interna. A esta entidad se la conoce como clase, y su principal característica es que enfatiza datos y algoritmos. A través de la OOP, o de otras técnicas de abstracción de alto nivel, se logra un diseño y una implementación que satisface la funcionalidad básica, y que presenta niveles de calidad aceptable. Sin embargo, existen conceptos entrecruzados que no pueden encapsularse dentro de una unidad funcional, debido a que atraviesan todo el sistema, o varias partes de él. Algunos de ellos son: sincronización, manejo de memoria, distribución, comprobación de errores, seguridad, y redes.

Las técnicas de implementación actuales tienden a plasmar los requerimientos usando metodologías de una sola dimensión, forzando a que todos los requerimientos sean

expresados en esa única dimensión. Esta dimensión resulta adecuada para la funcionalidad básica, pero no para los otros requerimientos, los cuales quedan diseminados a lo largo de la dimensión dominante. Es decir, que mientras el espacio de requerimientos es de n-dimensiones, el espacio de la implementación es de una sola dimensión. Dicha diferencia produce un mapeo deficiente de los requerimientos a sus respectivas implementaciones. Los dos síntomas más significativos de este problema son el *Código Mezclado (Code Tangling* y el *Código Diseminado (Code Scattering).* El *Código Mezclado* se presenta cuando en un mismo módulo de un sistema de software simultáneamente conviven más de un requerimiento.

Esto hace que en el modelo existan elementos de implementación de más de un requerimiento. El *Código Diseminado* se produce cuando un requerimiento está esparcido sobre varios módulos. Por lo tanto, la implementación de dicho requerimiento también queda diseminada sobre esos módulos. La combinación de estos síntomas afecta tanto al diseño como a la implementación de software.

La implementación simultánea de varios conceptos tiene dos efectos negativos. El primero es una baja correspondencia entre un concepto y su implementación. El segundo es una menor productividad de los desarrolladores, ya que se distraen del concepto principal.

Por otro lado, la implementación de varios conceptos en un mismo módulo lleva a un código poco reutilizable, de baja calidad, y propenso a errores. Esto se debe a que alguno de los tantos conceptos puede ser subestimado. Por último, la evolución es difícil de lograr debido a la insuficiente modularización. Los futuros cambios en un requerimiento implican revisar y modificar cada uno de los módulos donde esté presente ese requerimiento.

Por las razones expuestas se concluye que las técnicas tradicionales no soportan una separación completa de conceptos. Esto es, no permiten una modularización adecuada que facilite separar la implementación de determinados aspectos, distintos a la funcionalidad básica. Esta situación tiene un impacto negativo en la calidad del software, ya que esta característica es clave para producir un software comprensible y evolucionable.

Como respuesta a este problema nace la Programación Orientada a Aspectos (AOP). La AOP permite a los desarrolladores escribir, ver, y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e

intuitiva. La AOP es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los aspectos antes mencionados. Esto implica separar la funcionalidad básica de los aspectos, y los aspectos entre sí, a través de mecanismos que permitan la abstracción y la composición de los mismos, a fin de poder implementar todos los requerimientos del sistema. En la figura 6.1 se presenta las fases de desarrollo en la programación orientada a aspectos.

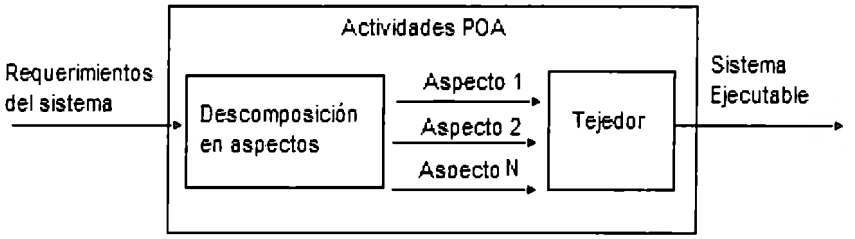

Figura 6.1 Fases de desarrollo en AOP

6.1.1 Diseño orientado a aspectos

Una filosofía de diseño orientado a aspectos, que se propone en esta tesis, consta de 4 puntos:

1. Un objeto es algo: un objeto existe por sí mismo, es una entidad.
2. Un aspecto no es algo. Es algo sobre algo: un aspecto se escribe para encapsular un concepto entrecruzado. Por definición un aspecto entrecruza diferentes componentes, los cuales en la OOP son llamados objetos. Si un aspecto no está asociado con ninguna clase, entonces entrecruza cero clases, y por lo tanto no tiene sentido en este contexto. Luego, para que un aspecto tenga sentido debe estar asociado a una o más clases; no es una unidad funcional por sí mismo.
3. Los objetos no dependen de los aspectos: Un aspecto no debe cambiar las interfaces de la clase asociada a él. Solo debe aumentar la implementación de dichas interfaces. Al afectar solamente la implementación de las clases y no sus interfaces, la encapsulación no se rompe. Las clases mantienen su condición original de cajas negras, aún cuando puede modificarse el interior de las cajas.
4. Los aspectos no tienen control sobre los objetos: Esto significa que el ocultamiento de información puede ser violado en cierta forma por los aspectos

> porque éstos conocen detalles de un objeto que están ocultos al resto de los objetos. Sin embargo, no deben manipular la representación interna de los objetos más allá de lo que sean capaces de manipular el resto de los objetos. A los aspectos se les permite tener esta visión especial, pero debería limitarse a manipular objetos de la misma forma que los demás objetos lo hacen, a través de la interfaz.

Esta filosofía de diseño permite a los aspectos hacer su trabajo de automatización y abstracción, respetando los principios de ocultamiento de información e interfaz manifiesta. Creo que esta filosofía es una política de diseño totalmente aceptable, y que se debe conocer y aplicar durante el desarrollo orientado a aspectos.

Si bien es una excelente política de diseño, no deberíamos quedarnos solamente en esto, sino que deberíamos llevarla más allá, haciéndola formar parte de las reglas del lenguaje orientado a aspectos. Esto es, que no sea una opción, sino que sea forzada por el lenguaje. El mismo problema que antes existía entre los distintos componentes de un sistema, está presente ahora entre objetos y aspectos. Es decir, los aspectos no deben inmiscuirse en la representación interna del objeto, por lo tanto se necesita una solución similar: el lenguaje debería proveer una interfaz para que aspectos y objetos se comuniquen respetando esa restricción.

Esta es una de las desventajas de los lenguajes orientados a aspectos actuales. El desafío está en construir lenguajes orientados a aspectos capaces de brindar mecanismos lingüísticos suficientemente poderosos para respetar por completo todos los principios de diseño.

6.1.2 Programación orientada a aspectos

En el capítulo dedicado a la orientación a aspectos se estableció la situación actual de AOP y se tienen algunas comparaciones entre OOP y AOP. Teniendo en cuenta los valores mencionados en ese punto y las métricas que ahí se describen podemos mencionar varios puntos de interés, por ejemplo, la programación orientada a aspectos trata de ayudar al problema del código enredado.

AspectJ apoya a reducir drásticamente la cantidad de código relacionado con la detección y manejo de excepciones. En algunos casos se ha podido reducir esa codificación hasta en un factor de 4 [9]. También se sabe que con respecto a la implementación en Java "plano", AspectJ da un mejor apoyo para configuraciones diferentes, para mejores desempeños, mayor tolerancia a cambios en las especificaciones, mayor apoyo para el desarrollo incremental, mejor re-uso, validación automática de contratos en aplicaciones que usan el *"frameworh"*, y programas (códigos) más claros.

6.1.2.1 Costo de AOP

Por otra parte, es válido preguntarse cuál es el costo de trabajar con AspectJ. Esta cuestión es lo que hace tan atractiva a la AOP, y en particular a AspectJ, y la respuesta es que el costo es mínimo. Si el programador ya cuenta con conocimientos sólidos en Java, capacitarse en AspectJ es una tarea trivial, debido a que este último lenguaje es una extensión del primero.

Aprender a trabajar con aspectos en AspectJ es sumamente sencillo, dada la naturaleza declarativa de los mismos. Una característica importante de los aspectos es que son fácilmente "enchufados" y "desenchufados" (plug-in, plug-out) a la funcionalidad básica.

Esto le permite al programador ir experimentando con aspectos, sin tener que modificar la funcionalidad básica, resultando en un aprendizaje gradual y efectivo. Todo lo que se experimentó y analizó, puede resumirse en una sola afirmación, que establece por sí misma la importancia de este nuevo paradigma *"con muy poco esfuerzo, se obtienen ganancias importantes en la calidad, no sólo del producto final, sino también en el proceso que desarrolla el producto"*. Lo que me hace pensar que la AOP es una de las ramas con mayor futuro dentro de la Ingeniería de Software.

6.1.2.2 Ventajas de AOP

En base a la experiencia lograda con el desarrollo de la presente tesis, es posible marcar varias ventajas:

- En primer término, al separar la funcionalidad básica de los aspectos, se aplica con mayor intensidad el principio de divide y vencerás.

- También hay que destacar que el ambiente de desarrollo es un ambiente de N-dimensiones, donde es posible implementar el sistema con las dimensiones que sean necesarias, y no en una única dimensión "sobrecargada".
- La AOP enfatiza el principio de mínimo acoplamiento y máxima cohesión.
- La separación de los aspectos, desde el diseño y no sólo en la implementación y ejecución, es un paso importante que se está dando en la ingeniería de software, pero que aún se está refinando, sobre todo en cuestión de eficiencia.
- Los lenguajes de aspectos de dominio específico juegan un papel crucial en la programación orientada a aspectos y son preferibles a los lenguajes de propósito general, porque soportan mejor la separación de funcionalidades. Pero desde el punto de vista de la empresa, es más cómodo y económico que se utilicen lenguajes de propósito general por lo que se refiere a la capacitación.

Al realizar comparaciones entre implementaciones realizadas con la programación orientada a objetos y las resultantes después de realizar las modificaciones para trabajar con programación orientada a aspectos [15], se tienen resultados muy interesantes:

- Al utilizar aspectos la ventaja más destacable es que el código que implementa la funcionalidad básica es mucho más limpio y entendible.
- Otra ventaja importante es una mayor facilidad para realizar cambios y modificaciones, como también acciones para remover o agregar comportamiento.
- En cuanto al desarrollo, el mismo se vuelve mucho más intuitivo y natural, dada la naturaleza declarativa de los *puntos de enlace*, y al comportamiento adicional que permite definir AspectJ.

6.2 Manejo de excepciones

El manejo de excepciones en sistemas grandes de software puede consumir una gran cantidad de recursos en el desarrollo. Se deben pensar en excepciones por todas partes del ciclo de desarrollo, lo cual es generalmente pesado.

En fases tempranas del desarrollo, los diseñadores deben identificar qué es lo que la aplicación debe y que no debe hacer. Por ejemplo calcular una distancia dado un parámetro en metros y otro parámetro en pies resulta en un valor incorrecto, un error. Tratar de calcular tal valor se puede considerar un "situación excepcional." Si desde el diseño se anticipa esta situación, se pueden decidir qué hacer si esto ocurre (e.g. convertir uno de los valores), o prevenir que ocurra. Esto es parte del esfuerzo humano para decidir exactamente qué es "normal" y qué es "excepcional." Tomar malas decisiones o dejar de identificar situaciones excepcionales puede llevar a resultados catastróficos.

El desempeño normal y excepcional del sistema, que fueron identificados en las fases tempranas de desarrollo, son implementados en el programa. Esta traducción es muy difícil y principalmente el proceso es manual, que se expone a la capa de errores humanos. Si nos enfocamos al proceso de combinar el desempeño normal de la aplicación y las conductas excepcionales se reduce la posibilidad de errores en la implementación cuando se hace esa combinación.

Muchos trabajos en lenguajes de programación han tenido la meta de proveer un mejor apoyo para la detección y manejo de excepciones. Contratos, introducido por Hoare son un ejemplo, que consiste en la definición de pre-condición, post-condición e invariantes que determinan cómo usar y qué esperar de una entidad computacional. Este concepto se convirtió en la metodología de diseño conocida como Diseño por Contratos [10]. Algunos lenguajes como Eiffel son la referencia de ejemplo, proveen constructores para soportar contratos. Mucha de la complejidad en la propagación de una excepción puede ser evitada tomando las ventajas de estos mecanismos. Muchos lenguajes de programación, de hoy en día, incluyen el poder construir la definición de las excepciones de la aplicación y el lanzar y capturar dichas excepciones.

Codificar la detección y manejo de errores es, de cualquier modo, todavía un proceso difícil que requiere una disciplina estricta de los programadores. Y la realidad es que la mayoría del software que se escribe hoy en día usa lenguajes de programación que proveen poca ayuda. Afortunadamente, esto está cambiando.

Pero existe un efecto secundario al codificar en las aplicaciones las excepciones que no pueden agregarse simplemente incluyendo un más poderoso mecanismo de detección y manejo de excepciones en el lenguaje de programación. Este efecto secundario consiste en

el código enredado entre el código para que el programa haga lo que debe hacer- i.e. su función básica- y la codificación para la detección y manejo de excepciones.

El código enredado es, en parte, una consecuencia del lenguaje de programación y, en parte, una consecuencia de decisiones de diseño. Pero el código enredado también refleja una decisión importante del diseño que está implícita en los lenguajes de programación: que la abstracción de "qué hacer" en una operación es la misma abstracción de "cómo descubrir y reacciona a excepciones" en esa operación. Esto implícita varios problemas:

- Re-uso: en la práctica, es improbable que la subclase pueda re-usar un método de la superclase que hace y redefine su reacción propia a la manipulación de la excepción (o vice-versa) sin redefinir el método entero. Estos requerimientos significan y prevención y disciplina en la parte de los programadores, que no es probable que ocurra.
- Evolución: en la práctica, es también difícil usar un método en un contexto con características diferentes de las excepciones de la aplicación.
- Documentación: el código puede llegar a ser muy engorroso, haciendo difícil de entender qué es exactamente lo que el método supuestamente debe hacer.
- Pérdida de abstracción: características de desempeño del tipo "todos los métodos que llamen el servicio de *"registry"* deben ser preparados para fallas de la red y debe reintentar llamar 3 veces antes de darse por vencido" no puede tener una correspondiente abstracción simple en el código. Por eso la implementación de este tipo de especificaciones se esparce por todas partes de la codificación, haciendo con esto un mantenimiento difícil de realizar.

Considerando la programación orientada a aspectos como una técnica de la programación para la modularización de intereses o asuntos (concerns) que atraviesan o cortan (cross-cut) la funcionalidad básica de los programas y teniendo en cuenta que el manejo de excepciones es referido como uno de esos cross-cut concerns, se sugiere que debiera ser posible lograr una separación relativa entre la codificación funcional y la codificación del manejo de excepciones.

6.3 Conclusiones de la investigación y herramienta desarrollada

Se ha presentado una herramienta que apoya el entendimiento y el uso de la programación orientada a aspectos y la programación declarativa, objetivo principal de la tesis. Lo que nos lleva tener las ventajas de que el código, manejado como aspectos dentro de la aplicación, será un código menos enmarañado y más natural; lo que facilita a razonar sobre los conceptos, tanto de la funcionalidad básica como de los que no lo son, ya que están separados y con dependencia mínima, es decir, el código que implementa la funcionalidad básica es un código mucho más limpio y entendible.

Ya que el manejo de excepciones de forma declarativa es un mecanismo que permite ahorrar tiempo a los programadores, durante el desarrollo y mantenimiento, la herramienta hace que posibles modificaciones tengan un impacto mínimo en los sistemas en general. Es decir, el código será reutilizable y se podrá acoplar y desacoplar cuando sea necesario.

También, al estar todas las acciones del manejo encapsuladas en el aspecto y claramente identificadas, se facilita enormemente la realización de cambios o modificaciones, como también acciones para remover o agregar comportamiento. En cuanto al desarrollo, el mismo se vuelve mucho más intuitivo, más natural, dada la naturaleza declarativa de los cortes ("poincuts") y los avisos ("advices") en AspectJ.

Se presentan, además del aspecto de manejo de excepciones, dos aspectos más (manejo de log de excepciones lanzadas y seguimiento de puntos de corte) y un ejemplo con el que se realizaron varias pruebas de la herramienta, que ayudan a entender el ámbito de la programación orientada a aspectos, en particular la utilización de AspectJ. Con esto se da un primer acercamiento al estudiante a lo que es la programación orientada a aspectos y a la programación declarativa. Además se apoya al mejor entendimiento del manejo de excepciones en Java y en particular a las excepciones de RMI.

Ya que el uso del manejo de excepciones puede afectar el re-uso de código, diversos papers reportan, que, "concurrencia y manejo de excepciones tienen un impacto substancial en estructuras de software y son factores importantes que afectan el re-uso." Pero no se encontró, en la investigación realizada, un estudio que provee soluciones para componer

los desempeños excepcionales con las clases. Pero hay que destacar que Walker et al. [31] ha hecho un estudio inicial de usabilidad de programación orientada a aspectos usando una versión anterior de AspectJ. Su estudio tenía un propósito y objetivo diferente al de esta investigación. Midieron cómo los programadores percibieron programas escritos en AspectJ vs programas escritos en Java plano. No se enfocaron a excepciones en particular. Nuestra experiencia con nuestro estudio refuerza algunos de sus hallazgos, a saber, que los aspectos deben tener un objetivo bien definido para efectos del código funcional. Los aspectos contrato, como los aspectos de manejo de excepciones, tienen una definición clara de su objetivo que se puede asegurar usando la combinación de aspectos abstractos y concretos.

La herramienta apoya la implementación y drásticamente reduce la porción de la codificación relacionada a la detección y manejo de excepciones. En el mejor escenario se pudo reducir esa codificación por un factor de cuatro, en base a lo descrito en el capítulo 3. Esa reducción refleja cortes de información redundante que era impuesto por Java. También hallamos, con respecto a Java plano, que con AspectJ y la herramienta desarrollada, se provee mejor apoyo para configuraciones diferentes de desempeños excepcionales, más tolerancia para cambios de especificaciones de desempeños excepcionales, mejor soporte para el desarrollo incremental, mejor re-uso y códigos más limpios.

6.3.1 Modificación al patrón de manejo de excepciones

Como ya se comentó, el patrón que se utilizó como base para el manejo de excepciones se adecuo, es decir, se le agregó el obtener la excepción original dentro del "*around*", ver la función QuitaWrapper de la clase ExHanAspAbstract, con lo que en el "*around*" se puede manejar la excepción que fue lanzada originalmente y llevar acabo su manejo correspondiente, en base a lo definido en el archivo de configuración XML, si después de ello se desea enviar una excepción encapsulada, dependiendo el manejo declarado, se puede encapsular con la excepción *runtime* propia, o inclusive se le puede decir que encapsule una excepción de otro tipo y después de ello el aspecto PreserveRemoteException trabajará normalmente para que el cliente reciba la excepción de

forma normal, si es que en el manejo declarado y generado por la herramienta no se proceso y/o se generó una nueva excepción.

6.4 Problemas durante el desarrollo

En las siguientes secciones se presenta una serie de errores y problemas que se tuvieron en el desarrollo de la herramienta, esperando que sirvan para el buen entendimiento, no solo de la herramienta, sino del paradigma de la programación orientada a aspectos y en particular a la programación del paradigma en Java con AspectJ.

6.4.1 Manejo de errores y problemas comunes en AspectJ

Por todas partes en el ciclo de desarrollo con AOP y AspectJ, se tendrán códigos que al intentar compilar o recompilar pueden producir errores. A continuación se muestran algunos de los errores y limitaciones que experimente al desarrollar la tesis, esperando que la información ayude en los tiempos difíciles de iniciar a utilizar el lenguaje AspectJ.

6.4.1.1 Errores al compilar

Un error de compilación, puede ocurrir cuando el compilador de Java o de AspectJ es ejecutado para un archivo individual o para un grupo de archivos.

Compilación errónea

Considere el siguiente código:

```
public class CompileTest {
        public CompileTest() {
                //Constructor de default
        }

        private aspect CompileTestAspect {
                pointcut grabConstructor() : execution(CompileTest.new());
                before() : grabConstructor() {
                        System.out.println("Estoy en el constructor");
                }
        }
}
```

Si se compila el código anterior, considerando que en el classpath se tiene el jar de AspectJ necesario:

```
java CompileTest.Java
```

El compilador producirá el siguiente error:

```
CompileTest.Java:6: ';' expected
private aspect CompileTestAspect {
^
CompileTest.Java:6: cannot resolve symbol
symbol : class aspect
location: class CompileTest
private aspect CompileTestAspect {
^
2 errors
```

Es decir, no se puede utilizar el compilador estándar de Java para compilar clases que contienen código para AspectJ. Se necesita usar el compilador de AspectJ, ajc.

Unable to Find Aspectjtools.jar

Usando el mismo código del ejemplo anterior, podemos ver otro error cuando se usa el compilador de AspectJ para compilar la clase. Considere el siguiente comando para la compilación:

```
ajc –classpath "c:\aspectj\lib]aspectjrt.jar" CompileTest.Java
```

No se compilara con éxito, pues se genera el siguiente error:

```
Can't find org.aspectj.lang.JoinPoint on your classpath anywhere.
```

Se necesita incluir aspectjrt.jar en el classpath.

Out of Memory Error

En ocasiones la JVM asociada al compilador se ejecuta fuera de memoria durante la compilación del proyecto. Esto para proyectos largos, con numerosas clases y aspectos. Cuando esto ocurre, se necesita modificar el script o archivo batch del compilador ajc y

cambiar la opción de memoria de la JVM con la bandera -Xmx64M con un valor grande como -Xmx128M o -Xmx256M.

Wrong JSDK

Cuando se instala el compilador de AspectJ t se soporta archivos JAR en el sistema local, el wizard pregunta la ubicación del Java Software Development Kit (JSDK). Este path es puesto en el script de startup del compilador, que se localiza en el directorio /bin de AspectJ, el archivo batch incluye la siguiente información:

```
@echo off
REM This file generated by AspectJ installer
REM Created on Thu Jan 27 19:26:03 CST 2005 by Hector

if "%JAVA_HOME%" == "" set JAVA_HOME=C:\Archivos de
programa\Java\jre1.5.0_01
if "%ASPECTJ_HOME%" == "" set ASPECTJ_HOME=c:\aspectj1.2

if exist "%JAVA_HOME%\bin\Java.exe" goto haveJava
if exist "%JAVA_HOME%\bin\Java.bat" goto haveJava
if exist "%JAVA_HOME%\bin\Java" goto haveJava
echo Java does not exist as %JAVA_HOME%\bin\Java
echo please fix the JAVA_HOME environment variable
:haveJava
"%JAVA_HOME%\bin\Java" -classpath
"%ASPECTJ_HOME%\lib\aspectjtools.jar;%JAVA_HOME%\lib\tools.jar;%CLASSPATH
%" -Xmx64M org.aspectj.tools.ajc.Main %*
```

Si se cambia la versión de JSDK, y para no reinstalar AspectJ, pues no es necesario, solo se debe editar y modificar la línea de JAVA_HOME

No Java Compiler

El compilador de AspectJ requiere el uso de varios archivos JAR en el sistema. El primero es el aspectjrt.jar localizado el directorio /lib de AspectJ. El segundo es el tool.jar que se localiza en el /lib del directorio de la instalación de Java SDK, y si el sistema no puede localizar el archivo tools.jar, se producirá un error.

O también se puede producir un error si el path JAVA_HOME no es valido, pues lo primero que hace el compilador AspectJ es tratar de ejecutar un comando Java, el error puede ser como:

```
Java does not exist as c:\j2sdk1.4.0_011\bin\Java
please fix the JAVA_HOME environment variable
```

```
The system cannot find the path specified.
```

Para este caso, se necesita verificar y modificar la variable de ambiente JAVA_HOME.

6.4.1.2 Errores de runtime

Errores runtime pueden ocurrir durante la ejecución de una aplicación que ha sido reforzada con código en AspectJ. Otras veces, el código no trabaja como se espera. En esta sección tenemos las situaciones más comunes de este tipo de problemas.

Stack Overflow

Sino se es cuidadoso, una condiciona de overflow puede ocurrir. Considere el siguiente código Java:

```
public class Recursive {
        public print() {
                System.out.println("Test");
        }

        public static void main(String args[]) {
                Recursive recursive = new Recursive();
                recursive.print();
        }
}
```

Ahora construyámosle un simple aspecto que coincida e imprima declaraciones en el código Java anterior:

```
public aspect RecursiveAspect {
        pointcut print() : call(* System.out.println(..));
                before() : print() {
                System.out.println(thisJoinPoint.getSignature());
        }
}
```

Este aspecto caza en la llamada al método println() y despliega la firma o signature donde el "match" fue encontrado. Si se compila y ejecuta este código, se recibirá un "stack overflow". Además, la salida no será lo que se esperaba- se esperaba una sencilla firma, pero las líneas de salida serán muchas y se necesitara el scroll en la pantalla.

El problema es en la llamada al método System.out.println() en el "advice" del aspecto, pues se cae en una situación recursiva. Es buena idea usar "this()" o "target()" para limitar el alcance del "match"; o se puede, también, usar el operador de negación "!", para excluir clases o aspectos.

