Integrando Sensibilidad al Contexto Mediante Aspect Oriented Programming

Arturo Federico Zambrano Polo y La Borda

Director: Dra. Silvia Gordillo

Tesis presentada para obtener el grado de
Magister en Ingeniería de Software

Facultad de Informática
Universidad Nacional de La Plata

Octubre de 2006

Índice general

Índice general 2

Índice de cuadros 5

Índice de figuras 6

Listings 7

1. Introducción 11

2. Computación Móvil, Ubicua, Pervasiva y su Relación con la Sensibilidad al Contexto 14
   - 2.1. Computación Móvil 14
   - 2.2. Computación Pervasiva 15
   - 2.3. Computación Ubicua 16
   - 2.4. Sensibilidad al Contexto 17
     - 2.4.1. Definiciones 19
     - 2.4.2. Taxonomía y Características 20
     - 2.4.3. Volatilidad 21

3. Separación de Concerns y *Aspect Oriented Programming* 23
   - 3.1. Separación de Concerns 23
   - 3.2. Modularización 24
   - 3.3. Problemas 25
     - 3.3.1. Identificación de Concerns 27
   - 3.4. Enfoques para la Separación de Concerns 27
     - 3.4.1. Subject Oriented Programming 29
     - 3.4.2. Composition Filters 32
     - 3.4.3. Demeter (Adaptive Programming) 34
   - 3.5. Aspect Oriented Programming 36
     - 3.5.1. Origen 36
     - 3.5.2. Mecanismos de Composición 38
     - 3.5.3. Definiciones 41
     - 3.5.4. Dynamic Aspect Oriented Programming 42
     - 3.5.5. Prescindencia y Cuantificación 43
     - 3.5.6. Elección del lenguaje para los prototipos: AspectJ . 45

4. Framework Orientado a Aspectos para Aplicaciones Sensibles al Contexto . . . 47
4.1. Estructura general . . . 47
4.2. La Posición y los Recursos como Parte de la Sensibilidad al Contexto . . . 47
4.2.1. Aplicación Base . . . 49
4.2.2. Capa de Adaptación . . . 50
4.2.3. Capa de Modelo de Contexto . . . 53
4.2.4. Agregando Información de Posicionamiento . . . 53
4.3. Personalización como Parte la Sensibilidad al Contexto . . . 55
4.3.1. Contexto: Definiciones y Trabajo Relacionado . . . 55
4.3.2. Proceso de Adaptación . . . 58
4.3.3. Personalizando Mediante Aspectos . . . 58
4.3.4. Implementando Personalización mediante AOP . . . 61
4.4. Otros Aspectos de la Sensibilidad al Contexto . . . 65
4.5. Conclusiones . . . 67

5. Reuso de Aspectos y Sensibilidad al Contexto . . . 69
5.1. Trabajos Relacionados . . . 70
5.2. Refinamiento de la Arquitectura . . . 70
5.3. Niveles de Reuso . . . 71
5.3.1. Primer Nivel: Definiendo Conectores en *Linkage Layer* 73
5.3.2. Segundo Nivel: Definiendo Conectores y Estrategias de Adaptación . . . 78
5.3.3. Tercer Nivel: Definiendo Nuevas Aplicaciones . . . 81
5.4. Conclusiones . . . 84

6. Conflictos entre Aspectos Sensibles al Contexto . . . 86
6.1. Conflictos Aspectuales . . . 86
6.1.1. Ejemplos . . . 87
6.2. Trabajos Relacionados . . . 90
6.3. Resolución de conflictos para *Resource-Awareness* . . . 92
6.3.1. *Semantic Labels* . . . 92
6.3.2. La Coordinación de los Aspectos . . . 93
6.3.3. Diseño . . . 94
6.3.4. Implementación . . . 96
6.4. Conclusiones . . . 97

7. Conclusiones . . . 99
7.1. Trabajo Futuro . . . 100

A. Referencia del Lenguaje AspectJ 102
A.1. Aspectos 102
A.2. Pointcuts 102
A.2.1. Expresiones de Pointcut 104
A.2.2. Definición Formal 107
A.3. Advices 107
A.3.1. Estructura 107
A.3.2. Before advice 108
A.3.3. After advice 108
A.3.4. Around advice 109
A.3.5. Definición Formal 110
A.4. Declaraciones Intertipo (Crosscutting Estático) 111
A.4.1. Introducción de Miembros 111
A.4.2. Modificación de la Estructura de Tipos 112
A.4.3. Declaración de Warnings y Errores en Compilación 112

Bibliografía 113

Índice de cuadros

Cuadro 5.1. Comparasión entre los niveles de reuso. Negro indica cambio en la capa. 84
Cuadro A.1. Ejemplos de utilización de comodines. 105
Cuadro A.2. Ejemplos de utilización de operadores. 105
Cuadro A.3. Categorías de *pointcuts* 106

Índice de figuras

Figura 2.1. Clasificación de los distintos tipos de computación 15

Figura 3.1. Requerimientos descompuestos forman un espectro de concerns. 28
Figura 3.2. Diferentes facetas de un sistema vistas como caras de un cuerpo geométrico 28
Figura 3.3. Varias perspectivas del objeto árbol 30
Figura 3.4. Composición de *subjects* [1] 31
Figura 3.5. Esquema de *Composition Filters* 33
Figura 3.6. Despacho de eventos en *Composition Filters* 34

Figura 4.1. Arquitectura basada en aspectos para *context-awareness*. 48
Figura 4.2. Distintos *concerns* que afectan a la aplicación base 51
Figura 4.3. Ciclo de la adaptación para personalización. 57
Figura 4.4. Separación entre la implementación de personalización y resto de la aplicación. 59
Figura 4.5. Mapeo de la arquitectura propuesta a objetos y aspectos. . . . 61
Figura 4.6. Diagrama de clases de la aplicación. 62
Figura 4.7. Diseño ampliado con aspectos 64
Figura 4.8. Aspecto original de la aplicación. 66
Figura 4.9. Aplicación adaptada 68

Figura 5.1. Arquitectura orientada a aspectos refinada 72
Figura 5.2. Lista de contactos de Agendus 73
Figura 5.3. Estructura de reuso mediante interfaces 82

Figura 6.1. Esquema abstracto de conflictos entre aspectos. 87
Figura 6.2. Esquema de conflictos entre aspectos. 89
Figura 6.3. Aspectos con metadatos relacionados a los recursos. 93
Figura 6.4. Esquema completo de resolución de conflictos: metadatos, coordinador y estrategias. 94
Figura 6.5. Diagrama de clases para la resolución de conflictos. 95

Figura A.1. Estructura de un aspecto 103
Figura A.2. Estructura de pointcut nombrado 103

Listings

4.1. XML request 52
4.2. Implementación en AspectJ del aspecto Bandwitdh 52
4.3. Request con el agregado de la posición geográfica. 54
4.4. Request enriquecido por varios aspectos 54
4.5. Aspecto para la captura de datos ingresados por el usuario. 65
4.6. Aspecto para el ofrecimiento de sugerencias al usuario 65
4.7. Aspecto para adaptación del color según el horario 67
5.1. Aspecto abstracto definiendo la adaptación. 71
5.2. Conector definiendo un pointcut concreto. 71
5.3. Aspecto abstracto para la notificación de objetos cercanos 74
5.4. interfaz de la aplicación base 75
5.5. Definición de un conector 76
5.6. Definición alternativa del pointcut 76
5.7. *Wrapping* de objetos del dominio 77
5.8. Aspecto para optimización del uso del ancho de banda mediante compresion. 79
5.9. Aspecto para encriptación de datos. 80
5.10. Aplicación del aspecto de encriptación de datos. 80
6.1. Aspecto anotado. 95
6.2. Apariencia final del código de un aspecto anotado 96
6.3. Pointcut para intercepción durante la creación de los aspectos que manejan recursos. 96
6.4. Ejemplo de regla simple implementada en JBoss Rules. 97
6.5. Intercepción de los cambios de estado en los recursos del sistema . . . 97
6.6. Aserción de cambios en el motor y evaluación de reglas 97

Resumen

La complejidad de los sistemas sensibles al contexto implica un gran número de *concerns* de software que deben ser modelados e implementados. Estos *concerns* normalmente se entremezclan con el resto de los módulos produciendo código esparcido y enredado que, por consiguiente, es muy difícil de evolucionar y mantener.

Para atacar este problema en el presente trabajo se estudia la utilización de *Aspect Oriented Programming*. Se relevan las características principales de las aplicaciones móviles sensibles al contexto y los principales enfoques de separación de *concerns*. Se describe una arquitectura basada en *Aspect Oriented Programming* que permite lograr un alto grado modularización, separando los *concerns* que no pertenecen a la dimensión principal de la aplicación. Dicha arquitectura es estudiada desde el punto de vista del reuso de los *concerns* de sensibilidad al contexto. Además se estudian los conflictos que surgen de la integración de los diversos *concerns* implementados como aspectos, proponiendo un mecanismo que permite resolverlos conservando a la vez la escencia modular de *Aspect Oriented Programming*.

Dedicatoria

A mi madre que hizo todo lo necesario para que mi hermano y yo podamos estudiar.

Agradecimientos

Agradezco a todos aquellos que ayudaron directa o indirectamente a completar este trabajo.

Al LIFIA por permitirme trabajar explorando aún aquellos temas que no pertenecían a su agenda de investigación.

A la Universidad Nacional de La Plata que, gracias a la gratuidad, permitió que me acercara a esta profesión.

Más concretamente le agradezco a Silvia por darme libertad en los temas de investigación, a Pola, Nacho y Tomás que colaboraron en el desarrollo de algunos de los trabajos publicados. A Gustavo por haberme acercado al LIFIA.

A mi familia, a la cual resté tiempo para desarrollar este trabajo.

1. Introducción

El desarrollo de sistemas móviles esta hoy en día en constante crecimiento y es de esperar que con el avance de las tecnologías destinadas al desarrollo de dispositivos cada vez más potentes, la tendencia se incremente. Los sistemas móviles, que se ejecutan en dispositivos tales como PDAs (*Personal Digital Assistants*) y teléfonos celulares, deben ofrecer al usuario una gran cantidad de información y de servicios, que en el futuro serán cada vez más numerosos y complejos. Entre los dispositivos móviles más utilizados están los teléfonos celulares de última generación (*smart phones*), las PDAs, *laptops* y *tabletPCs*.

Una de las características más importantes de los sistemas móviles es que deben aprovechar la posición geográfica para asistir al usuario en diversas tareas. Ejemplos de ello son los sistemas que guían a los usuarios en las visitas a museos, sistemas de navegación basados en GPS (como los utilizados en automv́iles), o Friend Finder de At&T [2] que notifica la presencia de personas geográficamente cercanas.

La posición geográfica es una de las partes más importantes del contexto de la aplicación y hasta ahora la más estudiada y aprovechada. Pero el contexto de una aplicación está formado por toda aquella información que caracteriza la situación en la cual un usuario interactúa con una aplicación. Esta información debe ser utilizada para aumentar la relevancia de la información y los servicios prestados. Otras piezas de información, además de la posición, que componen el contexto son: el perfil de usuario, sus preferencias, el tiempo, las actividades habituales, las restricciones propias del *hardware* del dispositivo como su *display*, memoria, procesador, *inputs*, etc., inclusive otras del ambiente de ejecución como el ancho de banda disponible, la presencia o ausencia de conectividad, etc.

Aquellas aplicaciones que utilizan la información del contexto para adaptar sus respuestas son llamadas sensibles al contexto (*context-aware*). Las adaptaciones son tan variadas como la información que hace al contexto. Por ejemplo, un teléfono sensible al contexto, que detecta que el usuario se encuentra en una reunión, podría vibrar en vez de sonar. Una aplicación corriendo en una PDA podría adaptar el nivel de brillo de la pantalla en función de la luz ambiente. Una aplicación que necesita de un servidor podría hacer una *cache* de datos para subsanar las pérdidas de conexión propias de las redes inalámbricas.

Podemos ver entonces que la sensibilidad al contexto conlleva una gran cantidad de *concerns* y adaptaciones que no pertenecen al dominio principal de la aplicación, y que pueden verse como requerimientos no funcionales de la misma. Desarrollar una aplicación teniendo en cuenta todos estos *concerns* a la vez es sumamente complejo y propenso a errores de diseño. El diseñador puede fácilmente perder el foco principal de la aplicación. Diseñar e implementar todos los *concerns* a la vez conduce al acoplamiento entre la aplicación y los mecanismos de adaptabilidad al contexto. Es importante destacar que, aún cuando se trate de un buen disenãdor el cual aplique reconocidas arquitecturas para minimizar el impacto del acoplamiento (como los *Design Patterns* [3]) el mismo no puede ser eliminado por completo [4].

En lo que se ha llamado la era post-objetos, han surgido numerosos enfoques que apuntan a resolver los problemas que el paradigma de orientación a objetos no ha podido resolver por completo. Entre ellos se encuentra *Aspect Oriented Pogramming* (AOP), que propone encapsular los *concerns* que atraviesan las aplicaciones en construcciones llamadas *aspectos.* Los aspectos definen comportamiento y de alguna manera expresan en qué puntos de la ejecución de otros módulos debe invocarse este comportamiento.

En este trabajo se explora la utilización de AOP para lograr una correcta modularización de sistemas móviles sensibles al contexto en general, haciendo particular énfasis en la posición geográfica, el perfil del usuario, y los recursos de dichos sistemas.

De esta exploración se espera obtener las siguientes contribuciones:

- Comprender la complejidad de las aplicaciones sensibles al contexto.
- Comprender los diferentes paradigmas de separación de *concerns*
- Presentar el paradigma de orientación a aspectos.
- Mostrar cómo el desarrollo de aplicaciones sensibles al contexto puede beneficiarse de este paradigma.
- Encontrar y solucionar aquellos problemas que surjan de aplicar AOP a un dominio relativamente nuevo y complejo como la sensibilidad al contexto.

Este trabajo se encuentra estructurado de la siguiente manera: el capítulo 2 presenta el dominio de las aplicaciones móviles, relacionándolas con términos afines como ubicuidad y pervasividad, a su vez se presenta el concepto de sensibilidad al contexto y los desafíos que representa para el desarrollo de aplicaciones. El capítulo 3 presenta la noción de separación de *concerns* y los distintos enfoques que han surgido como

respuesta a la necesidad de mayores niveles de modularidad; en este capítulo se hace hincapié en el más popular de estos paradigmas: *aspect oriented programming*. El capítulo 4 muestra cómo la orientación a aspectos puede ser aplicada al diseño de sistemas sensibles al contexto. En el capítulo 5 se analiza cómo el reuso de funcionalidad sensible al contexto se ve afectado por la utilización de AOP en el proceso de desarrollo de software. El capítulo 6 explica la forma en la que pueden resolverse conflictos entre aspectos de sensibilidad al contexto y propone un mecanismo para la coordinación de dichos aspectos. Finalmente, en el capítulo 7 se presentan las conclusiones de este trabajo y se delinean los trabajos futuros de investigación a realizar en este área.

2. Computación Móvil, Ubicua, Pervasiva y su Relación con la Sensibilidad al Contexto

En los últimos años los términos computación móvil, ubicua y pervasiva han ganado terreno dentro del ámbito de las ciencias de computación.

Dado el incremento de capacidad de cálculo en los procesadores, sus tamaños cada vez más reducidos y menor consumo de energía; hoy en día, más y más computadoras están a nuestro alcance.

Desde las computadoras personales portátiles (*laptops*, *notebooks*, *tabletPC*, etc), pasando por los teléfonos celulares, *handhelds*, *palmtops*, y hasta embebidas como computadoras de abordo en vehículos, edificios inteligentes, electrodomésticos, etc. vemos como nuestro mundo de va poblando de computadoras por doquier.

La existencia de múltiples computadoras a nuestro alrededor tiene relación con los conceptos de ubicuidad, pervasividad, movilidad y sensibilidad al contexto. En las siguientes secciones analizaremos qué significan en detalle estos términos, que desafíos implican desde el punto de vista de la ingeniería de software y cómo se relacionan con el resto del presente trabajo.

2.1. Computación Móvil

De acuerdo a Lyytinen et al. [6] *mobile computing* tiene que ver fundamentalmente con incrementar nuestra capacidad de mover físicamente los servicios computacionales junto con nosotros. Como resultado la computadora se convierte en un dispositivo que siempre está presente (porque nos acompaña), el cual expande nuestras capacidades para recordar, comunicarnos y razonar, independientemente de la ubicación física del dispositivo. Esto puede ocurrir ya sea por la reducción de tamaño de las computadoras y/o por la provisión de acceso a la capacidad computacional a través de una red, mediante dispositivos menos poderosos.

Esta evolución tiene que ver con la migración de los dispositivos desde los escritorios hasta nuestros bolsos (*notebooks*), bolsillos (*handhelds* y *smart-phones*) e inclusive el cuerpo (*wereable computers*). La parte interesante de esto es que entonces la computación se convierte en algo que literalmente nos puede acompanãr a cualquier lugar, soportando una gran variedad de actividades humanas. La Figura 2.1 [6]

ofrece una clasificación de los distintos tipos de computación de acuerdo al nivel de *embeddedness* y movilidad.

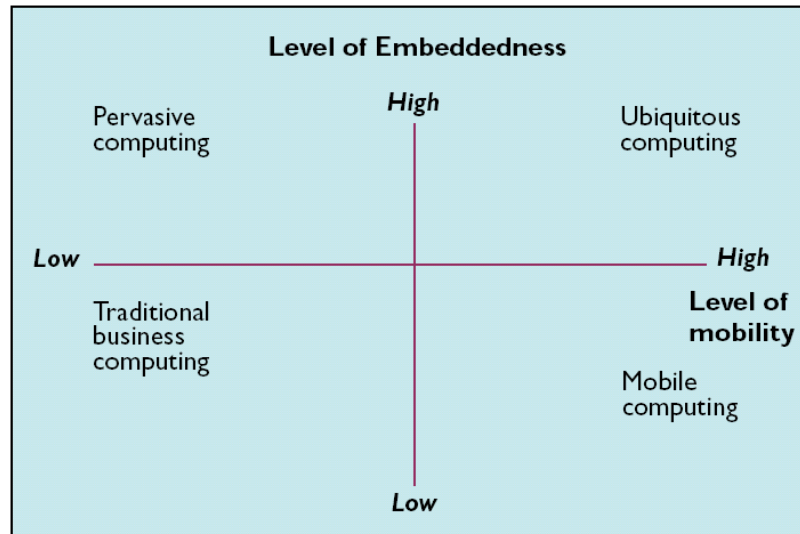

FIGURA 2.1. Clasificación de los distintos tipos de computación

Sin embargo, en *mobile computing* existe una limitación importante: el modelo computacional no cambia considerablemente con la posición (ni el resto del contexto). Esto ocurre porque en general los dispositivos no aprovechan la información del contexto en el que están ejecutando, por lo tanto no pueden adaptarse a éste. La información del contexto puede ingresar al sistema mediante el monitoreo de sensores; también es posible obtener esa información a través del ingreso manual por parte del usuario, pero este tipo de molestias deben ser minimizadas para evitar su distracción.