No Package Declaration

Cuando se usan "join points" que son parte de un paquete, se debe tener cuidado de como los "join points" son escritos en un "pointcut". Considere el siguiente código:

```
package my_package;
public class CompileTest2 {
        public CompileTest2() {
        }
        public int returnOne() {
                return 1;
        }
        public static void main(String args[]) {
            CompileTest2 compileTest = new CompileTest2();
            compileTest.returnOne();
        }
}
```

La clase CompileTest2 esta contenida en una declaración de paquete, my_package. Ahora considere el siguiente aspecto:

```
public aspect CompileTest2Aspect {
        pointcut grabOne() : call(* CompileTest2.returnOne());
        before() : grabOne() {
                System.out.println("I've got One");
        }
}
```

Este aspecto contiene una definición de un "join point" que hace "match" con el método returnOne() de la clase compileTest2. cuando el compilador AspectJ inicia el proceso de tejido, trata de encontrar el apropiado código base de clase/método en su directorio. Desafortunadamente no se tiene la clase CompilerTest2 en el directorio, pues es una clase en el path de my_package. Lo que se necesita hacer es usar el paquete o los wildcards, a continuación se muestran las dos opciones:

```
public aspect CompileTest2Aspect {
    pointcut grabOne() :
        call(*my_package.CompileTest2.returnOne());
    before() : grabOne() {
        System.out.println("I've got One");
    }
}

public aspect CompileTest2Aspect {
        pointcut grabOne() : call(* *.returnOne());
        before() : grabOne() {
                System.out.println("I've got One");
        }
}
```

More Subtype Matching

Otro problema común es cuando se escribe un aspecto que se quiere que obtenga todas las llamadas a métodos declarados en una clase o sus descendientes, es decir:

```
call(* Foo+.*(..))
```

Este call es para hacer "match" de todos los métodos de Foo o sus clases derivadas, sin importar el tipo de retorno, nombre del método, o el tipo o número de parámetros del método. Pero esto no hará "match" con los métodos toString(), hashCode(), y algunos otros, para resolver esto, se puede usar lo siguiente:

```
call(* *(..)) && target(Foo)
```

Aquí, call hace "match" con todas las llamadas a métodos en cualquier método del sistema, al combinarse con target(Foo) para obtener todas las llamadas cuando el "target" es un objeto Foo.

Matching All Types

Si se quiere obtener todos los tipos en un paquete o subpaquetes, por ejemplo de my_package, necesita:

```
call(my_package..*);
```

Note los dos puntos (..) entre el texto y el *. Estos dos puntos son un indicador wildcard que se usa también para indicar cualquier tipo o número de parámetros.

```
call(void int myFoo(..));
```

Improper Use of args()

Otro posible problema es cuando se definen "join points" y "pointcuts" con operadoradores lógicos y se utiliza "args()". Por ejemplo:

```
public void showTwo(int a, float b) {
}
```

Suponga que se quiere construir un "pointcut" para hacer "match" con la firma de showTwo, y el "pointcut" debe tener dos parámetros de tipo int y float. Se usa call() y "args()" combinados con el operador lógico AND. Una posibilidad es:

```
pointcut matchShowTwo() : call(public void showTwo(..)) &&
        args(int) &&
        args(float);
```

Parece que se ha creado el "pointcut" que hace el "match" con las llamadas al método showTwo(), que tiene cero o más parámetros y también dos argumentos. Pero. args(int) && args/float) realmente indican que debe hacer "match" con el método showTwo() que tenga un argumento donde el primer parámetro es int o float. Obviamente, esto no ocurrirá, con lo que el "pointcut" nunca será disparado. El problema se resuelve con "args()", que debe usar el separador coma (,) entre los parámetros:

```
pointcut matchShowTwo() : call(public void showTwo(..)) &&
        args(int, float);
```

Este nuevo "pointcut" le dice al sistema que le de las llamadas al método showTwo(), cuando los argumentos del método fueron definidos como un int y seguido de un float.

Using call() && execution()

Un error muy común ocurre durante el desarrollo de aspectos donde se quiere obtener las llamadas y ejecuciones a un método, por ejemplo:

```
call(* Foo.getFoo(..));
```

Con lo cual se obtienen las llamadas al método Foo.getFoo(), pero no se considera:

```
execution(* Foo.getFoo(..));
```

Con el cual se obtienen las ejecuciones del método Foo.getFoo(). Entonces, ¿Que se obtiene con el siguiente "join point"?

```
call(* Foo.getFoo(..)) && execution(* Foo.getFoo(..));
```

La respuesta es ¡NADA!, pues nunca se tendrá un punto en la ejecución del código donde se tengan ambas, la llamada y la ejecución, al mismo tiempo.

6.5 Trabajos relacionados

Una cantidad considerable de trabajos se han hecho para reforzar la programación orientada a objetos para la detección y manejo de excepciones. Varios de ellos se mencionan en las referencias de la presente tesis, y en ellos se describen diferentes mecanismos de sistemas orientados a objetos. Otro trabajos presentan la detección y manejo de excepciones para propósitos específicos como sistemas de información de banco de datos - intensivos, software de componentes basado en tiempo real o tolerancia a fallas. Esos papers tratan del mecanismo básico de habilitar un programa orientado a objetos para lanzar y cachar excepciones.

Además ha habido varias propuestas para integrar contratos en Java. Algunas de esas propuestas tienen un enfoque AOP. Es decir, se intenta soportar pre- y post-condiciones con una biblioteca mezclada y el uso de patrones de diseño.

6.6 Trabajos futuros

El campo de la orientación a aspectos es un campo de investigación "joven" en el que se abre un horizonte amplio. Desde la definición de los distintos aspectos mediante lenguajes de propósito especifico, hasta middleware orientados a aspectos, pasando por las etapas de análisis y diseño orientadas a aspectos, aún quedan muchos campos que abordar, por lo que se cree interesante abrir nuevas líneas de investigación en este tema.

El trabajo realizado en esta tesis y el análisis de AOP, me permite observar también, que la AOP es un nuevo paradigma que aún adolece de madurez y formalidad. Debido a esto, plantea nuevas líneas de investigación tales como: analizar cómo aplicar aspectos en las diversas etapas del ciclo de vida del software, en el análisis, en el diseño, las pruebas y en la documentación; investigar sobre la formalización de los aspectos; observar cómo es la integración de los aspectos con las aproximaciones, métodos, herramientas, y procesos de desarrollo existentes; desarrollar herramientas orientadas a aspectos que respeten los principios de diseño, entre otras.

Un próximo paso es seguir trabajando en cómo incluir aspectos en el diseño de un sistema de software. El diseño es una parte crítica de cualquier proceso, y es necesario investigar la forma de incluir los aspectos durante esta etapa. La mayoría de los trabajos de investigación relacionados, buscan maneras convenientes de extender UML, para que este lenguaje de diseño sea capaz también de expresar aspectos. Debido a que UML es el lenguaje de diseño con mayor respaldo dentro de la Ingeniería de Software, y es considerado como un estándar, se buscan los medios para expresar aspectos en UML.

Bibliografía

Referencias

[1] G. Kiezales. J. Lamping, A. Mendhekar, C.Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming, Xerox Palo Alto Research Center, 1997.

[2] Laddad *R., "I want my AOP"*.. Part 1,2,3 from JavaWorld, Enero-Marzo-Abril 2002.

[3] Cugola G., Ghezzi C., Monga M., "Language Support for Evolvable Software: An Initial Assessment of Aspect-Oriented Programming". Proceedings of the International Workshop on the Principles of Software Evolution. IWPSE99, Julio 1999.

[4] Soares S., Borda P., Progressive implementation with aspect-oriented programming. Informatics Center, Federal University of Penambuco

[5] Tristram C., "Untangling Code". in the January/February 2001 issue of the Technology Review.

[6] Laddad R., AspectJ in Action, Practical Aspect-Oriented Programming

[7] Flaviu C., Exception Handling and Software Fault Tolerance, in IEEE Transactions on Computers, Vol. c-31, No. 6, pp. 531540, June 1982.

[8] Katara M., Katz S., Architectural Views of Aspects

[9] Lippert M. y Lopes C., A Study on Exception Detection and Handling Using Aspect-Oriented Programming

[10] B. Meyer: Object-Oriented Software Construction, Prentice Hall, New Jersey, 1997.

[11] Sirvente A. y Murazzo M., Universidad Nacional de San Juan. Tendencias en educación en la sociedad t. i. http://www.portalzonda.com.ar/Congreso/papers/1998/EDU14.ppt (Abril 1, 2005).

[12] http://alejandria.hacer.ula.ve/2002/muelles/concepto.htm (enero 25, 2005).

[13] Congreso Internacional de Tecnología, Educación y Desarrollo sostenible. http://www.edutec.es/edutec01/edutec/TSE.html (Marzo 15, 2005).

[14] González J., Jiménez G., Implementación Automatizada de Patrones de Código. Instituto Tecnológico y de Estudios Superiores de Monterrey

[15] Asteasuain F., Contreras B., Estévez E., Fillottrani P., Programación Orientada a Aspectos: Metodología y Evaluación. Departamento de Ciencias e Ingeniería de la Computación Universidad Nacional del Sur.

[16] Chuck Cavaness. Programming Jakarta Struts, 2nd Edition. Second Edition June 2004.

[17] Lucena L. M. http://wwwdi.ujaen.es/~mlucena/decl.html (Marzo 14, 2005).

[18] http://eclipse.org/aspectj (Diciembre 10, 2004).

[19] Karl J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. 1996. http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html (Mayo 1, 2005).

[20] http://www.usabilityfirst.com/groupware/cscw.txl (Enero 20, 2005).

[21] Ricadela A. (Nov. 18, 2002) Aspect-oriented programming makes it easier to reflect complex processes. InformationWeek. http://www.informationweek.com/story/showArticle.jhtml?articleID=6504114

[22] Shomrat M. and Yehudai A. Obvious or Not? Regulating Architectural Decisions Using Aspect-Oriented Programmin.

[23] Pavón F. y otros (1998): Las nuevas tecnologías ayudan a la mejora del aprendizaje y comunicación en la docencia e investigación universitaria. Hipertexto- Hipermedia. En Sales, D. y otros (Eds.): II CONGRESO SOBRE REFORMA DE LOS PLANES DE ESTUDIOS Y CALIDAD UNIVERSITARIA: DOCENCIA INVESTIGACIÓN Y GESTIÓN. CITI. Universidad de Cádiz. (135-138) http://tecnologiaedu.us.es/bibliovir/pdf/9.pdf (Enero 23, 2005).

[24] http://www.diarioti.com/gate/p.php (Diciembre 10, 2004).

[25] Tecnologías de la Información http://www.wipo.int/it/es/index.html (Febrero 10, 2005).

[26] Contact: Steve Buckley e.steve@commedia.org.uk Press Point: 12 diciembre 2003, Ginebra, Palexpo, Room Saleve http://www.choike.org/nuevo/informes/1382.html (Enero 12, 2005).

[27] SEBASTIÁN, Sistema Educativo BASado en TecnologÍA INternet http://www.it.uc3m.es/~sebas/doc-publica/indice.html (Marzo 25, 2005).

[28] bibliotecnic.org http://www.bibliotecnic.org/campus/mod/forum/discuss.php?d=6 (Febrero 19, 2005).

[29] Calzadilla Ma. E., Aprendizaje Colaborativo y tecnologías de la información y la comunicación. Universidad Pedaógica Experimental Libertador, Venezuela. http://www.campus-oei.org/revista/deloslectores/322Calzadilla.pdf (Marzo 10, 2005).

[30] Lopes C., "D: A Language Framework for Distributed Programming", Ph.D. thesis, Northeastern University, Noviembre de 1997.

[31] R. J. Walker, E. L. A. Baniassad, G. C. Murphy: An Initial Assessment of Aspect-oriented Programming, in Proceedings of 21st International Conference on Software Engineering (ICSE), IEEE Computer Society Press, 1999.

[32] log4j. http://logging.apache.org/log4j/docs/manual.html (Mayo 29, 2005).

[33] http://creativecommons.org/licences/by/1.0 (Junio 2, 2005).

[34] http://Java.sun.com/docs/books/tutorial (Enero 17, 2005).

Referencias de interés

Java

- Doug Lea, SUNY Oswego. Java: 1.5 and Beyond, dl@cs.oswego.edu http://gree.cs.oswego.edu

Programación orientada aspectos

- Kiezales G., Lamping J., Mendhekar A., Lopes C., Loingtier J. y Irwin J. Aspect-Oriented Programming, Xerox Palo Alto Research Center, 1997. gregor@parc.xerox.com, http://www.parc.xerox.com/aop
- Papapetrou O. y Papadopoulos G., Aspect Oriented Programming for a component based real life application: A case study, Department of Computer Science University of Cyprus 75 Kallipoleos Str., P.O.Box 20537, Nicosia, Cyprus cspapap,george@cs.ucy.ac.cy
- Naoyasu U., Aspect-Oriented Programming with Model Checking, Systems Integration Technology Center, Toshiba Corporation Tokyo, Japan naoyasu.ubayashi@toshiba,co,jp. Tetsuo Tamai, Interfaculty Initiative in Information Studies, Graduate School, University of Tokyo Tokyo, Japan tamai@graco.c.u-tokyo.ac.jp
- Martin G., Introduction to Aspect-Oriented Programming

AspectJ

- AspectJ 1.2 Readme
- AspectJ Documentation and Resources
- Nieto J., AspectJ en la Programación Orientada a Aspectos
- Griswold B., Hilsdale E., Hugunin J., Kersten M., Kiczales G., Palm J., Aspect-Oriented Programming with AspectJ, the AspectJ.org team, Xerox PARC, Aspectj.org
- Gradecki J., Lesiecki N., Mastering AspectJ, 2003.
- AspectJ Programming Guide
- Viega J. y Voas J., Can Aspect-Oriented Programming Lead to More Reliable Software?
- Lesiecki N., Improve modularity with aspect-oriented programming AspectJ brings AOP to the Java language ndlesiecki@apache.org?cc=&subject=Improve modularity with aspect-oriented programming) Technical Team Lead, eBlox, Inc. 01 Jan 2002

Manejo de excepciones

- A. Bordiga: Exceptions in object-oriented languages, in ACM SIGPLAN Notices, Vol, 21, No. 10, pp. 107-119, October 1986.

- A. Bordiga: Language Features for Flexible Handling of Exceptions in Information Systems, in ACM Transactions on Database Systems, Vol. 10, No. 4, pp. 565-603, Dec. 1985.
- C. Dony: Exception Handling and Object-Oriented Programming: towards a synthesis, in Proceesings of OOPSLA/ECOOP '90, SIGPLAN Notices, Vol. 25, No. 10. ACM Press, October 1990.
- J. B. Goodenough: Exception Handling: Issues and Proposed Notation, in Communications of the ACM, Vol. 18, No. 12, 1975, pp. 683-696.
- J. Lang, D. B. Stewart: A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology, in ACM Transactions on Programming Languages and Systems, Vol. 20, No. 2, pp. 274-301, March 1998.
- P. Black: Exception Handling: The Case Against, Technical Report 82-01-02, Department of Computer Science, University of Washington, 1982. Reprinted in May 1983.
- P.M. Melliar-Smith, B. Randell: Software Reliability: the Role fo Programmed Exception Handling, in Proceedings of ACM Conference Language Design for Reliable Systems, SIGPLAN Notices, Vol, 12, No. 3, pp. 95-100, March 1977.

Patrones de diseño

- Gamma Erich y otros. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley; 1998.

Otros

- Zhhao J., Data-Flow-Based Unit Testing of Aspect-Oriented Programs, Department of Computer Science and Engineering, Fukuoka Institute of Technology 3-30-1Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan zhao@cs.fit.ac.jp
- Bryant A., Catton A., Kris De Voider y Gail C. Murphy, Explicit Programming
- Clemente P., Hernandéz J., Interconexión de Componentes CCM Mediante Programación Orientada a Aspectos
- J. Bruno, U. Hölzle, M. Karaorman: jContractor: A Reflective Java Library to Support Design By Contract, Technical Report TRCS98-31, Department of Computer Science, University of California, Santa Barbara, December 1998.
- R. Kramer: iContract – The Java Design By Contract Tool, in Proceedings of Tools 26 - USA '98, IEEE Computer Society.

Programación declarativa

- Narayanan Jayaratchagan, O'Really on Java.com; 04/21/2004, http://www.onJava.com/pub/a/onJava/2004/04/21/declarative.html (Diciembre 3, 2004).

Programación orientada a aspectos

- http://aosd.net home de la conferencia anual de Aspect.Oriented software Developmen

Lenguajes orientados a aspectos

- AspectJ homepage. http://eclipse.org.aspectj.
- HyperJ Homepage. http://alphaworks.ibm.com/tech/hypetj
- JBossAOP homepage. http://www.jboss.com.products/aop
- AspectWerkz http://aspectwerkz.codehaus.org
- Spring www.springframework.org
- Dynaop http://dynaop.dev.Java.net

Eclipse y AspectJ

- http://eclipse.org/aspectj

Trabajo colaborativo soportado por computadora, CSCW

- http://ksi.cpsc.ucalgary.ca/courses/547-95/pfeifer/cscw_domain.html
- http://groupware.openoffice.org/ home de Open Office.

APÉNDICES

Apéndice A

A Programación orientada a aspectos, una visión general

A.1 Introducción

Dentro de la constante evolución de la ingeniería de software ha surgido una nueva forma de descomponer los sistemas: orientación a aspectos. A continuación se da una visión general de este tipo de programación.

A.1.1 Historia

En relación a la evolución de la ingeniería de software, los progresos o cambios más significativos se han dado gracias a un principio fundamental para resolver problemas, el de dividir el problema en sub-problemas.

Con lo cual, se intenta evitar tener en el código mezclados los conceptos, los datos y las funcionalidades. Con el concepto de descomposición funcional se identifican las partes que se pueden separar del dominio del problema; esto, a la larga lo que favorece es el poder integrar nuevas funciones, pero un inconveniente seria que esas funciones, en ocasiones, quedan poco claras por los datos compartidos que utilizan y los datos quedan esparcidos por todo el código.

Después de la descomposición funcional, se dio la programación orientada a objetos (OOP) (que ha permitido desarrollar sistemas muy complejos) este modelo de objetos se ajusta mejor a los problemas del dominio real que la descomposición funcional.

La OOP facilita la integración de nuevos datos aunque quedan las funciones esparcidas por todo el código, lo que da como desventaja, que en ocasiones para integrar una nueva funcionalidad hay que modificar varios objetos.

A.1.2 Diseño, implementación y otros aspectos

Para el desarrollo de software se consideran no sólo el diseño y la implementación de la funcionalidad, se consideran también aspectos tales como la sincronización, la distribución, el manejo de errores, la optimización del manejo de memoria, seguridad, entre otros.

Ni la programación orientada a objetos ni la descomposición funcional son suficientes para atacar de manera eficiente este tipos de "problemas" (estos aspectos); es decir estas técnicas no soportan la separación de competencias para aspectos distintos a la

funcionalidad básica del sistema, lo que impacta de manera negativa en la calidad del software.

Una metodología que intenta soportar la separación de competencia de los diferentes aspectos es la programación orientada a aspectos (AOP); intenta separar los componentes y los aspectos, teniendo mecanismos para abstraerlos y componerlos para formar todo el sistema.

La programación orientada a aspectos lo que persigue es implementar una aplicación de forma eficiente y fácil de entender. Sigue el paradigma de la orientación a objetos, soportando por tanto la descomposición orientada a objetos, así como la procedural y la descomposición funcional; es decir, como puede utilizarse con los diferentes estilos de programación, no se considera una extensión de la orientación a objetos.

A.2 Programación orientada a aspectos

El concepto fue introducido por Gregor Kiczales y su grupo, el equipo de Demeter [19] estaba centrado en la programación adaptativa (alrededor de 1991) una instancia temprana de la programación orientada a objetos). En 1995 se publicó la primera definición del concepto de aspecto, por el grupo Demeter: Un aspecto es una unidad que se define en términos de información parcial de otras unidades.

- Los principales objetivos de la programación orientada a objetos son el de separar los conceptos y minimizar las dependencias entre ellos.
- Con separar los conceptos se consigue que cada cosa esté en su sitio (que cada decisión se tome en un lugar concreto) y con minimizar las dependencias, se consigue tener una pérdida del acoplamiento entre los distintos elementos.

  Al perseguir estos objetivos se tienen varias ventajas, como son:

  - Código menos enmarañado, más natural y más reducido.
  - Mayor facilidad para razonar sobre los conceptos, ya que están separados y con dependencia mínima.
  - Más facilidad para depurar y hacer modificaciones en el código.
  - Que un conjunto de modificaciones en la definición de un concepto tenga un impacto mínimo en las otras.
  - Código reutilizable y se puede acoplar y desacoplar cuando sea necesario.

A.2.1 Un aspecto

La definición de aspecto ha ido evolucionando, desde la primera definición mencionada en los párrafos anteriores, hasta la que se maneja actualmente: Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa es una unidad modular que aparece en otras unidades modulares del programa (G. Kiczales).

Una definición más informal: los aspectos son la unidad básica de la AOP, y son las partes de una aplicación que describen las cuestiones claves relacionadas a la semántica esencial o el rendimiento, pueden verse como los elementos que se diseminan por todo el código y que son difíciles de describir localmente con respecto a otros componentes.

A.2.2 Componente y aspecto

Se puede diferenciar entre un componente y un aspecto viendo el primero como aquella propiedad que se puede encapsular claramente en un procedimiento mientras que un aspecto no se puede encapsular.

Los aspectos no suelen ser unidades de descomposición funcional del sistema, sino propiedades que afectan al rendimiento o la semántica de los componentes. Algunos ejemplos de aspectos son: los patrones de acceso a memoria, la sincronización de procesos concurrentes, el manejo de errores, etc.

La idea es, tener un programa formado por un conjunto de aspectos, más un modelo de objetos. Con el modelo de objetos se tiene la funcionalidad básica, mientras que con los aspectos se tienen las características de rendimiento y otras no relacionadas con la funcionalidad básica del programa. Y en general este tipo de programas es más compacto y modular, teniendo cada aspecto su propia competencia.

A.2.3 Fundamentos

Englobando los conceptos mencionados, tenemos que con las clases se implementa la funcionalidad básica o principal de la aplicación; mientras que con los aspectos se tienen los conceptos técnicos, tales como, la persistencia, el manejo de errores, la sincronización de procesos, la comunicación de procesos, etc.

Los aspectos se escriben mediante lenguajes de descripción de aspectos, por lo que los lenguajes orientados a aspectos definen una nueva unidad de programación para encapsular las funcionalidades que cruzan todo el código aspectos. Para que los aspectos y componentes de las aplicaciones se puedan entremezclar se deben tener puntos en común, los cuales se conocen como puntos de enlace, los cuales son una clase especial de interfaz; el encargado de realizar el proceso de mezclarlos se conoce como tejedor ("weaver") que mezcla los diferentes mecanismos de abstracción y composición.

A.2.4 Definiciones para la programación orientada a aspectos

Se necesita contar con:

- Lenguaje para definir funcionalidad básica (lenguaje base), el cual puede ser un lenguaje de propósito general.
- Lenguaje de aspectos, define la forma de los aspectos.
- Tejedor, combina los lenguajes, el proceso de mezcla puede realizarse en tiempo de compilación o en tiempo de ejecución.

A.3 Tejedor

Los aspectos nos describen apéndices al comportamiento de los objetos. Hacen referencia a las clases de los objetos y definen en que puntos se han de colocar estos apéndices (puntos de enlace, que pueden ser métodos o asignaciones de variables). Las clases y aspectos pueden mezclarse de dos formas: estática o dinámicamente.

Enlazado estático. Implica modificar el código fuente de una clase insertando sentencias en esos puntos de enlace. La principal ventaja, es que se evita que el nivel de abstracción derive en un impacto negativo en el rendimiento. Como desventaja, es difícil identificar los aspectos en el código, lo cual para remplazar aspectos de forma dinámica en tiempo de ejecución da un problema de eficiencia. Un ejemplo de esté tipo de tejedor es AspectJ.

Enlazado dinámico. Una precondición para que se pueda realizar un enlazado dinámico es que los aspectos existan de forma explicita (tanto en compilación como en ejecución), para lo cual los aspectos y las estructuras enlazadas se modelan como objetos. Y el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica.

A.4 Lenguajes de aspectos de dominio específico vs. de propósito general

De dominio especifico. Soportan uno o más de los sistemas de aspectos que se han mencionado (distribución, manejo de errores, sincronización, etc.) NO pueden soportar otros aspectos distintos. Tienen un nivel de abstracción mayor al lenguaje base. Normalmente imponen restricciones en la utilización del lenguaje base (para garantizar que los conceptos del dominio de los aspectos se programen con el lenguaje de aspectos, para no producir un conflicto).

De propósito general. Se diseñan para ser utilizados con cualquier clase de aspecto, no pueden imponer restricciones al lenguaje base. Tienen un severo inconveniente, permiten la separación de código pero no garantizan la separación de funcionalidades.

Desde el punto de vista empresarial, siempre le es más fácil a los programadores el aprender un lenguaje de propósito general, que el tener que estudiar varios lenguajes distintos de propósito específico.

A.5 AspectJ, lenguaje de propósito general

AspectJ es una extensión de Java orientada a aspectos y de propósito general [18], en el apéndice B se da una descripción detallada del lenguaje. Los aspectos cortan las clases, las interfaces y otros aspectos. Los aspectos mejoran la separación de competencias haciendo posible localizar de forma limpia los conceptos de diseño del corte. Un aspecto es una clase con la particularidad que puede contener unos constructores de corte.

Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos.

Los aspectos son constructores que trabajan al cortar de forma transversal la modularidad de las clases de forma limpia y cuidadosamente diseñada. Por lo tanto, un aspecto puede afectar a la implementación de un número de métodos en un número de clases, lo que permite capturar la estructura de corte modular de este tipo de conceptos de forma limpia.

Un aspecto en AspectJ contiene:

- Cortes. ("pointcut") capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos. Se utilizan para definir el código de los aspectos utilizando avisos.
- Introducciones. ("introduction") para introducir elementos completamente nuevos en las clases dadas. Entre estos elementos podemos añadir:
  Un método
  Un constructor
  Un atributo
  Varios de los elementos anteriores a la vez
  Varios de los elementos anteriores en varias clases
- Avisos ("advices") definen partes de la implementación del aspecto que se ejecutan en puntos bien definidos. El cuerpo de un aviso, puede añadir distintos puntos de código:
  "Before". Se ejecuta justo antes de las acciones asociadas
  "After". Se ejecuta justo después de las acciones asociadas.
  "Catch". Se ejecuta cuando durante la ejecución de las acciones asociadas se lanza una excepción.
  "Finally". Se ejecuta justo después de las acciones asociadas; incluso aunque haya producido una excepción.
  "Around". Atrapa la ejecución de los métodos designados por el evento.

A.6 Aspectos, en el diseño (uso de UML)

Cuando las aplicaciones necesitan un alto grado de adaptabilidad o que se requiere que se puedan reutilizar, se debe considerar desde el diseño la orientación a aspectos. Los trabajos sugeridos proponen utilizar UML como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar el diseño funcional de los objetos separando del diseño no funcional del mismo, o lo que es lo mismo, representar la funcionalidad básica separada de los otros aspectos.

Las ventajas que tiene el capturar los aspectos ya desde la fase de diseño son:

- Facilita la creación de documentación y el aprendizaje. El tener los aspectos como constructores de diseño permite que los desarrolladores los reconozcan en los primeros estadios del proceso de desarrollo, teniendo así una visón de más alto nivel y ayudando así a que los diseñadores de aspectos y los principiantes puedan aprender y documentar los modelos de aspectos de un modo más intuitivo, pudiendo incluso utilizar herramientas CASE para tener representado el modelo de forma visual.

- Facilita la reutilización de los aspectos. La facilidad de documentación y aprendizaje influye en la reutilización de la información de los aspectos. Al saber como se diseña y como afecta a otras clases, es más fácil ver como se pueden utilizar de otra forma, lo que incrementaría la reutilización de los aspectos.

A.7 Conclusiones

La separación de los aspectos, desde el diseño y no solo en la implementación y ejecución), es un paso importante que se esta dando en la ingeniería de software, pero que aún se esta refinando, sobre todo en cuestión de eficiencia.

Los lenguajes de aspectos de dominio específico juegan un papel crucial en la programación orientada a aspectos y que son preferibles a los lenguajes de propósito general, porque soportan mejor la separación de funcionalidades. Pero desde el punto de vista de la empresa, es más cómodo y económico que se utilizan lenguajes de propósito general por lo que se refiere a la capacitación.

El campo de la orientación a aspectos es un campo de investigación "joven" en el que se abre un horizonte amplio. Desde la definición de los distintos aspectos mediante lenguajes de propósito específico, hasta middleware orientados a aspectos, pasando por las etapas de análisis y diseño orientadas a aspectos, aun quedan muchos campos que aborda, por lo que se cree interesante abrir nuevas líneas de investigación en este tema.

Apéndice B

B AspectJ

La Programación Orientada a Aspectos (AOP) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la AOP se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reutilizables.

AspectJ es una extensión al lenguaje Java orientada a aspectos y de propósito general, que extiende el popular lenguaje con una nueva clase de módulos llamados aspectos. Los aspectos cortan las clases, las interfaces y a otros aspectos mejorando la separación de competencias y haciendo posible localizar de forma limpia los conceptos de diseño del corte.

B.1 Introducción al lenguaje

En AspectJ, un aspecto es una clase, exactamente igual que las clases Java, pero con una particularidad, que pueden contener unos constructores de corte, que no existen en Java. Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, de constructores, lanzamiento de excepciones, acceso a atributos. Los cortes no definen acciones, sino que describen eventos. De modo que en AspectJ los aspectos son constructores que trabajan cortando de forma transversal la modularidad de las clases de forma clara y específica.

Cualquier programa Java que sea válido, es también un programa valido en AspectJ. El compilador de AspectJ genera archivos class conformes a la especificación Java byte-code, por lo que cualquier máquina virtual Java (JVM) debe poder ejecutarlos. Al usar Java se obtienen todos los beneficios de este lenguaje, los cuales han quedado demostrados por su extensa aceptación entre la comunidad de programadores. Ello acelera además el proceso de aprendizaje en el lenguaje, puesto que sólo se ha de aprender a utilizar la parte nueva introducida por AspectJ.

Debemos diferenciar en AspectJ dos partes: la especificación del lenguaje y su implementación. La primera, define el lenguaje en el que se escribe el código; en AspectJ la funcionalidad de los conceptos encapsulados se escribe en Java estándar y las extensiones del lenguaje se usan para definir dónde y cómo han de entrelazarse con el código de nuestra aplicación. La segunda parte es la implementación del lenguaje, provista de herramientas para compilar, depurar e integrar AspectJ con conocidos entornos de desarrollo integrados (*"integrated development environment"*, IDE) como JBuilder, Eclipse, Emacs.

Existen una serie de conceptos que son necesarios entender y conocer antes de poder utilizar AspectJ y que no tienen por qué entenderse de igual manera en los diferentes lenguajes que implementan los conceptos de la programación orientada a aspectos. Se hará referencia siempre al término inglés utilizado para designar estos conceptos en la documentación del lenguaje para facilitar así la lectura de otros documentos. No obstante al principio se da una explicación del significado de éstos.

B.2 Crosscuting en AspectJ

En AspectJ, la implementación de las reglas de entrelazado se denomina *crosscutting,* y a los conceptos encapsulados que originalmente estaban repartidos por todo o parte de la aplicación, se les denomina *crosscutting concerns*. El proceso de combinación de este código con el de la aplicación Java se conoce como *"weaving process"*, es decir, proceso de entrelazado. Las reglas de entrelazado, definen de una manera sistemática, las diferentes partes del código que deben ser afectadas por los conceptos encapsulados en los aspectos, respondiendo a la cuestión qué hacer cuando cierto punto de la ejecución de un programa se alcanza. A dichos puntos de ejecución se les llama *"pointcuts"*. En AspectJ se definen dos tipos de *crosscutting*: estático y dinámico.

B.2.1 Crosscuting dinámico

Se dice que un aspecto afecta a la aplicación de manera dinámica, o usando el término inglés, se habla de *dynamic crosscutting*, cuando dicho aspecto afecta al comportamiento global o una parte de la aplicación.

Crosscutting dinámico, es la forma más frecuente en AspectJ de entrelazado y es la manera de dotar de nuevo comportamiento a la ejecución de un programa. De esta forma se puede enriquecer o según sea el caso, reemplazar el flujo de ejecución de un programa, de forma que éste vaya por diferentes requisitos que en nuestro código se tengan encapsulados en diferentes módulos. Por ejemplo, algo muy frecuente, es indicar que cierta acción ha de ser realizada antes de la ejecución de ciertos métodos, o de la captura de determinadas excepciones dentro de un conjunto de clases, definiendo en un módulo aparte los puntos dónde el código se ha de entremezclar (*"weaving points"*) y la acción a realizar en el momento en el que se alcancen dichos puntos.

B.2.2 Crosscuting estático

Se dice que un aspecto afecta a la aplicación de manera estática, o usando el término inglés, se habla de *static crosscutting*, cuando se añaden modificaciones en la estructura estática del programa, esto es, en las clases, interfaces y aspectos. Por tanto, este tipo de técnica no implica modificación del comportamiento del sistema.

En la mayor parte de los casos, su uso se requiere para dar soporte a la implementación de los conceptos de *crosscutting* dinámico. Por ejemplo, puede simplificarse la implementación de éstos, modificando las relaciones de herencia entre clases e interfaces, para poder trabajar así con las clases abstractas en lugar de tener que hacerlo con las implementaciones específicas; o puede ser de utilidad, modificar los

atributos o métodos de una clase o interfaz para poder así definir estados específicos y comportamientos que pueden ser usados en las acciones que se lleven a cabo con entrelazado dinámico. También es posible con entrelazado estático, definir advertencias (*warnings*) y errores (*errors*) en tiempo de compilación de una manera sencilla y que afecte a todo o una parte de la aplicación, permitiendo definir los puntos de corte de una manera general con predicados lógicos.

B.3 Las reglas de entrelazado

AspectJ es una extensión al lenguaje de programación Java, para especificar las reglas de entrelazado para los aspectos que afectan a una aplicación, ya sea de forma estática o dinámica. Las extensiones están diseñadas de forma que el proceso de aprendizaje de un programador Java para usarlas sea mínimo. A continuación, se explican los constructores que utiliza AspectJ para especificar las reglas de entrelazado de los aspectos con la aplicación de manera precisa (al igual que en casos anteriores, especifico en primer lugar el término inglés):

- **"Join points"**, puntos de enlace. Un punto de enlace es un sitio identificable en la ejecución de un programa. Podría ser una llamada a un método o la asignación de un nuevo valor a un atributo de una clase. En AspectJ todo gira alrededor de los puntos de enlace, ya que son los lugares donde los valores añadidos de los aspectos son enlazados con el código.
- **"Pointcuts"**, puntos de corte. Estos son constructores donde se especifican los puntos de enlace y se captura información referente al contexto. Por ejemplo, un punto de corte puede referirse a un punto de enlace que sea una llamada a un método, pudiendo, en tal caso, capturar el contexto del método, siendo éste, el objeto sobre el que se invoca el método, así como los argumentos de la llamada. Para entender la diferencia entre puntos de enlace y puntos de corte, se puede pensar en estos últimos como las reglas de especificación y en los primeros como las situaciones que satisfacen dichas reglas.
- ***"Advice"***, avisos. Este es el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o alrededor. Alrededor quiere decir que se puede, con el aviso, modificar la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si es eso lo que se desea. Por ejemplo, se podría especificar un aviso para registrar en un histórico cierto mensaje, siempre antes de la ejecución de ciertas funciones que se encuentran dispersas por diferentes módulos. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte junto con los avisos, son las herramientas para implementar entrelazado dinámico. Los puntos de corte identifican dónde y los avisos lo completan indicando qué hacer.
- ***"Introduction"***, introducciones. Si lo anterior era para el entrelazado dinámico, las introducciones se usan para el estático. Con ella es posible introducir cambios a las clases, interfaces y aspectos del sistema para así permitir el entrelazado dinámico. Conlleva cambios estáticos en los módulos que no afectan directamente a su comportamiento. Por ejemplo, se pueden añadir métodos o atributos a clases y después usarlos tal como si hubieran sido definidos por la propia clase.

- ***"Compile-Time declaration"***, declaraciones en tiempo de ejecución. Esta es otra forma de implementar técnicas de entrelazado estático. Permite añadir advertencias y errores para en tiempo de compilación detectar ciertos patrones de uso que queramos advertir o prohibir (en el caso de que los identifiquemos como errores). Por ejemplo, se puede declarar que es un error hacer uso de las llamadas del paquete Java.sql fuera del paquete dataAccess de la aplicación.
- ***"Aspect"***, aspectos. Estos son la unidad central de AspectJ, al igual que lo son las clases en Java. Contienen el código que expresa las reglas de entrelazado tanto para los aspectos dinámicos como los estáticos. Puntos de corte, avisos, introducciones y declaraciones de compilación se combinan en los aspectos. Además de estos elementos propios de AspectJ, los aspectos pueden contener atributos, métodos y clases anidadas, tal como los tienen las clases normales en Java.

Veamos cómo funciona todo ello junto. Cuando se diseña un comportamiento que se encuentra disperso por la aplicación, lo primero que hay que hacer es identificar los puntos de enlace donde se quiere enriquecer el comportamiento o bien modificarlo. Una vez hecho, se diseña cual será el nuevo comportamiento. Para hacer esto, primero se escribe un aspecto que sirva como módulo para el comportamiento global. En él se escriben los puntos de corte para capturar los puntos de enlace que se deseen. Finalmente, se crean avisos para esos puntos de corte y se define dentro del cuerpo de los avisos las acciones que se deben realizar cuando se ejecute el código que definen los puntos de enlace. Es posible que para permitir ciertos avisos sea necesario hacer uso de entrelazado estático.

B.4 Detalles del lenguaje

En esta sección se presentan los aspectos más relevantes del lenguaje AspectJ, tratando en primer lugar los puntos de corte, viendo su sintaxis y modo de uso; los avisos, sintaxis y tipos; el manejo de la información contextual; las reglas de entrelazado estático disponibles; extensión e instancias de los aspectos.

B.4.1 Puntos de corte

Los puntos de corte (*"pointcuts"*) capturan o identifican puntos de enlace en el flujo del programa. Una vez definidos con los constructores que proporciona el lenguaje, puede especificarse mediante reglas de entrelazado cómo combinar los comportamientos definidos en los aspectos con el código original. Además los puntos de corte dan acceso al contexto del programa y las acciones se pueden beneficiar de la información contextual para llevar a cabo su cometido.

En AspectJ las *"pointcuts"* se pueden declarar anónimos o con un nombre. Al igual que las clases, un punto de corte anónimo se define en el lugar donde se usa, que bien podría ser en un *"advice"* o en la definición de otro *"pointcut"*. Los puntos de corte que se definen con un nombre pueden después ser referenciados desde múltiples partes del código, haciéndoles reutilizables. Su sintaxis es la siguiente:

pointcut nombre (argumentos): definición

Por definición se entiende el conjunto de reglas que definen los puntos de enlace donde se quiere insertar cierto comportamiento. El nombre que se utilice para definir el *"pointcut"* es el que luego ha de utilizarse en los *"advices"* para hacer referencia a ellos. Véase el siguiente ejemplo:

```
pointcut operacionesPublicas(): call(public *.*(..));
before() : operacionesPublicas(){
        //acciones a realizar antes de las llamadas a las funciones públicas
}
```

Es posible también definir puntos de corte anónimos, es decir, sin un nombre que los referencíe. Se definen en el lugar donde se usan y no pueden reutilizarse. No se aconseja, por tanto, usarlos si la expresión que los define es complicada o interesa utilizarlos en varios *"advices"*. Su sintaxis dentro de la definición de los *"advices"* es la siguiente:

```
[especificación-del-advice]: definición-del-pointcut
```

Si quisiéramos escribir el mismo ejemplo anterior pero con el *"pointcut"* anónimo, se haría como lo muestra el siguiente ejemplo –obsérvese que lo importante, la definición del punto de enlace, es igual al anterior:

```
before(): call(public *.*(..)) {
//acciones a realizar antes de las llamadas a las funciones públicas
}
```

B.4.1.1 Sintaxis y semántica de los puntos de enlace

Los puntos de corte harán referencia a una serie de puntos de interés (*"joinpoints"*) en el programa, por lo que se necesita un lenguaje eficiente para definirlos. AspectJ utiliza una sintaxis basada en elementos comodines que sirven para capturar puntos de enlace que comparten características comunes. Dichos elementos son los siguientes:

* un asterisco denota cualquier número de caracteres excepto el punto.
.. dos puntos seguidos denota cualquier número de caracteres incluyendo el punto.
+ el símbolo de la suma denota los descendientes de una clase.

Más adelante se verá que el significado de estos caracteres dependerá de los elementos a los que acompañen. Al igual que en Java y en muchos lenguajes las expresiones se pueden combinar para dar lugar a expresiones más complejas. Para ello se utilizan los siguientes operadores unarios y binarios:

! la exclamación de cierre es un operador unario que denota todos los puntos de enlace, excepto aquellos que se especifiquen en la expresión que siga. Por ejemplo, !within(aspects.*) denota todos los puntos de enlace excepto aquellos del paquete aspects.

|| dos barras verticales es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumpla alguno de los lados de la expresión.

&& es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumplan ambos lados de la expresión.

Además pueden usarse los paréntesis combinados con estos operadores con objeto de alterar la precedencia por defecto de las expresiones y facilitar también la legibilidad. Los elementos expuestos tendrán diferente significado según acompañen a paquetes, tipos (clases, interfaces o aspectos), métodos o atributos.

Veamos algunos ejemplos del significado que adquieren según cómo y donde se usen:

Patrón	Elementos referidos
Manager	Tipos de nombre Manager.
*Manager	Tipos con un nombre que termine en Manager.
Java.*.Name	Tipo Name en cualquiera de los subpaquetes del paquete Java.
Java..*	Cualquier tipo del paquete Java o todos sus subpaquetes
Javax..*Set+	Todos los tipos en el paquete Javax o sus subpaquetes cuyo nombre termine en Set y sus subtipos.
public void Stack.clean()	El método clean() de la clase Stack que tenga acceso público, devuelva void y no tenga argumentos.
public void Stack.push(Integer) throws RangeExceded	El método push() de la clase Stack que tenga acceso público, devuelva void, tenga un único argumento de tipo Integer y lance la excepción RangeExceded.
public void Stack.push*(*)	Todos los métodos públicos de la clase Stack cuyo nombre empiece con push y tenga un único argumento de cualquier tipo.
public void Stack.*()	Todos los métodos en de la clase Stack que tengan acceso público devuelvan void y no tengan argumentos.
public * Stack.*()	Todos los métodos públicos de la clase Stack que no tengan argumentos y devuelvan cualquier tipo.
public * Stack.*(..)	Todos los métodos públicos de la clase Stack que tomen cualquier número de argumentos.
* Stack.*(..)	Todos los métodos de la clase Stack.
!public * Stack.*(..)	Todos los métodos que no tengan acceso público de la clase Stack.
public static void MyClass.main (String[] args)	El método estático main() de la clase MyClass con acceso público.
* Stack+.*(..)	Todos los métodos de la clase Stack o sus subclases.
* Java.io.Reader.read(char[],..)	Métodos read() de la clase Reader con cualquier número de argumentos, con el primero del tipo char[].
* Javax..*.add*List(List+)	Métodos que empiecen con add y terminen en List en el paquete Javax o sus subpaquetes y que tomen un argumento del tipo List o sus subtipos.
* *.*(..) throws SQLException	Cualquier método que lance SQLException.
public MyClass.new()	Constructores públicos de la clase MyClass sin argumentos.
public MyClass.new(Integer)	Constructores públicos de la clase MyClass que tomen un único argumento de tipo Integer.
public MyClass.new(..)	Todos los constructores de la clase MyClass que tomen cualquier número de argumentos.
public *MyClass.new(..)	Constructores públicos de clases cuyo nombre termine en MyClass.

public MyClass+.new(..)	Constructores públicos de la clase MyClass o sus subclases.
public MyClass(..) throws InvalidFormatException	Constructores públicos de la clase MyClass que lancen InvalidFormatException.
private float MyClass.value	Atributo privado value de la clase MyClass.
* MyClass.*	Todos los atributos de la clase MyClass.
!public static * MyClass..*.*	Todos los atributos no públicos estáticos de la clase MyClass y de sus subpaquetes.
Public !final *.*	Atributos no finales públicos de cualquier clase.
!Vector	Todos los tipos que no sean Vector
Vector || Hashtable	Tipos Vector o Hashtable
Javax..*Model || Javax.swing.test.Document	Todos los tipos en el paquete Javax o sus subpaquetes directos o indirectos que su nombre termine con Model o Javax.swing.test.Document
Java.util.RandomAccess+ && Java.util.List+	Todos los tipos que implementen las dos interfaces especificadas.
* *.*(..) thrwos RemoteException	Todos los métodos que declaren que pueden lanzar RemoteException
public Account.new()	Un constructor publico de la clase Account que no tiene argumentos
public Account.new(int)	Un constructor publico de la clase Account que tiene un argumento int
public Account.new(..)	Todos los constructores públicos de la clase Account, sin importar sus argumentos
public Account+.new(..)	Cualquier constructor publico de la clase Account o sus subclases
public *Account.new(..)	Cualquier constructor publico de una clase cuyo nombre termine con Account
public Account.new(..) throws InvalidAccountException	Cualquier constructor publico de la clase Account que pueda lanzar la excepción InvalidAccountException
private float Account._balance	El campo privado _balance de la clase Account
* Account.*	Todos los campos de la clase Account, sin importar el tipo de acceso, su tipo o su nombre
!public static * banking..*.*	Todos los no públicos y estáticos campos de banking y sus paquetes directos o indirectos
public !final *.*	Campos públicos no finales de cualquier clase

Tabla B.1 Patrones de puntos de enlace

Como se ve en la tabla B.1, el significado de los caracteres comodines depende si se aplican a tipos, métodos o atributos. Referido a los tipos, el asterisco especifica una parte de la clase, interfaz o paquete; los dos puntos seguidos se usan para denotar todos los subpaquetes; y el símbolo de suma denota los subtipos (subclases o subinterfaces). Mientras que referido a los métodos, los dos puntos seguidos denotan cualquier tipo y número de argumentos que tome el método.

B.4.1.2 Implementando los puntos de corte

Una vez que tenemos una manera eficiente de definir los puntos de enlace en el código, se necesita una manera de utilizarlos. Para referirnos a ellos podemos hacerlo según el tipo de punto de enlace al que representan, como pueden ser llamadas a métodos, ejecución de estos, lectura de atributos. Véase la siguiente tabla, B.2, para una descripción de las diferentes posibilidades que ofrece AspectJ.

Categoría	Sintaxis
Ejecución de métodos	execution(MethodSignature)
Llamada a métodos	call(MethodSignature)
Ejecución de constructores	execution(ConstructorSignature)

Llamada a constructores	call(ConstructorSignature)
Inicialización de clases	staticinitialization(TypeSignature)
Lectura de atributos	get(FieldSignature)
Escritura de atributos	set(FieldSignature)
Captura de excepciones	handler(TypeSignature)
Inicialización de objetos	initialization(ConstructorSignature)
Pre-inicialización de objetos	preinitialization(ConstructorSignature)
Ejecución de avisos	adviceexecution()

Tabla B.2 Tipos de puntos de corte

Para poder ser más concretos en la definición de los puntos de corte nos podemos servir de un buen número de *pointcuts* disponibles en AspectJ y que se muestran en las siguientes tablas (B.3 y B.4).

Pointcut	Descripción
cflow(call(* MyClass.operation(..))	Puntos de enlace durante la ejecución del método operation() de la clase MyClass, incluyendo la llamada al método operation() de MyClass.
cflowbelow(call(* MyClass.operation(..))	Puntos de enlace durante la ejecución del método operation() de la clase MyClass, excluyendo la llamada al método operation() de MyClass.
cflow(Operations())	Todos los puntos de enlace en el contexto de los puntos de enlace capturados por el punto de corte Operations().
cflow(staticinitializer(StClass))	Puntos de enlace durante la inicialización de la clase StClass.
cflowbelow(execution(MyClass.new(..))	Puntos de enlace durante la ejecución de alguno de los constructores de la clase MyClass, excluyendo la ejecución misma del constructor.

Tabla B.3 Pointcuts basados en control-flow

Pointcut	Descripción
within(MyClass)	Puntos de enlace dentro de la clase MyClass.
within(MyClass+)	Puntos de enlace dentro de la clase MyClass y sus subclases.
withincode(* MyClass.operation(..))	Puntos de enlace dentro del método operation() de la clase MuClass.
withincode(* *printer.flow(..))	Puntos de enlace dentro del método flow() en clases cuyo nombre acabe en printer.