2.2. Computación Pervasiva

El concepto *pervasive computing* implica que la computadora tiene la capacidad de obtener información del ambiente donde opera y utilizarla para construir modelos computacionales de manera dinámica. Este proceso es recíproco, pues el ambiente también debería ser capaz de detectar a aquellos dispositivos que están entrando en él. De esta manera ambos, dispositivo y ambiente, son conscientes uno del otro y pueden interactuar de manera inteligente para brindar una mejor calidad en los servicios. Esto es el núcleo de la idea de *pervasive computing*: Un área poblada con

sensores, *pads*, *badges* y un conjunto de modelos del ambiente (físico, social, cognitivo, etc.).

Los servicios de computación pervasiva pueden ser construidos ya sea embebiendo modelos de ambientes específicos en computadoras dedicadas, o bien construyendo capacidades genéricas en las computadoras para consultar, detectar, explorar y construir dinámicamente modelos de los ambientes.

Actualmente, desde el punto de vista del software, uno de los mayores desafíos en el campo de *pervasive computing* es "enseñarle" al dispositivo/aplicación sobre el contexto en el cual se desempeña.

2.3. Computación Ubicua

Mark Weiser es el responsable de la definición del concepto *Ubiquitous Computing*. En el artículo *Some Computer Science Problems in Ubiquitous Computing* [7] esta se define como:

> ...el método por el cual se mejora el uso de las computadoras haciendo disponibles muchas de ellas a través del ambiente físico ...
>
> ... un ambiente en el cual cada persona está continuamente interactuando con cientos de computadoras, situadas en las cercanías, interconectadas de manera inalámbrica. El objetivo es lograr el tipo más efectivo de tecnología que sea escencialmente invisible al usuario.

Es interesante notar que estas definiciones tienen un significado mucho más profundo que el simple hecho de llevar una *notebook* a cualquier lugar. El mensaje más importante[1] que se puede extraer de las definiciones de Weiser tiene que ver con el hecho de embeber los dispositivos computacionales en el ambiente haciéndolos desaparecer literalmente de la vista del usuario. De esta manera el usuario utiliza la capacidad computacional del ambiente de manera inconsciente, por lo tanto su atención no es distraída. Estas características tienen un impacto notable en lo que a interacción con el usuario se refiere.

En las ultimas décadas gran parte del esfuerzo de los investigadores se ha puesto al servicio del desarrollo de tecnologías cada vez más distractivas para el usuario. Los ejemplos más destacables son las tecnologías de realidad virtual y multimedia. La realidad virtual intenta construir un mundo dentro de la computadora, y trata además, de sumergir al usuario dentro de ese mundo, haciéndolo vestir anteojos que

[1] desde el punto de vista de este trabajo

proyectan el mundo virtual, sensores en sus ropas, que detectan sus movimientos y le permiten manejar objetos virtuales. La realidad virtual emplea una maquinaria enorme para simular el mundo, en lugar de, invisiblemente, mejorar el mundo que ya existe[8]. En el caso de las tecnologías multimedia, ellas están pensadas para demandar la atención del usuario. Nótese que esto no es una crítica a estas áreas de las ciencias de la computación, sino que se trata simplemente de sentar posiciones, dejando en claro cual es el enfoque de *ubiquitous computing* mediante la comparación con sus opuestos más destacados.

La frase que mejor resume esta idea es del propio Weiser:

> *The most profund technologies are those that disappear. They wave themselves into the fabric of the everyday life.*

Pero cómo lograr que la tecnología que nos rodea desaparezca, en especial la que tiene que ver con la computación?

Computadoras más pequeñas Teniendo computadoras físicamente pequeñas, con un poder de cálculo razonable, es posible pensar en empotrarlas en los aparatos que usamos todos los días.

Computadoras más baratas *Ubiquitous Computing* supone que muchas computadoras son utilizadas por el mismo usuario. Esta gran cantidad de computadoras, presupone un costo no demasiado elevado por unidad.

Conexiones inalámbricas Las conexiones *wireless* facilitan la instalación y permiten a los dispositivos conectados en red moverse libremente.

2.4. Sensibilidad al Contexto

Nuestros sentidos captan continuamente información del ambiente en el que nos desenvolvemos. Esta información es procesada en nuestro cerebro. Toda esta información junto con informaciones precedentes, nuestra memoria y relaciones sociales, forman el contexto. Nuestras decisiones se encuentran influenciadas en su mayoría por este contexto, y actuamos de manera consistente con nuestro contexto. El contexto es el que nos permite decidir qué cosas son importantes y cuales no.

Nuestra sensibilidad al contexto está continuamente activa, estemos o no conscientes de ella. De hecho, en general, no lo estamos y es allí de donde viene gran parte de su valor. Al no *pensar* en ella podemos concentrarnos en otros objetivos. Lo mismo ocurre con cualquier actividad en la cual hayamos sido bien entrenados. En

algún punto desarrollamos la habilidad de hacer algo sin pensar ni enfocarnos en ello, pudiendo realizar más de una tarea a la vez. Como expresa Weiser, aquellas cosas que han sido bien aprendidas, desaparecen y nos permiten enfocarnos en nuevas metas.

Con estas ideas en mente entonces podemos preguntarnos: por qué no construir software que utilice la información del contexto, adaptándose a éste y mejorando a la vez la experiencia del usuario? Las aplicaciones sensibles al contexto pueden optimizar sus resultados de acuerdo a la situación actual y limitar la necesidad de entradas por parte del usuario, logrando reducir la distracción del mismo. Si la interrupción al usuario no puede ser evitada, es necesario que la aplicación busque el momento adecuado y minimice la distracción, por ejemplo reduciendo el espacio de selección [9].

Imaginamos entonces un mundo donde las aplicaciones tienen cierta noción de las actividades que estamos desarrallondo y del contexto en el cual se llevan a cabo. De esta manera son capaces de adaptarse y brindar una respuesta de mayor valor para el usuario. Estas adaptaciones pueden tomar varias formas:

- Una barra de herramientas puede mostrar aquellas tareas que siguen a la que estamos realizando, colocando en primera instancia la tarea que tenemos en mente.

- Una aplicación móvil, que presenta datos con referencias geográficas, podría ir actualizándolos automáticamente a medida que nos movemos.

- La misma aplicación podría predecir el próximo punto geográfico que visitaremos, basándose en los anteriores y estableciendo nuestra trayectoria. De esta manera podría adelantarse mostrándonos información de lo que vamos a encontrar en el futuro cercano.

- Una aplicación que corre en una PDA, puede estar consciente de los recursos de *hardware* disponibles y adaptar su interfaz y comportamiento teniendo en cuentas las restricciones del caso.

Esta lista podría ser muy extensa, debido a la variedad de dispositivos, de aplicaciones, tareas, usuarios, etc. que pueden entrar en juego, haciendo que las combinaciones crezcan desmesuradamente.

Dado que es imposible predecir de antemano todas las combinaciones posibles, y los potenciales estados del contexto, es necesario que las adaptaciones se realicen de la manera más automática que sea posible. Esto quiere decir que aquellas adaptaciones

deberían poder generarse de manera dinámica. Desde el punto de vista del software esto implica un tremendo desafío, pues es necesario modularizar aquellos elementos contextuales y sus adaptaciones de manera tal que sean combinables con unos con otros.

2.4.1. Definiciones

Según el diccionario de la Real Academia Española, la definición de contexto es:

> Contexto.(Del lat. contextus).
>
> 1. Entorno lingüístico del cual depende el sentido y el valor de una palabra, frase o fragmento considerados.
> 2. Entorno físico o de situación, ya sea político, histórico, cultural o de cualquier otra índole, en el cual se considera un hecho.
> 3. p. us. Orden de composición o tejido de un discurso, de una narración, etc.
> 4. desus. Enredo, maraña o unión de cosas que se enlazan y entretejen.

De estas acepciones nos quedamos con la segunda. Desde el punto de vista del software, una de las definiciones más aceptadas es la que brindan Dey et al [10]:

> Contexto es cualquier información que puede ser usada para caracterizar la situación de una entidad. Una entidad puede ser una persona, un lugar o un objeto que es considerado relevante para la interacción entre el usuario y la aplicación; incluyendo al usuario y la aplicación mismos.

Dey [11] define a las aplicaciones sensibles al contexto de la siguiente manera:

> Un sistema es sensible al contexto si utiliza el contexto para proveer información y/o servicios relevantes para el usuario, donde la relevancia depende de la actividad del usuario.

Además establece tres categorías de características que una aplicación sensible al contexto puede soportar.

- Presentación de información y servicios al usuario.
- Ejecución automática de un servicios para un usuario.

- Etiquetado del contexto para su posterior recuperación.

Bill Schilit, uno de los pioneros en el área, define al contexto en los siguientes términos: [12]

> Los aspectos importantes del contexto son: dónde uno está, con quién y qué recursos hay en las cercanías. El contexto comprende más que la posición del usuario, porque otros elementos de interés son móviles y cambiantes. El contexto incluye la luminosidad del ambiente, el ruido, la conectividad, el ancho de banda e inclusive la situación social; por ejemplo, si uno está con su jefe o con un compañero de trabajo.

2.4.2. Taxonomía y Características

En la literatura se encuentran muchas referencias a tipos de información contextual que, dependiendo de la aplicación, pueden ser todo o parte del contexto. Dentro de una misma aplicación el contexto puede estar formado por diferentes elementos contextuales, dependiendo del momento en la ejecución de la aplicación.

Algunos trabajos de investigación han aportado taxonomías y clasificaciones para los diferentes tipos de contextos o elementos contextuales. Según Abowd y Dey [10], las categorías más importantes de información contextual son:

- Posición
- Identidad
- Actividad
- Tiempo

Tazari et al. [13] describe el contexto físico en función de una serie de factores como temperatura, iluminación, nivel de ruido, etc. Otra categoría de contexto está representada por el contexto de usuario, su identidad, su perfil, situación social y actividad. Una tercera categoría es llamada contexto computacional, esta categoría incluye: conectividad, ancho de banda, recursos de *display*, memoria, procesador, etc.

Dey y Abowd [10] introducen el concepto de contexto mediante las cinco Ws, del inglés *who, what, where, when and why*. Estas preguntas en realidad encierran una clasificación de la información contextual:

Who Los sistemas actuales enfocan su interacción en la identidad de un usuario en particular. También se han creado sistemas que tienen en cuenta la presencia de otras personas, por ejemplo *Friend Finder* de AT&T Labs [2].

What La interacción con dispositivos sensibles al contexto requiere que en cierto punto se realice una interpretación de la actividad que está realizando el usuario (para comprender qué información esta accediendo).

Where En cierta forma, la pregunta *donde?* ha sido más explorada que el resto. Prueba de esto es la existencia de un conjunto de tecnologías conocidas como Location Based Services [14]. El ejemplo del turista recorriendo una ciudad mientras es guiado por sistema ejecutándose en una PDA es recurrente en la literatura y analizado desde diferentes perspectivas. Sin embargo, queda mucho por estudiar en el campo de la localización geográfica y su impacto sobre el contexto, por ejemplo la utilización de trayectorias para preparar respuestas del sistema a futuro.

When En general el tiempo se ha utilizado únicamente para etiquetar información de contexto, pero no como parte activa del contexto en sí mismo. Un área importante en la que puede ser aplicada la ubicación temporal es la comprensión de las actividades del usuario, por ejemplo, si un usuario pasa poco tiempo en cierta parte de una aplicación puede ser un indicador de poco interés o de actividad poco frecuente. Otros usos podrían incluir asociar actividades con ciertos horarios, de manera tal que sea posible disparar ciertos servicios anticipándose a las necesidades del usuario.

Why Esta pregunta es más difícil de responder que el *what*, ya que requiere modelar los objetivos del usuario, y utilizar este contexto para inferir los motivos y objetivos de ciertas acciones.

2.4.3. Volatilidad

Una de las características más importantes del contexto es su dinamismo. La información que caracteriza un contexto en un momento dado posiblemente no se repita nunca. Si miramos el contexto como una función de muchas variables (elementos contextuales), lo podemos pensar en un espacio multidimensional donde una de esas dimensiones es el tiempo. Si nos movemos por el eje del tiempo, veremos que el resto de variables cambian su valor con diferentes frecuencias. Por lo tanto no es posible

definir una frecuencia común, que permita capturar todos los cambios. El problema subyacente es la continuidad de la información, pero sobre estos aspectos de los contextos hasta ahora no hay bibliografía ni resultados de investigación disponibles. Independiente de esto, las aplicaciones sensibles al contexto deben lidiar con cambios continuos de contexto en tiempo de ejecución, por lo tanto las arquitecturas de software que se desarrollaron hasta el momento (Context Toolkit [15], Hydrogen[16], etc.) y las que se desarrollen el futuro deberán soportar un nivel cada vez más alto de cambio en los valores que definen el contexto.

3. Separación de Concerns y *Aspect Oriented Programming*

En este capítulo se presenta la problemática de separación de *concerns* y los enfoques que se desarrollaron para solucionarla. Se explican los más importantes, como *Subject Oriented Programming*, *Composition Filters* y *Aspect Oriented Programming*; haciendo hincapié en este último, por ser el utilizado para el desarrollo de este trabajo.

3.1. Separación de Concerns

La palabra *concern* puede ser traducida como *competencia* o *preocupación*. Estas traducciones no denotan el significado del término en cuanto a construcción de *software* se refiere. Un *concern* es una faceta o una perspectiva del *software*. Un *concern* puede estar ligado a uno más requerimientos del *software*. Si fueran varios estos están íntimamente relacionados. Por ejemplo, si para un sistema de *software* distribuido existen varios requerimientos relacionados, como pueden ser:

- Que partes del sistema sólo sean accesibles al administrador.
- Que los usuarios no puedan alterar datos de otros usuarios (sandbox).
- Que las comunicaciones en la red sean encriptadas.

Se puede ver que estos requerimientos están claramente relacionados con la seguridad. La faceta del sistema correspondiente a la seguridad es un *concern*. Utilizaremos la palabra *concern*, sin traducción, para no complicar innecesariamente el texto.

Cada *concern* puede plasmarse en una o muchas secciones de código fuente. A su vez, una sección puede alojar código correspondiente a uno o varios *concerns*. Los beneficios de contar con un *concern* en una sola sección de código son muchos y bien conocidos. El más importante tiene que ver con que es suficiente inspeccionar un sólo lugar para entender cómo se materializa este *concern* en el software. De lo contrario sería necesario revisar varias secciones y mentalmente filtrar el código de los demás *concerns*. Por otro lado, un *concern* compacto y expresado en una sola porción de código puede ser analizado, extendido y depurado más fácilmente.

Según Laddad [17] un *concern* es un requerimiento específico que debe ser atacado para satisfacer el objetivo global de un sistema. Un sistema es la materialización de

un conjunto de *concerns*. Un sistema bancario, por ejemplo, es la materialización de los siguientes *concerns*:

- Administración de cuentas y usuarios.
- Cálculo de intereses.
- Transacciones interbancarias.
- Transacciones ATM.
- Persistencia de todas las entidades.
- Autorización de acceso a los servicios
- Etc.

Además de los *concerns* del sistema, un proyecto abarca otros *concerns* de proceso, tales como compresibilidad, mantenibilidad, rastreabilidad y facilidad de evolución.

Dijstra [18] introdujo el principio de separación de *concerns* como la necesidad de tratar un asunto por vez. Sin duda, el hecho de tener los *concerns* de un sistema correctamente separados y modularizados, es una cualidad importante, pero lamentablemente no existen mecanismos obvios para lograr esta característica.

La separación de *concerns* es un principio fundamental de ingeniería, basado en la noción de *dividir y conquistar*, que es aplicado en el análisis, diseño e implementación de software. Todos los paradigmas de programación y modelización proveen construcciones que expresan unidades modulares. Por ejemplo, el paradigma procedural provee procedimientos, la orientación a objetos clases y la programación funcional funciones. Estas unidades se conocen con el nombre de módulos. El objetivo es organizar los sistemas mediante la composición jerárquica de estos módulos. Estas unidades modulares están orientadas a dividir el software de manera funcional y estos módulos funcionales son invocados únicamente de manera explícita.

3.2. Modularización

La correcta modularización de un sistema es un tema que inquieta a los investigadores en informática desde que surgieron los primeros lenguajes de alto nivel. Ya en 1972, Parnas [19] reconocía la necesidad de la modularización, e inclusive delineó un criterio que permite obtener modularizaciones correctas. Parnas establece tres beneficios de la programación modular:

1. Tiempos de desarrollo menores. Porque distintos grupos pueden trabajar en módulos aislados, solo conociendo las interfaces que los relacionarán.

2. Flexibilidad en los productos: debería ser posible realizar cambios drásticos en un módulo sin necesidad de modificar los demás.

3. Comprensibilidad, debería ser posible estudiar un módulo del sistema a la vez.

Un módulo es una asignación de responsabilidades más que un subprograma.

La afirmación más valiosa que hace Parnas en [19] es que cada módulo debe esconder una (o más) decisión (nes) de diseño. De manera tal que el resto de los módulos no tengan conocimiento alguno al respecto. La interfaz del módulo debe ser definida revelando tan poco como sea posible acerca de su funcionamiento interno.

Como criterios adicionales para la modularización se nombran:

1. Una estructura de datos y los procedimientos para operarla son parte de un módulo. (Notar la similitud con la definición de lo que es un *objeto* en el paradigma de POO).

2. La secuencia de instrucciones necesaria para llamar una rutina y la rutina en sí misma forman parte del mismo módulo.

3. El formato de los bloques de control debe ser escondido en algún "módulo de control".

4. Los códigos, tablas de símbolos y datos similares también deben ser encapsulados en un módulo para una mayor flexibilidad.

3.3. Problemas

Es interesante notar cómo los paradigmas han evolucionado brindando diferentes abstracciones que se concentran en identificar y componer unidades funcionales, permitiendo representar los módulos de un sistema, como los nombrados anteriormente: funciones, procedimientos, objetos, etc.

Sin embargo, existen propiedades de los sistemas que impactan en varios de sus módulos. Estas propiedades no pueden ser acotadas a una parte de un diseño

funcional, sino que se encuentran embebidas a lo largo y a lo ancho de todo o gran parte del diseño.

Cuando un arquitecto de software encara un nuevo diseño, comienza por atacar la funcionalidad primaria del sistema, que se conoce como *lógica de negocios*. Por ejemplo en una aplicación bancaria los módulos que representan la lógica de negocios se diseñan para manejar transacciones y cuentas. En el caso de una aplicación de venta, la funcionalidad esta centrada en registrar las ventas y el manejo de inventario. En ambas veremos que existen *concerns* abarcativos a nivel de sistema que se relacionan como el *logging*, acceso restringido, persistencia y otros elementos comunes a otras aplicaciones. Estos *concerns* que afectan múltiples módulos son llamados *crosscuttings concerns*.

Kiczales [20] destacó que los módulos que proveen los lenguajes y notaciones en la actualidad[1] no son lo suficientemente flexibles para encapsular el comportamiento referente a esas propiedades que atraviezan[2] los sistemas.

Un *crosscutting concern* es aquel que por su naturaleza afecta a otros, por lo tanto, no puede ser completamente modularizado. Como resultado, se obtiene código esparcido, ya que el *concern* en cuestión no se encuentra en un solo lugar (módulo); y enredado, porque el código de los *módulos* afectados además contiene trozos del *crosscutting concern*.