Tabla B.4 Pointcuts basados en localización

Un uso común de within() es para excluir los puntos de enlace del aspecto donde se define. Por ejemplo, el siguiente punto de corte excluye los puntos de enlace correspondientes a las llamadas a los métodos print de la clase Java.io.Stream que ocurran dentro del aspecto TraceAspect:

```
call(* Java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

Los siguientes puntos de corte, "this" y "target", están basados en los tipos de los objetos en tiempo de ejecución (tabla B.5). Con "this" se referencia al objeto actual, mientras que con "target" es al objeto sobre el que se llama al método. Estos puntos de corte sirven además para recoger el contexto en el punto de enlace; "this" toma la forma this(Type) y así captura los puntos de enlace asociados con objetos del tipo especificado Type. Al igual, "target" es también de la forma target(Type).

Pointcut	Descripción
this(MyClass)	Puntos de enlace donde this es una instancia de MyClass o sus subclases (this es el objeto que se esté ejecutando actualmente).
target(MyClass)	Puntos de enlace en los que el objeto donde se llama el método es instancia de MyClass o sus subclases.

Tabla B.5 Pointcuts de objetos

Nótese como "this()" y "target()" toman como argumentos Type y no TypePattern. Eso quiere decir que no podremos usar los comodines asterisco o dos puntos seguidos (el comodín referente a los subtipos, +, no se necesita, pues éstos son capturados por las reglas de herencia de Java). Obsérvese también que, al no ejecutarse los métodos estáticos sobre un objeto, éstos no tendrán un objeto "this" asociado, por lo que el punto de corte no incluirá este tipo de métodos (igualmente ocurre con "target"). Esta última es la diferencia de uso que puede hacerse con within.

El siguiente *"pointcut"* que se basa en el tipo de los argumentos del punto de enlace. Por argumentos ha de entenderse lo siguiente: en los métodos y constructores los argumentos son los mismos que los argumentos de éstos; para los puntos de enlace que capturan excepciones, la excepción capturada es considerada el argumento; y para los accesos a atributos en escritura, se considera como argumento el nuevo valor a escribir sobre el atributo. Este tipo de punto de corte sirve también para capturar la información contextual en el punto de enlace, ver tablas B.6 y B.7.

Pointcut	Descripción
args(String,..,int)	Puntos de enlace en los que el método tenga como primer argumento un String y como último un int.
args(SQLException)	Puntos de enlace con un único argumento de tipo SQLException. Capturaría un método que tome un único argumento de tipo SQLException, la escritura sobre un atributo con un valor de tipo SQLException, o la captura de una excepción del tipo SQLException.

Tabla B.6 Pointcuts de argumentos

Pointcut	Descripción
if(Buffer.size() > MaxAllowed)	Puntos de enlace que ocurran a partir de que el tamaño del buffer supere el valor de MaxAllowed.
if(thisJoinPoint.getTarget()!=null)	Puntos de enlace donde el objeto donde se estén ejecutando los métodos capturados no sea nulo.

Tabla B.7 Pointcuts condicionales

Hemos examinado todas las posibilidades para definir puntos de corte en el lenguaje AspectJ. En la siguiente sección se explica el concepto de *"advice"*, que hará uso de los puntos de corte que hemos definido, para dotar de cierto comportamiento al programa en los puntos de enlace que se especifiquen.

B.4.2 Avisos

Los puntos de corte son, junto con los avisos (*"advices"*), las herramientas para implementar conceptos de entrelazado dinámico en AspectJ. Los avisos son constructores parecidos a los métodos, donde se indica el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o "alrededor" del punto de enlace. Alrededor quiere decir que puede modificarse con el aviso la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si es eso lo que se desea. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte identifican *dónde* y los avisos lo completan indicando *qué* hacer.

B.4.2.1 Sintaxis de los avisos

Los avisos pueden descomponerse en tres partes: su declaración, la definición del punto de corte (o el nombre del punto de corte que vaya a utilizar) y el cuerpo. Por claridad, los siguientes ejemplos harán uso del mismo punto de corte y que será el que a continuación se define:

```
pointcut connectionOperation(Connection connection):
call(* Connection.*(..) throws SQLException) && target(connection);
```

El *"pointcut"* connectionOperation consta de dos puntos de corte anónimos: el primero captura todas las llamadas a métodos de la clase Connection que lancen la excepción SQLException independientemente de los argumentos que tome o el valor que devuelva; y el segundo, target(connection), captura el objeto sobre el que se hacen las llamadas. Al no ser anónimo y tener un nombre, podemos utilizarlo en la definición de los avisos cuantas veces queramos.

B.4.2.2 Tipos de avisos

Como se ha comentado, existen tres posibilidades para definir el momento de actuación de los avisos ("before", "after" y "around"). Cada una de estas se explica a continuación.

Los avisos **"before"** se ejecutan antes de la ejecución del punto de enlace capturado. Al terminar, dan paso al método original. En el caso de que se lanzara una excepción en el cuerpo del aviso, el método original no se ejecutará. Este tipo de aviso se usa típicamente para llevar a cabo tareas de control o precondiciones, como el control de los parámetros, logging, autentificación. Véase el siguiente ejemplo:

```
before(Connection connection): connectionOperation (connection){
        checkConnectionValidity(connection);
```

```
}
```

En este caso, el cuerpo del aviso podría comprobar mediante el método checkConnectioValidity y antes de llamar a las operaciones que captura connectionOperation, que el objeto connection que se va a utilizar no está corrupto.

Los avisos **"after"** se ejecutan después de la ejecución del punto de enlace capturado. En este caso hay que distinguir además el caso en el que un método termine con el lanzamiento de una excepción. Para ello, AspectJ ofrece tres variantes para el aviso "after": después de terminar normalmente, después de terminar lanzando una excepción y después de terminar, sea como sea (es decir, incluyendo los dos casos anteriores). La sintaxis para
cada uno de esos casos es la siguiente:

```
after () : connectionOperation (connection) { ... }
```

Este aviso se ejecuta después de las llamadas capturadas por el punto de corte connectionOperation, ya terminen éstas normalmente o con el lanzamiento de una excepción.

```
after () returning (Object x) : connectionOperation(connection) { ... }
```

Este aviso se ejecutará después de las llamadas a los métodos capturados por el punto de corte connectionOperation, en el caso de que la ejecución de éstas se complete sin lanzar ninguna excepción. El valor devuelto recibe el nombre x, por lo que se podrá referenciar y usar en el cuerpo del aviso. Si el método termina con una excepción, el cuerpo del aviso no se ejecutará.

```
after () throwing (SQLException exc) : connectionOperation(connection){ ... }
```

Este aviso se ejecuta después de las llamadas a los métodos capturados por el punto de corte connectionOperation, en el caso de que éstas terminen de forma anómala devolviendo la excepción SQLException, según se ha indicado. A la excepción se le da el nombre exc en el cuerpo del aviso, por lo que así podrá utilizarse. Si el método termina normalmente, el cuerpo del aviso no se ejecutará.

Los avisos **"around"** envuelven el punto de enlace capturado, de tal forma que se puede anular la ejecución del punto de enlace o alterar el contexto antes de proceder. Permite también ejecutar varias veces la lógica capturada en el punto de enlace, cada vez, por ejemplo, con diferentes argumentos. Es sin duda la más flexible de las posibilidades que ofrece AspectJ. Para permitir la ejecución por el punto de enlace capturado dentro del cuerpo del aviso, hay que utilizar una llamada a un método clave de nombre proceed(). Este debe ser invocado con el mismo número y tipo de argumentos que el original y devolver el mismo tipo de argumento que este. Véase el siguiente ejemplo:

```
Object around() memoryOperations(){
        try{
                proceed();
        }catch(Exception e){ /*Manejar la excepción e */ }
```

```
}
```

La intención del ejemplo es tratar de manera uniforme las excepciones que puedan lanzar las operaciones capturadas por el punto de corte memoryOperations. Se envuelve la ejecución de proceed() entre "try/catch", lo que permite eliminar este código de cada uno de los métodos y manejarlo aquí de manera igual para todos.

B.4.3 Paso de información contextual

Una de las mayores utilidades en los avisos es el paso de la información contextual del punto de enlace. De esta forma se tiene acceso a información respecto a los métodos que se capturan, así como a los argumentos que estos reciben. Esto permite poder actuar de diferentes formas según el método o el tipo de los objetos capturados, así como alterar o hacer comprobaciones sobre los parámetros antes de pasar el control a los puntos de enlace. Dicha información es lo que se conoce como contexto. AspectJ dispone de una serie de herramientas para exponer dicho contexto en los avisos, como son "target()", "this()" y "args()". La sintaxis es muy similar a la utilizada en Java para el paso de parámetros en funciones: En la declaración de los avisos y los puntos de corte hay que especificar los parámetros con sus tipos. Dependiendo de si se utilizan puntos de corte anónimos o con nombre, la sintaxis varía ligeramente. Véanse los siguientes ejemplos, donde se ven las diferencias en los casos citados:

```
before ( Statement stmt, String query ) :
call (* Statement.execute*( String )) && target ( stmt ) && args ( query ) {
        checkSQLQueryValidity(query);
}
```

Este primer ejemplo muestra un aviso que hace uso de un punto de corte anónimo. En la definición de "before" se han de especificar todos los argumentos asociados con la ejecución del método capturado. "Target" recoge el objeto en el que se invoca el método capturado cuyo nombre ha de comenzar por execute, mientras que args captura los argumentos de dicho método. La parte del "advice" antes de los dos puntos especifica el tipo y nombre de cada uno de los argumentos que se capturan y que luego en el cuerpo pueden usarse referenciándolos con dichos nombres.

Cuando se usan puntos de corte con nombres, dichos puntos de corte deben recoger la información contextual y pasársela al aviso. El siguiente ejemplo es similar al anterior salvo que usa puntos de corte con nombre.

```
pointcut sqlQueryChecker( Statement stmt, String query ) :
call (* Statement.execute*( String )) && target ( stmt ) && args ( query );
before (Statement stmt, String query ) :
sqlQueryChecker (stmt, query ) {
        checkSQLQueryValidity(query);
}
```

Funcionalmente el código es idéntico al anterior. El punto de corte sqlQueryChecker, además de definir los puntos de enlace, captura el contexto para que

pueda ser usado en el aviso. Al igual que antes, se recoge el objeto sobre el que se invoca el método y sus argumentos. El mismo punto de corte declara el tipo y nombre de cada elemento que se recoge, tal como se declara en un método normal. La primera parte del aviso es igual que en el caso anterior, no varía. Y a continuación se usa el punto de corte que se ha definido más arriba, haciendo coincidir los nombres de los argumentos, pero sin necesidad de indicar los tipos, pues ya se ha hecho. Otro ejemplo de paso de información contextual es el que se muestra a continuación:

```
after() returning(Connection conn ) :
call(Connection DriverManager.getConnection(..)) {
        System.out.println("Obtenida la conexión: "+ conn );
}
```

El aviso "after" returning captura el valor devuelto por el método getConnection para poder ser utilizado en el cuerpo del aviso. Para ello se ha especificado el tipo y el nombre del valor devuelto en la parte returning() del aviso. De esta forma puede usarse el valor devuelto tal como cualquier otro objeto del contexto. Igualmente puede hacerse con el aviso "after" throwing para capturar la excepción lanzada, tal como se muestra en el siguiente ejemplo:

```
after() throwing (RemoteException ex ):
call(* *.*(..) throws RemoteException) {
        System.out.println("Lanzada " + ex+ " al ejecutar "+ thisJointPoint );
}
```

Haciendo uso de la API de AspectJ puede también accederse a la información contextual. Así se tienen las funciones getThis(), getTarget() y getArgs() que acceden a la misma información explicada antes. Sin embargo existe una diferencia: Usar parámetros en la definición de los puntos de corte, como hemos visto primero, proporciona comprobación de tipos en tiempo de compilación, así como la seguridad de que tales parámetros existen. Por ejemplo, al usar el punto de corte target(TipoX) aseguramos que sólo se han llamado a funciones sobre un objeto de tipo TipoX y no de otro. Mientras que si usamos dentro del aviso la función JoinPoint.getTarget(), no podremos asegurar nada sobre el tipo del objeto que vamos a obtener y, ni siquiera, de si existe o no.

Otro aspecto a tener en cuenta es la devolución de valores desde los avisos "around". Frecuentemente el valor que se devuelve se declara del mismo tipo del valor devuelto en los puntos de enlaces a los que afecta el aviso. La llamada a proceed() devuelve el valor retornado por el método en el punto de enlace. A no ser que se necesite manipular el valor devuelto, el aviso "around" simplemente devuelve el valor que devuelva proceed() –sino se invocara este método, habría que devolver un valor apropiado para la lógica del aviso. Se puede dar el caso de que el conjunto de puntos de enlaces al que afecta un "advice" tengan diferentes valores de retorno. En tal caso la solución es definir el valor devuelto por el "advice" como Object.

B.4.4 Reglas de entrelazado estático

En AOP a menudo se necesita no sólo alterar el comportamiento dinámico de la aplicación, sino también modificar la estructura estática. Es lo que se conoce como entrelazado estático, frente a lo que hemos visto hasta ahora que se correspondía con el entrelazado dinámico. Hay cuatro maneras en las que AspectJ implementa lo que la terminología inglesa define como *static crosscutting*. Estas son: introducciones, modificaciones de la estructura jerárquica, errores y avisos en tiempo de compilación y suavizado de excepciones. Veamos a continuación cada una de ellas.

B.4.4.1 Introducciones

Los aspectos a menudo necesitan introducir atributos y métodos en las clases a las que afectan. AspectJ dispone de un mecanismo llamado *introduction* para introducir dichos miembros en las clases e interfaces que se especifiquen, de forma que no sea necesario modificar las clases afectadas. Tales miembros pasan a ser parte de las clases entrelazadas y pueden tratarse como tal, con las condiciones de visibilidad que se especifiquen. La sintaxis para las introducciones es como se muestra a continuación:

```
Modifiers Type TypePattern.Id(Formals) { Body }
abstract Modifiers Type TypePattern.Id(Formals);
```

El efecto que tiene tal introducción, es hacer que todos los tipos indicados en TypePattern dispongan de un nuevo método de nombre Id y cuerpo el especificado entre llaves. Igualmente se pueden introducir constructores de la siguiente forma:

```
Modifiers TypePattern.new(Formals) { Body }
```

A diferencia del ejemplo anterior, en este caso no está permitido introducir constructores en interfaces. Así que si TypePattern incluye una interfaz, se producirá un error de compilación. De forma similar, se pueden introducir atributos:

```
Modifiers Type TypePattern.Id = Expression;
Modifiers Type TypePattern.Id;
```

En una declaración se pueden introducir varios elementos en diferentes clases a la vez. En el siguiente ejemplo, se introducen tres métodos en tres clases diferentes, Point, Line y Square.

```
public String (Point || Line || Square).getName() { return name; }
```

Los tres métodos tienen el mismo nombre (getName), no tienen parámetros y devuelven un String. Los tres tendrán además el mismo cuerpo, lo cual es la gran ventaja de esta técnica, pues facilita la adaptación del código.

En cuanto a la visibilidad, AspectJ permite introducciones de miembros públicos, privados o con visibilidad de paquete (esto es, *protected*). El uso del modificador private implica una visibilidad privada con relación al aspecto, no necesariamente con el destino de la introducción. Así si un aspecto hace una introducción privada de un atributo en una clase, como podría ser, private int A.x, entonces el código en el aspecto puede hacer

referencia al atributo x, pero ninguna otra clase puede hacerlo. Similarmente, si un aspecto hace una introducción con el modificador protected, tal como, protected int A.x, entonces cualquiera desde el paquete del aspecto (que no tiene porqué ser el mismo que el de A) podrá acceder al atributo x.

Por último, comentar que si la palabra reservada "this" aparece en el cuerpo de los nuevos constructores o métodos introducidos, esta hará referencia al destino de TypePattern y no al aspecto donde se declara.

B.4.4.2 Modificación de la estructura estática

Muchas veces la implementación de los problemas que afectan de manera horizontal a las aplicaciones, requieren actuar sobre un conjunto de clases o interfaces que comparten una base común, de tal forma que los avisos y aspectos trabajen únicamente con la API que ofrece tal tipo base. De tal forma, los avisos y aspectos serán dependientes del tipo base, en lugar de las clases e interfaces específicas de cada aplicación. Por ejemplo, un aspecto de manejo de la caché puede declarar que ciertas clases implementen la interfaz Cacheable. Así el aviso en el aspecto podrá trabajar solamente con la interfaz de Cacheable. El resultado de tal práctica es el desacoplamiento del aspecto de las clases específicas de la aplicación, haciendo así que los aspectos sean reutilizables.

Con AspectJ, se puede modificar la estructura jerárquica de las clases haciendo que extiendan a una nueva clase o que implementen una lista de interfaces (siempre y cuando no se violen las reglas de herencia de Java). La sintaxis para tales declaraciones es la siguiente:

declare parents: [*ChildTypePattern*] implements [*InterfaceList*];
declare parents: [*ChildTypePattern*] extends [*Class* or *InterfaceList*];

Por ejemplo, el siguiente aspecto declara que todas las clases e interfaces del paquete B del paquete A tienen que implementar la interfaz C:

```
aspect MyAspect {
        declare parents : A..B.* implements C;
        //...
}
```

La declaración de superclases debe seguir las reglas jerárquicas que impone Java. Por ejemplo, no se puede declarar que una clase sea el padre de una interfaz. Y de igual forma, no se pueden declarar padres, de tal forma que resulte en herencia múltiple. Todo ello resultaría en errores en tiempo de compilación.

B.4.5 Errores y warnings en tiempo de compilación

AspectJ dispone de un mecanismo para declarar errores y advertencias en tiempo de compilación basado en los puntos de corte explicados anteriormente. De esta forma pueden implementarse comportamientos similares a los que se obtienen con las directivas de preprocesamiento #error y #warning que soportan algunos preprocesadores C y C++.

El constructor declare error permite declarar errores de compilación cuando el compilador detecta la presencia de cierto punto de enlace que se haya especificado con un punto de corte. En tal caso, el compilador informa del error mediante el mensaje proporcionado por el programador y aborta el proceso de compilación. La sintaxis del constructor es como se muestra a continuación:

```
declare error : <pointcut> : <message>;
```

De igual forma se pueden declarar advertencias mediante el siguiente constructor:

```
declare warning : <pointcut> : <message>;
```

Al afectar estas declaraciones al comportamiento en tiempo de compilación es evidente que no se pueden usar puntos de corte basados en el contexto como "this()", "target()", "args()", "if()", "cflow()", y "cflowbelow()".

Un uso típico de estos constructores es para definir reglas que eviten el uso de ciertos métodos no soportados o bien para limitar la localización de determinados usos del código. Un ejemplo sacado de la aplicación que sirve como ejemplo para este proyecto es el siguiente. A efectos de evitar la dispersión del código de base de datos, si limita el uso de éste al paquete bd y se permite, aunque avisando con un warning, en el paquete aspects.

```
pointcut dbCode(): call(Connection DriverManager.getConnection(..));
pointcut notRecommendedDbCode(): dbCode() && !within(db.*);
pointcut forbidedDbCode(): notRecommendedDbCode() && !within(aspects.*);
declare warning: notRecommendedDbCode(): "Codigo no recomendado!";
declare error: forbidedDbCode(): "Código no permitido!";
```

B.4.6 Suavizado de excepciones

Es un dato estadístico el hecho de que en las aplicaciones Java aproximadamente el treinta por cierto del código corresponde al tratamiento de errores. El uso en las mismas de los mecanismos de lanzamiento de excepciones y su captura es una solución al problema, pero sin duda uno de los síntomas más evidentes de enredamiento del código. En AspectJ un aspecto puede especificar que cuando en los puntos de enlace indicados se lance cierta excepción, ésta debe saltarse el sistema usual de tratamiento de excepciones de Java y lanzarse como del tipo org.aspectj.lang.SoftException, que es subtipo de RuntimeException y que por lo tanto no necesita ser declarada. Se dice que la excepción ha sido suavizada, del inglés *softened*. La sintaxis para especificar tal comportamiento en AspectJ es la siguiente:

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

En el siguiente ejemplo, un aspecto MyAspect declara Exception en una cláusula declare soft, lo cual implica que cualquier excepción que se lance será encapsulada en una excepción del tipo org.aspectj.lang.SoftException y relanzada como tal.

```
aspect MyAspect {
        declare soft: Exception: execution(void main(String[] args));
```

```
}
```

El efecto conseguido es el mismo que se conseguiría con el siguiente aviso:

```
aspect MyAspect {
        void around() execution(void main(String[] args))
        {
                try {
                        proceed();
                }catch (Exception e) {
                        throw new org.aspectj.lang.SoftException(e);
                }
        }
}
```

B.4.7 Extensión de los aspectos

Al igual que las clases, los aspectos se pueden extender, de forma que permitan la abstracción y composición de los problemas que implementan. Sin embargo, introducen algunas reglas a la hora de extenderse, tal como son:

- Un aspecto, abstracto o concreto, puede extender una clase e implementar un conjunto de interfaces.
- Las clases no pueden extender aspectos. Intentar que una clase extienda o implemente un aspecto resultará en un error de compilación.
- Aspectos que extienden aspectos. Los aspectos pueden extender otros aspectos, con lo cual no sólo se heredan los atributos y métodos, sino también los puntos de corte. Sin embargo, los aspectos sólo pueden extender aspectos abstractos. Intentar que un aspecto extienda un aspecto concreto resultará en un error de compilación.

Por ejemplo, algo usual es implementar los sistemas de log como aspectos. Así si queremos mantener un registro o historial de ciertas acciones, se puede definir un aspecto abstracto con un punto de corte abstracto, donde se indique cómo se quiere escribir la información, pero no cuales van a ser los puntos de enlace. Los aspectos que extiendan dicho aspecto, son los que concretan los puntos de corte, lo que da la posibilidad de definir clara y separadamente los elementos que se quieren controlar. El siguiente fragmento de código muestra el aspecto abstracto:

```
abstract public aspect Logger {
        abstract pointcut logPoint();
        before(): logPoint() && !within(Logger+){
                System.log(thisJoinPoint.getTarget()+"/"+thisJoinPoint.getThis());
        }
    }
```

Del punto de corte se excluyen el propio aspecto y sus subclases para evitar la recursividad. El aviso recoge el objeto sobre el que se llaman las funciones y el objeto

actual y los escribe en el log del sistema. El siguiente fragmento muestra un aspecto concreto que extiende a este aspecto:

```
public aspect LogDataBaseExecutions extends Logger{
        pointcut logPoint(): call(* DataBase.*.*(..));
}
```

Este aspecto proporciona una definición del punto de corte logPoint, que en este caso son las todas las llamadas a funciones de un supuesto paquete de base de datos DataBase.

B.4.8 Instancias aspectos

A diferencia de las clases, los aspectos no se pueden instanciar con expresiones new. En lugar de eso, las instancias de los aspectos se crean automáticamente. Debido a que los "advices" sólo se ejecutan en el contexto de una instancia de un aspecto, el cómo se instancian los aspectos controla indirectamente cuando se ejecutan los avisos. El criterio utilizado para determinar cómo se instancia un aspecto se hereda del aspecto al que extiende. Si éste no existiera, entonces por defecto, los aspectos siguen una política *singleton* (es decir, sólo existe una instancia de ellos). Los siguientes tres subapartados indican las diferentes posibilidades a la hora de instanciar aspectos y la sintaxis para cada una de ellas.

B.4.9 Aspectos singleton

Sintaxis:

aspect *Id* { ... }
aspect *Id* issingleton { ... }

Por defecto, o usando el modificador issingleton, los aspectos tienen exactamente una instancia que se ejecuta para todo el programa. Dicha instancia está disponible en cualquier momento durante la ejecución del programa usando el método estático aspectOf() definido en los aspectos. Esta instancia se crea tal como se carga el archivo que contiene el *class* del aspecto y vive durante toda la ejecución del programa, por lo que sus avisos podrán ejecutarse en cualquier punto de enlace.

B.4.10 Aspectos per-object

Sintaxis:

aspect *Id* perthis(Pointcut) { ... }
aspect Id pertarget(*Pointcut*) { ... }

Si un aspecto A se define perthis("Pointcut") entonces existirá una instancia diferente del aspecto A cada vez que el objeto que se está ejecutando en los puntos de enlace definidos en "Pointcut" (esto es, el referenciado por "this") sea diferente. Así, los avisos definidos en A pueden ejecutarse en cualquier punto de enlace donde el objeto

actual en ejecución haya sido asociado con una instancia de A. Sino se ha creado aún una instancia de A para el objeto en cuestión, se crea una nueva.

B.4.11 Aspectos per-control-flow

Sintaxis:

aspect *Id* percflow(*Pointcut*) { ... }
aspect Id percflowbelow(Poin*tcut*) { ... }

Si un aspecto A se define percflow("Pointcut") o percflowbelow("Pointcut") entonces un objeto de tipo A se crea cada vez que el control pasa a uno de los puntos de enlace capturados por el "Pointcut", bien tal como el flujo de control pasa al punto de enlace o a partir de éste, pero excluyéndolo, respectivamente.

B.4.12 Aspect privilege

Sintaxis:

privileged aspect *Id* { ... }

El código escrito en los aspectos está sujeto a las mismas reglas de control de acceso que el código Java en lo que respecta a los miembros de las clases o los aspectos. Así, por ejemplo, el código escrito en un aspecto no puede hacer referencia a miembros con visibilidad *protected*, a no ser que el aspecto esté definido en el mismo paquete. Mientras que estas restricciones son aceptables para muchos aspectos, puede haber casos en los que los aspectos necesiten en sus *"advices"* o *"introductions"* acceder a recursos que son privados o protegidos. Para permitir esto, los aspectos tienen que ser declarados privileged. En el código de los miembros privileged se tiene acceso a todos los miembros, incluidos los privados.

B.4.13 Aspectos dominantes

Sintaxis:

aspect *Id* dominates *TypePattern* { ... }

Un aspecto puede declarar que los *"advices"* que define dominan sobre los *"advices"* en otros aspectos, lo que implica que el aviso dominante tiene mayor precedencia que los especificados en los aspectos que coincidan con TypePattern. Por defecto en AspectJ no existe ningún orden definido, por lo que, si se precisa ejecutar los avisos en determinado orden, es necesario especificarlo con la cláusula dominates. La semántica es que si un aspecto A domina sobre un aspecto B, entonces los avisos de A tienen más prioridad y se ejecutan antes que los de B. En cuanto a las situaciones de conflicto como puede ser la siguiente, el compilador 1.0.6 no informa del error y simplemente actúa ignorando una de las dos.