Concerns como el *logging*, la sincronización y la persistencia, entre otros, aparecen mezclados con otros *concerns* de base de los sistemas. Por ejemplo, imaginemos cualquier sistema diseñado e implementado mediante orientación a objetos. Si el sistema posee persistencia veremos que en diferentes puntos del programa existen referencias al *concern* de persistencia. En general, no importa como modularicemos nuestro sistema, el *concern* de persistencia siempre se materializará como secciones de código esparcidas en entre el código de otros módulos.

Una vez entendida la necesidad de definir nuevas herramientas que permitan tratar con estos *concerns*, cuya modularización es dificultosa, es posible plantear una serie de temas que deberán ser abordados por estas herramientas:

- **¿Cuáles son los *concerns* que deben ser separados?** Esta pregunta implica saber, cḿo detectamos estos *crosscuttings concerns*.

- **¿Cómo capturamos cada uno de los *concerns* de manera localizada?**

[1]Aquí *actualidad* se refiere a fines de los 90s cuando Kiczales desarrolló su trabajo sobre AOP

[2] *Cut across* de allí el término *crosscutting*

¿Cuáles son los nuevos mecanismos, tipos de módulos y construcciones sintácticas que nos permiten modularizar los *crosscuttings concerns*?

- **¿Qué mecanismos de composición podríamos usar?** Aún cuando logremos descomponer correctamente el sistema, es necesario definir mecanismos que permitan componer el comportamiento de los módulos para lograr el funcionamiento general deseado.

3.3.1. Identificación de Concerns

Hemos visto entonces que existen dos categorías principales de *concerns*, los llamados *core concerns*[3] y los *crosscutting concerns*. Los primeros se refieren a la lógica principal de la aplicación en cuestión mientras que los segundos tienen que ver con requerimientos periféricos que afectan varios *core concerns*.

Si podemos identificar correctamente los *concerns* y por lo tanto categorizarlos correctamente, reducideremos la complejidad del problema, enfocándonos en los diferentes *concerns* de manera individual.

Al comienzo del ciclo de desarrollo de software nos encontramos con los requerimientos. Partiendo de ellos debemos identificar y separar los *concerns*. Para ilustrar esta idea Laddad [17] utiliza la metáfora de un prisma. Los requerimientos son un haz de luz entrando al prisma, que al salir se ha convertido en un conjunto de rayos correspondientes al espectro. Cada color indica un *concern* identificado (Figura 3.1). El proceso de análisis de requerimientos debería funcionar de la misma manera, permitiendo descomponer correctamente los requerimientos en un conjunto de *concerns*.

Otra forma de ver un sistema compuesto por diferentes *concerns* es imaginarlo como un cuerpo geométrico en tres dimensiones donde cada cara se corresponde con un *concern* (Figura 3.2).

Algunos enfoques para separación de *concerns* se alínean muy bien con esta idea de las facetas, permitiendo que los conceptos en una de las facetas se mapeen a los de otra, mediante mecanismos de composición. En las siguiente sección veremos uno de ellos.

3.4. Enfoques para la Separación de Concerns

Existen varios enfoques que fueron desarrollados con el objetivo de subsanar las falencias que los paradigmas tradicionales tienen, en particular la orientación a

[3]Forman lo que en la jerga de AOP recibe el nombre de programa o aplicación base, y en otros paradigmas de SOC se los denomina dimensión predominante

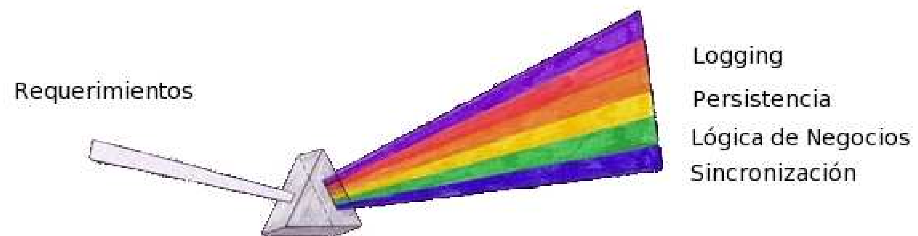

FIGURA 3.1. Requerimientos descompuestos forman un espectro de concerns.

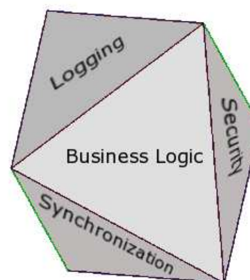

FIGURA 3.2. Diferentes facetas de un sistema vistas como caras de un cuerpo geométrico

objetos. Estos desarrollos extienden el modelo puro de objetos agregando nuevos conceptos y mecanismos necesarios para su composición.

3.4.1. Subject Oriented Programming

El modelo clásico de objetos sugiere crear clases que representen entidades (y otras abstracciones) y que los clientes manipulen estos objetos mediante las operaciones expuestas (interfaces). Esto nos asegura obtener las ventajas del encapsulamiento, por ejemplo la libertad de cambiar la implementación interna de un objeto y que sus clientes no sean afectados por el cambio.

El diseñador de un objeto debe incluir en este a aquellas propiedades y comportamientos intrínsecos del objeto en cuestión. Dado que en el mundo del software es necesario reutilizar los objetos ya definidos a lo largo de varias aplicaciones vemos que nuevas propiedades y comportamientos irán surgiendo en este proceso. Este nuevo comportamiento, que en el momento inicial de la definición del objeto se consideró como extrínseco, si es incluído en el objeto se convierte en intrínseco[21].

Desde el punto de vista de cada aplicación existirán propiedades y comportamiento naturales de los objetos. Estas características de los objetos no tienen por qué ser las mismas en las distintas aplicaciones.

Al momento de reutilizar un objeto existente se plantea el siguiente problema en el modelo tradicional de objetos. Es necesario decidir dónde colocar el nuevo comportamiento que se espera de un objeto. Si se decide utilizar el objeto en su estado original, el nuevo comportamiento quedará en la aplicación (separado del objeto), perdiendo el encapsulamiento y el polimorfismo en los objetos de la aplicación. Por otro lado, si se decide introducir comportamiento extrínseco en el objeto original, potencialmente podría verse afectado el funcionamiento de las aplicaciones que ya lo utilizaban. Es importante notar que cada aplicación que utilice el objeto agregaría nuevo comportamiento extrínseco al objeto de acuerdo a su propio punto de vista, afectando notablemente su mantenibilidad.

Harrison y Ossher [21] propusieron Subject Oriented Programming (SOP) como una extensión al paradigma de objetos con el fin de tratar diferentes *perspectivas subjetivas* de los objetos modelados. Por ejemplo, un objeto *libro* modelado desde el punto de vista del departamento de marketing tendrá atributos como *área temática* o *resumen*, mientras que para el departamento de manufacturación los atributos de interés podrían ser tipo de papel, calidad de encuadernación, etc. Es evidente que el mismo objeto modelado desde diferentes puntos de vista posee diferentes atributos, lo mismo ocurre con el comportamiento. Como se puede observar en la figura 3.3,

distintos sujetos tienen diferentes perspectivas del mismo objeto. En la figura, el objeto árbol es percibido de diferentes maneras según quien sea su *usuario*.

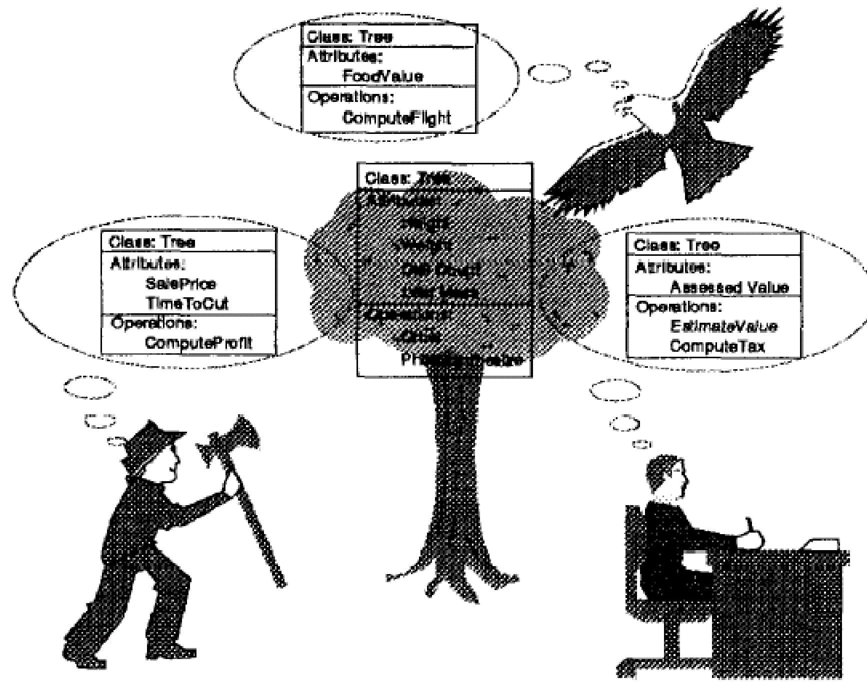

FIGURA 3.3. Varias perspectivas del objeto árbol

Los diferentes contextos de utilización no son la única razón para usar perspectivas. Este enfoque también sirve para tratar los conflictos que pudieran surgir en la integración cuando el software es desarrollado con un alto grado de independencia. Por ejemplo, si dos aplicaciones de la misma empresa son desarrolladas siguiendo las perspectivas mencionadas anteriormente, por dos equipos de desarrollo separados, sería posible integrar estas distintas perspectivas del mismo objeto. Inclusive, también podría agregarse comportamiento no previsto.

Cada perspectiva es llamada *subject*. Una perspectiva es una colección de clases y/o fragmentos de clases (en el sentido de los *mixins*[4]) relacionados por herencia u otras relaciones definidas por el *subject*. Por lo tanto, el *subject* es un modelo de objetos completo o parcial. Los *subjects* pueden ser compuestos siguiendo las siguientes reglas de composición:

[4]Los *mixins* son fragmentos de clases pensados para ser compuestos con otras clases o *mixins*, pero no son utilizables de manera aislada, sin en el contexto de otro módulo.

Reglas de correspondencia Las reglas de correspondencia especifican, si existiere, la correspondencia entre clases, atributos y métodos de diferentes *subjects*. Por ejemplo, podemos usar una regla de correspondencia para expresar que los *subjects* libro del departamento de marketing y el libro del departamento de manufactura son en realidad dos perspectivas de la misma entidad. Esta correspondencia puede establecerse aún si las clases tienen distintos nombres. De la misma manera pueden establecerse correspondencias entre métodos y atributos.

Reglas de combinación Podemos utilizar reglas de combinación para establecer cómo dos *subjects* deben ser combinados. Las clases resultantes tendrán los métodos y atributos independientes, y aquellos que se correspondan serán compuestos de acuerdo a las reglas de combinación especificadas. Estas reglas pueden establecer, por ejemplo, que un método en un subject preceda a otro de otro subject, o que lo reemplace.

Reglas de correspondencia y composición Este tipo de reglas son una manera abreviada de expresar a la vez correspondencia y combinación.

La composición de uno o más *subjects* se realiza mediante estas reglas de composición, el resultado de la composición es una nueva perspectiva (Figura 3.4).

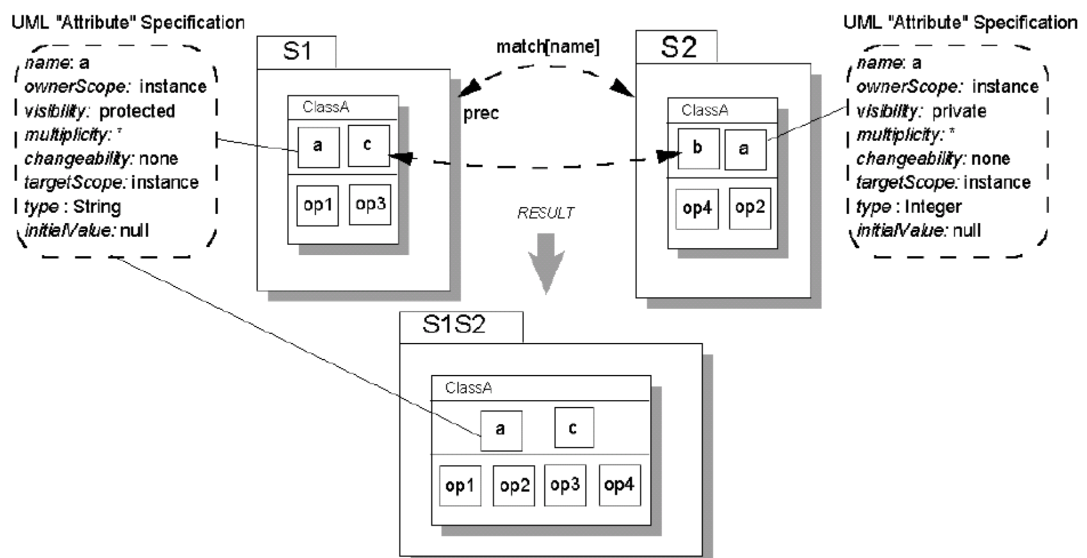

FIGURA 3.4. Composición de *subjects* [1]

Subject Oriented Programming ataca los siguientes problemas:

- La creación de extensiones y configuraciones de software
  - Sin modificar el código original.
  - Encapsulando la variaciones para múltiples plataformas, versiones y características.
- Customización e integración de sistemas y componentes reusables.
- Provisión de desarrollos entre múltiples equipos utilizando modelos de dominio compartidos, relacionados o independientes.
- Desarrollo de clases descentralizado, eliminando conflictos por concurrencia y cuellos de botella debidos a la centralización.
- Mantenimiento de la correspondencia entre los requerimientos y el código.
- Simplificación del código de muchos *Design Patterns* [3]. Por ejemplo, *decorator, abstract factory, visitor, strategy, adapter*, etc.
- Aplicación de extensiones, inclusive aquellas no planeadas.
- Desarrollo de *suites* de aplicaciones variando los grados de independencia entre los equipos de desarrollo.

3.4.2. Composition Filters

Composition Filters (CF) fueron introducidos por Akşit [22] [23], y están motivados por las dificultades surgidas al expresar la coordinación de los mensajes en el modelo de objetos tradicional. Por ejemplo, expresar sincronización a nivel de la interfase de un objeto requiere de alguna manera inyectar código referente a la sincronización en todos los métodos que deben ser sincronizados. Extender una clase con nuevos métodos, mediante subclasificación, podría requerir cambios en el esquema de sincronización. Este y otros problemas relacionados son conocidos como anomalías de herencia [24]. Se puede decir que uno de los problemas del modelo de objetos convencional es la ausencia de mecanismos adecuados para separar la funcionalidad del código para la coordinación de los mensajes.

CF extiende el modelo convencional de objetos agregando los filtros de mensajes (ver figura 3.5).

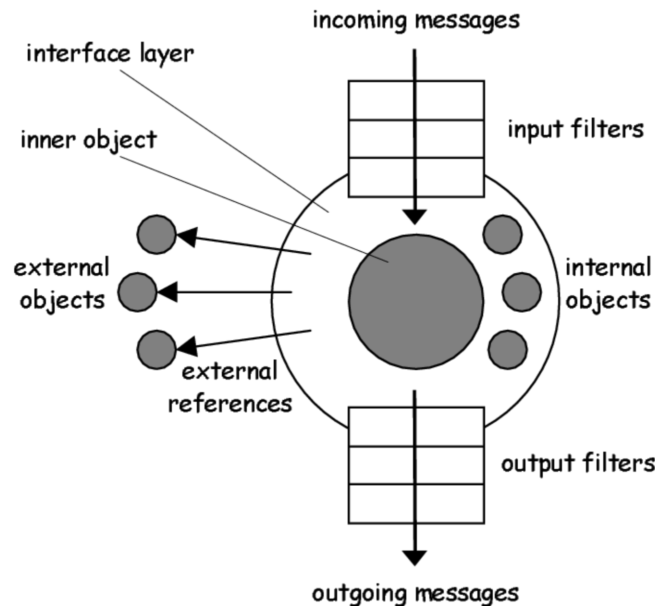

Figura 3.5. Esquema de *Composition Filters*

El objeto ahora consiste de un objeto interno también conocido como *kernel o inner object* y una capa de interfaz. Este objeto núcleo puede ser imaginado como un objeto estándar del paradigma, expresado en algún lenguaje, por ejemplo Java o C++. La *capa de interfase* contiene un número arbitrario de filtros, separados según se trate de filtros de entrada o de salida. Estos filtros pueden modificar la invocación de los mensajes, modificando el selector del método o el receptor del mensaje. En efecto los filtros pueden ser usados para redireccionar mensajes (delegación) a objetos internos o externos, también es posible traducir los mensajes cambiando el selector. Estos filtros pueden retrasar la ejecución poniendo los mensajes en *buffers*, o bien pueden generar excepciones. Un esquema del despacho de mensajes puede verse en la figura 3.6.

Nuevos tipos de filtros pueden ser agregados, permitiendo implementar restricciones de sincronización, de tiempo real, transacciones atómicas, chequeo de precondiciones y otros aspectos de una manera bien localizada y con un alto nivel de modularización. En general, cualquier *concern* que requiera interceptar mensajes o extender métodos con acciones previas y/o posteriores a la ejecución del método, puede ser expresado usando CF.

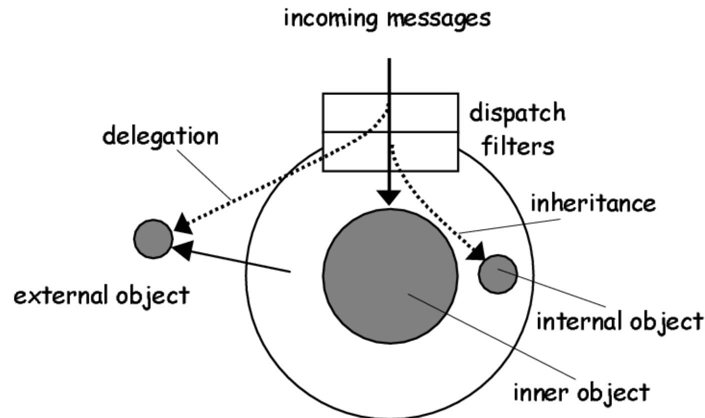

FIGURA 3.6. Despacho de eventos en *Composition Filters*

La capacidad de redireccionar mensajes puede ser usada para implementar delegación y herencia dinámica. En el modelo de CF, delegación significa redireccionar mensajes a objetos externos y asegurar que las referencias a *self* apunten al receptor original del mensaje. Por otro lado, herencia significa redireccionar mensajes a objetos internos y asegurarse que las referencias a *self* apunten al receptor original.

Un filtro puede delegar mensajes en diferentes objetos internos de acuerdo al estado del objeto núcleo. En consecuencia, esto significa que la superclase de un objeto puede cambiar dependiendo su estado, este efecto se conoce como herencia dinámica.

3.4.3. Demeter (Adaptive Programming)

El objetivo de Demeter [25] es desacoplar la estructura de los objetos de ciertos algoritmos que la atraviesan. El nombre *Demeter* proviene de un principio de diseño conocido en la orientación a objetos que dice que un método solo debe contener mensajes al objeto receptor (*self*), a variables de instancia locales, o a objetos pasados como parámetro a dicho método. Otra forma de ver este principio es pensar en que los bloques de código del estilo `object.getPart1().getPart2().getPart3()` deberán ser evitados, pues navegan profundamente en la estructura de los objetos, esto significa que dichos métodos atraviesan el diagrama de clases.

Siguiendo la ley de Demeter [26], deberíamos definir métodos (*accessors*) para exponer partes de los objetos referenciados como variables de instancia. De esta manera, cambiamos los accesos profundos en las estructuras de los objetos por métodos que

exponen parte del diagrama de clases. Normalmente esto es preferible a la primera situación, pero no es lo ideal porque sólo reduce parcialmente el problema de exponer partes de la estructura.

Imaginemos un sistema de una empresa que contiene una función que permite contabilizar el total a pagar en concepto de salarios. Suponiendo que la empresa se encuentra dividida en departamentos, podemos intuir que la empresa recolectará la sumatoria de parciales calculados por los departamentos, y que los departamentos sumarán los salarios de sus empleados. En este caso el diagrama de clases se atraviesa siguiendo la secuencia:

$$Empresa \rightarrow Departamento \rightarrow Empleado \rightarrow Salario$$

Supongamos que por cuestiones administrativas, se agregan *divisiones* entre la empresa y los departamentos. En este caso la secuencia cambia y se convierte en:

$$Empresa \rightarrow División \rightarrow Departamento \rightarrow Empleado \rightarrow Salario$$

Deberíamos agregar métodos en la clase *División* que permitan atravesar la estructura de objetos, pero que no tienen que ver con el cálculo en sí mismo, que está dado por la sumatoria de los salarios de todos los empleados de la empresa, sino con la capacidad de acceder a partes remotas del diagrama de clases.

Lieberherr [27, 25] propone una solución que consiste en escribir el código de los algoritmos basándose en especificaciones parciales del diagrama de clases. Estas especificaciones parciales sólo nombran aquellas clases que son importantes para el algoritmo en sí mismo, en nuestro ejemplo ellas son Empresa y Salario. Las especificaciones parciales del diagrama de clases son llamadas *traversal strategies*. Para nuestro ejemplo una *traversal strategy* podría ser *desde Empresa a Salario*. Esta estrategia indica que algún diagrama de clases concreto debe ser atravesado desde la clase `Empresa` hasta la clase `Salario`, pero la manera de hacerlo es inferida y el código necesario es generado automáticamente. Esto se conoce como *structure-shy behavior*

El algoritmo de cálculo de salario se reduce a expresar que siempre que un objeto de la clase `Salario` es visitado, su monto debe ser sumado al total. Finalmente, el monto total es retornado a la instancia de `Empresa`.

Analizando nuestro ejemplo, el hecho de agregar la clase `División` o cualquier otra clase que modifique el camino desde la `Empresa` hasta el `Salario`, no altera la definición abstracta del algoritmo. En caso de agregar nuevas clases sólo es necesario regenerar el programa y recompilarlo.

El término *adaptive programming* fue introducido en 1991 y hace referencia a los aspectos estructura de clases y *strucuture-shy behavior* ya descriptos. Luego se agregó el aspecto de sincronización. Este trabajo es una de las raíces de AOP. De acuerdo a su autor, Lieberherr, *adaptive programming* es un caso particular de AOP donde existen dos tipos bloques de construcción del software. Uno de ellos expresable en términos de grafos y el otro como estrategias para atravesar los grafos. Una estrategia es una especificación parcial del diagrama de clases, definiendo algunas clases clave y sus relaciones.

3.5. Aspect Oriented Programming

3.5.1. Origen

Como se expresó con anterioridad, las estructuras modulares de los sistemas se organizan en jerarquías. Los aspectos atraviesan (*cuts across*, de allí el nombre *crosscutting concern*) estas jerarquías en cualquier sentido.

Algunos ejemplos de aspectos que hemos nombrado son: reglas de composición de SOP, sincronización, restricciones de tiempo-real, chequeo de errores, *structure-shy behavior*, etc. Existen muchos ejemplos de aspectos usados en la literatura: administración de memoria, persistencia, seguridad, políticas de *caching*, *profiling*, monitoreo, testing, estructura y representación de los datos.

Ciertos aspectos son propios de determinados dominios, y surgen del análisis de las características particulares de estos. Por ejemplo, en el dominio de los sistemas distribuidos otros ejemplos de aspectos podrían ser: interacción de componentes, invocación remota, estrategias de pasaje de parámetros, *load balancing*, manejo de errores, transacciones distribuidas, etc. En posteriores capítulos de este trabajo analizaremos los aspectos que tienen que ver en particular con el dominio de las aplicaciones móviles sensibles al contexto.

Uno de los problemas que dan origen a AOP es el llamado código enredado (*tangled code*). Si partimos de la presunción que ciertos aspectos realmente atraviesan la estructura modular/funcional del software, la manera de implementar dichos aspectos es incluyendo porciones de código que referencian a la dimensión no funcional, de manera tal que el código de dicha dimensión queda mezclado con el código original del módulo funcional. Este fenómeno, fue bautizado como *tangled code* y es el resultado la imposibilidad de separar claramente el código proveniente de diferentes *concerns*.

Es interesante ver que el mismo efecto ocurre con la escritura **automática** de código optimizado. El programador escribe código no optimizado, por otro lado, exis-

ten ciertas heurísticas y políticas de optimización que son aplicadas al código original (por un precompilador); generando un nuevo código, seguramente enredado, que es óptimo. En este caso las dimensiones del programa original y de las optimizaciones están complemente separadas y pueden evolucionar independientemente.

Los *crosscutting concerns* y el código enredado son las motivaciones principales de AOP.

Como ocurre en SOP, donde las perspectivas representan distintos modelos computacionales, en general notamos que muchas veces varios modelos convergen en un punto. Lo que generalmente hacemos (cuando usamos únicamente el paradigma de objetos) es modelar ese "gran" modelo, donde todas las perspectivas quedan plasmadas, produciendo un *tangled-model*. Por ejemplo, si estamos diseñando un sistema administrativo donde los objetos son presistentes y en los cuales se manejan los errores que pudieran ocurrir, generaremos un modelo donde ciertos objetos del dominio cumplen responsabilidades relacionadas con la persistencia y el manejo de errores; esto quiere decir que tanto la persitencia como el manejo de errores no pudieron ser correctamente modularizados.

El objetivo de AOP es definir estos distintos modelos por separado, diseñándolos y programándolos independientemente, e integrándolos de manera automática. Esto permite que todos evolucionen con un alto grado de independencia.

Czarnecki [4] afirma que *un modelo es un aspecto de otro modelo si lo atraviesa*. Atravesar aquí significa que dicho modelo no se encuentra bien modularizado y localizado.

Obviamente siempre existe la posibilidad de refactorizar nuestro modelo de manera tal que un aspecto "calce" en él, y ya no sea más un aspecto, sino parte de la dimensión principal. Esto lleva al problema antes mencionado del *tangled model*.

Es ingenuo pensar que uno puede "deshacerse"de los aspectos mediante la refactorización del modelo, aunque sea aplicando los reconocidos patrones de diseño [3]. Los patrones en ciertas situaciones pueden acarrear complejidad extra y problemas de performance. Aún así, permiten convertir los aspectos en componentes (objetos), pero los patrones de diseño quedan esparcidos en la aplicación y no se encuentran claramente modularizados. En muchos casos un mismo objeto puede jugar más de un rol, a esos roles debemos sumarles aquellos derivados de la participación en patrones. Es fácil ver que el código del objeto resultante contendrá secciones pertenecientes a diversos *concerns*, y por lo tanto será muy difícil de mantener.

Existen publicaciones como [28] donde se propone la utilización de AOP para aislar el código correspondiente a los patrones, de esta manera se logra encapsular la

implementación de los patrones.

Los aspectos surgen para resolver una categoría de problemas que aún no han sido mitigados por otros paradigmas, en este contexto es necesario contar con tecnologías que permitan:

1. Expresar aspectos: esto es, definir mecanismos lingüísticos que faciliten la materialización de los aspectos en la forma de código fuente, localizado y modularizado.
2. Componer eficientemente los aspectos. Esta composición es la integración entre los módulos de los aspectos que definen un sistema y los módulos funcionales.

3.5.2. Mecanismos de Composición

Los paradigmas tradicionales cuentan con sus respectivos mecanismos de composición: invocación de funciones, procedimientos, mensajes, y herencia son algunos ejemplos de ellos. Estos mecanismos, al igual que los módulos, no siempre son adecuados para expresar *crosscutting concerns*, por lo tanto, nuevos mecanismos deben ser introducidos.

De los enfoques analizados en la sección 3.4 podemos ver que, por ejemplo, las *traversal strategies* de Demeter son mecanismos composición útiles para el *concern* que tiene que ver con la estructura de clases del sistema. De la misma manera, las *composition rules* de *Subject Oriented Programming* permiten componer las distintas perspectivas (facetas) de un modelo. Los filtros de mensajes de *Composition Filters* también son ejemplos de mecanismos de composición no tradicionales.

Czarnecki [4] establece ciertos requerimientos para los mecanismos de composición de aspectos, que nos ayudarán a analizar y comprender la evolución de AOP en términos de composición. Estos requerimientos son:

- Acoplamiento mínimo.
- Diferentes tiempos y modos para el enlace (*binding*) de aspectos.
- Adición no invasiva de aspectos respecto del código existente.

Acoplamiento Mínimo

La razón para desear acoplamiento mínimo y no independencia total es puramente práctica. En la realidad sólo en casos triviales es posible desacoplar por completo

los aspectos, ya que en general los aspectos se refieren a distintas perspectivas sobre un mismo modelo. Aunque las perspectivas sea ortogonales, esto no implica una independencia total. Todas las perspectivas integradas deben conducir a un modelo consistente, por lo tanto las perspectivas no deben ser contradictorias entre sí. Esta no contradicción implica por sí misma cierto acoplamiento.

Demeter y sus *traversal strategies* son un buen ejemplo de composición con acoplamiento mínimo. Tradicionalmente en los programas orientados a objetos se nombran muchas más clases de las que en realidad se necesitan para la computación de los algoritmos, exponiendo así la estructura de clases y produciendo acoplamiento innecesario. Este acoplamiento es eliminado en Demeter, nombrando sólo aquellas clases clave para la computación que desea realizarse y dejando sin especificar la navegación en el grafo de clases. Esta navegación es luego inferida, por lo tanto los cambios en la estructura de la clases no afectan la definición de los algoritmos.

Uno de los modelos más aceptados de aspectos es el de *pointcuts*; en este modelo el punto donde se acoplan los aspectos se conoce como *join point*. En el caso de Demeter, las *traversal strategies* son los puntos donde se une el comportamiento del aspecto "estructura de clases" con el resto del comportamiento. En el caso de SOP, las reglas de combinación definen la manera en que distintas perspectivas se enlazan, estas reglas son los *join points* para SOP.

Existen 3 tipos de *join points* [4]:

- Por nombre: cuando el *join point* hace referencia a algún constructor del lenguaje en cuestión. Por ejemplo, un `join point` de un aspecto puede indicar el nombre de un método en una clase determinada.

- Por nombre calificado: similar al anterior, pero indicando cierta información de contexto. Por ejemplo el mensaje `m1()` de clase `X` cuando es invocado desde alguna instancia de la clase `Y`.

- Por patrones: la referencia se hace mediante la especificación de un patrón, en el sentido de *pattern-matching*. Esto permite referenciar muchos *join points* al mismo tiempo.

Con esta idea en mente podemos decir que si vemos porciones de código que siempre aparecen acompañando ciertos constructores o patrones de código, es posible convertirlos en aspectos, estableciendo como *join points* a los constructores o patrones. Así evitamos la replicación del código de los *concerns* en cuestión. Es importante notar que los módulos no invocan a los aspectos, sino que estos atraviesan a los módulos

nombrando los *join points*. Por lo tanto, un aspecto extiende o modifica la semántica del modelo al cual refiere.

Diferentes modos y tiempos de enlace (binding)

El enlace entre los módulos y los aspectos puede realizarse en tiempo de compilación, antes de la ejecución o en tiempo de ejecución. La elección de uno u otro acarrea consecuencias en cuanto al dinamismo y la performance del sistema final.

El enlace puede ser estático o dinámico. Si es estático estamos hablando de enlaces optimizados pero "congelados", es decir no pueden ser modificados ulteriormente con facilidad, ya que cualquier modificación en general implica regeneración de código. El *binding* dinámico conlleva algún tipo de indirección entre el código de las entidades y los aspectos, que debe ser vinculado en el momento de la ejecución. El *binding* dinámico es especialmente útil cuando las operaciones de *re-binding* son muy frecuentes. El dinamismo impone un costo en performance, lo cual limita su uso.

Adaptabilidad No-Invasiva

Entendemos por adaptabilidad no-invasiva a la capacidad de adaptar un componente de software sin modificar su código fuente manualmente. Es clave que el mecanismo de composición provea los medios necesarios para realizar adaptaciones no-invasivas, de manera tal que sea posible incluir adaptaciones no previstas.

Idealmente, deberíamos poder manejar las adaptaciones de manera aditiva, mediante algún operador provisto por el lenguaje. Esto es, dado el código original de un componente nosotros queremos **adicionar** ciertos cambios en su comportamiento o estructura. Es importante notar que esto no implica que el código fuente del componente no sea físicamente modificado. La modificación o no del código es una consecuencia secundaria que no hace a la esencia de la aditividad de los cambios, pues si existiera dicha modificación sería hecha en forma automática por alguna herramienta. De hecho la expresión de la composición en el aspecto nos está diciendo que ahora el componente original tiene su semántica modificada.

Un ejemplo de operador de composición es la herencia en la orientación a objetos. Este tipo de composición es no-invasiva únicamente respecto a la superclase, pero los objetos cliente deben reflejar que ahora utilizan instancias de la subclase y no ya de la superaclase. Concluimos entonces que el operador de composición *herencia* es invasivo.

Otro ejemplo de operador de composición son las reglas de combinación de SOP.

Es posible definir reglas que combinan una clase con modificaciones de la misma, y publicar esa clase con el mismo nombre. De esta manera el código de los objetos cliente permanece en su estado original. En este caso el operador de composición es no invasivo respecto de los clientes.

Obviamente, la adaptación no-invasiva no es posible en todos los casos. El factor determinante es que existan constructores o patrones en el código del componente a adaptar, de manera que puedan usarse como *hooks* para adicionar el nuevo comportamiento. De esta manera exteriormente no se observan cambios y la adaptación se torna no invasiva.

3.5.3. Definiciones

Antes de continuar con la clasificación de los distintos enfoques para la implementación de *aspect oriented programming* es necesario introducir las definiciones comunes a todos ellos.

Join Point Es un punto bien definido en la ejecución de un programa. Ejemplos son: llamados a métodos, manejadores de excepciones (palabra clave `catch` en Java), acceso a miembros de la clase, constructores, etc.

Advice Código arbitrario y ejecutable. Un *advice* es conceptualmente independiente de los *join points*, aunque puede tomar información de contexto proveniente de los *join points* si fuera necesario.

Pointcut Un conjunto de *join points* de un programa, generalmente definido por comprensión. Los *pointcuts* proveen un mecanismo que permite a un *advice* afectar varios *join points*.

Aspecto Es un módulo que implementa un *crosscutting concern*. Este módulo contiene *pointcuts* (referencias a *join points*) y *advices* que indican el comportamiento a ejecutar cuando se alcanza un *join point*.

Es importante notar la diferencia entre los *join points* y los *pointcuts*. Los *join points* residen en el código original que será afectado por uno o más aspectos. Los *pointcuts*, en cambio, se encuentran del lado del aspecto y denotan un conjunto de *join points*.

3.5.4. Dynamic Aspect Oriented Programming

Los sistemas de AOP dinámica deben soportar tres tareas fundamentales en tiempo de ejecución:

- Reemplazo de *join points*.
- Reemplazo de *advice*.
- Reemplazo del *binding* entre estos 2 elementos.

Un sistema AOP dinámico es aquel que posee una representación en tiempo de ejecución y que, a través de esta representación, es capaz de reemplazar, remover e insertar nuevos *join points* arbitrarios en tiempo de ejecución, así como cargar y descargar *advices* y, arbitrariamente, reemplazar *pointcuts* (enlace entre *join points* y *advices*).

Esta definición implica que:

- Un sistema que permite agregar advices **sólo** a través de código no es un sistema AOP dinámico. Ya que el advice no tiene una representación en *runtime*.
- Un sistema que permite intercambiar advices a través de una representación en runtime, pero solo en *join points* existentes, aunque sean todos los *join points* posibles, no es un sistema AOP dinámico.
- En contraste, un sistema que no permite la redefinición de *join points durante el período de vida del aspecto* es considerado un sistema AOP dinámico.

Estas restricciones dejan muy pocos sistemas en la categoría de dinámicos. En general los sistemas dinámicos traen como consecuencia negativa una performance muy pobre, porque requieren de muchas indirecciones para acomodar la flexibilidad que proveen.

En el mundo Java, la gran mayoría de los sistemas AOP dinámicos, entre los que podemos citar a PROSE [29], trata de utilizar el soporte de reflexión Java y a partir de allí construyen una capa que provee funcionalidad aspectual. Es aquí donde la performance se degrada, pues para lograr la intercepción de los *joint points* recurren por ejemplo a utilización de la JDI (*Java Debugger Interface*) para notificarse de dichos *joint points* y actuar en consecuencia. Un sistema corriendo en estas condiciones es inherentemente lento.

Un enfoque diferente fue elegido en Steamloom [30, 31] donde el soporte para AOP dinámico es provisto a nivel máquina virtual. De más está decir que esto involucra la utilización de una máquina virtual no estándar, pero en la actualidad Steamloom es un proyecto de investigación y no aplicable a sistemas que deban entrar en producción.

Steamloom

Steamloom [30, 31] está basado en la máquina virtual de IBM Jikes [32]. Esta JVM (Java Virtual Machine) intenta cumplir tanto como sea posible la especificación de Sun respecto de JVM. Jikes es un proyecto *open source*, y se encuentra bajo una licencia (*IBM Public License*) que permite basar otros desarrollos en ella. Jikes fue extendida por el Software Modularity Lab de la Darmstadt Technishe Universtät, haciendo uso del proyecto BAT2 de la misma universidad. BAT es un *toolkit* orientado a la manipulación de *bytecodes* que le permite a Steamloom hacer *weaving* y *unweaving* de aspectos en tiempo de ejecución. Dado que está construido sobre Jikes, Steamloom utiliza el sistema optimización adaptativo, generando código óptimo que se traduce en una performance comparable a los sistemas AOP estáticos como AspectJ, pero manteniendo todas las ventajas derivadas de la programación orientada a aspectos dinámica.

3.5.5. Prescindencia y Cuantificación

Según Filman, existen dos propiedades dentro de AOP que son importantes para que un lenguaje dado sea considerado orientado a aspectos [33].

Cuantificación

AOP permite escribir sentencias de la forma:

En el programa P, cuando la condición C sea verdadera, realizar la acción A.

Esto implica tres aspectos fundamentales para el diseñador de un lenguaje AOP.

Cuantificación Qué clases de condiciones C podemos especificar.

Interfaz Cuál es la interfaz de las acciones A. Cómo interactúan con el programa base P y entre ellas.

Weaving Cómo hace el sistema para entretejer la ejecución de las acciones del programa base P con las acciones A.

En un sistema orientado a aspectos podemos cuantificar sentencias para especificar qué código se ejecutará en qué circunstancias. Esta cuantificación puede realizarse sobre la estructura estática de los sistemas o sobre su comportamiento dinámico.