```
aspect A dominates B{...}
aspect B dominates A{...}
```

Hay que advertir que en la versión 1.1 de AspectJ la sintaxis de dominates ha cambiado, desapareciendo dominates y pasando a usarse la siguiente declaración:

declare precedence ":" TypePatternList ";"

Por poner un ejemplo, si quisiéramos definir una regla de precedencia, de modo que los avisos de los aspectos cuyo nombre contenga la cadena Security precedan a cualquier otro y que además, los avisos del aspecto Logging y sus subclases precedan a los restantes, entonces deberíamos escribir lo siguiente:

declare precedence: *..*Security*, Logging+, *;

Sería un error en una sentencia precedence que un aspecto pudiera ser referido por más de un patrón de tipos.

Así por ejemplo, la siguiente declaración produciría un error:

declare precedence: A, B, A ;

Sin embargo no es un error tener en múltiples sentencias este tipo de problema de circularidad, como puede producirse con las siguientes dos declaraciones:

declare precedence: B, A;
declare precedence: A, B;

Y de hecho un sistema así definido puede ser correcto, siempre y cuando los avisos de A y B no compartan puntos de enlace. De esa forma queda claramente expresado que A y B son independientes.

B.5 AspectJ quick reference

Aspectos

aspect *A* { ... }	Define el aspecto *A*.
privileged aspect *A* { ... }	*A* puede acceder campos y métodos privados.
aspect *A* **extends** *B* **implements** *I, J* { ... }	*B* es una clase o aspecto abstracto, *I* y *J son* interfaces.
aspect *A* **percflow(** *call(void Foo.m())*) { ... }	Una instancia de *A* es instanciada para todo flujo de control que llame a *m()*.

forma general:

[**privileged**] [*Modifiers*] **aspect** *Id*
[**extends** *Type*] [**implements** *TypeList*] [*PerClause*]
{ *Body* }

where *PerClause* is one of

pertarget (*Pointcut*)
perthis (*Pointcut*)
percflow (*Pointcut*)
percflowbelow (*Pointcut*)
issingleton

Definiciones de Pointcut en tipos

private pointcut *pc*() : *call(void Foo.m())* ;	Un pointcut visible solo para el tipo definido.
pointcut *pc(int i)* : *set(int Foo.x) && args(i)* ;	Un pointcut visible solo para cuando se expone un *int*.
public abstract pointcut *pc*() ;	Un pointcut abstracto que puede ser referenciado en cualquier parte.

abstract pointcut *pc(Object o)* ; Un pointcut abstracto visible solo para el paquete que lo define. Cualquier pointcut que lo implemente debe exponer un *Object*.

general form:

abstract [*Modifiers*] **pointcut** *Id* (*Formals*) ;
[*Modifiers*] **pointcut** *Id* (*Formals*) : *Pointcut* ;

Declaraciones Advice en aspectos

before () : *get(int Foo.y)* { ... } Se ejecuta antes de leer el campo *int Foo.y*

after () **returning** : *call(int Foo.m(int))* { ... } Se ejecuta después de llamar a *int Foo.m(int)* que retorna de forma normal.

after () **returning** (*int x*) : *call(int Foo.m(int))* { .. } Igual que el anterior, pero el valor de retorno es llamado *x* en el body.

after () **throwing** : *call(int Foo.m(int))* { ... } Se ejecuta después de llamar a *m* que retorna lanzando una excepción.

after () **throwing** (*NotFoundException e*) : *call(int Foo.m(int))* { ... } Se ejecuta despupes de llamar a *m* que retorna una excepción *NotFoundException*. La excepción es llamada *e* en el body.

after () : *call(int Foo.m(int))* { ... } Se ejecuta después de llamar a *m* sin importar como retorne.

before(*int i*) : *set(int Foo.x) && args(i)* { ... } Se ejecuta antes de asignar el campo *int Foo.x*. El valor que será asignado es llamado *i* en el body.

before(*Object o*) : *set(* Foo.*) && args(o)* { ... } Se ejecuta antes de asignar cualquier campo de *Foo*. El valor que será asignado se convierte a tipo object (*int* a *Integer*, por ejemplo) y es llamado *o* en el body.

int **around** () : *call(int Foo.m(int))* { ... } Se ejecuta en lugar de la llamada a *int Foo.m(int)*, y retorna un *int*. En el body, continua la llamada usando **proceed**(), con la misma firma que el advice around.

int **around** () **throws** *IOException* : *call(int Foo.m(int))* { ... } Igual que el anterior, pero el body puede lanzar una *IOException*

Object **around** () : *call(int Foo.m(int))* { ... } Igual que el anterior, pero el valor de **proceed**() es convertido a un *Integer*, y el body deberá también retornar un *Integer* el cual será convertido a un *int*

general form:

[**strictfp**] *AdviceSpec* [**throws** *TypeList*] : *Pointcut* { *Body* }

where *AdviceSpec* is one of

before (*Formals*)
after (*Formals*)
after (*Formals*) **returning** [(*Formal*)]
after (*Formals*) **throwing** [(*Formal*)]
Type **around** (*Formals*)

Special forms in advice

thisJoinPoint Información reflectiva acerca del join point.

thisJoinPointStaticPart Equivalente a **thisJoinPoint.getStaticPart()**, pero usa menos recursos.

thisEnclosingJoinPointStaticPart La parte estática de el join point que lo encierra.

proceed (*Arguments*) Solo para advice **around**. Los *Argumentos* deberan de ser el mismo numero y tipo que los parámetros del advice.

Inter-type Member Declarations in aspects

int Foo . m (*int i*) { ... } Un método *int m(int)* de *Foo*, visible en cualquier lugar en el paquete definido. En el body, **this** se refiere a la instancia de *Foo*, no al aspecto.

private *int Foo . m* (*int i*) **throws** *IOException* { ... } Un método *int m(int)* que es declarado que lanza *IOException*, *solo* visible en el aspecto definido. En el body, **this** se refiere a la instancia de *Foo*, no al aspecto.

abstract *int Foo . m* (*int i*) ; Un método abstracto *int m(int)* de *Foo*

Point . **new** (*int x, int y*) { ... } Un constructor de *Point*. In el body, **this** se refiere al nuevo *Point*, no al aspecto.

private static *int Point . x* ; Un campo estático *int* llamado *x* de cualquier *Point* y visible solo en el aspecto declarado.

private *int Point . x = foo()* ; Una inicialización de un campo no-estático con el resultado de llamar a *foo()*. En la inicialización, **this** se refiere a la instancia de *Foo*, no al aspecto.

general form:

[*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] { *Body* }

abstract [*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] ;
[*Modifiers*] *Type* . new (*Formals*)
[**throws** *TypeList*] { *Body* }
[*Modifiers*] *Type Type . Id* [= *Expression*] ;

Other Inter-type Declarations in aspects

declare parents : *C* **extends** *D*;	Declara que la superclases de *C* es *D*.
declare parents : C **implements** *I, J* ;	*C* implementa *I* y *J*.
declare warning : *set(* Point.*) && !within(Point)* : *"bad set"* ;	El compilador advierte *"bad set"* si encuentra un set de cualquier campo de *Point* fuera del código de *Point*.
declare error : *call(Singleton.new(..))* : *"bad construction"* ;	El compilador manda el error *"bad construction"* si encuentra una llamada a cualquier constructor de *Singleton*.
declare soft : *IOException* : *execution(Foo.new(..))*;	Cualquier IOException lanzada por ejecución de constructores de *Foo* son re envueltas en **org.aspectj.SoftException**
declare precedence : *Security, Logging, ** ;	A cada join point, advice para *Security es precedido* por el advice para *Logging*, que tiene precedencia sobre los otros advice's.

general form

declare parents : *TypePat* **extends** *Type* ;
declare parents : *TypePat* **implements** *TypeList* ;
declare warning : *Pointcut* : *String* ;
declare error : *Pointcut* : *String* ;
declare soft : *Type* : *Pointcut* ;
declare precedence : *TypePatList* ;

Primitive Pointcuts

call (*void Foo.m(int)*)	Una llamada al método *void Foo.m(int)*
call (*Foo.new(..)*)	Una llamada cualquier constructor de *Foo*.
execution (* *Foo.*(..) throws IOException*)	La ejecución de cualquier método de *Foo* que es declarado que lanza *IOException*.
execution (*!public Foo .new(..)*)	La ejecución de cualquier constructor no publico de *Foo*.
initialization (*Foo.new(int)*)	La inicialización de cualquier objeto *Foo* que es iniciado con el constructor *Foo(int)*
preinitialization (*Foo.new(int)*)	La pre-initialización (antes de que el constructor de **super** sea llamado) que es iniciado con el constructor *Foo(int)*.
staticinitialization(*Foo*)	Cuando el tipo *Foo* es inicializado, después cargado.
get (*int Point.x*)	cuando *int Point.x* es leido.
set (*!private * Point.**)	Cuando cualquier campo no privado de *Pointe e*s asignado.
handler (*IOException+*)	Cuando una *IOException* o un subtipo de ella es manejada con un cloque catch.
adviceexecution()	La ejecución de todos los advice bodies.
within (*com.bigboxco.**)	Cualquier join point donde el código asociado es definido en el paquete *com.bigboxco*
withincode (*void Figure.move()*)	Cualquier join point donde el código asociado es definido en el método *void Figure.move()*
withincode (*com.bigboxco.*.new(..)*)	Cualquier join point donde el código asociado es definido en cualquier constructor en el paquete *com.bigoxco.*
cflow (*call(void Figure.move())*)	Cualquier join point el flujo de control de cada llamada a *void Figure.move()*. Incluye la llamada a el mismo.
cflowbelow (*call(void Figure.move())*)	Cualquier join point abajo del flujo de control de cada llamada a *void Figure.move()*. No incluye la llamada a el mismo.
if (*Tracing.isEnabled()*)	Cualquier join point donde *Tracing.isEnabled()* es **true**. La expresión boleana usada solo puede acceder miembros estáticos, variables bound en el mismo pointcut, y de la forma **thisJoinPoint**.
this (*Point*)	Cualquier join point donde el objeto en ejecución es una instancia de *Point*.

target (*Java.io.InputPort*)	Cualquier join point donde el objeto target es una instancia de *Java.io.InputPort.*
args (*Java.io.InputPort, int*)	Cualquier join point donde hay dos argumentos, el primero una instancia de *Java.io.InputPort,* y el segundo un *int.*
args (**, int*)	Cualquier join point donde hay dos argumentos, el segundo debe ser un *int.*
args (*short, .., short)*	Cualquier join point con al menos dos argumentos, el primero y el último deben ser *shorts.*

general form:

call(*MethodPat*)
call(*ConstructorPat*)
execution(*MethodPat*)
execution(*ConstructorPat*)
initialization(*ConstructorPat*)
preinitialization(*ConstructorPat*)
staticinitialization(*TypePat*)
get(*FieldPat*)
set(*FieldPat*)
handler(*TypePat*)
adviceexecution()
within(*TypePat*)
withincode(*MethodPat*)
withincode(*ConstructorPat*)
cflow(*Pointcut*)
cflowbelow(*Pointcut*)
if(*Expression*)
this(*Type* | *Var*)
target(*Type* | *Var*)
args(*Type* | *Var* , ...)

where *MethodPat* is:

[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat* (*TypePat* | .., ...)
[**throws** *ThrowsPat*]

ConstructorPat is:

[*ModifiersPat*] [*TypePat* .] **new** (*TypePat* | .. , ...)
[**throws** *ThrowsPat*]

FieldPat is:

[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat*

TypePat is one of:

IdPat [+] [[] ...]
! *TypePat*
TypePat **&&** *TypePat*
TypePat || *TypePat*
(*TypePat*)

Tabla B.8 AspectJ quick reference

Apéndice C

C Invocación remota de métodos (RMI)

En el presente apéndice se describe la arquitectura de una de las principales alternativas para el desarrollo de aplicaciones basadas en objetos distribuidos, Java RMI. Además se presenta de manera detallada la capacidad de activar "sobre demanda" a los objetos remotos.

En la actualidad, el cómputo distribuido ocupa un lugar preponderante tanto en las ciencias de la computación como en la industria, debido a que muchos de los problemas a los que se enfrentan son inherentemente 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 cómputo distribuido 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 a la distribución física de estos objetos en diferentes procesos o computadoras una extensión natural.

La invocación remota de métodos de Java es un modelo de objetos distribuidos, diseñado específicamente para el lenguaje Java, por lo que mantiene la semántica del modelo de objetos locales de Java, facilitando de esta manera la implantación y el uso de objetos distribuidos. En el modelo de objetos distribuidos de Java, un objeto remoto es aquel cuyos métodos pueden ser invocados por objetos que se encuentran en una máquina virtual (MV) diferente. Los objetos de este tipo se describen por una o más interfaces remotas que contienen la definición de los métodos del objeto que es posible invocar remotamente.

La invocación remota de un método (RMI) es la acción de invocar un método de una interfaz remota de un objeto remoto. La invocación de un método de un objeto remoto tiene exactamente la misma sintaxis de invocación que la de un objeto local.

C.1 Metas del sistema RMI de Java

Las metas que se pretenden alcanzar al soportar objetos distribuidos en Java, son:

- Proporcionar invocación remota de objetos que se encuentran en MVs diferentes.
- Soportar llamadas a los servidores desde los applets.
- Integrar el modelo de objetos distribuidos en el lenguaje Java de una manera natural, conservando en medida de lo posible la semántica de los objetos Java.
- Hacer tan simple como sea posible, es decir, facilitar la escritura de aplicaciones distribuidas.
- Preservar la seguridad de tipos (type safety) proporcionada por el ambiente Java.
- Proporcionar varias semánticas para las referencias de los objetos remotos (persistentes, no persistentes y de "activación retardada o lenta"). En [26] se

describen los modelos de seguridad que implementan los sistemas en tiempo de ejecución Java.
- Distinción entre objetos remotos y no remotos.
- Mantener la seguridad del ambiente dada por los Security Managers, en particular, en presencia de carga dinámica de clases.

C.2 Aplicaciones con objetos distribuidos en Java

Un programa orientado a objetos consta de una colección de objetos que interactúan entre sí, solicitando y proporcionando servicios mediante el intercambio de mensajes, por lo que se ajustan bastante bien al diseño de sistemas distribuidos, debido a que estos mensajes enviados y recibidos por los objetos, pueden ser extrapolados a mensajes enviados a objetos en diferentes máquinas a través de la red. Los objetos distribuidos radican en aplicaciones que pueden actuar como clientes, servidores o ambos, dependiendo de si contienen objetos remotos, referencias de ellos o ambos.

Una aplicación servidora que utiliza RMI, generalmente crea cierto número de objetos remotos, permite que se creen referencias a dichos objetos y espera a que algún cliente invoque de manera remota los métodos de dichos objetos. Una aplicación cliente típica obtiene referencias de uno o más objetos remotos, e invoca sus métodos por medio del sistema RMI de Java, el que se encarga de proporcionar los mecanismos mediante los cuales, tanto clientes como servidores, pueden comunicarse.

Las funciones esenciales que deben desarrollar las aplicaciones distribuidas, son:

- Localizar objetos remotos. Las aplicaciones cliente tienen dos alternativas para obtener referencias de objetos remotos. Una aplicación puede registrar sus objetos remotos ante un servidor de nombres llamado "rmiregistry", o la aplicación puede pasar referencias a objetos remotos como parámetro de una invocación o como valor de retorno.
- Comunicarse con los objetos remotos. Los detalles de la comunicación entre los objetos remotos, son manejados por el sistema RMI. Para el programador, la comunicación entre objetos se asemeja a la utilizada normalmente en programas Java.
- Cargar el código de operación que implementa a las clases que son pasadas por valor. Debido a que RMI permite pasar objetos Java puros, como parámetros en la invocación de métodos de objetos remotos, proporciona los mecanismos necesarios para, por medio de un servidor HTTP o FTP, cargar el código y los datos de dichos objetos. En la figura C.1 se muestra una aplicación distribuida, basada en RMI, que utiliza al servidor de nombres "rmiregistry" para obtener referencias de objetos remotos. El servidor que implementa los objetos remotos, invoca al "rmiregistry" para asociarle un nombre a un objeto remoto. El cliente busca al objeto remoto utilizando su nombre como argumento y, finalmente, invoca alguno de sus métodos. La figura C.1 también muestra cómo el sistema RMI puede usar un servidor web para cargar códigos de operación, de clientes a servidores y de servidores a clientes, mediante el empleo de cualquier protocolo URL.
- Lazy activation. Como se verá más adelante, consiste en activar un objeto hasta que se invoca alguno de sus métodos. Un localizador de recursos uniforme (Uniform Resourse Locator) es una representación compacta de la localización y del medio de

acceder a algún recurso disponible vía Internet. El URL proporciona un apuntador a cualquier objeto que sea accesible en cualquier máquina conectada a Internet. Debido a que los objetos son accesibles de diferentes maneras (ftp, http, gopher, file, etc.), el URL indica además el método de acceso que se debe utilizar para obtener el objeto deseado.

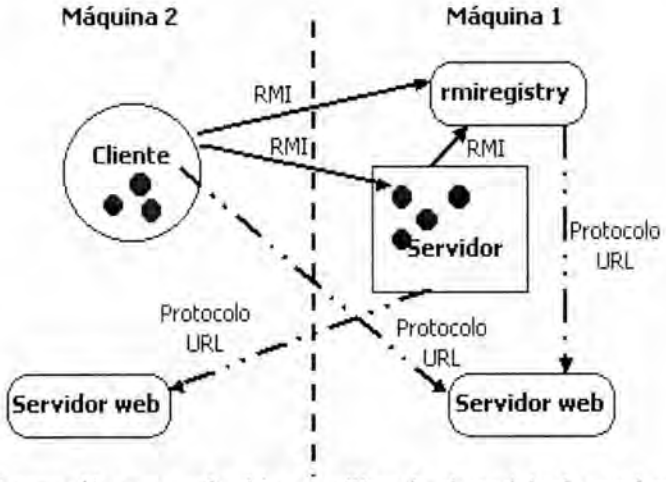

Figura C.1 Invocación remota de objetos, utilizando el servicio de nombres "rmiregistry"

C.3 RMI, arquitectura

El sistema RMI está formado por 3 niveles, ver la figura C.2; los niveles son:

- El nivel de los cabos (Stubs) y esqueletos (Skeletons).
- El nivel de referencias remotas.
- El nivel de transporte.

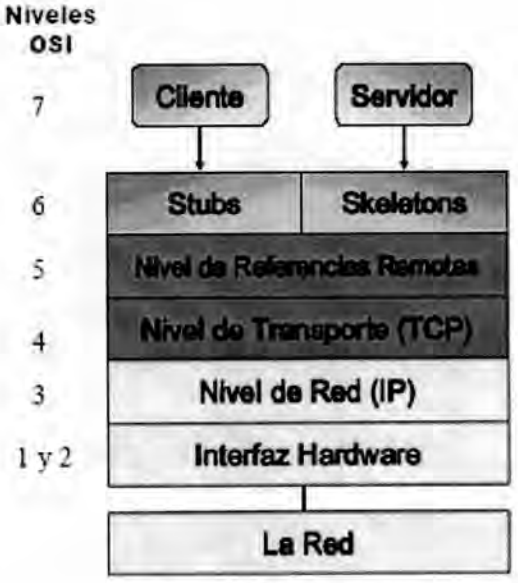

Figura C.2 RMI, arquitectura

El cliente y el servidor desarrollan sus aplicaciones en paralelo, pues el cliente invoca objetos remotos a través de los cabos; los esqueletos permiten hacer accesibles los objetos servidores al cliente.

En la figura C.3, se muestran las clases principales del sistema Java RMI.

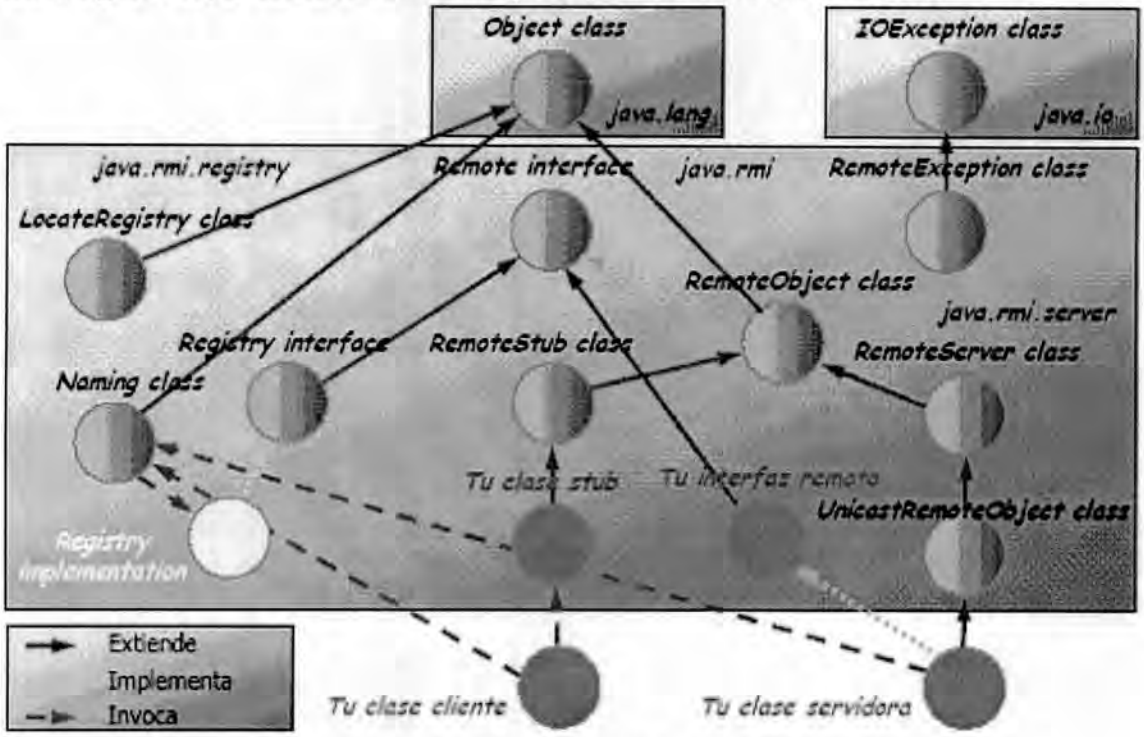

Figura C.3 RMI, clases principales

C.4 Coincidencias y diferencias entre el modelo de objetos local y distribuido de Java

Como se mencionó, 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

En la tabla C.1 se tiene una comparación de objeto remoto vs. objeto no remoto.

Objeto no remoto	Objeto remoto
Instancia de una clase Java	Instancia de una clase Java que implementa una o varias interfaces que extienden Java.rmi.Remote
Se crea una nueva instancia con new()	En la misma JVM se crea una nueva instancia con new(); desde otra JVM sólo se puede crear una instancia por activación remota
Acceso directo	Acceso por un Stub o cabo
Eliminado por el garbage collector cuando no hay referencias locales que apuntan a él	Eliminado por el garbage collector cuando no hay ni referencias locales ni referencias remotas activas que apuntan a él; una referencia remota se considera activa sino ha sido solicitado y se ha realizado un acceso hace poco (con tiempo configurable)
Sus métodos no pueden lanzar Java.rmi.Remoteexception	Sus métodos pueden lanzar Java.rmi.RemoteException

Tabla C.1 Objeto no remoto vs. objeto remoto

C.5 Esqueletos y cabos (stubs)

RMI utiliza el mismo mecanismo que los sistemas RPC para implementar la comunicación con los objetos remotos, el basado en esqueletos y cabos. Los cabos forman parte de las referencias y actúan como representantes de los objetos remotos ante sus clientes. En el cliente se invocan los métodos del cabo, quien es el responsable de invocar de manera remota al código que implementa al objeto remoto. En RMI un cabo 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 cabo, realiza las siguientes acciones:

- Inicia una conexión con la MV que contiene al objeto remoto.
- Aplana (marshals) y transmite los parámetros de la invocación a la MV remota.
- Espera por el resultado de la invocación.
- Desaplana (unmarshals) y devuelve el valor de retorno o la excepción.
- Devuelve el valor a quien lo llamó.

Los cabos se encargan de ocultar el aplanado de los parámetros, así como los mecanismos de comunicación empleados. En la MV remota, cada objeto debe poseer su esqueleto correspondiente. El esqueleto es responsable de despachar la invocación al objeto remoto.

Cuando un esqueleto recibe una invocación, realiza las siguientes acciones:

- Desaplana los parámetros necesarios para la ejecución del método remoto.
- Invoca el método de la implantación del objeto remoto.
- Aplana los resultados y los envía de vuelta al cliente.

A partir del JDK 1.2 se introdujo un protocolo adicional a los cabos, para eliminar la necesidad de los esqueletos en el lado de las implantaciones de los objetos remotos, y en lugar de ellos, se utiliza código genérico para realizar todas las operaciones que eran

responsabilidad de los esqueletos en el JDK 1.1. Tanto cabos como esqueletos, son generados por un compilador llamado rmic.

C.6 Utilización de hilos en la invocación de métodos remotos

El sistema de tiempo de ejecución de RMI, no garantiza que la activación de las invocaciones de los objetos remotos se proyecte sobre diferentes hilos de ejecución. Pero debido a que es posible que las invocaciones remotas de un mismo objeto se ejecuten concurrentemente, es necesario que las implantaciones de los objetos sean de hilo seguro (thread-safe).

C.7 Recolección de basura de objetos distribuido

En un sistema de objetos distribuidos, como en un sistema local, es deseable contar con un servicio que automáticamente elimine todos aquellos objetos que no posean una referencia activa hacia ellos. RMI utiliza un algoritmo de recolección de basura, el cual está basado en un contador de referencias, para lo cual, el sistema en tiempo de ejecución de RMI da seguimiento a todas las referencias activas de los objetos remotos. La obtención de la primera referencia a un objeto dentro de una MV remota, provoca el envío de un mensaje al servidor que contiene el objeto remoto, indicando que el objeto tiene al menos una referencia activa (referenced) en esa MV. Cuando posteriormente se crean nuevas referencias activas al objeto remoto en dicha MV, en ella se incrementa un contador local de referencias. Cuando se identifican referencias que ya no están activas, el contador local se disminuye, y cuando el contador llega a cero, se envía un mensaje al servidor, indicándole que el objeto remoto no posee referencias activas en esa MV remota (unreferenced). Existen varias sutilezas en este protocolo para asegurar la entrega ordenada de los mensajes "referenceed" y "unreferenced", debido a que pueden provocar que un objeto sea eliminado prematuramente.

Cuando un objeto remoto no posee referencias activas, el sistema en tiempo de ejecución de RMI utiliza una referencia débil ("weak reference") hacia el objeto en cuestión y así, en el momento en que no existan referencias locales, el objeto sea susceptible de ser eliminado por el recolector de basura. El algoritmo de recolección de basura distribuido, interactúa con el recolector de basura de la MV local, manteniendo referencias normales y débiles a los objetos. Cuando se pasa un objeto remoto como parámetro o como valor de retorno, el identificador de la MV destino se añade al conjunto de referencias.

Cuando se necesite una notificación de que un objeto remoto dado ya no tiene referencias activas, se debe implementar la interfaz Java.rmi.server.Unreferenced, que contiene al método unreferenced que se invoca en el momento en que dejen de existir las referencias remotas, es decir, cuando el conjunto de referencias es el vacío. Es importante notar que un objeto puede ser eliminado prematuramente, debido a un fallo en la red y por lo tanto no es posible garantizar integridad referencial a las referencias remotas. En otras palabras, es posible que una referencia remota haga referencia a un objeto que en realidad no exista.

C.8 Carga dinámica de clases

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.

Cuando se desaplanan los valores de retorno o los parámetros de una invocación para convertirlos en objetos activos dentro de la MV que los recibe, es necesario poseer las definiciones de las clases de todos estos objetos. En primera instancia, el proceso de desaplanado intenta resolver las clases por su nombre en el contexto del cargador de clases local (el contexto del cargador de clases del hilo actual). En caso de no encontrar las definiciones de manera local, RMI proporciona la facilidad de cargar dinámicamente la definición de las clases de los objetos recibidos, desde las direcciones especificadas por la información contenida en la invocación. Esto incluye cargar dinámicamente las clases de los cabos, así como cualquier tipo pasado por valor en la llamada RMI.

Para soportar la carga dinámica de clases, RMI utiliza subclases especiales de Java.io.ObjectOutputStream y Java.io.ObjectInputStream, para el manejo de flujos de objetos aplanados. Estas subclases especializan al método annotateClass de la clase ObjectOutputStream y al método resolveClass de la clase ObjectInputStream, para incluir información sobre la localización de los archivos de clase, que contienen la definición de los objetos contenidos en el flujo.

C.9 Utilización de RMI a través de firewalls por medio de proxies

Normalmente, la capa de transporte de RMI intenta abrir sockets directamente a puertos de los servidores a los que se desea conectar, pero en el caso de que los clientes se encuentren detrás de algún firewall que no permita este tipo de conexiones, la capa de transporte estándar de RMI proporciona un mecanismo alterno basado en el protocolo HTTP (confiable para el firewall), para permitir a los clientes que se encuentran detrás del firewall, invocar métodos de objetos que se encuentren del otro lado de él.

Para atravesar el firewall, la capa de transporte de RMI incluye la llamada remota dentro del protocolo HTTP, como el cuerpo de una solicitud POST, mientras que los valores de retorno se reciben en el cuerpo de la respuesta HTTP. La capa de transporte de RMI puede formular la solicitud POST, de alguna de las dos maneras siguientes:

1. Si el proxy firewall permite entregar una solicitud HTTP directamente sobre cualquier puerto de la máquina destino, la solicitud se dirige directamente al puerto donde el servidor RMI se encuentra escuchando. La capa de transporte RMI en la máquina destino escucha con un socket que es capaz de entender y decodificar llamadas RMI, que se encuentren dentro de una solicitud POST.
2. Si el proxy firewall sólo entrega solicitudes HTTP a ciertos puertos HTTP bien conocidos, la llamada se envía a un servidor HTTP que se encuentre escuchando en el puerto 80, donde se puede ejecutar un guión (script) CGI que se encargue de enviar la llamada RMI al puerto específico del servidor.

El sistema de transporte de RMI especializa la clase Java.rmi.server.RMISocketFactory, para proporcionar una implantación por defecto de una fábrica de sockets que se encargue de proveer tanto a clientes como servidores de dichos recursos. La fábrica de sockets por defecto, crea sockets que proporcionan el mecanismo para hacer transparente el paso por firewalls, con las siguientes características:

- Los sockets clientes automáticamente intentan conexiones HTML cuando el servidor no puede contestar directamente.
- Los sockets del lado del servidor, automáticamente detectan si es que una nueva conexión se trata de una solicitud POST de HTTP. En ese caso, devuelven un socket que únicamente expone el cuerpo de la solicitud al sistema de transporte y da formato a sus salidas, como una respuesta HTML.

C.10 Activación de objetos remotos

Los sistemas de objetos distribuidos están diseñados para soportar objetos persistentes de larga vida. Debido a que dichos sistemas pueden estar constituidos por miles o quizás millones de objetos, no es razonable que sus instancias siempre estén activas y consumiendo recursos, aún cuando no se encuentren participando en ninguna tarea. Por otro lado, los clientes necesitan tener la capacidad de almacenar referencias persistentes de objetos, de manera tal que se pueda restablecer la comunicación con ellos después de un fallo en el sistema, o simplemente en una posterior ejecución del cliente.

El mecanismo para proveer referencias persistentes y administrar la ejecución de las implantaciones de los objetos, es la activación automática de dichos objetos. En RMI es posible activar dinámicamente los objetos en el momento en que son necesitados, es decir, cuando se accede a un objeto remoto "activable" vía la ejecución de alguno de sus métodos, el sistema se encarga de ejecutar el objeto en una MV Java apropiada.

C.11 Terminología de objetos activables

Un objeto remoto activo, es aquel que es instanciado y exportado en una MV Java. Un objeto remoto pasivo, es aquel que no ha sido instanciado o exportado, pero es susceptible de pasar al estado activo. El proceso de llevar un objeto pasivo a activo se conoce como activación. La activación requiere la asociación de un objeto con una MV. Dicha asociación incluye la carga del código que implementa la clase en la MV, así como la restauración del estado persistente del objeto, si es que lo tuviese.

En el sistema RMI se utiliza la activación tardía (lazy activation), que consiste en retrasar la activación del objeto hasta que algún cliente lo utilice por primera vez, es decir, la primera vez que se invoque alguno de sus métodos.

C.12 Activación tardía (lazy activation)

La activación tardía de objetos remotos se implanta utilizando lo que se conoce como una "referencia remota responsable" (faulting remote reference), algunas veces denominada "bloque responsable". Una referencia remota responsable a un objeto remoto, "responde" por la referencia del objeto activo durante la primera invocación de algún método del

objeto. Cada referencia responsable mantiene tanto un manipulador persistente (un identificador de activación) y una referencia remota transitoria al objeto remoto destinatario. El identificador de activación del objeto remoto contiene información suficiente para iniciar la activación del objeto remoto y la referencia transitoria es la referencia "viva" real al objeto remoto activo, que puede utilizarse para comunicarse con él.

En una referencia responsable, si la referencia viva a un objeto remoto es nula, no se sabe si el objeto destinatario está activo. Durante la invocación del método, la referencia responsable se compromete en el protocolo de activación a obtener una referencia "viva", que es una referencia remota para el objeto recién activado. Una vez que la referencia responsable obtiene la referencia "viva", le reenvía la invocación del método para que esta última a su vez reenvíe la invocación al objeto remoto.

En términos más concretos, una referencia de objeto remoto contiene un tipo referencia remota responsable, que contiene dos cosas:

- Un identificador de activación para un objeto remoto.
- Una referencia "viva" (posiblemente nula) que contiene el tipo referencia remota activa del objeto remoto.

C.13 Protocolo de activación

Durante la invocación de un método remoto, sino se conoce la referencia "viva" para el objeto destino, la referencia responsable se compromete en el protocolo de activación. El protocolo de activación involucra muchas entidades: la referencia responsable, el activador, un grupo de activación en una MV Java y el objeto remoto que será activado.

El activador (generalmente uno por máquina) es la entidad que supervisa la activación, y actúa como:

- Una base de datos que contiene las asociaciones entre los identificadores de activación y la información necesaria para activar al objeto (la clase del objeto, su ubicación -un camino URL-, de donde se puede obtener la clase, información que el objeto podría necesitar al momento de arrancar, etc.).
- Un administrador de máquinas virtuales, que se encarga de ejecutar MVs (cuando sea necesario) y de encaminar solicitudes de activación de objetos (junto con la información necesaria) al grupo de activación correcto dentro una MV remota.

Un grupo de activación (uno por cada MV Java) es la entidad que recibe la solicitud de activación de un objeto en una MV y devuelve el objeto activado al activador.

El protocolo de activación es como sigue. Una referencia responsable utiliza un identificador de activación y llama al activador (mediante una interfaz RMI interna) para activar al objeto asociado con el identificador. El activador busca el descriptor de activación (activation descriptor) del objeto (previamente registrado). El descriptor del objeto contiene:

- Un identificador del grupo del objeto (especifica la MV en que será activado).
- El nombre de la clase del objeto.
- Un URL que contenga el camino de dónde obtener el código de la clase del objeto.
- Datos de inicialización aplanados específicos al objeto (por ejemplo el nombre del archivo que contiene el estado persistente del objeto).

Si el grupo de activación en el que debe de residir el objeto ya existe, el activador transfiere la solicitud de activación a dicho grupo. Sino existe, el activador inicia una nueva

MV y ejecuta un grupo de activación, para luego transferirle la solicitud de invocación. El grupo de activación carga las clases del objeto y lo inicializa haciendo uso de un constructor especial que utiliza varios parámetros, incluyendo el descriptor de activación que había sido registrado previamente. Cuando finalmente se activa al objeto, el grupo de activación devuelve al activador una referencia del objeto aplanado (marshalled object reference). El activador almacena el par identificador de activación, referencia activa, y devuelve una referencia activa (viva) a la referencia responsable. La referencia responsable (dentro de la referencia) encamina la invocación de los métodos, vía la referencia viva, directamente al objeto remoto.

En el JDK, RMI proporciona una implantación de las interfaces del sistema de activación por medio del demonio rmid, por lo que es necesario que se esté ejecutando para poder realizar una activación.

C.14 Proceso de desarrollo de una aplicación RMI

El proceso de desarrollo de una aplicación RMI, la podemos enumerar en 10 actividades, ver figura C.4, que son:

1. Extender de Java.rmi.Remote.
2. Implementar la interfaz, extendiendo de Java.rmi.UnicastRemoteObject (o llamando a exportObject ()).
3. Compilar la implementación con Javac.
4. Compilar la implementación con rmic.
5. Arrancar "rmiregistry".
6. Arrancar los objetos del servidor.
7. Registrar los objetos remotos, es decir, asociar un nombre con el objeto remoto, utilizando Java.rmi.Naming.
8. Escribir el código cliente utilizando Java.rmi.Naming para localizar el objeto remoto.
9. Compilar el código del cliente con Javac.
10. Arrancar el cliente.

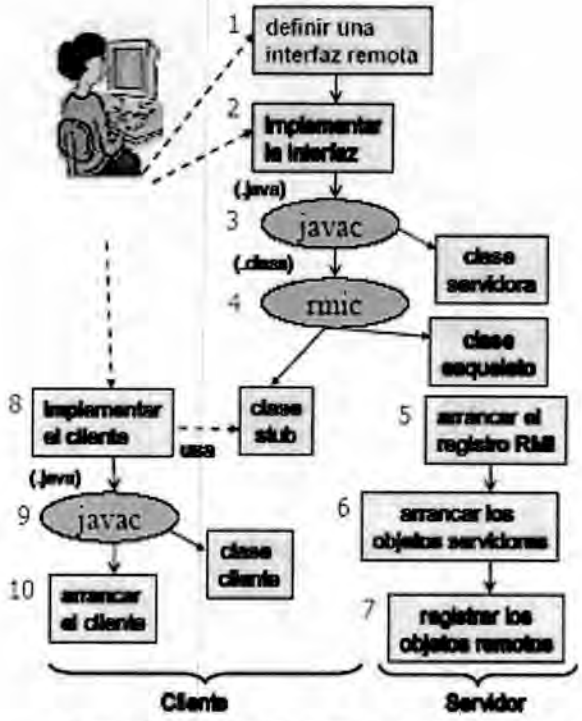

Figura C.4 Proceso de desarrollo de aplicación RMI

C.15 Conclusiones

La utilización de objetos en el desarrollo de sistemas distribuidos, presenta varias ventajas que permiten ocultar las dificultades inherentes a la distribución en niveles de abstracción inferiores, dejando así a los programadores la tarea de resolver únicamente su problema en particular.

La invocación remota de métodos en Java parte del hecho de correr sobre una plataforma homogénea. Está diseñada para tomar ventaja de esta característica, lo que le permite presentar propiedades que otros modelos de objetos como CORBA no poseen. Ejemplo de esto lo tenemos en la capacidad que tiene RMI de migrar dinámicamente a las implantaciones de los objetos, lo que le puede permitir a un cliente enviar un objeto para que se ejecute en una máquina con mayor poder de cómputo. Por otro lado, RMI posee todas las características de seguridad que hereda de la plataforma Java misma. Pero a diferencia de CORBA, que posee una arquitectura que proporciona independencia del lenguaje de programación, RMI está diseñada exclusivamente para Java.

Por último, podemos mencionar que Java RMI se utiliza como infraestructura fundamental de otras tecnologías Java sumamente importantes, como los Enterprise JavaBeans y JINI.

Apéndice D

D Excepciones RMI

	Excepciones	Paquetes			
		Java.rmi	Java.rmi.activation	Java.rmi.server	
1	AccessException	X			Una AccessException es lanzada por métodos de la clase Java.rmi.Naming (específicamente bind, rebind y unbind) y métodos de la interfaz Java.rmi.activation.ActivationSystem para indicar que la llamada no tiene permiso de ejecutar la acción solicitada (El cliente no tiene permiso para la acción solicitada). Si el método fue invocado por un host no-local, entonces se lanza un AccessException.
2	ActivateFailedException		X		Esta excepción es lanzada por el RMI (runtime) cuando la activación falla durante una llamada remota a un objeto activable.
3	ActivationException		X		Excepción general usada por la interfaz de activación. En el relase 1.4, esta excepción ha sido mejorada para conformar el propósito general del mecanismo de exception-chaining (encadenamiento de excepciones). El "detalle de la excepción" se provee en tiempo de construcción y es accedida vía el campo publico del detalle, es como se sabe la causa, y se accede vía el método Throwable.getCause ().
4	AlreadyBoundException	X			Se lanza una AlreadyBoundException si se hace un intento de ligar (bind) a un objeto en el registro (registry) a un nombre que ya tiene asociada un ligado (binding), es decir, el nombre que se intenta utilizar ya existe en el registro (bind ()).
5	ConnectException	X			Se lanza una ConnectException si una conexión es rechazada, por el host remoto, para una llamada de un método remoto (Error de conexión al host remoto).
6	ConnectIOException	X			Se lanza una ConnectIOException si una IOException ocurre mientras se realiza una conexión al host remoto por una llamada a un método remoto, es decir, error de tipo IOException al intentar conectar al host remoto.
7	ExportException			X	Una ExportException es una RemoteException lanzada si al intentar exportar un objeto remoto se produce una falla.
8	MarshalException	X			Se lanza una MarshalException si un Java.io.IOException ocurre mientras se realiza el marshalling del header, argumentos o valor de retorno de la llamada remota. Se lanza una MarshalException también si el receptor no soporta la versión del protocolo del remitente.
9	NoSuchObjectException	X			Se lanza una NoSuchObjectException si se intenta invocar un método en un objeto que no existe en la máquina remota virtual.
10	NotBoundException	X			Se lanza una NotBoundException si se intenta hacer un lookup o unbind en el registro (registry) a un nombre que no tiene liga (bind) asociada.

11	RemoteException	X	_	_	Una RemoteException es la superclase común para varias excepciones relacionadas a la comunicación que pueden ocurrir durante la ejecución de una llamada a un método remoto. Cada método de un interfaz remota, una interfaz que extiende Java.rmi.Remote, debe listar RemoteException en su cláusula de los throws.
12	RMISecurityException	X	_	_	Deprecated. Usar SecurityException en su lugar. El código de la aplicación nunca debe referenciar directamente esta clase, y RMISecurityManager no lanza esta subclass de java.lang.SecurityException. Una RMISecurityException señala que una excepción de seguridad ha ocurrido durante la ejecución de uno de los métodos de Java.rmi.RMISecurityManager.
13	ServerCloneException	_	_	X	Se lanza una ServerCloneException si una excepción remota ocurre durante el cloning de un UnicastRemoteObject.
14	ServerError	X	_	_	Una ServerError se lanza como resultado de una invocación de un método remoto cuando una RemoteException es lanzada mientras se procesa la invocación en el servidor, o mientras el unmarshalling de los argumentos, ejecución de un método remoto, o marshalling de un valor de retorno.
15	ServerException	X	_	_	Se lanza una ServerException como resultado de una invocación de un método remoto cuando una RemoteException es lanzada mientras se procesa la invocación en el servidor, o mientras el unmarshalling de los argumentos, ejecución de un método remoto, o marshalling del valor de retorno. Un ServerException contiene el RemoteException original que ocurrió.
16	ServerNotActiveException	_	_	X	Una ServerNotActiveException es una Excepción lanzada durante una llamada a RemoteServer.getClientHost si el método getClientHost se llamaba fuera del servicio de un método remoto.
17	ServerRuntimeException	X	_	_	Cuando un servidor procesa un programa en JDK 1.1, se lanza una ServerRuntimeException como resultado de una invocación de un método remoto cuando un RuntimeException de lanza mientras se procesa la invocación en el servidor, o mientras el unmarshalling de los argumentos, la ejecución de un método remoto, o el marshalling del valor de retorno. Una ServerRuntimeException contiene el RuntimeException original que ocurrió. No se lanza una ServerRuntimeException por servidores en plataformas con versión 1.2 o superiores.
18	SkeletonMismatchException	_	_	X	Se lanza esta excepción cuando se recibe una llamada que no coincide con el skeleton disponible. Indica que el nombre del método remoto o sus firmas en esa interfaz han cambiado o que la clase del stub usada para hacer la llamada y el skeleton receptor de la llamada fueron no generados por la misma versión del compilador stub (rmic).
19	SkeletonNotFoundException	_	_	X	
20	SocketSecutityException	_	_	X	Se lanza una SocketSecurityException durante la exportación de un objeto remoto si la codificación de exportación del objeto remoto (o por construcción o por llamada explícita al método del exportObject de UnicastRemoteObject o Java.rmi.activation.Activatable) no tiene permiso para crear un Java.net.ServerSocket en el número de puerto especificado durante la exportación del objeto remoto.
21	StubNotFoundException	X	_	_	Se lanza una StubNotFoundException sino se puede hallar una clase stub valida para el objeto remoto que se exporta. Se lanza una StubNotFoundException también cuando se registra un objeto activable vía el método Java.rmi.activation.Activatable.register; es decir, no se encuentra la clase stub.
22	UnexpectedException	X	_	_	Se lanza una UnexpectedException si el cliente de un método remoto recibe una llamada, como resultado de una llamada, una excepción "checked" (verificada) que no está entre los tipos de "checked exception" declarados en la cláusula de los throws en la interfaz remota del método.

23	UnknownGroupException	_	X	_	Una UnknownGroupException es lanzada por métodos de clases e interfaces en el paquete Java.rmi.activation cuando el parámetro ActivationGroupID para el método se determina como inválido, i.e., no conocido por el ActivationSystem. Se lanza una UnknownGroupException también si el ActivationGroupID en una referencia a ActivationDesc a un grupo que no esta registrado con el ActivationSystem.
24	UnknownHostException	X	_	_	Se lanza una UnknownHostException si un Java.net.UnknownHostException ocurre mientras se crea una conexión al host remoto por una llamada al método remoto (host desconocido).
25	UnknownObjectException	_	X	_	Una UnknownObjectException se lanza por métodos de clases e interfaces en el paquete Java.rmi.activation cuando el parámetro ActivationID se determina como inválido para método. Un ActivationID es inválido sino se conoce ciertamente por el ActivationSystem. Un ActivationID es obtenido por el método ActivationSystemregisterObject Se obtiene también durante la llamada Activatable.register
26	UnmarshalException	X	_	_	Se puede lanzar una UnmarshalException mientras el unmarshalling de los parámetros o de resultados de una llamada a método remoto, es decir, error en la fase de unmarshalling de parámetros o valores de retorno, si cualquiera de las condiciones siguientes ocurre: si una excepción ocurre mientras el unmarshalling del header de la llamada; si el protocolo por el valor del retorno es inválido; si un Java.io.IOException ocurre en el unmarshalling de los parámetros (en el lado del servidor) o el valor de retorno (en el lado del cliente); si un java.lang.ClassNotFoundException ocurre durante el unmarshalling de los parámetros o del valor del retorno; sino se puede cargar ningún skeleton en el lado del servidor (nota los skeleton son requeridos en el 1.1 protocolo del stub, pero no en el 1.2); si el método hash es invalido(i.e., método faltante); si hay fracaso al crear un objeto de la referencia remota por el stub de un objeto remoto cuando es unmarshalled.
27	SecutityException	X	_	_	Lanzado por el manejador de seguridad para indicar una violación de seguridad

Tabla D.1 Excepciones RMI

Apéndice E

E Listados

Por cuestión de espacio en la presente tesis, solo se presentan los listados a los archivos que se hace referencia en alguno de los capítulos o apéndices anteriores. El código completo se tiene dentro de la misma aplicación y en el CD que se anexa.

E.1 PreserveRemoteExcepcion.Java