Cuantificación Estática La estructura estática es el programa como texto. Dos vistas comunes son: la estructura estática en términos de interfaces públicas del programa (métodos) y como un programa parseado en forma de árbol sintáctico.

Los mecanismos de aspectos *black-box* cuantifican sobre la interfaz pública de los componentes como métodos de objetos y funciones. Ejemplos de estos sistemas son los Composition Filters [34]. Una implementación simple para esta clase de sistemas es envolver[5] los componentes con el comportamiento aspectual.

Los sistemas *clear-box* permiten la cuantificación sobre la estructura parseada de los componentes. Ejemplos de estos sistemas incluyen a: AspectJ que permite cuantificar en la llamada y la recepción de mensajes, Subject Oriented Programming, cuyas reglas de composición permiten cuantificar sobre elementos tales como la interpretación de variables dentro de los módulos.

Ambas técnicas tienen sus puntos fuertes y débiles. *Clear-box*, por ejemplo, requiere el código fuente; Pero en ambientes donde no se pueden construir *proxies*, *clear-box* es efectiva para crear aspectos "caller-side", es decir asociados al ambiente de la invocación de un subprograma. Por otro lado, las técnicas *black-box* son típicamente más fáciles de implementar y pueden ser utilizadas en componentes donde el código fuente no está disponible. Dado que las técnicas *black-box* no permiten cuantificar nada excepto la interfaz de los componentes no son buenas para debugging, como sí lo son las *clear-box*.

Cuantificación Dinámica La cuantificación dinámica permite ligar el comportamiento de los aspectos con situaciones en runtime. Por ejemplo:

- La ocurrencia de una excepción.
- La llamada a un subprograma X dentro del ámbito temporal de la llamada a otro subprograma dado.
- El tamaño del *call stack*.
- Patrones en la historia del programa.

[5] En en el sentido del patrón de disenño *Wrapper* [3].

Prescindencia

Prescindencia se refiere a que el programador del código base no es consciente que su código será aumentado mediante aspectos. Esto implica que siempre debería ser posible agregar los aspectos en posteriores etapas del desarrollo.

Si bien no es imprescindible esta característica para que un sistema sea considerado orientado a aspectos, Fillman afirma que los mejores sistemas AO tienden a presentar un alto grado de prescindencia.

La mayoría de los sistemas orientados a aspectos brindan prescindencia.

3.5.6. Elección del lenguaje para los prototipos: AspectJ

Como hemos visto, existen muchos enfoques para la separación de *concerns*, uno de ellos es la orientación a aspectos. Dentro de la orientación a aspectos existe una gama de lenguajes y sistemas de aspectos que proveen tanto *AOP* dinámica como estática, diferentes tipos de cuantificadores y descriptores de *pointcuts*, distintos niveles de polimorfismo aspectual, etc.

Si bien la idea de aplicar *AOP* dinámica es tentadora en primera instancia, pues brinda el mayor grado de flexibilidad, permitiendo el *weaving* y *unweaving* en *runtime*, también es cierto que estos sistemas son experimentales y en muchos casos su performance es pobre. Steamloom, la única máquina virtual con soporte nativo de aspectos, únicamente puede correr en una estación de trabajo linux, lo cual restringe su aplicabilidad. Por lo tanto la aplicación de programación orientada a aspectos dinámica no es posible hoy en día para los dispositivos móviles.

Un aspecto importante de los dispositivos móviles es su variedad. Existen innumerables modelos, cada uno con sus características de hardware. Además estos utilizan distintos sistemas operativos propietarios. Es importante desarrollar sistemas que sean independientes del hardware y el sistema operativo subyacente. La propuesta de Java "*write once, run anywhere*" es especialmente atractiva en este sentido, y es por eso que la mayoría de los fabricantes de dispositivos móviles proveen implementaciones de máquinas virtuales Java para sus teléfonos y otros dispositivos. La versión de Java para dispositivos móviles se conoce como J2ME [35]. También existen implementaciones de J2ME para los principales sistemas operativos utilizados en *handhelds*, que son PalmOS [36] y Windows [37].

El campo de las orientación a aspectos estática el lenguaje más maduro es AspectJ [38]. AspectJ fue desarrollado inicialmente en Xerox Parc, por un grupo de investigadores liderado por Gregor Kiczales [39]. AspectJ fue el primer lenguaje de aspectos

de propósito general y evolucionó a partir de trabajos en el área de concurrencia y sincronización. Hoy el proyecto es soportado como un subproyecto de Eclipse [40] y ha evolucionado lo suficiente como para ser utilizado en ambientes de producción. Por otro lado, AspectJ cuenta con una gran comunidad de usuarios y desarrolladores. Es por eso que se ha seleccionado a AspectJ para la implementación de los prototipos. En el apéndice A encontrará un referencia básica del lenguaje.

Con esto terminamos el relevamiento sobre las técnicas de separación de *concerns* disponibles en la actualidad, en particular AOP. En los siguientes capítulos analizaremos cómo es posible integrar sensibilidad al contexto utilizando *aspect oriented programming*.

4. Framework Orientado a Aspectos para Aplicaciones Sensibles al Contexto

En este capítulo se describen los componentes que forman un *framework* orientado a aspectos para aplicaciones sensibles al contexto.

El objetivo del *framework* es adicionar comportamiento referido a la sensibilidad al contexto de la forma más transparente posible, para lograr mejorar aplicaciones existentes o desarrollar nuevas aplicaciones que en principio no incluyan la sensibilidad al contexto.

4.1. Estructura general

La arquitectura general se divide en tres capas que serán explicadas en las siguientes secciones (figura 4.1) .

La capa inferior contiene a la aplicación base que implementa los requerimientos funcionales del sistema. En esta capa podría encontrarse una aplicación desarrollada con anterioridad. La siguiente capa está formada por un conjunto de aspectos que se corresponden con los diferentes *concerns* que afectan el comportamiento de la aplicación base. La tercera capa define un modelo de contexto que incluye la posición espacio-temporal, los recursos del sistema, perfil del usuario, etc. Los aspectos que residen en la segunda capa modifican el procesamiento realizado en la aplicación base utilizando el modelo de contexto.

Un primer esquema del framework se presenta en la figura 4.1

En las próximas secciones veremos como se puede utilizar esta arquitectura para incluir distintos aspectos de la sensibilidad al contexto. En particular se analizará la sensibilidad a los recursos (*resource awareness*) y la personalización.

4.2. La Posición y los Recursos como Parte de la Sensibilidad al Contexto

Los recursos son una parte importante del contexto en el cual se desempeña una aplicación. La adaptación del comportamiento de la aplicación a los recursos disponibles en el sistema permite su utilización de manera eficaz, mejorando así la experiencia

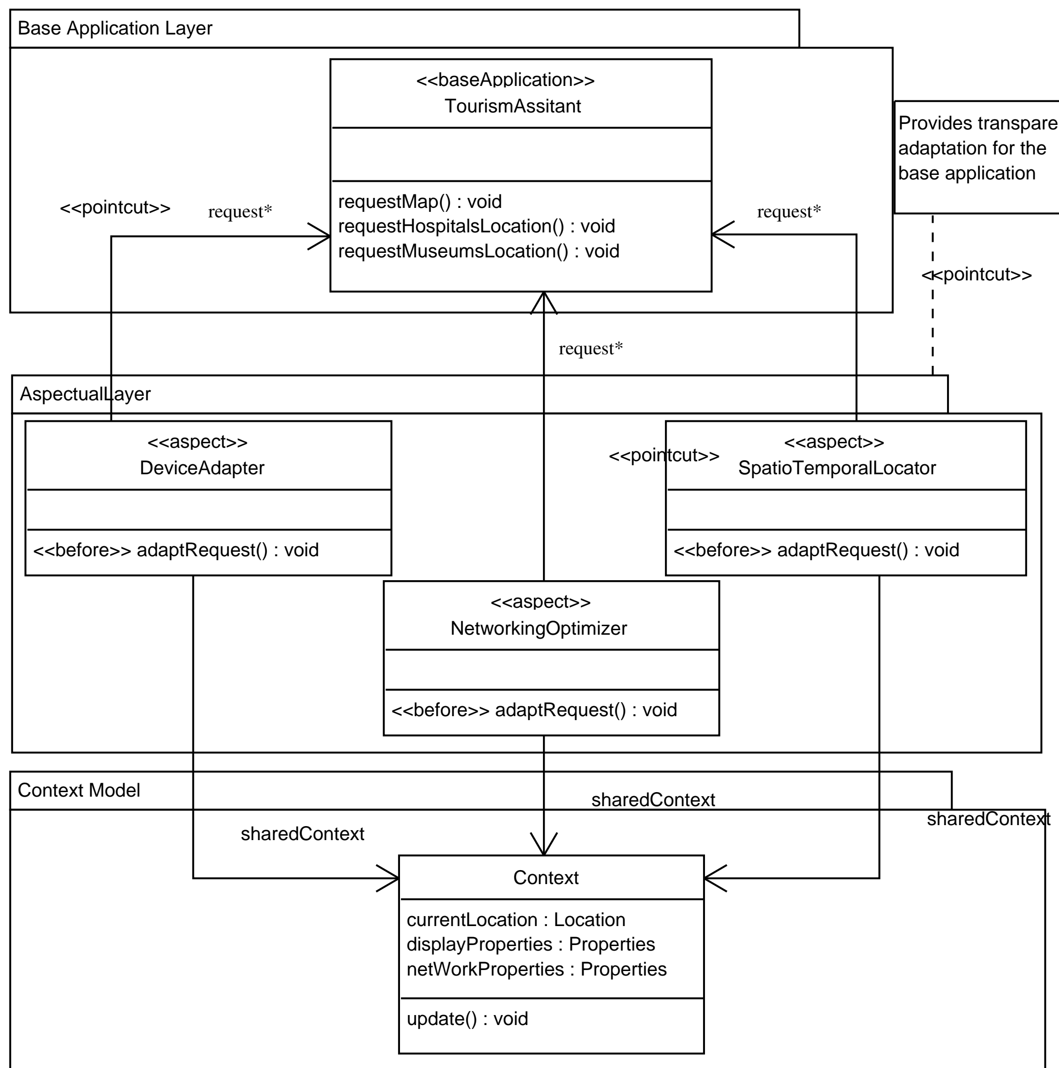

Figura 4.1. Arquitectura basada en aspectos para *context-awareness.*

del usuario. En esta sección veremos mediante un ejemplo cómo puede descomponerse una aplicación móvil en las capas de la arquitectura descriptas anteriormente.

4.2.1. Aplicación Base

Cualquier aplicación o sistema móvil existente puede ser considerado la aplicación base (*base program* en la terminología de orientación aspectos). La aplicación base implementa los requerimientos funcionales de una especificación para un sistema de software. Para ilustrar, tomemos la siguiente especificación de un sistema móvil:

> Se debe diseñar y desarrollar una aplicación que sirva de asistente para turistas. El objetivo de la aplicación es proveer información relevante respecto al lugar donde se encuentra el usuario, desde su posición geográfica hasta, por ejemplo, hoteles, restaurantes, museos, etc. Debido a la variedad de dispositivos móviles existentes, es deseable que la misma aplicación sea capaz de ejecutarse tanto en *laptops*, como en PDAS y teléfonos celulares de cuarta generación. El sistema debe funcionar aún cuando el ancho de banda disponible fluctúe o hayan perdidas esporádicas de conectividad. Existen muchos servidores, y la aplicación se conecta a aquel servidor más cercano en términos de su posición geográfica.
>
> Ya que el sistema se ejecutará en distintos ambientes con características cambiantes (ancho de banda, posición), con diferentes recursos de hardware a disposición (memoria, resolución de pantalla, mecanismos de entrada de datos, etc), el sistema debe ser capaz de adaptarse cambiando su comportamiento de acuerdo a las circunstancias.

Analizando la especificación planteada debemos decidir qué parte del sistema será el programa base, dónde se implementan los requerimientos funcionales, y cuáles son los *crosscutting concerns*.

La decisión sobre cómo modularizar el sistema tiene que ver con la descomposición del comportamiento, separándolo entre la parte que permanece fija y otras partes que requieren adaptación.

Por ejemplo, si el sistema debe mostrar imágenes de los restaurantes cercanos, podemos intuir que las imágenes se mostrarán de manera diferente de acuerdo a la resolución de la pantalla. Un teléfono celular podrá mostrar una imagen de menor de resolución que una *handheld*, sin importar la calidad original de la misma.

Otros tipos de adaptaciones son posibles: si el ancho de banda es menor que el ideal, la información puede comprimirse. Si la memoria del dispositivo se encuentra prácticamente llena, el servidor deberá enviar sólo información indispensable.

La funcionalidad principal de la aplicación es mostrar información al usuario. Por lo tanto, asumiremos que ésta formará nuestro programa base, cuyo comportamiento debe ser adaptado de acuerdo al contexto. Este *concern* de adaptación será aislado y encapsulado en aspectos.

4.2.2. Capa de Adaptación

Identificando Concerns de Adaptación

La funcionalidad principal de la aplicación puede resumirse como asistir al usuario durante sus recorridos. Los *concerns* que identificamos en este punto son:

1. La funcionalidad principal de la aplicación: asistente de turismo.

2. *Concern* de Visualización: afecta la forma en la que se muestr la información al usuario, por ejemplo si el dispositivo sólo permite mostrar texto en su pantalla, se deben realizar las adaptaciones necesarias para reemplazar imágenes por textos descriptivos. En el caso de poseer un *display* gráfico, este *concern* nos indica que las imágenes que deben ser mostradas escalándolas de acuerdo a la resolución y cantidad de colores soportados por el dispositivo.

3. *Concern* de comunicación: este *concern* tiene que ver con la optimización del recurso de red y el soporte a interrupciones de conectividad. Las adaptaciones en este *concern* incluyen compresión de datos, *caching*, etc.

4. *Concern* de administración de memoria: este *concern* afecta a los *requests* al servidor y a la información que debe ser guardada en el dispositivo. Si éste tiene poco espacio disponible es posible informarlo al servidor para que reduzca el tamaño de las respuestas a ciertos *requests*.

5. *Concern* espacio-temporal: este *concern* afecta la forma en la que se hacen los *requests* al servidor, dichos *requests* pueden ser adaptados para incluir metainformación espacio-temporal, que luego es utilizada por el servidor para brindar respuestas más apropiadas a los *requests* del usuario.

Definición de Aspectos para Adaptación

Asumiendo que estamos trabajando con un modelo orientado a objetos de la aplicación debemos decidir cuales serán los objetos que pertenezcan a la dimensión principal de la aplicación e identificar a aquellos *concerns* que atraviesan la aplicación. Debemos además, identificar los puntos de la aplicación base donde el comportamiento debe ser adaptado.

Dada la naturaleza del sistema, la mayoría de las actividades serán mapeadas a *requests* que resuelve algún servidor. De los *concerns* nombrados anteriormente podemos ver que los últimos cuatro afectan el comportamiento del primero. Podemos entonces pensarlos como aspectos[1], que modifican el funcionamiento de la aplicación base (ver Figura 4.1).

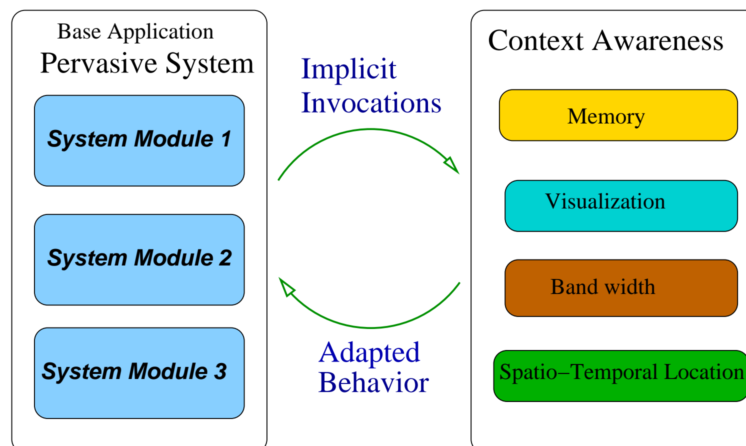

FIGURA 4.2. Distintos *concerns* que afectan a la aplicación base

Cada uno de estos *concerns* tendrá como efecto algún tipo de adaptación. Desde la perspectiva de los modelos de orientación a aspectos basados en la noción de *pointcut*, es necesario definir los puntos de la aplicación base donde se realizarán las adaptaciones. En la segunda capa de la Figura 4.1 se observan los aspectos de adaptación y sus relaciones con la aplicación base. Las relaciones estereotipadas como «*pointcut*» indican que el comportamiento del objeto al final de la relación es modificado por los *advices* definidos en cada aspecto. Para ejemplificar esta situación podemos imaginar el código de un *request* de la aplicación base como el que sigue:

Un método de la aplicación (listado 4.1) genera un *request* XML [41] pidiendo

[1]en el sentido de *aspect oriented programming*

```
1    public String request(){
2       return "<REQUEST>
3                   <USER_ID>232</USER_ID>
4                </REQUEST>";
5
6    }
```

Listing 4.1. XML request

```
1   public aspect BandwidthAspect {
2     String around(): call(BaseApplication.request(..)){
3     String request = proceed();
4     request+= "<BANDWIDTH_CONSTRAINTS>
5           <MAX_SIZE>" + context.currentBandwidth()
6            + "</MAXSIZE>
7           </BANDWIDTH_CONSTRAINTS>";
8     return request;
9     }
10   }
11  }
```

Listing 4.2. Implementación en AspectJ del aspecto Bandwitdh

algún tipo de información.

El aspecto que corresponde al *concern* de comunicaciones puede adaptar el formato del *request* para agregar información sobre el estado del ancho de banda. Esta información podrá ser utilizada por el servidor para generar una respuesta que sea transmisible en un tiempo razonable considerando velocidad disponible de la conexión. El listado de código 4.2 presenta una posible implementación (simplificada) de esta funcionalidad en AspectJ [42].

Nótese que la línea 2 define el *pointcut* que referencia al método `request()` en la aplicación base. Asociado a este *pointcut* se define un *advice* donde se implementa el comportamiento adaptativo. La cláusula `around` indica que la ejecución del *advice* se realiza reemplazando la ejecución del método original `request()`. En la línea 3 la palabra clave `proceed()` realiza una invocación del método original (`request()`). El resto del código muestra cómo se enriquece el XML original agregándole una restricción de ancho de banda[2]. Si existieran varios métodos generadores de *requests*, esta adaptación del comportamiento podría aplicarse a todos ellos escribiendo una expresión de *pointcut* que coincida con los mismos.

[2]Se trata de una simplificación con el fin de ahorrar espacio y no obscurecer el código, pues debería insertarse el bloque de XML extra antes del fin del request y no concatenarse.

De esta manera las adaptaciones correspondientes a cada *concern* pueden ser agregadas de forma independiente. Como resultado cada *concern* permanece encapsulado en un aspecto. Varias adaptaciones correspondientes al mismo *concern* deberían ser implementadas como diferentes *advices* en el mismo aspecto. Nuevos puntos de adaptación deberían ser definidos como *pointcuts* también dentro del mismo aspecto.

4.2.3. Capa de Modelo de Contexto

En el listado 4.2 línea 5 se hace referencia al objeto *context*. Dicho objeto representa el modelo de contexto en el cual se basan las decisiones para realizar adaptaciones.

Un modelo de contexto debe ser lo suficientemente flexible para permitir expresar los diferentes elementos que definen el ambiente de la aplicación.

El objetivo del modelo de contexto permitir a los aspectos acceder a la información que necesitan para determinar las adaptaciones a realizar. El modelo de contexto se encuentra encapsulado en una capa separada para facilitar su evolución y promover su independencia. Este no tiene que seguir ninguna estructura en particular. Puede ser expresado como una lista de tuplas indicando el valor de cada parámetro observado o ser un complejo modelo de objetos.

Obviamente los aspectos serán dependientes de la interfaz que provea el modelo de contexto, pero es importante destacar que no existe interacción desde el modelo de contexto hacia los aspectos ni a la aplicación base. De esta manera se logra una reusabilidad completa del modelo de contexto.

4.2.4. Agregando Información de Posicionamiento

Hemos visto que es posible enriquecer la aplicación con información sobre los recursos disponibles. Siguiendo el mismo ejemplo podemos extender la funcionalidad del sistema de manera que ahora los *requests* contengan información sobre la posición geográfica del usuario, de esta manera un servidor podría interactuar con un Sistema de Información Geográfica (GIS) para que las respuestas estén orientadas a la posición del usuario (ver listado de código 4.3).

Como ejemplo, una aplicación tipo asistente turístico ejecutándose en una PDA puede listar los restaurantes en una ciudad. Este *request* puede beneficiarse si agregamos la información sobre la posición actual de usuario. El servidor puede utilizar esto para devolver una respuesta que incluya a aquellos restaurantes cercanos en primer lugar.

```
1  <REQUEST>
2    <USER ID> 3232 </USER ID>
3     <CURRENT POS>
4      <LAT>20 20\' 21"</LAT>
5      <LONG>24 21\' 0"</LONG>
6     </CURRENT POS>
7  </REQUEST>
```

Listing 4.3. Request con el agregado de la posición geográfica.

```
1  <REQUEST>
2   <USER ID> 3232 </USER ID>
3   <POSITION INFO> <TYPE> MAP </TYPE>
4   </POSITION INFO>
5   <CURRENT POS>        <!-- Concern de posicionamiento-->
6   <LAT>20 20' 21"</LAT>
7   <LONG>24 21' 0"</LONG>
8   </CURRENT POS>
9   <IMAGE CONSTRAINTS>  <!-- Concern visualizacion -->
10    <WIDTH>320</WIDTH>
11    <HEIGTH>200</HEIGTH>
12  </IMAGE CONSTRAINTS>
13  <BANDWIDTH CONSTRAINTS> <!-- Concern comunicaciones -->
14    <MAX SIZE> 2KB </MAX SIZE>
15  </BANDWIDTH CONSTRAINTS>
16  <MEMORY CONSTRAINTS>    <!-- Concern administracion memoria -->
17    <MAX SIZE> 6MB </MAX SIZE>
18  </MEMORY CONSTRAINTS>
19 </REQUEST>
```

Listing 4.4. Request enriquecido por varios aspectos

Si además de esto nuestro aspecto de posicionamiento es capaz de censar continuamente la posición podrá deducir nuestra trayectoria y velocidad de movimiento. Esta información también puede incluirse para generar una respuesta acorde al desplazamiento que esta haciendo el usuario en ese momento.

Ahora bien, considerando algunos posibles aspectos que enriquecerán nuestro `request` original, como ser: el de ancho de banda, resolución del *display*, posicionamiento y memoria, el resultado final sería similar al mostrado en el listado de código 4.4

4.3. Personalización como Parte la Sensibilidad al Contexto

Como mencionamos en la sección 2.4.1 el usuario es parte del contexto de la aplicación y por lo tanto debe ser tenido en cuenta al definir los mecanismos que introducen la sensibilidad al contexto.

En esta sección veremos cómo la arquitectura propuesta permite integrar el proceso de personalización como parte de las adaptaciones de sensibilidad al contexto. Estos resultados fueron expuestos en [43].

4.3.1. Contexto: Definiciones y Trabajo Relacionado

Personalización Según Blom [44] se entiende por personalización al proceso que cambia la funcionalidad, interfaz, contenido de información, o que distingue a un sistema para incrementar su relevancia personal ante el usuario.

Para que las aplicaciones sean personalizables, deben conocer su contexto, el cual hace referencia a diferentes características relacionadas con el ambiente de ejecución como por ejemplo la información del usuario y sus preferencias.

Las aplicaciones personalizadas reflejan modelos de los objetivos, características, preferencias y conocimientos de cada usuario. Estos modelos son utilizados para mantener información actualizada sobre cada usuario (denominada perfil de usuario), con el objetivo de proveer servicios adaptados a sus preferencias para satisfacer sus necesidades [45].

Basándose en los perfiles de usuario y en otros elementos relativos al contexto y al dominio de la aplicación, las aplicaciones o servicios adaptan de forma autónoma y para cada usuario, tanto su presentación, como su organización y su gestión.

Entre los trabajos más importantes en el área de personalización se encuentra WUML (*Web Unified Modeling Language*) cuyo propósito es definir una metodología que soporte el desarrollo de aplicaciones web con especial atención en las de tipo omnisciente [46].

Los objetivos de WUML son:

- uso de UML como formalismo básico,
- extensión de UML utilizando únicamente el sistema de extensiones UML,
- especificación de WUML en términos de un framework genérico,
- separación en tres niveles del proceso de desarrollo (contenido, hiperbase, representación), y

- soporte para la omnisciencia (ubicuidad) a través de la personalización.

Este *framework* incluye en su modelo un *contexto* y *perfiles*, que proveen información de entorno, *reglas de personalización*, encargadas de monitorear cambios en el entorno, y a la aplicación propiamente dicha. La aplicación se divide en una parte *estable* que es la que se encarga de aquellas tareas que no dependen del entorno, y una parte *variable* para todo lo que sea dependiente del contexto, y en consecuencia sujeto de adaptación.

El *contexto* representa características pertenecientes al ambiente que cambian continuamente, como ser la ubicación, ancho de banda, etc. Además este contexto no es modificable por la aplicación.

Por otro lado existe el **perfil**, el cual es muy similar al **contexto** en relación a la información almacenada, pero en este caso esta información es mucho más estable. Esta información, por ejemplo, puede ser establecida explícitamente por el usuario. Para manipular estos datos, se utiliza por lo general el framework definido por el World Wide Web Consortium (W3C) denominado Composite Capabilities/Preference Profiles (CC/PP), el cual está basado en el *Resource Description Framework* (RDF) [47]. Este framework especifica las capacidades de los dispositivos y las preferencias de los usuarios, que utilizaran los servidores de aplicaciones para responder a las demandas de las aplicaciones.

Las *reglas de personalización* utilizan un mecanismo de eventos/condiciones/acciones para poder realizar la adaptación de la aplicación:

Eventos Los eventos determinan si es posible aplicar la regla de personalización.

Condiciones La condición es evaluada tan pronto se dispara el evento asociado y determina si es necesario realizar algún tipo de adaptación.

Acciones Son las responsables de activar algún tipo de adaptación en la aplicación

WUML es completo en cuanto a personalización pero no ataca la necesidad de modularizar las adaptaciones requeridas. Como dijimos anteriormente, estos trabajos están orientados principalmente al modelado de los datos. En el caso de WUML, aunque se modela comportamiento para poder adaptar aplicaciones, el código que realiza la adaptación esta embebido dentro de la aplicación, lo cual impide su reutilización o su aplicación en sistemas existentes, ya que esta adaptación no se realiza en forma transparente y debe ser explícitamente programada en la aplicación.

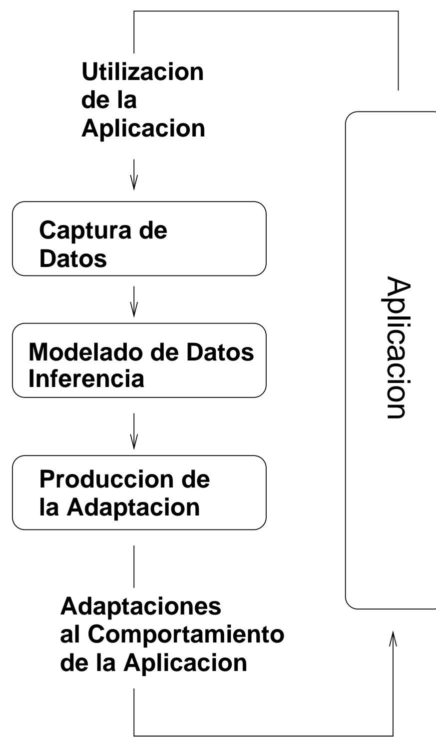

FIGURA 4.3. Ciclo de la adaptación para personalización.

4.3.2. Proceso de Adaptación

El proceso de adaptar la aplicación al usuario puede ser descompuesto en tres tareas principales (Figura 4.3):

1. **Adquisición de datos del usuario**: mediante la cual se identifican los datos disponibles sobre las características personales, de su comportamiento y del ambiente en que actúa, todo esto a través del monitoreo. Con esta información se construyen modelos iniciales de las preferencias del usuario.

2. **Representación del perfil de usuario e inferencia secundaria**: para expresar el contenido de los modelos apropiadamente. Se elaboran presunciones (secundarias) sobre el usuario y/o grupos de usuarios (usuarios con las mismas características), comportamiento y ambiente.

3. **Producción o adaptación**: que genera las adaptaciones de contenido, presentación y/o modalidad, que luego son introducidas en la aplicación.

Es interesante notar que, debido a las características antes mencionadas, al momento de adaptar una aplicación para que soporte la personalización de su comportamiento, deberemos introducir código en varios puntos dispersos en la aplicación. Esto hace muy dificultosa la tarea de transformar aplicaciones ya existentes para que sean personalizables.

Entre las acciones típicas utilizadas en la adaptación de la información presentada al usuario podemos mencionar:

Filtrado: Consiste en eliminar información o comportamiento que no es del interés del usuario.

Ordenamiento o Priorización: Se realiza mediante la reorganización de la información de forma acorde las preferencias del usuario.

Sugerencia: Consiste en realizar sugerencias *espontáneas* al usuario, presentando información o sugiriendo realizar tareas que se supone son de su interés.

4.3.3. Personalizando Mediante Aspectos

Considerando el carácter intrusivo de la adaptación necesaria para implementar la personalización, se hace necesario definir una arquitectura que permita la completa separación de los componentes de la aplicación en sí misma y los que agregan el

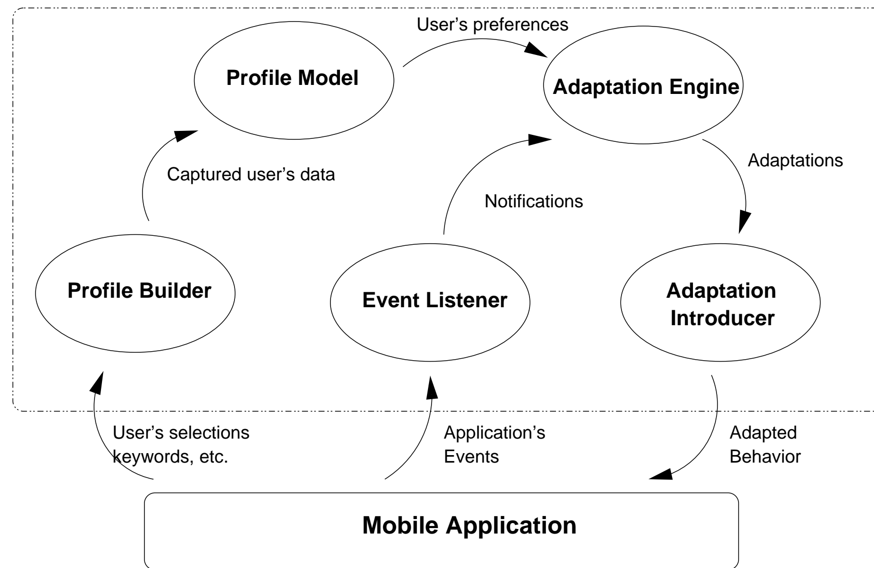

FIGURA 4.4. Separación entre la implementación de personalización y resto de la aplicación.

comportamiento adaptativo. Dicha separación sólo es beneficiosa si se complementa con un mecanismo de integración adecuado, que permita realizar la adaptación de manera transparente.

Lograr una arquitectura con estas características proporciona ventajas en términos de extensibilidad. El desacoplamiento entre ambos modelos hace posible que evolucionen de manera independiente. Por otro lado, la abstracción de las características de personalización permite diseñar la aplicación base concentrándose en su funcionalidad, dejando para una etapa posterior cuestiones accesorias (como la personalización).

El sistema en cuestión puede ser dividido en 2 partes. En primer lugar la aplicación en sí misma, que implementa los requerimientos funcionales del sistema; esta aplicación móvil es completamente operativa. Esta dimensión corresponde a aquellas partes que proveen la funcionalidad principal del sistema, sin tener en cuenta su costado personalizable. En segundo lugar, existe un meta-nivel, donde se encuentran materializadas las características propias de un sistema personalizable. Este meta-nivel representa la porción del sistema que es capaz de capturar las preferencias del usuario, almacenarlas de alguna manera, e intervenir en la ejecución normal de la aplicación móvil subyacente, adaptándola en función de dichas preferencias.

Es importante notar en este punto, que la aplicación base es completamente funcional independientemente de la personalización. Esto significa que no es necesario

ningún tipo de interacción desde la aplicación hacia el mecanismo de personalización. Esta independencia es beneficiosa desde el momento mismo del diseño, ya que hace posible diseñar el núcleo de la aplicación abstrayéndose de los detalles referentes a la personalización

Cabe destacar que una de las características novedosas de este enfoque consiste en colocar la funcionalidad relacionada con personalización del lado del cliente, es decir, en el dispositivo móvil. Esta característica brinda mayor robustez comparándola con una solución basada en personalización implementada en el servidor, que dependería de la disponibilidad de conexión. Dado que los mecanismos de captura y almacenamiento de información referida al perfil de usuario, así como el de adaptación están implementados en el cliente móvil, es posible enfrentar situaciones donde la conectividad no está disponible el 100 % del tiempo. Aunque el dispositivo móvil se encuentre *offline* es posible que el perfil del usuario y las características de la adaptación evolucionen.

Podemos modelar nuestro mecanismo de personalización mediante cinco componentes fundamentales (ver figura 4.4):

Profile Model Es la componente encargada del almacenamiento de las preferencias del usuario.

Adaptation Engine Encargado de inferir el tipo de adaptación que debería realizarse, basándose para esto en el profile model.

Profile Builder Esta componente esta formada por aspectos que interceptan ciertos puntos en la ejecución de la aplicación con el fin de alimentar el profile model.

Event Listener Componente constituido por aspectos que detectan la ocurrencia de determinados eventos en la aplicación que son potenciales disparadores de una acción adaptativa.

Adaptation Introducer Una vez determinado el tipo de adaptación que debe ser introducida en la aplicación, esta componente (un aspecto que modifica el comportamiento de la aplicación) es la encargada de modificar el flujo normal de control y agregar el comportamiento necesario para llevar a cabo la adaptación.

La Figura 4.5 muestra el mapeo entre las componentes de la arquitectura y un potencial diseño (simplificado) de una aplicación. La aplicación base es interceptada en aquellos métodos que tienen que ver con su utilización por parte del usuario

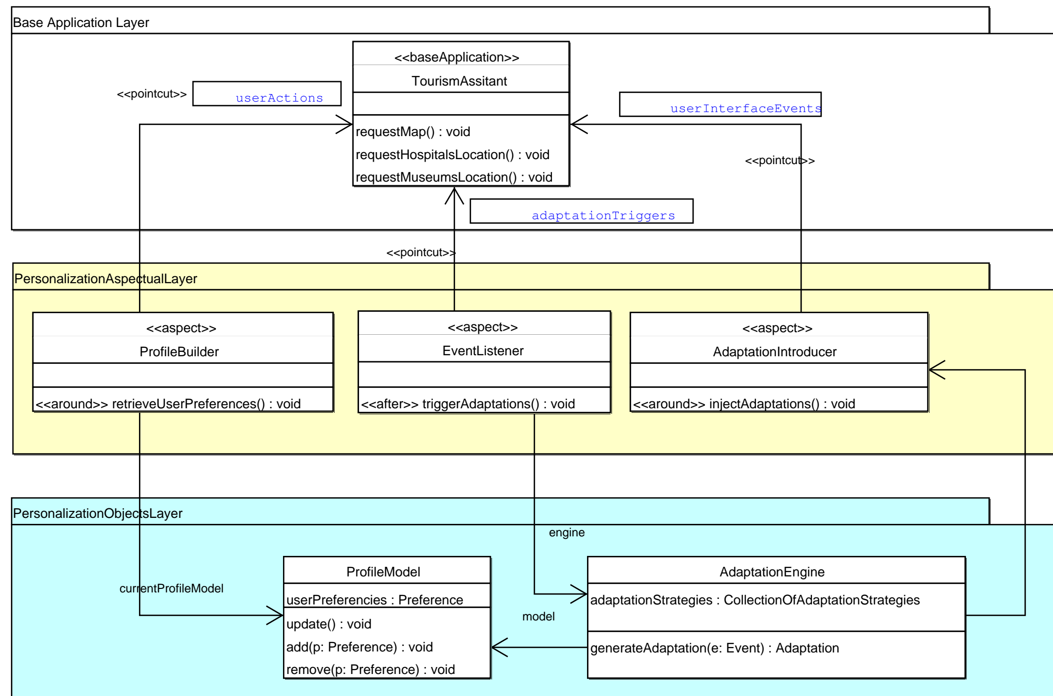

Figura 4.5. Mapeo de la arquitectura propuesta a objetos y aspectos.

(mediante la definición de puntos de enlace para estos métodos). Esta intercepción es realizada por el *ProfileBuilder*, el cual notifica las preferencias del usuario al *ProfileModel*. Por otro lado, el *EventListener* captura aquellos eventos que son potenciales disparadores de adaptaciones y los notifica al *AdaptationEngine*, el cual genera las adaptaciones del caso. Estas adaptaciones son introducidas en la aplicación por el *AdaptationIntroducer*, quien define puntos de enlace sobre las partes de la aplicación donde el comportamiento adaptativo puede ser introducido (en general se trata métodos relacionados con acciones del usuario y eventos de interfase).

4.3.4. Implementando Personalización mediante AOP

En esta sección mostraremos como una aplicación existente puede ser enriquecida mediante la integración de personalización.

La aplicación base en este caso es una agenda que permite mantener una lista de eventos. Este tipo de aplicaciones son de las más usadas en dispositivos móviles.

La aplicación permite agregar eventos a la agenda, editarlos o borrarlos. Además notifica al usuario cuando alguno de los eventos agendados está por ocurrir. Obviamente los eventos son persistentes.