```
import Java.rmi.*;
import Java.rmi.server.*;
import Java.rmi.activation.*;

public aspect PreserveRemoteException {
        declare precedence: PreserveRemoteException , ExHanAspect;

        after() throwing(ExHanAspRuntimeException ex) throws AccessException
        : call(* *.*(..) throws AccessException) {
                Throwable cause = ex.getCause();
                if (cause instanceof AccessException) {
                        System.out.println("**** Preserve " + cause);
                        throw (AccessException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws AlreadyBoundException
        : call(* *.*(..) throws AlreadyBoundException) {
                Throwable cause = ex.getCause();
                if (cause instanceof AlreadyBoundException) {
                        System.out.println("**** Preserve " + cause);
                        throw (AlreadyBoundException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws ActivateFailedException
        : call(* *.*(..) throws ActivateFailedException) {
                Throwable cause = ex.getCause();
                if (cause instanceof ActivateFailedException) {
                        System.out.println("**** Preserve " + cause);
                        throw (ActivateFailedException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws ActivationException
        : call(* *.*(..) throws ActivationException) {
                Throwable cause = ex.getCause();
                if (cause instanceof ActivationException) {
                        System.out.println("**** Preserve " + cause);
                        throw (ActivationException)cause;
                }
```

```
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws ConnectException
	: call(* *.*(..) throws ConnectException) {
		Throwable cause = ex.getCause();
		if (cause instanceof ConnectException) {
			System.out.println("**** Preserve " + cause);
			throw (ConnectException)cause;
		}
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws ConnectIOException
	: call(* *.*(..) throws ConnectIOException) {
		Throwable cause = ex.getCause();
		if (cause instanceof ConnectIOException) {
			System.out.println("**** Preserve " + cause);
			throw (ConnectIOException)cause;
		}
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws ExportException
	: call(* *.*(..) throws ExportException) {
		Throwable cause = ex.getCause();
		if (cause instanceof ExportException) {
			System.out.println("**** Preserve " + cause);
			throw (ExportException)cause;
		}
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws MarshalException
	: call(* *.*(..) throws MarshalException) {
		Throwable cause = ex.getCause();
		if (cause instanceof MarshalException) {
			System.out.println("**** Preserve " + cause);
			throw (MarshalException)cause;
		}
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws NoSuchObjectException
	: call(* *.*(..) throws NoSuchObjectException) {
		Throwable cause = ex.getCause();
		if (cause instanceof NoSuchObjectException) {
			System.out.println("**** Preserve " + cause);
			throw (NoSuchObjectException)cause;
		}
		throw ex;
	}

	after() throwing(ExHanAspRuntimeException ex) throws NotBoundException
	: call(* *.*(..) throws NotBoundException) {
		Throwable cause = ex.getCause();
		if (cause instanceof NotBoundException) {
			System.out.println("**** Preserve " + cause);
			throw (NotBoundException)cause;
		}
		throw ex;
	}
```