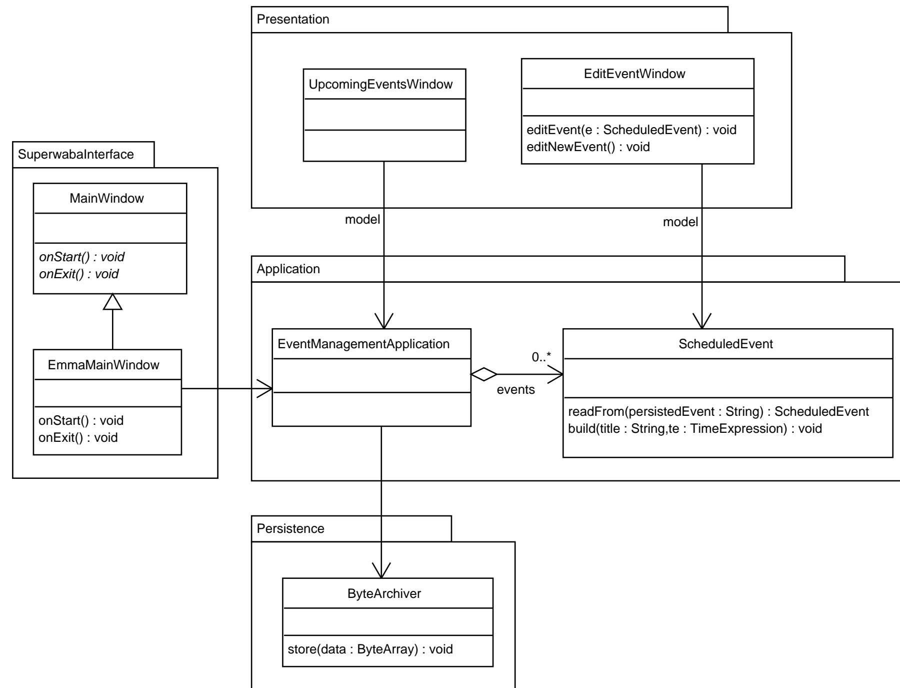

FIGURA 4.6. Diagrama de clases de la aplicación.

La figura 4.6 muestra el modelo de objetos subyacente. Allí se puede observar que la aplicación esta divida en 3 capas. La capa de persistencia, la de aplicación y la de presentación. En el diagrama cada capa ha sido encerrada en un paquete de UML. Debido a que esta aplicación ha sido implementada sobre la plataforma Superwaba [48] existen clases que permiten hacer la conexión con los servicios de dicha plataforma (paquete `SuperwabaInterface`).

Este es el escenario sobre el cual queremos implementar el siguiente comportamiento de personalización:

> Dado que en general un usuario ingresa un alto porcentaje de eventos que se repiten es deseable que el sistema *aprenda* o tome consciencia de los eventos más habituales. De esta manera cuando el usuario crea un evento para la agenda el sistema es capaz de agilizar la entrada de datos ofreciendo aquellos nombres de eventos más habituales.

Es evidente que esta nueva funcionalidad, que tiene que ver con conocer las actividades que el usuario ingresa más frecuentemente, no es parte del núcleo de la aplicación. Esta característica de personalización puede implementarse de forma

totalmente aislada de la aplicación. De hecho, veremos que no es necesario modificar en lo más mínimo el código de la aplicación original.

En primer lugar debemos plantearnos cómo queremos introducir el comportamiento deseado tomando como base el diseño abstracto propuesto en la sección 4.3.3.

Esta adaptación puede ser llevada a cabo siguiendo los pasos que se enuncian a continuación:

1. Captura de datos ingresados por el usuario, en nuestro ejemplo el nombre de los eventos.

2. Almacenamiento de los datos capturados en el modelo de personalización.

3. Intercepción en aquellos puntos donde se puede ofrecer la información almacenada en el modelo de personalización. En nuestro ejemplo, cuando se va a crear un nuevo evento.

4. Ofrecimiento de aquellos datos más comúnmente ingresados. En nuestro ejemplo se le presentan al usuario los nombres de eventos extraídos del modelo de personalización.

De los pasos enunciados se puede deducir que existirá al menos un aspecto que permita la captura de la información y el ofrecimiento de la misma. También será necesaria un interfaz de usuario en la cual se pueda realizar dicho ofrecimiento. Obviamente también es necesario un modelo de personalización que permita almacenar la información capturada y organizarla de tal manera que sea posible hacer ofrecimientos altamente eficaces. Además es posible que el modelo de personalización necesite persistir datos, por lo tanto podría hacer uso de la funcionalidad implementada en la capa de persistencia. La figura 4.7 muestra diseño resultante de integrar los conceptos enunciados.

En la figura 4.7 podemos observar, indicados mediante fondo sombreado, los paquetes que tienen que ver con el *concern* de personalización. Vemos que además del modelo de perfil del usuario y el aspecto que permite capturar datos y hacer sugerencias al usuario, existe un nuevo componente de interfaz gráfica que le permite al usuario seleccionar alguna (o ninguna) de las opciones ofrecidas. En este caso se trata de la clase ChoiceBox, que es un cuadro de diálogo donde el usuario puede seleccionar mediante un *combo-box*, una de varias opciones ofrecidas.

Es interesante notar que el modelo de perfil del usuario es completamente independiente de la aplicación y encapsula toda la lógica de adaptación. Por ejemplo es

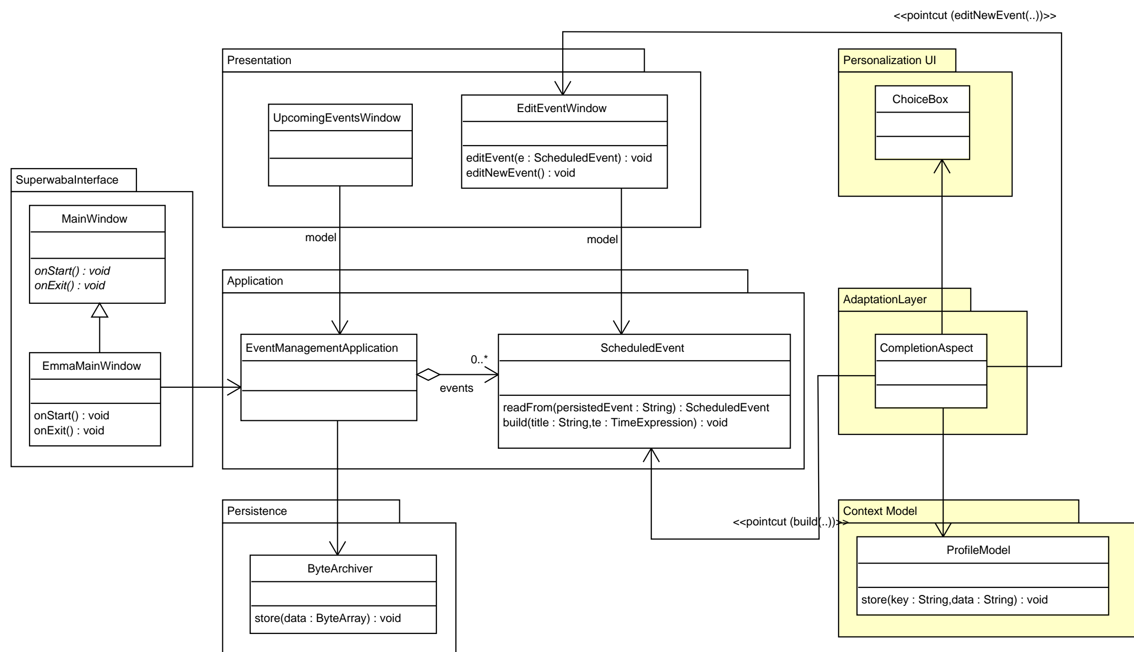

Figura 4.7. Diseño ampliado con aspectos

posible cambiar fácilmente la estrategia utilizada para construir el ranking de sugerencias. Distintas variantes de esta lógica podrían incluir:

- Ranking de los más usados.
- Ranking de los más recientemente usados.
- Variantes complejas como: asociar los nombres de los eventos a la hora del día en que se los ingresa. De esta manera el sistema cambiaría las sugerencias de acuerdo al momento en que ésta se realiza.

Veamos cómo podría implementarse la capa de adaptación de este ejemplo, en particular la captura de datos. En el listado de código 4.5 se puede ver el encabezamiento del aspecto. En particular nos interesa hacer notar que existe un *pointcut* que referencia la creación de objetos `ScheduledEvent`. Nótese que ese *pointcut* en particular permite capturar el evento (la instancia misma) que está siendo construido. En el correspondiente *advice* se usa el título del evento para alimentar al modelo de perfil del usuario.

Respecto al ofrecimiento de las sugerencias del sistema, este puede ser implementado interceptando el momento en el que el usuario va a ingresar un nuevo evento.

```
1
2  public privileged aspect CompletionAspect {
3      public static final String EVENT_TITLE = "Title"
4      // Intercepting event construction to capture event titles
5      pointcut schedEventConstruction(ScheduleEvent se):
6        call (private void buildEvent(ScheduleEvent)) && args(se);
7
8     after(ScheduleEvent se):schedEventConstruction(se){
9        Vm.debug("Captured " + se.getTitle());
10       profileModel.store(EVENT_TITLE,se.getTitle());
11     }
```

Listing 4.5. Aspecto para la captura de datos ingresados por el usuario.

```
1   ...
2   pointcut showingNewEventWindow(EditEventWindow window):
3    call(public void EditEventWindow.editNewEvent(..))
4         && target(window);
5
6   after(EditEventWindow window ):showingNewEventWindow(window){
7      Vm.debug("[Completion Concern]");
8      String[] options  = profileModel.suggestionsFor(EVENT_TITLE);
9      ChoiceBox.getInstance("Past Events",options).show();
10   }
11  ...
```

Listing 4.6. Aspecto para el ofrecimiento de sugerencias al usuario

Vemos en el listado de código 4.6 que se intercepta uno de los eventos de interfaz que indica la creación de un nuevo evento. En este punto se muestra una instancia de nuestro `ChoiceBox` con los elementos provistos por el `ProfileModel`.

4.4. Otros Aspectos de la Sensibilidad al Contexto

Hasta aquí hemos ilustrado tres subdominios de *context-awareness*:

- Posicionamiento
- Recursos
- Personalización

Ellos sirven para ilustrar la factibilidad del enfoque propuesto. Además de los expuestos existen diversas facetas del contexto que pueden ser incluidas en una aplica-

Figura 4.8. Aspecto original de la aplicación.

ción, como ser: el tiempo, la trayectoria del usuario y su dispositivo, otros dispositivos cercanos, etc.

Veamos por ejemplo la forma que toma una adaptación simple dependiente del horario:

> Dada una aplicación que se ejecuta en una PDA se desea que dicha aplicación cambie el conjunto de colores utilizados para su *rendering* de acuerdo a la hora del día, con el fin de favorecer la visibilidad.

Durante el día, cuando hay mucha luz disponible, es preferible utilizar fondos claros y letras obscuras, mientras que por las noches, cuando la luz escasea, un fondo obscuro con letras claras es más fácilmente legible.

La aplicación base en cuestión es una agenda de reuniones que permite hacer alta, baja y modificaciones de las mismas. Esta aplicación ha sido implementada originalmente usando Superwaba [48], y es necesario agregarle la funcionalidad mencionada anteriormente. La aplicación luce originalmente como se muestra en la figura 4.8.

La adaptación necesaria consiste en interceptar el flujo de control cuando se inicia la aplicación para lograr cambiar los colores de acuerdo al horario. De la misma manera pueden interceptarse otras acciones frecuentes para contemplar el caso de corridas largas, donde sea necesario realizar la adaptación ya no en el momento de carga de la aplicación sino durante su uso.

El aspecto que realiza la adaptación se encuentra detallado en el listado de código 4.7.

```
1  public aspect ColorAdaptation {
2
3  public static int intervalStart = 20;
4  public static int intervalEnd = 7;
5
6  pointcut start():
7      call(void lifia.app.emma.EmmaMainWindow.initialize());
8
9  before():start() {
10    Vm.debug("ColorAdaptation: Changing Colors according time");
11    Time now = new Time();
12    if(now.hour>intervalStart ||  now.hour<intervalEnd) {
13     lifia.gui.WindowAdapter.APP_BACK_COLOR = Color.BLACK;
14     lifia.gui.WindowAdapter.APP_FORE_COLOR = Color.WHITE;
15      }
16    }
17 }
```

Listing 4.7. Aspecto para adaptación del color según el horario

Podemos ver en el código cómo en la líneas 6 y 7 se define el *pointcut* que referencia el comienzo en la ejecución de la aplicación. Dentro del código del *advice* (lineas 9 a 16) se toma la hora actual y luego se la compara contra los valores que permiten determinar qué perfil de colores usar. Finalmente se cambian los colores utilizados para el *rendering* por toda la aplicación.

En la figura 4.9 podemos ver el resultado del *weaving* del aspecto con la aplicación original.

4.5. Conclusiones

Hasta aquí hemos visto como se puede organizar el comportamiento de una aplicación sensible al contexto basándonos en una arquitectura orientada a aspectos. En particular se han analizado dos escenarios:

- Sensibilidad al contexto referida a los recursos de hardware disponibles (resource -awareness) y a la posición geográfica.
- Personalización de aplicaciones de acuerdo al perfil de usuario.

En ambos casos se trata de *crosscutting concerns* pues afectan el modelo de la aplicación base en muchos puntos. En una aplicación orientada a objetos estos *concerns* no pueden ser modularizados ni aislados completamente de la aplicación.

FIGURA 4.9. Aplicación adaptada

En este capítulo se ha demostrado cómo la utilización de AOP brinda los medios para realizar diseños e implementaciones más modulares de aplicaciones sensibles al contexto.

5. Reuso de Aspectos y Sensibilidad al Contexto

La reusabilidad es una de las principales ventajas de las tecnologías de objetos y componentes. La reutilización de módulos brinda dos importantes beneficios:

- Disminuye los tiempos de desarrollo: pues no es necesario reescribir el código, simplemente se lo utiliza.
- Brinda mayor confiabilidad: pues el componente ya desarrollado y utilizado en otros contextos tiene como valor agregado un estado de maduración, ya que su funcionalidad ha sido ejercitada y corregida.

Por lo tanto es altamente recomendable maximizar la reusabilidad, aunque existe un costo de desarrollo extra asociado. Todo módulo es originalmente pensado para un uso en un contexto particular. Para reusar el módulo en otros contextos, este debe ser lo suficientemente genérico como para enfrentar su nuevo contexto. Lo mismo ocurre respecto de su funcionalidad, las posibles variantes de su funcionalidad deben ser contempladas durante su implementación, y los mecanismos para activar dichas variantes provistos para su efectiva utilización.

Considerando la importancia de la reutilización de componentes de software es conveniente comprender los efectos de la aplicación de *AOP* en el diseño e implementación de sistemas sensibles al contexto. En el Capítulo 4 se mostró cómo la separación entre el modelo de sensibilidad al contexto y la aplicación base provee los medios para reusar el modelo de *context awareness*, ya que éste es complementamente independiente de la aplicación base. Sin embargo, la capa de aspectos que permite conectar el modelo de sensibilidad al contexto con el resto de la aplicación y que define la manera en la que se materializan las adaptaciones, es muy dependiente de la aplicación adaptada, por lo tanto poco reusable.

En este capítulo presentaremos una serie de mecanismos que permiten lograr diferentes grados de reusabilidad de la capa de aspectos. Dichos niveles de reusabilidad serán introducidos mediante ejemplos concretos de aplicaciones móviles sensibles al contexto.

5.1. Trabajos Relacionados

Dantas [49] propone el uso de AOP para desacoplar la adaptabilidad de los sistemas móviles. Este trabajo presenta una comparación entre tres implementaciones del sistema: una basada en objetos sin patrones de diseño, otra con patrones y finalmente una basada en aspectos. En este aporte se afirma que la arquitectura propuesta promueve el reuso, pero no muestra ningún caso concreto que permita corroborarlo.

El trabajo de Yang [50] propone un proceso de dos fases para adaptar un sistema. La primera fase consiste en lograr un programa *adaptation ready*, esto se logra definiendo los potenciales puntos donde la adaptación será introducida. Estos puntos son instrumentados mediante aspectos y son usados por un kernel de adaptación, donde esta es definida mediante pares condición-acción. Dichos pares son definidos en la segunda fase. En este enfoque se asume que todos los puntos de adaptación pueden ser definidos por anticipado. Aún cuando el kernel de adaptación podría ser reusado en otras aplicaciones, no existe un análisis sobre cómo reusar los aspectos para adaptar otros programas.

5.2. Refinamiento de la Arquitectura

Como se mostró en el capítulo 4 la arquitectura propuesta está compuesta por tres capas principales.

1. Aplicación Base
2. Capa de Adaptación
3. Modelo de Contexto

El orden expuesto esta relacionado con el grado de especificidad. La aplicación base es una aplicación concreta y contiene todas las entidades características de su dominio. La capa de adaptación, formada por aspectos, observa y modifica el comportamiento de la aplicación base. El modelo de contexto es independiente de la aplicación base, y puede ser reutilizado a lo largo de varias aplicaciones.

En este capítulo nos concentraremos en describir diferentes maneras de reutilizar los aspectos de la capa de adaptación. Para esto es necesario diferenciar dos niveles de abstracción dentro de la capa aspectual de adaptación.

Como muestra la figura 5.1 la funcionalidad de la capa de adaptación se puede dividir en 2 niveles, un nivel que llamamos de "conectores"(*Connectors Layer*) que son aspectos muy ligados a la aplicación base y otro más abstracto que llamaremos

```
1  public abstract aspect GenericColorAdapter {
2   abstract pointcut start();
3   before():start() {
4   ...// adapting the original behavior
5   }
6  }
```

Listing 5.1. Aspecto abstracto definiendo la adaptación.

```
1  public aspect ColorAndTimeAdaptation
2                                extends GenericColorAdapter{
3   pointcut start():
4    call(void lifia.app.emma.EmmaMainWindow.initialize());
5  }
```

Listing 5.2. Conector definiendo un pointcut concreto.

Abstract Adaptation Layer, donde se define de forma genérica la lógica de adaptación. De esta manera esta lógica es independiente del lugar concreto donde será aplicada.

A nivel implementativo (utilizando AspectJ) esto se logra definiendo aspectos abstractos que poseen la lógica de adaptación en *advices* ligados a *pointcuts* abstractos. Luego, mediante la especialización del aspecto, se definen los *pointcuts* concretos que realizan el enlace real con el código base. El listado de código 5.1 muestra la definición de un aspecto que define el comportamiento de la adaptación ligado a un *pointcut* abstracto. Por otro lado, el listado 5.2 define el *pointcut* concreto donde se aplicará la adaptación.

5.3. Niveles de Reuso

Para cada uno de los niveles de reuso que se presentarán a continuación analizaremos el escenario, la solución propuesta, un ejemplo y las consecuencias que acarrea. Con el fin estudiar las distintas opciones de reuso para todos ellos asumiremos que:

- La capa de modelo de contexto ya se encuentra implementada.
- Para cada *concern* existe un aspecto abstracto que implementa la lógica de adaptación correspondiente en *Abstract Adaptation Layer*.
- Si una adaptación ya fue aplicada en alguna aplicación base al menos existe un conector que permite aplicar concretamente dicha adaptación.