```
after() throwing(ExHanAspRuntimeException ex) throws RemoteException
: call(* *.*(..) throws RemoteException) {
        Throwable cause = ex.getCause();
        if (cause instanceof RemoteException) {
                System.out.println("**** Preserve " + cause);
                throw (RemoteException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws RMISecurityException
: call(* *.*(..) throws RMISecurityException) {
        Throwable cause = ex.getCause();
        if (cause instanceof RMISecurityException) {
                System.out.println("**** Preserve " + cause);
                throw (RMISecurityException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws ServerCloneException
: call(* *.*(..) throws ServerCloneException) {
        Throwable cause = ex.getCause();
        if (cause instanceof ServerCloneException) {
                System.out.println("**** Preserve " + cause);
                throw (ServerCloneException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws ServerNotActiveException
: call(* *.*(..) throws ServerNotActiveException) {
        Throwable cause = ex.getCause();
        if (cause instanceof ServerNotActiveException) {
                System.out.println("**** Preserve " + cause);
                throw (ServerNotActiveException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws ServerError
: call(* *.*(..) throws ServerError) {
        Throwable cause = ex.getCause();
        if (cause instanceof ServerError) {
                System.out.println("**** Preserve " + cause);
                throw (ServerError)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws ServerException
: call(* *.*(..) throws ServerException) {
        Throwable cause = ex.getCause();
        if (cause instanceof ServerException) {
                System.out.println("**** Preserve " + cause);
                throw (ServerException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws ServerRuntimeException
```

```
: call(* *.*(..) throws ServerRuntimeException) {
        Throwable cause = ex.getCause();
        if (cause instanceof ServerRuntimeException) {
                System.out.println("**** Preserve " + cause);
                throw (ServerRuntimeException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws SkeletonMismatchException
: call(* *.*(..) throws SkeletonMismatchException) {
        Throwable cause = ex.getCause();
        if (cause instanceof SkeletonMismatchException) {
                System.out.println("**** Preserve " + cause);
                throw (SkeletonMismatchException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws SkeletonNotFoundException
: call(* *.*(..) throws SkeletonNotFoundException) {
        Throwable cause = ex.getCause();
        if (cause instanceof SkeletonNotFoundException) {
                System.out.println("**** Preserve " + cause);
                throw (SkeletonNotFoundException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws SocketSecurityException
: call(* *.*(..) throws SocketSecurityException) {
        Throwable cause = ex.getCause();
        if (cause instanceof SocketSecurityException) {
                System.out.println("**** Preserve " + cause);
                throw (SocketSecurityException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws StubNotFoundException
: call(* *.*(..) throws StubNotFoundException) {
        Throwable cause = ex.getCause();
        if (cause instanceof StubNotFoundException) {
                System.out.println("**** Preserve " + cause);
                throw (StubNotFoundException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws UnexpectedException
: call(* *.*(..) throws UnexpectedException) {
        Throwable cause = ex.getCause();
        if (cause instanceof UnexpectedException) {
                System.out.println("**** Preserve " + cause);
                throw (UnexpectedException)cause;
        }
        throw ex;
}

after() throwing(ExHanAspRuntimeException ex) throws UnknownGroupException
: call(* *.*(..) throws UnknownGroupException) {
        Throwable cause = ex.getCause();
```

```
                if (cause instanceof UnknownGroupException) {
                        System.out.println("**** Preserve " + cause);
                        throw (UnknownGroupException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws UnknownObjectException
        : call(* *.*(..) throws UnknownObjectException) {
                Throwable cause = ex.getCause();
                if (cause instanceof UnknownObjectException) {
                        System.out.println("**** Preserve " + cause);
                        throw (UnknownObjectException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws UnknownHostException
        : call(* *.*(..) throws UnknownHostException) {
                Throwable cause = ex.getCause();
                if (cause instanceof UnknownHostException) {
                        System.out.println("**** Preserve " + cause);
                        throw (UnknownHostException)cause;
                }
                throw ex;
        }

        after() throwing(ExHanAspRuntimeException ex) throws UnmarshalException
        : call(* *.*(..) throws UnmarshalException) {
                Throwable cause = ex.getCause();
                if (cause instanceof UnmarshalException) {
                        System.out.println("**** Preserve " + cause);
                        throw (UnmarshalException)cause;
                }
                throw ex;
        }

} // aspect
```

E.2 ExHanAspAbstract.Java

```
import Java.rmi.*;
import Java.rmi.RemoteException;
import Java.rmi.activation.*;
import Java.rmi.server.*;

public abstract aspect ExHanAspAbstract {

        Throwable ex_final=null;
        int retry = 0;

        // *** pointcut
        // *** datos nombre=operations descripcion=primera prueba.
        abstract pointcut operations();

        Object around() throws Java.rmi.RemoteException: operations() {

                try {
                        return proceed();
                } catch (Throwable ex_ori) {
```

```
System.out.println("ExHanAspAbstract operations cacho una Throwable ex=" + ex_ori);
Throwable ex= this.QuitaWrapper(ex_ori);

// aqui el manejo de ese punto de corte

// *** excepcion
// *** referencia=ref_1_a_AlreadyBoundException
// *** nombre=Java.rmi.AlreadyBoundException
// *** descripcion=descripcion de AlreadyBoundException.
// *** if (1)
if (ex.getClass().equals(Java.rmi.AlreadyBoundException.class)) {
        System.out.println("if (1)");
        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
        // *** encapsular="ExHanAspRuntimeException
        throw new ExHanAspRuntimeException(ex);

} // if Java.rmi.AlreadyBoundException

// *** excepcion
// *** referencia=ref_2_a_ActivationException
// *** nombre=Java.rmi.activation.ActivationException
// *** descripcion=descripcion de ActivationException.
// *** if (2)
else if (ex.getClass().equals(Java.rmi.activation.ActivationException.class)) {
        System.out.println("if (2)");
        // *** para reintentar
        retry = 0;
        ex_final=null;

        while (true) {
                // *** intenta 3 veces mas
                if (++retry > 3)
                        break;
                // *** delay=1000
                this.espera(1000);
                try{
                        ex_final=null;
                        System.out.println("en around a proceed es intento "+ retry);
                        return proceed();
                } catch(Throwable ex_2){
                        System.out.println("cacho Throwable en intento:"+ retry);
                        ex_final=ex_2;
                } // catch
        } // while

        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                // *** se genero otro tipo de excepción
                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                throw new ExHanAspRuntimeException(ex_final);
        } //if

        // *** lanzar Java.rmi.RemoteException
        System.out.println("lanza Java.rmi.RemoteException");
        throw new Java.rmi.RemoteException();

} // if Java.rmi.activation.ActivationException

// *** grupo
// *** referencia=ref_gpo_1
```

```
                        // *** descripcion=descripcion grupo uno
                        // *** excepciones
                        // *** datos nombre=Java.rmi.activation.UnknownGroupException descripcion=descripcion
de Java.rmi.activation.UnknownGroupException
                        // *** datos nombre=Java.rmi.activation.UnknownObjectException descripcion=descrip
Java.rmi.activation.UnknownObjectException
                        // *** if (3)
                        else if ( (ex.getClass().equals(Java.rmi.activation.UnknownGroupException.class)) ||
                                (ex.getClass().equals(Java.rmi.activation.UnknownObjectException.class))  ) {
                                System.out.println("if (3)");
                                // *** para reintentar
                                retry = 0;
                                ex_final=null;

                                while (true) {
                                        // *** intenta 1 veces mas
                                        if (++retry > 1)
                                                break;
                                        try{
                                                ex_final=null;
                                                System.out.println("en around a proceed es intento "+ retry);
                                                return proceed();
                                        } catch(Throwable ex_2){
                                                System.out.println("cacho Throwable en intento:"+ retry);
                                                ex_final=ex_2;
                                        } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(ex);

                        } // if ref_gpo_1

                        // *** grupo
                        // *** referencia=ref_gpo_2
                        // *** descripcion=descripcion grupo dos
                        // *** excepciones
                        // *** datos nombre=Java.rmi.activation.ActivateFailedException descripcion=This
exception is thrown by the RMI runtime when activation fails during a remote call to an activatable object
                        // *** datos nombre=Java.rmi.activation.ActivationException descripcion=General
exception used by the activation interfaces.  As of release 1.4, this exception has been retrofitted to conform to the
general purpose exception-chaining mechanism.
                        // *** if (4)
                        else if ( (ex.getClass().equals(Java.rmi.activation.ActivateFailedException.class)) ||
                                (ex.getClass().equals(Java.rmi.activation.ActivationException.class))  ) {
                                System.out.println("if (4)");
                                System.out.println("lanza Java.rmi.RemoteException");
                                // *** lanzar=Java.rmi.RemoteException
                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                        } // if ref_gpo_2
```

```
// *** paquete
// *** referencia=ref_1_paquete_server
// *** datos nombre=Java.rmi.server descripcion=descripcion paquete paq server
// *** if (5)
else if ((ex.getClass().getName()).contains("Java.rmi.server")) {
        System.out.println("if (5)");
        System.out.println("paquete Java.rmi.server");
        // *** para reintentar
        retry = 0;
        ex_final=null;

        while (true) {
                // *** intenta 3 veces mas
                if (++retry > 3)
                        break;
                // *** delay=1000
                this.espera(1000);
                try{
                        ex_final=null;
                        System.out.println("en around a proceed es intento "+ retry);
                        return proceed();
                } catch(Throwable ex_2){
                        System.out.println("cacho Throwable en intento:"+ retry);
                        ex_final=ex_2;
                } // catch
        } // while

        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                // *** se genero otro tipo de excepción
                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                throw new ExHanAspRuntimeException(ex_final);
        } //if

        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
        // *** encapsular="ExHanAspRuntimeException
        throw new ExHanAspRuntimeException(ex);

} // if ref_1_paquete_server

// *** default
// *** if (default)
else {
        System.out.println("if (default)");
        // *** para reintentar
        retry = 0;
        ex_final=null;

        while (true) {
                // *** intenta 6 veces mas
                if (++retry > 6)
                        break;
                // *** delay=100
                this.espera(100);
                try{
                        ex_final=null;
                        System.out.println("en around a proceed es intento "+ retry);
                        return proceed();
                } catch(Throwable ex_2){
                        System.out.println("cacho Throwable en intento:"+ retry);
```

```
                                        ex_final=ex_2;
                                    } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                    // *** se genero otro tipo de excepción
                                    System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                    throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                // *** ejecutar
                                System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
                                // *** parámetros
                                // *** parámetro
                                Integer para1= new Integer(2);
                                // *** parametro
                                String para2= new String("01234567");
                                // *** parámetro
                                boolean para3= false;

                                ex_final=null;

                                try {
                                    // *** retorno
                                    int a;
                                    // *** ejecuto
                                    a= MiClaseRemota2.metodo_paquete(para1,para2,para3);
                                    System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_paquete(para1,para2,para3);");
                                } catch(Throwable t){
                                    System.out.println("cacho Throwable del metodo ");
                                    ex_final=t;
                                }

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                    // *** se genero otro tipo de excepción
                                    System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                    throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                System.out.println("lanza Java.rmi.RemoteException");
                                // *** lanzar=Java.rmi.RemoteException
                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                            } // else default
                        } // catch
                    } // around operations


                    // *** pointcut
                    // *** datos nombre=operations_2 descripcion=segunda prueba.
                    abstract pointcut operations_2();

                    Object around() throws Java.rmi.RemoteException: operations_2() {

                        try {
```

```
                return proceed();
            } catch (Throwable ex_ori) {

                System.out.println("ExHanAspAbstract operations_2 cacho una Throwable ex=" + ex_ori);
                Throwable ex= this.QuitaWrapper(ex_ori);

                // aqui el manejo de ese punto de corte

                // *** excepción
                //  ***  datos  nombre=Java.rmi.NotBoundException  descripcion=descripcion  de
NotBoundException.
                // *** if (1)
                if (ex.getClass().equals(Java.rmi.NotBoundException.class)) {
                        System.out.println("if (1)");
                        // *** ejecutar
                        System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
                        // *** parámetros
                        // *** parámetro
                        Integer para1= new Integer(10);
                        // *** parametro
                        String para2= new String("si");

                        ex_final=null;

                        try {
                                // *** retorno
                                // *** void
                                // *** ejecuto
                                MiClaseRemota2.metodo_paquete(para1,para2);
                                System.out.println("ejecuto  método,  "  +
"MiClaseRemota2.metodo_paquete(para1,para2);");
                        } catch(Throwable t){
                                System.out.println("cacho Throwable del metodo ");
                                ex_final=t;
                        }

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                // *** se genero otro tipo de excepción
                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                throw new ExHanAspRuntimeException(ex_final);
                        } //if

                        System.out.println("lanza original encapsulada");
                        throw new ExHanAspRuntimeException(ex);

                } // if Java.rmi.NotBoundException

                // *** excepcion
                //  ***  datos  nombre=Java.rmi.RemoteException  descripcion=descripcion  de
RemoteException.
                // *** if (2)
                else if (ex.getClass().equals(Java.rmi.RemoteException.class)) {
                        System.out.println("if (2)");
                        // *** para reintentar
                        retry = 0;
                        ex_final=null;

                        while (true) {
                                // *** intenta 1 veces mas
                                if (++retry > 1)
```

```
                                        break;
                                    // *** delay=500
                                    this.espera(500);
                                    try{
                                        ex_final=null;
                                        System.out.println("en around a proceed es intento "+ retry);
                                        return proceed();
                                    } catch(Throwable ex_2){
                                        System.out.println("cacho Throwable en intento:"+ retry);
                                        ex_final=ex_2;
                                    } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass())) ) ){
                                    // *** se genero otro tipo de excepción
                                    System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                    throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(ex);

                            } // if Java.rmi.RemoteException

                            // *** excepcion
                            // *** referencia=ref_1_a_AlreadyBoundException
                            // *** nombre=Java.rmi.AlreadyBoundException
                            // *** descripcion=descripcion de AlreadyBoundException.
                            // *** if (3)
                            else if (ex.getClass().equals(Java.rmi.AlreadyBoundException.class)) {
                                System.out.println("if (3)");
                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(ex);

                            } // if Java.rmi.AlreadyBoundException

                            // *** excepcion
                            // *** referencia=ref_2_b_a_ActivationException
                            // *** nombre=Java.rmi.activation.ActivationException
                            // *** descripcion=descripcion de ActivationException dos.
                            // *** if (4)
                            else if (ex.getClass().equals(Java.rmi.activation.ActivationException.class)) {
                                System.out.println("if (4)");
                                // *** para reintentar
                                retry = 0;
                                ex_final=null;

                                while (true) {
                                    // *** intenta 1 veces mas
                                    if (++retry > 1)
                                        break;
                                    // *** delay=500
                                    this.espera(500);
                                    try{
                                        ex_final=null;
                                        System.out.println("en around a proceed es intento "+ retry);
                                        return proceed();
                                    } catch(Throwable ex_2){
```

```
                                        System.out.println("cacho Throwable en intento:"+ retry);
                                        ex_final=ex_2;
                                } // catch
                        } // while

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                // *** se genero otro tipo de excepción
                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                throw new ExHanAspRuntimeException(ex_final);
                        } //if

                        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                        // *** encapsular="ExHanAspRuntimeException
                        throw new ExHanAspRuntimeException(ex);

                } // if Java.rmi.activation.ActivationException

                // *** grupo
                // *** referencia=ref_gpo_3
                // *** descripcion=descripcion grupo tres
                // *** excepciones
                //    ***    datos    nombre=Java.rmi.MarshalException    descripcion=descripcion    de
Java.rmi.MarshalException
                //    ***    datos    nombre=Java.rmi.NoSuchObjectException    descripcion=descrip
Java.rmi.NoSuchObjectException
                // *** if (5)
                else if ( (ex.getClass().equals(Java.rmi.MarshalException.class)) ||
                        (ex.getClass().equals(Java.rmi.NoSuchObjectException.class))  ) {
                        System.out.println("if (5)");
                        // *** para reintentar
                        retry = 0;
                        ex_final=null;

                        while (true) {
                                // *** intenta 2 veces mas
                                if (++retry > 2)
                                        break;
                                // *** delay=2500
                                this.espera(2500);
                                try{
                                        ex_final=null;
                                        System.out.println("en around a proceed es intento "+ retry);
                                        return proceed();
                                } catch(Throwable ex_2){
                                        System.out.println("cacho Throwable en intento:"+ retry);
                                        ex_final=ex_2;
                                } // catch
                        } // while

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                // *** se genero otro tipo de excepción
                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                throw new ExHanAspRuntimeException(ex_final);
                        } //if

                        // *** ejecutar
                        System.out.println("ejecutar MiClaseRemota2.metodo_grupo");
                        // *** parámetros
```

```
                                        // *** parámetro
                                        Integer para1= new Integer(40);
                                        // *** parametro
                                        String para2= new String("no");

                                        ex_final=null;

                                        try {
                                                // *** retorno
                                                Integer a=null;
                                                // *** ejecuto
                                                a= MiClaseRemota2.metodo_grupo(para1,para2);
                                                System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_grupo(para1,para2);");
                                        } catch(Throwable t){
                                                System.out.println("cacho Throwable del metodo ");
                                                ex_final=t;
                                        }

                                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                                // *** se genero otro tipo de excepción
                                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                                throw new ExHanAspRuntimeException(ex_final);
                                        } //if

                                        System.out.println("lanza original encapsulada");
                                        throw new ExHanAspRuntimeException(ex);

                                } // if ref_gpo_3

                                // *** grupo
                                // *** referencia=ref_gpo_2
                                // *** descripcion=descripcion grupo dos
                                // *** excepciones
                                // *** datos nombre=Java.rmi.activation.ActivateFailedException descripcion=This
exception is thrown by the RMI runtime when activation fails during a remote call to an activatable object
                                // *** datos nombre=Java.rmi.activation.ActivationException descripcion=General
exception used by the activation interfaces. As of release 1.4, this exception has been retrofitted to conform to the
general purpose exception-chaining mechanism.
                                // *** if (6)
                                else if ( (ex.getClass().equals(Java.rmi.activation.ActivateFailedException.class)) ||
                                        (ex.getClass().equals(Java.rmi.activation.ActivationException.class)) ) {
                                        System.out.println("if (6)");
                                        System.out.println("lanza Java.rmi.RemoteException");
                                        // *** lanzar=Java.rmi.RemoteException
                                        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                        // *** encapsular="ExHanAspRuntimeException
                                        throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                                } // if ref_gpo_2

                                // *** paquete
                                // *** referencia=ref_2_paquete_activation
                                // *** datos nombre=Java.rmi.activation descripcion=descripcion paquete paq activation
                                // *** if (7)
                                else if ((ex.getClass().getName()).contains("Java.rmi.activation")) {
                                        System.out.println("if (7)");
                                        System.out.println("paquete Java.rmi.activation");
                                        // *** ejecutar
                                        System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
```

```
                        // *** parámetros
                        // *** parámetro
                        Integer para1= new Integer(2);
                        // *** parámetro
                        char para2= 'a';

                        ex_final=null;

                        try {
                                // *** retorno
                                String a=null;
                                // *** ejecuto
                                a= MiClaseRemota2.metodo_paquete(para1,para2);
                                System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_paquete(para1,para2);");
                        } catch(Throwable t){
                                System.out.println("cacho Throwable del metodo ");
                                ex_final=t;
                        }

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                // *** se genero otro tipo de excepción
                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                throw new ExHanAspRuntimeException(ex_final);
                        } //if

                        System.out.println("lanza original encapsulada");
                        throw new ExHanAspRuntimeException(ex);

                } // if ref_2_paquete_activation

                // *** paquete
                // *** referencia=ref_1_b_paquete_server
                // *** datos nombre=Java.rmi.server descripcion=descripcion paquete paq server
                // *** if (8)
                else if ((ex.getClass().getName()).contains("Java.rmi.server")) {
                        System.out.println("if (8)");
                        System.out.println("paquete Java.rmi.server");
                        // *** para reintentar
                        retry = 0;
                        ex_final=null;

                        while (true) {
                                // *** intenta 2 veces mas
                                if (++retry > 2)
                                        break;
                                // *** delay=5000
                                this.espera(5000);
                                try{
                                        ex_final=null;
                                        System.out.println("en around a proceed es intento "+ retry);
                                        return proceed();
                                } catch(Throwable ex_2){
                                        System.out.println("cacho Throwable en intento:"+ retry);
                                        ex_final=ex_2;
                                } // catch
                        } // while

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
```

```
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                // *** lanzar Java.rmi.RemoteException
                                System.out.println("lanza Java.rmi.RemoteException");
                                throw new Java.rmi.RemoteException();

                        } // if ref_1_b_paquete_server

                        // *** default
                        // *** if (default)
                        else {
                                System.out.println("if (default)");
                                // *** para reintentar
                                retry = 0;
                                ex_final=null;

                                while (true) {
                                        // *** intenta 6 veces mas
                                        if (++retry > 6)
                                                break;
                                        // *** delay=100
                                        this.espera(100);
                                        try{
                                                ex_final=null;
                                                System.out.println("en around a proceed es intento "+ retry);
                                                return proceed();
                                        } catch(Throwable ex_2){
                                                System.out.println("cacho Throwable en intento:"+ retry);
                                                ex_final=ex_2;
                                        } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                // *** ejecutar
                                System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
                                // *** parámetros
                                // *** parámetro
                                Integer para1= new Integer(2);
                                // *** parametro
                                String para2= new String("01234567");
                                // *** parámetro
                                boolean para3= false;

                                ex_final=null;

                                try {
                                        // *** retorno
                                        int a;
                                        // *** ejecuto
                                        a= MiClaseRemota2.metodo_paquete(para1,para2,para3);
```

```
                                        System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_paquete(para1,para2,para3);");
                                } catch(Throwable t){
                                        System.out.println("cacho Throwable del metodo ");
                                        ex_final=t;
                                }

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                System.out.println("lanza Java.rmi.RemoteException");
                                // *** lanzar=Java.rmi.RemoteException
                                System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                // *** encapsular="ExHanAspRuntimeException
                                throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                        } // else default
                } // catch
        } // around operations_2


        // *** pointcut
        // *** datos nombre=operations_3 descripcion=tercera prueba.
        abstract pointcut operations_3();

        Object around() throws Java.rmi.RemoteException: operations_3() {

                try {
                        return proceed();
                } catch (Throwable ex_ori) {

                        System.out.println("ExHanAspAbstract operations_3 cacho una Throwable ex=" + ex_ori);
                        Throwable ex= this.QuitaWrapper(ex_ori);

                        // aqui el manejo de ese punto de corte

                        // *** excepción
                        // *** datos nombre=Java.rmi.NotBoundException descripcion=descripcion de
NotBoundException.
                        // *** if (1)
                        if (ex.getClass().equals(Java.rmi.NotBoundException.class)) {
                                System.out.println("if (1)");
                                // *** ejecutar
                                System.out.println("ejecutar MiClaseRemota2.metodo_ex2");
                                // *** parámetros
                                // *** parámetro
                                String para1= new String("pba");

                                ex_final=null;

                                try {
                                        // *** retorno
                                        RemoteException a=null;
                                        // *** ejecuto
                                        a= MiClaseRemota2.metodo_ex2(para1);
                                        System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_ex2(para1);");
```

```
                        } catch(Throwable t){
                                System.out.println("cacho Throwable del metodo ");
                                ex_final=t;
                        }

                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                // *** se genero otro tipo de excepción
                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                throw new ExHanAspRuntimeException(ex_final);
                        } //if

                        System.out.println("lanza original encapsulada");
                        throw new ExHanAspRuntimeException(ex);

                } // if Java.rmi.NotBoundException

                // *** excepcion
                //  ***  datos  nombre=Java.rmi.RemoteException  descripcion=descripcion  de
RemoteException.
                // *** if (2)
                else if (ex.getClass().equals(Java.rmi.RemoteException.class)) {
                        System.out.println("if (2)");
                        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                        // *** encapsular="ExHanAspRuntimeException
                        throw new ExHanAspRuntimeException(ex);

                } // if Java.rmi.RemoteException

                // *** grupo
                // *** referencia=ref_gpo_2
                // *** descripcion=descripcion grupo dos
                // *** excepciones
                //  ***  datos  nombre=Java.rmi.activation.ActivateFailedException  descripcion=This
exception is thrown by the RMI runtime when activation fails during a remote call to an activatable object
                //  ***  datos  nombre=Java.rmi.activation.ActivationException  descripcion=General
exception used by the activation interfaces.  As of release 1.4, this exception has been retrofitted to conform to the
general purpose exception-chaining mechanism.
                // *** if (3)
                else if ( (ex.getClass().equals(Java.rmi.activation.ActivateFailedException.class)) ||
                        (ex.getClass().equals(Java.rmi.activation.ActivationException.class))  ) {
                        System.out.println("if (3)");
                        System.out.println("lanza Java.rmi.RemoteException");
                        // *** lanzar=Java.rmi.RemoteException
                        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                        // *** encapsular="ExHanAspRuntimeException
                        throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                } // if ref_gpo_2

                // *** paquete
                // *** referencia=ref_2_paquete_activation
                // *** datos nombre=Java.rmi.activation descripcion=descripcion paquete paq activation
                // *** if (4)
                else if ((ex.getClass().getName()).contains("Java.rmi.activation")) {
                        System.out.println("if (4)");
                        System.out.println("paquete Java.rmi.activation");
                        // *** ejecutar
                        System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
                        // *** parámetros
                        // *** parámetro
```

```
                                Integer para1= new Integer(2);
                                // *** parámetro
                                char para2= 'a';

                                ex_final=null;

                                try {
                                        // *** retorno
                                        String a=null;
                                        // *** ejecuto
                                        a= MiClaseRemota2.metodo_paquete(para1,para2);
                                        System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_paquete(para1,para2);");
                                } catch(Throwable t){
                                        System.out.println("cacho Throwable del metodo ");
                                        ex_final=t;
                                }

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                System.out.println("lanza original encapsulada");
                                throw new ExHanAspRuntimeException(ex);

                        } // if ref_2_paquete_activation

                        // *** paquete
                        // *** referencia=ref_1_b_paquete_server
                        // *** datos nombre=Java.rmi.server descripcion=descripcion paquete paq server
                        // *** if (5)
                        else if ((ex.getClass().getName()).contains("Java.rmi.server")) {
                                System.out.println("if (5)");
                                System.out.println("paquete Java.rmi.server");
                                // *** para reintentar
                                retry = 0;
                                ex_final=null;

                                while (true) {
                                        // *** intenta 2 veces mas
                                        if (++retry > 2)
                                                break;
                                        // *** delay=5000
                                        this.espera(5000);
                                        try{
                                                ex_final=null;
                                                System.out.println("en around a proceed es intento "+ retry);
                                                return proceed();
                                        } catch(Throwable ex_2){
                                                System.out.println("cacho Throwable en intento:"+ retry);
                                                ex_final=ex_2;
                                        } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
```

```
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                // *** lanzar Java.rmi.RemoteException
                                System.out.println("lanza Java.rmi.RemoteException");
                                throw new Java.rmi.RemoteException();

                        } // if ref_1_b_paquete_server

                        // *** default
                        // *** if (default)
                        else {
                                System.out.println("if (default)");
                                // *** para reintentar
                                retry = 0;
                                ex_final=null;

                                while (true) {
                                        // *** intenta 6 veces mas
                                        if (++retry > 6)
                                                break;
                                        // *** delay=100
                                        this.espera(100);
                                        try{
                                                ex_final=null;
                                                System.out.println("en around a proceed es intento "+ retry);
                                                return proceed();
                                        } catch(Throwable ex_2){
                                                System.out.println("cacho Throwable en intento:"+ retry);
                                                ex_final=ex_2;
                                        } // catch
                                } // while

                                if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                        // *** se genero otro tipo de excepción
                                        System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                        throw new ExHanAspRuntimeException(ex_final);
                                } //if

                                // *** ejecutar
                                System.out.println("ejecutar MiClaseRemota2.metodo_paquete");
                                // *** parámetros
                                // *** parámetro
                                Integer para1= new Integer(2);
                                // *** parametro
                                String para2= new String("01234567");
                                // *** parámetro
                                boolean para3= false;

                                ex_final=null;

                                try {
                                        // *** retorno
                                        int a;
                                        // *** ejecuto
                                        a= MiClaseRemota2.metodo_paquete(para1,para2,para3);
                                        System.out.println("ejecuto método, " + "a=
MiClaseRemota2.metodo_paquete(para1,para2,para3);");
```

```
                                        } catch(Throwable t){
                                                System.out.println("cacho Throwable del metodo ");
                                                ex_final=t;
                                        }

                                        if ((ex_final != null) && !(ex_final.getClass().equals(ex.getClass()) ) &&
!(ex_final.getCause().getClass().equals(ex.getClass()) ) ){
                                                // *** se genero otro tipo de excepción
                                                System.out.println("OJO se genero otro tipo de excepción, ori: " +
ex.getClass() + " generada: " + ex_final.getClass() + " la lanza en wrapper");
                                                throw new ExHanAspRuntimeException(ex_final);
                                        } //if

                                        System.out.println("lanza Java.rmi.RemoteException");
                                        // *** lanzar=Java.rmi.RemoteException
                                        System.out.println("ENCAPSULA ExHanAspRuntimeException WRAPPER");
                                        // *** encapsular="ExHanAspRuntimeException
                                        throw new ExHanAspRuntimeException(new Java.rmi.RemoteException());

                                } // else default
                        } // catch
                } // around operations_3


                private void espera (int delay) {

                        try{
                                System.out.println("espera "+ delay);
                                Thread.sleep(delay);
                        } catch(InterruptedException ex_i){
                                System.out.println("espera, cacho interrupted");
                        } // catch
                        return;
                } // espera


                private Throwable QuitaWrapper(Throwable ex_ori) {

                        Throwable ex= null;
                        Throwable cause = ex_ori.getCause();
                        System.out.println("ExHanAspAbstract nvo la causa " + cause);

                        if (cause instanceof AccessException) {
                                System.out.println("ExHanAspAbstract nvo lanza AccessException");
                                ex = (AccessException)cause;
                        } else if (cause instanceof AlreadyBoundException) {
                                System.out.println("ExHanAspAbstract nvo lanza AlreadyBoundException");
                                ex = (AlreadyBoundException)cause;
                        } else if (cause instanceof ActivateFailedException) {
                                System.out.println("ExHanAspAbstract nvo lanza ActivateFailedException");
                                ex = (ActivateFailedException)cause;
                        } else if (cause instanceof ActivationException) {
                                System.out.println("ExHanAspAbstract nvo lanza AtivationException ");
                                ex=(ActivationException)cause;
                        } else if (cause instanceof ConnectException) {
                                System.out.println("ExHanAspAbstract nvo lanza ConnectException ");
                                ex=(ConnectException)cause;
                        } else if (cause instanceof ConnectIOException) {
                                System.out.println("ExHanAspAbstract nvo lanza ConnectIOException ");
                                ex=(ConnectIOException)cause;
                        } else if (cause instanceof ExportException) {
                                System.out.println("ExHanAspAbstract nvo lanza ExportException ");
```

```
            ex=(ExportException)cause;
        } else if (cause instanceof MarshalException) {
            System.out.println("ExHanAspAbstract nvo lanza MarshalException ");
            ex=(MarshalException)cause;
        } else if (cause instanceof NoSuchObjectException) {
            System.out.println("ExHanAspAbstract nvo lanza NoSuchObjectException ");
            ex=(NoSuchObjectException)cause;
        } else if (cause instanceof NotBoundException) {
            System.out.println("ExHanAspAbstract nvo lanza NotBoundException ");
            ex=(NotBoundException)cause;
        } else  if (cause instanceof RemoteException) {
            System.out.println("ExHanAspAbstract nvo lanza Remote");
            ex = (RemoteException)cause;
        } else if (cause instanceof RMISecurityException) {
            System.out.println("ExHanAspAbstract nvo lanza RMISecurityException ");
            ex=(RMISecurityException)cause;
        } else if (cause instanceof ServerCloneException) {
            System.out.println("ExHanAspAbstract nvo lanza ServerCloneException ");
            ex=(ServerCloneException)cause;
        } else if (cause instanceof ServerError) {
            System.out.println("ExHanAspAbstract nvo lanza ServerError ");
            ex=(ServerError)cause;
        } else if (cause instanceof ServerNotActiveException) {
            System.out.println("ExHanAspAbstract nvo lanza ServerNotActiveException ");
            ex=(ServerNotActiveException)cause;
        } else if (cause instanceof ServerException) {
            System.out.println("ExHanAspAbstract nvo lanza ServerException ");
            ex=(ServerException)cause;
        } else if (cause instanceof ServerRuntimeException) {
            System.out.println("ExHanAspAbstract nvo lanza ServerRuntimeException ");
            ex=(ServerRuntimeException)cause;
        } else if (cause instanceof SkeletonMismatchException) {
            System.out.println("ExHanAspAbstract nvo lanza SkeletonMismatchException ");
            ex=(SkeletonMismatchException)cause;
        } else if (cause instanceof SkeletonNotFoundException) {
            System.out.println("ExHanAspAbstract nvo lanza SkeletonNotFoundException ");
            ex=(SkeletonNotFoundException)cause;
        } else if (cause instanceof SocketSecurityException) {
            System.out.println("ExHanAspAbstract nvo lanza SocketSecurityException ");
            ex=(SocketSecurityException)cause;
        } else if (cause instanceof StubNotFoundException) {
            System.out.println("ExHanAspAbstract nvo lanza StubNotFoundException ");
            ex=(StubNotFoundException)cause;
        } else if (cause instanceof UnexpectedException) {
            System.out.println("ExHanAspAbstract nvo lanza UnexpectedException ");
            ex=(UnexpectedException)cause;
        } else if (cause instanceof UnknownGroupException) {
            System.out.println("ExHanAspAbstract nvo lanza UnknownGroupException ");
            ex=(UnknownGroupException)cause;
        } else if (cause instanceof UnknownObjectException) {
            System.out.println("ExHanAspAbstract nvo lanza UnknownObjectException ");
            ex=(UnknownObjectException)cause;
        } else if (cause instanceof UnknownHostException) {
            System.out.println("ExHanAspAbstract nvo lanza UnknownHostException ");
            ex=(UnknownHostException)cause;
        } else if (cause instanceof UnmarshalException) {
            System.out.println("ExHanAspAbstract nvo lanza UnmarshalException ");
            ex=(UnmarshalException)cause;
        } else {
            System.out.println("ExHanAspAbstract nvo OJO deja original "+ ex_ori);
            ex = ex_ori;
        }
```

```
            return ex;
        } // QuitaWrapper

} // aspect
```

E.3 excepciones.XML

Un archivo ejemplo de la forma en que se declara el manejo de excepciones es el siguiente:

```
<ManejoExcepciones XMLns ="http://www.hecc.org" XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.hecc.org esquema.xsd" nombre ="aspecto_excepciones"
ruta_salida="D:/ITESM/tesis/programas/SALIDA"
        genera_aspecto_logging="true" genera_aspecto_seguimiento_PC="true">

        <pointcut advice ="call">
                <datos nombre="operations" descripcion="primera prueba."/>
                <clausula>
                        <joinPoints>
                                <joinPoint retorno="*" clase ="MiInterfazRemoto" metodo="Pba_throw">
                                        <parametro_str nombre="(..)"/>
                                </joinPoint>
                                <joinPoint retorno="*" clase ="MiInterfazRemoto" metodo="Pba_throw_3">
                                        <parametro_str nombre="(int)"/>
                                </joinPoint>
                        </joinPoints>
                </clausula>
                <excepcion>
                        <referencia nombre="ref_1_a_AlreadyBoundException"/>
                </excepción>
                <excepción>
                        <referencia nombre="ref_2_a_ActivationException"/>
                </excepción>
                <grupo_ref nombre="ref_gpo_1"/>
                <grupo_ref nombre="ref_gpo_2"/>
                <paquete_ref nombre="ref_1_paquete_server"/>
        </pointcut>

        <pointcut advice ="call">
                <datos nombre="operations_2" descripcion="segunda prueba."/>
                <clausula>
                        <joinPoints>
                                <joinPoint retorno="*" clase ="MiInterfazRemoto" metodo="Pba_throw_2">
                                        <parametro_str nombre="(..)"/>
                                </joinPoint>
                        </joinPoints>
                </clausula>
                <excepción>
                        <datos nombre="Java.rmi.NotBoundException" descripcion="descripcion de
NotBoundException."/>
                        <ejecutar método ="MiClaseRemota2.metodo_paquete" retorno="void">
                                <parámetros>
                                        <parámetro param="para1" tipo="Integer" valor="10"/>
                                        <parámetro param="para2" tipo="String" valor="si"/>
                                </parámetros>
                        </ejecutar>
                </excepción>
                <excepción>
                        <datos nombre="Java.rmi.RemoteException" descripcion="descripcion de
RemoteException."/>
```

```
                    <configura intentar="1" delay="500" encapsular="ExHanAspRuntimeException"/>
                </excepción>
                <excepción>
                    <referencia nombre="ref_1_a_AlreadyBoundException"/>
                </excepción>
                <excepción>
                    <referencia nombre="ref_2_b_a_ActivationException"/>
                </excepción>
                <grupo_ref nombre="ref_gpo_3"/>
                <grupo_ref nombre="ref_gpo_2"/>
                <paquete_ref nombre="ref_2_paquete_activation"/>
                <paquete_ref nombre="ref_1_b_paquete_server"/>
        </pointcut>

        <pointcut advice ="call">
                <datos nombre="operations_3" descripcion="tercera prueba."/>
                <clausula>
                    <joinPoints>
                        <joinPoint retorno="void" clase ="MiInterfazRemoto" metodo="Pba_throw_2">
                            <parámetros>
                                <parámetro param="el_para1" tipo="String"/>
                                <parámetro param="el_para2" tipo="boolean"/>
                            </parámetros>
                        </joinPoint>
                    </joinPoints>
                </clausula>
                <excepción>
                    <datos nombre="Java.rmi.NotBoundException" descripcion="descripcion de
NotBoundException."/>
                    <ejecutar método ="MiClaseRemota2.metodo_ex2" retorno ="RemoteException">
                        <parámetros>
                            <parámetro param="para1" tipo="String" valor="pba"/>
                        </parámetros>
                    </ejecutar>
                </excepción>
                <excepción>
                    <datos nombre="Java.rmi.RemoteException" descripcion="descripcion de
RemoteException."/>
                    <configura encapsular="ExHanAspRuntimeException"/>
                </excepción>
                <grupo_ref nombre="ref_gpo_2"/>
                <paquete_ref nombre="ref_2_paquete_activation"/>
                <paquete_ref nombre="ref_1_b_paquete_server"/>
        </pointcut>

        <excepción>
                <referencia nombre="ref_2_a_ActivationException"/>
                <datos nombre="Java.rmi.activation.ActivationException" descripcion="descripcion de
ActivationException."/>
                <configura intentar="3" delay="1000" lanzar="Java.rmi.RemoteException"/>
        </excepción>

        <excepción>
                <referencia nombre="ref_2_b_a_ActivationException"/>
                <datos nombre="Java.rmi.activation.ActivationException" descripcion="descripcion de
ActivationException dos."/>
                <configura intentar="1" delay="500" encapsular="ExHanAspRuntimeException"/>
        </excepción>

        <excepción>
                <referencia nombre="ref_1_a_AlreadyBoundException"/>
```

```
                <datos nombre="Java.rmi.AlreadyBoundException" descripcion="descripcion de
AlreadyBoundException."/>
                <configura encapsular="ExHanAspRuntimeException"/>
        </excepción>

        <grupo>
                <datos nombre="ref_gpo_1" descripcion="descripcion grupo uno"/>
                <configura intentar="1" encapsular="ExHanAspRuntimeException"/>
                <excepciones>
                        <datos nombre="Java.rmi.activation.UnknownGroupException" descripcion="descripcion
de Java.rmi.activation.UnknownGroupException"/>
                        <datos nombre="Java.rmi.activation.UnknownObjectException" descripcion="descrip
Java.rmi.activation.UnknownObjectException"/>
                </excepciones>
        </grupo>

        <grupo>
                <datos nombre="ref_gpo_2" descripcion="descripcion grupo dos"/>
                <configura lanzar="Java.rmi.RemoteException" encapsular="ExHanAspRuntimeException"/>
                <excepciones>
                        <datos nombre="Java.rmi.activation.ActivateFailedException" descripcion="This exception
is thrown by the RMI runtime when activation fails during a remote call to an activatable object"/>
                        <datos nombre="Java.rmi.activation.ActivationException" descripcion="General exception
used by the activation interfaces.  As of release 1.4, this exception has been retrofitted to conform to the general
purpose exception-chaining mechanism."/>
                </excepciones>
        </grupo>

        <grupo>
                <datos nombre="ref_gpo_3" descripcion="descripcion grupo tres"/>
                <configura intentar="2" delay="2500"/>
                <excepciones>
                        <datos nombre="Java.rmi.MarshalException" descripcion="descripcion de
Java.rmi.MarshalException"/>
                        <datos nombre="Java.rmi.NoSuchObjectException" descripcion="descrip
Java.rmi.NoSuchObjectException"/>
                </excepciones>
                <ejecutar método ="MiClaseRemota2.metodo_grupo" retorno ="Integer">
                        <parámetros>
                                <parámetro param="para1" tipo="Integer" valor="40"/>
                                <parámetro param="para2" tipo="String" valor="no"/>
                        </parámetros>
                </ejecutar>
        </grupo>

        <paquete>
                <referencia nombre="ref_1_paquete_server"/>
                <datos nombre="Java.rmi.server" descripcion="descripcion paquete paq server"/>
                <configura intentar="3" delay="1000" encapsular="ExHanAspRuntimeException"/>
        </paquete>

        <paquete>
                <referencia nombre="ref_1_b_paquete_server"/>
                <datos nombre="Java.rmi.server" descripcion="descripcion paquete paq server"/>
                <configura intentar="2" delay="5000" lanzar="Java.rmi.RemoteException"/>
        </paquete>

        <paquete>
                <referencia nombre="ref_2_paquete_activation"/>
                <datos nombre="Java.rmi.activation" descripcion="descripcion paquete paq activation"/>
                <ejecutar método ="MiClaseRemota2.metodo_paquete" retorno ="String">
                        <parámetros>
```

```
                        <parámetro param="para1" tipo="Integer" valor="2"/>
                        <parámetro param="para2" tipo="char" valor="a"/>
                    </parámetros>
                </ejecutar>
        </paquete>

        <default>
                <configura intentar="6" delay="100" lanzar="Java.rmi.RemoteException"
encapsular="ExHanAspRuntimeException"/>
                <ejecutar método ="MiClaseRemota2.metodo_paquete" retorno ="int">
                    <parámetros>
                        <parámetro param="para1" tipo="Integer" valor="2"/>
                        <parámetro param="para2" tipo="String" valor="01234567"/>
                        <parámetro param="para3" tipo="boolean" valor="false"/>
                    </parámetros>
                </ejecutar>
        </default>

</ManejoExcepciones>
```

E.4 excepciones.XML

```
<?XML version="1.0"?>
<xsd:schema XMLns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.hecc.org"
        XMLns="http://www.hecc.org" elementFormDefault="qualified">

        <xsd:element name="ManejoExcepciones">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="pointcut" type="tipoPointCut" minOccurs="0"
maxOccurs="unbounded" />
                        <xsd:element name="excepcion" type="tipoExcepcion" minOccurs="0"
maxOccurs="unbounded" />
                        <xsd:element name="grupo" type="tipoGrupo" minOccurs="0"
maxOccurs="unbounded" />
                        <xsd:element name="paquete" type="tipoPaquete" minOccurs="0"
maxOccurs="unbounded" />
                        <xsd:element name="default" type="tipoDefault" minOccurs="1" maxOccurs="1"
/>
                    </xsd:sequence>
                    <xsd:attribute name="nombre" type="xsd:string" use="required" />
                    <xsd:attribute name="ruta_salida" type="xsd:string" use="required" />
                    <xsd:attribute name="genera_aspecto_logging" type="xsd:boolean" use="required" />
                    <xsd:attribute name="genera_aspecto_seguimiento_PC" type="xsd:boolean" use="required"
/>
                </xsd:complexType>
        </xsd:element>

        <xsd:complexType name="tipoPointCut">
                <xsd:sequence>
                    <xsd:element name="datos" type="tipoDatos" minOccurs="1" maxOccurs="1"/>
                    <xsd:element name="clausula" type="tipoClausula" minOccurs="1" maxOccurs="1"/>
                    <xsd:element name="excepcion" type="tipoReferenciaExcepcion" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element name="grupo_ref" type="tipoReferencia" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element name="paquete_ref" type="tipoReferencia" minOccurs="0"
maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="advice" use="required">
```

```
                        <xsd:simpleType>
                                <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="call"/>
                                </xsd:restriction>
                        </xsd:simpleType>
                </xsd:attribute>
        </xsd:complexType>

        <xsd:complexType name="tipoClausula">
                <xsd:sequence>
                        <xsd:element name="joinPoints" type="tipoJoinPoints" minOccurs="1" maxOccurs="1" />
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoJoinPoints">
                <xsd:sequence>
                        <xsd:element name="joinPoint" type="tipoJoinPoint" minOccurs="1"
maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoJoinPoint">
                <xsd:choice>
                        <xsd:group ref="losParametros"/>
                                <xsd:element name="parametro_str" type="tipoReferencia" />
                </xsd:choice>
                <xsd:attribute name="retorno" type="xsd:string" use="required" />
                <xsd:attribute name="clase" type="xsd:string" use="required" />
                <xsd:attribute name="metodo" type="xsd:string" use="required" />
        </xsd:complexType>

        <xsd:group name="losParametros">
                <xsd:sequence>
                        <xsd:element name="parametros" type="tipoParametros" minOccurs="1" maxOccurs="1"
/>
                </xsd:sequence>
        </xsd:group>

        <xsd:complexType name="tipoDatos">
                <xsd:attribute name="nombre" type="xsd:string" use="required" />
                <xsd:attribute name="descripcion" type="xsd:string" use="required" />
        </xsd:complexType>

        <xsd:complexType name="tipoReferencia">
                <xsd:attribute name="nombre" type="xsd:string" use="required" />
        </xsd:complexType>

        <xsd:complexType name="tipoReferenciaExcepcion">
                <xsd:sequence>
                        <xsd:choice>
                                <xsd:group ref="campos"/>
                                        <xsd:element name="referencia" type="tipoReferencia"/>
                        </xsd:choice>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:group name="campos">
                <xsd:sequence>
                        <xsd:element name="datos" type="tipoDatos" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="configura" type="tipoConfigura" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="ejecutar" type="tipoEjecutar" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
```

```
        </xsd:group>

        <xsd:complexType name="tipoExcepcion">
                <xsd:sequence>
                        <xsd:element name="referencia" type="tipoReferencia" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="datos" type="tipoDatos" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="configura" type="tipoConfigura" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="ejecutar" type="tipoEjecutar" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoConfigura">
                <xsd:attribute name="intentar" type="xsd:string" use="optional" />
                <xsd:attribute name="delay" type="xsd:string" use="optional" />
                <xsd:attribute name="lanzar" type="xsd:string" use="optional" />
                <xsd:attribute name="encapsular" type="xsd:string" use="optional" />
        </xsd:complexType>

        <xsd:complexType name="tipoEjecutar">
                <xsd:sequence>
                        <xsd:element name="parametros" type="tipoParametros" minOccurs="0" maxOccurs="1"
/>
                </xsd:sequence>
                <xsd:attribute name="metodo" type="xsd:string" use="required" />
                <xsd:attribute name="retorno" type="xsd:string" use="required" />
        </xsd:complexType>

        <xsd:complexType name="tipoParametros">
                <xsd:sequence>
                        <xsd:element name="parametro" type="tipoParametro" minOccurs="1"
maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoParametro" >
                <xsd:attribute name="param" type="xsd:string" use="required"/>
                <xsd:attribute name="tipo" type="xsd:string" use="required" />
                <xsd:attribute name="longitud" type="xsd:integer" use="optional" />
                <xsd:attribute name="valor" type="xsd:string" use="optional" />
        </xsd:complexType>

        <xsd:complexType name="tipoGrupo">
                <xsd:sequence>
                        <xsd:element name="datos" type="tipoDatos" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="configura" type="tipoConfigura" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="excepciones" type="tipoExcepcionGrupo" minOccurs="1"
maxOccurs="unbounded" />
                        <xsd:element name="ejecutar" type="tipoEjecutar" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoExcepcionGrupo">
                <xsd:sequence>
                        <xsd:element name="datos" type="tipoDatos" maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoPaquete">
                <xsd:sequence>
                        <xsd:element name="referencia" type="tipoReferencia" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="datos" type="tipoDatos" minOccurs="1" maxOccurs="1"/>
                        <xsd:element name="configura" type="tipoConfigura" minOccurs="0" maxOccurs="1"/>
```

```
                    <xsd:element name="ejecutar" type="tipoEjecutar" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="tipoDefault">
                <xsd:sequence>
                    <xsd:element name="configura" type="tipoConfigura" minOccurs="0" maxOccurs="1"/>
                    <xsd:element name="ejecutar" type="tipoEjecutar" minOccurs="0" maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>

</xsd:schema>
```

E.5 log4j.properties.XML

```
log4j.rootCategory=ALL,Default
log4j.appender.Default=org.apache.log4j.FileAppender
log4j.appender.Default.file=log_salida\\log_ejecucion.log
log4j.appender.Default.layout=org.apache.log4j.XML.XMLLayout
log4j.appender.Default.append=false
```

E.6 log4j.properties.XML

```
log4j.rootCategory=ALL,Default
log4j.appender.Default=org.apache.log4j.FileAppender
log4j.appender.Default.file=log_salida\\log_ejecucion.log
log4j.appender.Default.layout=org.apache.log4j.PatternLayout
log4j.appender.Default.append=false
log4j.appender.Default.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
```

Apéndice F

F Contenido del CD

En el CD que se proporciona con esta tesis, se tiene la siguiente información:

- Se tiene un subdirectorio "Man_Exc_Asp_XML" el cual contiene:
  - Subdirectorio "aspecto", ver punto 5.3.2.1
  - Subdirectorio "XML", ver punto 5.3.1.1
  - Subdirectorio "utilerias", ver punto 5.3.2.2
  - Subdirectorio "log_config", ver punto 5.3.3
  - Subdirectorio "log_salida", ver punto 5.3.3
  - Subdirectorio "doc", contiene la documentacion de la herramienta generada con javaDoc, es decir en formato XML.
  - Los archivos:
    - comp_genera.bat, archivo batch, con el que se compila la aplicación, ver punto 5.3.4.
    - lista.lst, archivo necesario para la compilación con AspectJ, el cual contiene la lista de los archivos que se necesitan "tejer".
    - run_genera.bat, archivo batch, con el que se ejecuta la aplicación, ver punto 5.3.4.
  - Subdirectorio ProyectoBDArq, contiene la docuemtnación y código de la aplicación mostrada como ejemplo para las pruebas de la herramienta desarrollada.
  - Subdirectorio Tesis, contiene este docuemto en formato electronico.