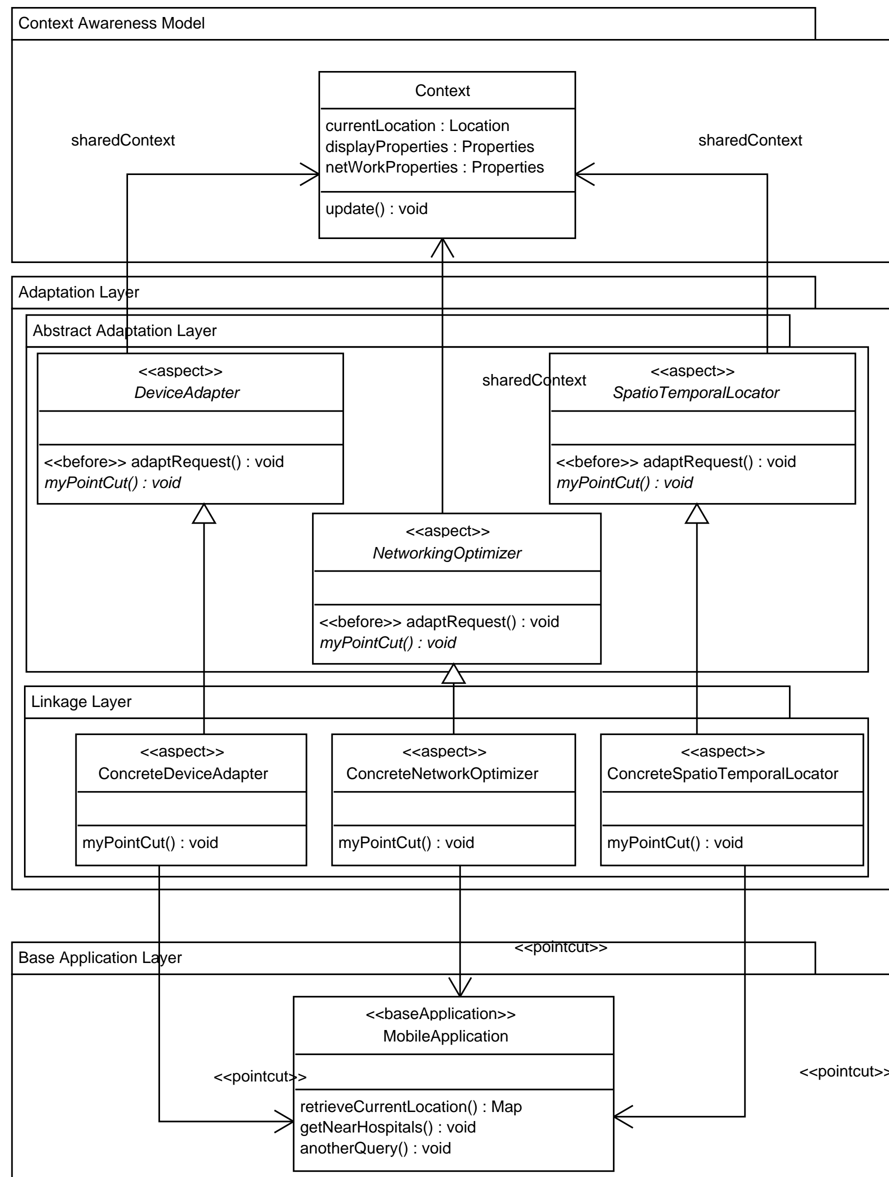

Figura 5.1. Arquitectura orientada a aspectos refinada

5.3.1. Primer Nivel: Definiendo Conectores en *Linkage Layer*

Escenario

En este nivel asumiremos que existe una aplicación base que en principio no es sensible al contexto y a la cual deseamos agregarle dicha funcionalidad.

Si el o los *concerns* a los cuales pertenece la funcionalidad en cuestión ya se encuentran implementados y fueron utilizados alguna vez, existirá al menos un aspecto en *Abstract Adaptation Layer* cuyos *pointcuts* abstractos podrían ser redefinidos de manera tal capturen los *join points* que provee la nueva aplicación.

Ejemplo

Supongamos un sistema capaz de ejecutarse en un dispositivo móvil, por ejemplo Agendus [51] (Figura 5.2), donde una de las funciones del sistema es mantener una lista de contactos.

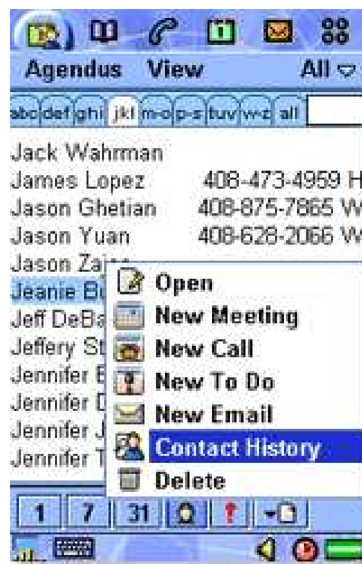

FIGURA 5.2. Lista de contactos de Agendus

Dado este sistema consideramos necesario agregarle sensibilidad al contexto que permita al usuario saber cuales de sus contactos se encuentran cerca en términos geográficos.

La aplicación base cuenta con la información de los contactos. El modelo de contexto permite saber cual es nuestra posición y la de otros dispositivos cercanos

```
1   public abstract aspect ReportNearObjects {
2    abstract pointcut appEventReportNearObjects();
3    after():appEventReportNearObjects() {
4       for (Iterator objects = getInterestingObjects();
5                                objects.hasNext();) {
6          GeoObject obj = (GeoObject) objects.next();
7          if (context.distanceTo(obj) < threshold) {
8             informNearObject(obj)
9          }
10      }
11   }
12
13  //Subaspects must define the appropiate
14   //notification mechanism
15   public abstract void informNearObject(Object o);
16
17  //Subaspects must define how to get the list
18   //of object to query distance.
19   public abstract Collection getInterestingObjects();
20
21 }
```

Listing 5.3. Aspecto abstracto para la notificación de objetos cercanos

mediante algún mecanismo de posicionamiento como GPS.

Un aspecto abstracto que notifica la presencia cercana de objetos interesantes podría ser definido como se muestra en el listado 5.3. Nótese que, en la línea 2, se define un *pointcut* abstracto que indica cuando debe censarse a los objetos cercanos. En la siguiente línea se declara el *advice* de manera tal que el código del mismo sea ejecutado luego (palabra clave *after*) de alcanzar un *join point* capturado por el *pointcut* abstracto. El código del *advice* (líneas 3 a la 11) expresa cómo se lleva a cabo la adaptación. En el caso de este ejemplo se itera en una colección de objetos geográficos y se consulta la distancia. Nótese que la manera en la que se obtienen los objetos en cuestión es dependiente de la aplicación adaptada, y en este caso queda definida como un método abstracto (línea 19) del aspecto que deberá ser redefinido cuando se haga el *binding* con una aplicación concreta. Dentro del `loop` se testea por cada objeto de la lista si la distancia a él es menor que un umbral configurable. Si es así, se debe notificar a la aplicación o al usuario. Una vez más este comportamiento es propio de la aplicación en particular, así que es expresado en el aspecto abstracto como un método abstracto (línea 15).

Habiendo definido la forma que tendría un aspecto abstracto en la capa de adaptación AAL (*Abstract Adaptation Layer*) podemos examinar la aplicación que deseamos

```
1   public class ContactManager{
2    ...
3    public Collection getContacts(){
4     return contacts;
5    }
6    public void performAction(){
7    ...
8    }
9   }
10  public class ContactManagerUI{
11   public void displayMesssage(String message)
12   {
13        ....
14   }
15   ...
16  }
```

Listing 5.4. interfaz de la aplicación base

adaptar con el fin de diseñar el *binding* más apropiado.

La clase `ContactManager` (listado 5.4) es la encargada de mantener la lista de contactos en nuestra aplicación. El método `getContacts()` retorna una lista de objetos propios del dominio de la aplicación, en este caso podríamos suponer instancias de la clase `Contact`. En la misma clase, el método `performAction` representa en nuestro ejemplo parte de la lógica de la aplicación que se ejecuta periódicamente. Finalmente, existe una clase que modela la interfaz con el usuario. De dicha clase sólo nos interesa saber que existe un método que permite mostrar un mensaje.

Hasta aquí hemos analizado el código existente, tanto desde el punto de vista del *framework* aspectual como de la aplicación. En este punto debemos definir los conectores que permiten hacer el *binding* del aspecto con la aplicación.

El listado de código 5.5 muestra cómo se especializa el aspecto abstracto del *framework*, haciendo el mapeo entre las abstracciones de la dimensión de *context awareness* y los conceptos propios del dominio de la aplicación adaptada.

Analizando el código en 5.5, vemos que en la línea 2 se redefine el `pointcut` que en el super-aspecto había sido declarado como abstracto. En este ejemplo el nuestro *pointcut* abstracto se mapea al mensaje *performAction()* de la clase *ContactManager*. Nótese que el *pointcut* captura todos los mensajes cuyo nombre sea `performAction`, sin importar la cantidad ni tipos de los parámetros. Alternativamente, podría declararse el *pointcut* de manera que capture todos los mensajes de la clase *ContactManager* como se muestra en el listado 5.6.

```
1  public aspect ReportNearContacts extends ReportNearObject{
2   pointcut appEventReportNearObjects:
3    call(void ContactManager.performAction(..));
4
5   public void informNearObject(Object){
6    contactManagerUI.displayMessage(
7       "Contact " + object + " is nearby.");
8   }
9
10  public Collection getInterestingObjects()
11   {
12    return contactManager.getContacts();
13   }
14 }
```

Listing 5.5. Definición de un conector

```
1   pointcut appEventReportNearObjects:
2                call(* ContactManager.*(..));
```

Listing 5.6. Definición alternativa del pointcut

El método *getInterestingObjects()* se implementa retornando la lista de contactos de nuestra aplicación. Si fuera necesario que los objetos retornados en la colección contengan cierta funcionalidad particular que no sea propia del dominio de la aplicación, en este punto se podría decorar[1] a los objetos con el comportamiento necesario (ver listado 5.7).

En el listado de código 5.7 vemos como los contactos que a nivel de dominio de la aplicación no contienen funcionalidad de posicionamiento geográfico, son decorados con dicha funcionalidad. La clase `GeoObject` envuelve a un objeto cualquiera agregándole posicionamiento y las operaciones topológicas propias de los objetos geográficos. La línea 6 de la misma figura muestra cómo se crean instancias de GeoObject para cada contacto.

Finalmente, es necesario definir el método *informNearObject*, para esto su implementación utiliza funcionalidad existente en la aplicación de base, de manera que la integración sea completa, inclusive desde la perspectiva de la interfaz de usuario.

[1] Con el verbo decorar nos referimos a la aplicación del patrón de diseño *Decorator* [3]

```
1   public Collection getInterestingObjects()
2    {
3     ArrayList geoObjects= new ArrayList();
4     for (Iterator i = contactManager.getContacts().iterator;
5                                                       i.hasNext();)
6            geoObjects.add(new GeoObject(i.next()));
7     return contactManager.getContacts();
8    }
```

Listing 5.7. *Wrapping* de objetos del dominio

Solución

Dependiendo del soporte de aspectos (lenguaje o ambiente) que se este utilizando la forma de materializar la redefinición de un *pointcut* puede variar. En el caso particular de AspectJ esta operación se realiza heredando del aspecto abstracto y redefiniendo los *pointcuts* abstractos de manera tal que capturen los *join points* de nuestra aplicación.

Consecuencias

En este primer nivel se maximiza el reuso, puesto que estamos asumiendo dos condiciones claves para su éxito:

- Que el tipo de adaptaciones que deseamos incluir en la aplicación se encuentra predefinidas en la capa AAL (*Abstract Adaptation Layer*).
- Que los *bindings* que define y requiere dicha estrategia de adaptación pueden mapearse sin mucho esfuerzo a los conceptos subyacentes en la aplicación base.

Estas dos premisas no siempre se cumplirán, existen casos donde la adaptación no fue prevista y debe ser incluida como una nueva estrategia de adaptación, esto implica extender el *framework* de aspectos incluyendo la nueva funcionalidad. Por otro lado, es posible que los puntos de *binding* de ciertas adaptaciones no puedan mapearse a la aplicación que deseamos adaptar, como en los ejemplos que se presentan en la sección 5.3.2. En todos estos casos es necesario modificar el *framework* de aspectos, escapando al primer caso de reuso expuesto en esta sección.

El primer nivel de reuso permite reutilizar el modelo de sensibilidad al contexto y las estrategias de adaptación, por lo tanto no requiere alterar la aplicación base.

Únicamente es necesario definir el *binding* entre la estrategia de adaptación y la aplicación adaptada.

Primer nivel de reuso: reutilizamos el modelo de contexto y las estrategias genéricas de adaptación. Requiere definir los conectores.

5.3.2. Segundo Nivel: Definiendo Conectores y Estrategias de Adaptación

Escenario

Como se mencionó en 5.3.1 existen ciertos casos donde la definición de conectores entre la aplicación base y las estrategias de adaptación es insuficiente para lograr la funcionalidad deseada. A continuación describimos algunas situaciones problemáticas que pueden presentarse:

- **Nuevos Concerns:** Cuando es necesario modelar y generar la adaptación referente a un *concern* que no había sido previsto, es necesario modificar la capa de adaptación (AAL) incluyendo uno o más aspectos.
- **Firmas de los métodos no coinciden con la forma de los *pointcuts* abstractos:** Dependiendo del lenguaje de implementación, puede ser necesario proveer parámetros para los *pointcuts* abstractos. No puede asegurarse que dichos parámetros estarán disponibles definiendo conectores, ya que se trata de información de contexto que no esta disponible en todas las aplicaciones, o que puede estarlo en diferentes formas. En estos casos es necesario definir nuevos *pointcuts* abstractos acordes a los parámetros que pueden ser extraídos de la aplicación base. Esto a su vez puede involucrar la introducción de ciertas variantes en el código original de la estrategias de adaptación.
- **No existe el método apropiado para hacer el binding:** Si una estrategia de adaptación fue implementada para trabajar sobre un determinado tipo de *join point*, puede ocurrir que no exista un mapeo para tal *join point*. Imaginemos que un aspecto que optimiza la utilización del ancho de banda de red (llamémoslo *Network Optimizer Aspect*) realiza dicha optimización comprimiendo los *streams* de datos que salen del dispositivo y descomprimiendo a aquellos que llegan al mismo. Ahora supongamos que el aspecto fue pensando de manera que, al especializarlo, sea ligado a un método de la aplicación base que realiza el envío de datos (*poincut* `sendData`, ver listado 5.8). Al intentar adaptar una

```
1  public abstract aspect NetworkOptimizer
2   abstract pointcut sendData(String data);  // the pointcut
3   Object around(String data):sendData(data)  {
4    compressedData = compress(data);// encrypts original data
5    //sends the compressed data
6    compressedResponse = proceed(compressedData);
7    return decompress(compressedResponse);
8   }
```

LISTING 5.8. Aspecto para optimización del uso del ancho de banda mediante compresion.

nueva aplicación nos encontramos con que no existe manera de mapear el *pointcut* `sendData` porque la aplicación realiza el envío de datos mediante una serie de sentencias más primitivas que no han sido correctamente factorizadas, por ejemplo mediante una secuencia invocaciones a los métodos `openSocket()`, `writeData(byte[])` y `closeSocket()` que se encuentra entremezclada con otras sentencias. Es claro que el mapeo de `sendData` no puede hacerse y será necesario, si no se quiere refactorizar la aplicación base, alterar la estrategia de adaptación o crear una nueva.

- **El orden de invocación de mensajes no es el esperado:** Los aspectos pueden tener estado interno; dependiendo de cual sea el propósito y la manera en que fue implementado, su estado puede depender del orden de ejecución de los *join points*. Si un aspecto *stateful* es empleado sobre una aplicación donde los *join points* se invocan en un orden diferente, pueden ocurrir inconsistencias en el aspecto. En este caso es necesario rediseñar y reimplementar el aspecto en cuestión para adaptarlo al nuevo flujo de control de la aplicación, o bien, hacerlo *stateless*.

Ejemplo

Un ejemplo típico en la bibliografía de computación móvil y *context-awareness* es el campus universitario. En el campus los estudiantes cuentan con dispositivos móviles que les permiten saber dónde y cuándo se dictarán las clases a las que deben asistir. También es posible enviar mensajes a otros dispositivos dentro del campus. Imaginemos que una primera versión del sistema no se provee seguridad en las comunicaciones entre dispositivos, pero dada la vulnerabilidad de la redes inalámbricas se hace necesario asegurar la privacidad de alguna manera.

```
1   public abstract aspect DataCipher{
2    abstract pointcut sendData(String data);  // the pointcut
3        Object around(String data):sendData(data){
  // advice definition
4          encryptedData = encrypt(data)  // encrypts original data
5          //sends the encrypted data
6          encryptedResponse = proceed(encryptedData)
7          decryptedResponse = decrypt(encryptedResponse)
8          /* reads the response */
9          return decryptedResponse; //returns decrypted response
10       }
```

Listing 5.9. Aspecto para encriptación de datos.

```
1   public aspect CampusDataCipher extends DataCipher {
2          pointcut sendData(String data):
3                            call ( * pkg.CampusApplication.
4                                           sendData(String))
5                            && args(data);
6   }
```

Listing 5.10. Aplicación del aspecto de encriptación de datos.

En este caso el *framework* no contemplaba adaptación alguna para la segurización de comunicaciones. Por lo tanto, es necesario definir una nueva estrategia de adaptación que provea seguridad a los usuarios. Para esto decidimos encriptar los mensajes antes de ser enviados. Definimos entonces el aspecto `DataCipher` como se muestra en el listado 5.9.

Una aplicación concreta de este aspecto puede observarse en el listado de código 5.10.

Solución

El segundo nivel de reuso está compuesto por aquellos casos que involucran cambios en la capa *Abstract Adaptation Layer*. Los cambios introducidos en esta capa permiten crear nuevas adaptaciones para *concerns* no previstos como también adaptar las estrategias existentes creando nuevas versiones de ellas que puedan ser utilizadas en contextos diferentes al originalmente pensado.

Consecuencias

La definición de nuevas estrategias de adaptación es una tarea compleja que responde a una serie de factores:

- Es necesario entender el flujo del control en la aplicación base.
- El desarrollador debe diseñar e implementar la estrategia de adaptación.
- El desarrollador debe desarrollar por adelantado *pointcuts* abstractos adecuados de manera tal que obtenga una adaptación reusable. Como ocurre en la orientación a objetos, encontrar el nivel apropiado de abstracción es un punto clave para conseguir módulos reusables. Un nivel excesivo de abstracción hace el software tan genérico que no puede ser usado rápidamente; por otro lado, un nivel pobre de abstracción reduce notablemente su campo de aplicación.

En el segundo nivel reutilizamos el modelo de contexto, no alteramos la aplicación base pero nuevas estrategias de adaptación y los correspondientes conectores deben ser definidos.

5.3.3. Tercer Nivel: Definiendo Nuevas Aplicaciones

Escenario

El escenario para este caso es el siguiente: las capas que definen el modelo de contexto y las adaptaciones se encuentran desarrolladas y deben ser reusadas para aplicar su comportamiento a un nuevo sistema que será construido desde cero. Es posible que inclusive los conectores se encuentren definidos. Por lo tanto, el desarrollador de la nueva aplicación debe proveer los *join points* adecuados para que los aspectos puedan aplicarse.

Ejemplo

Para este ejemplo asumiremos que las capas de modelo de contexto y la de adaptación proveen todo el comportamiento deseable para conseguir una aplicación sensible al contexto. En este caso el comportamiento adaptativo tiene que ver con el tipo de dispositivo, optimizaciones en el uso de la conexión de red y adaptaciones de acuerdo la posición espacio-temporal. Además, en la capa de enlace existen conectores

definidos cuyos *pointcuts* capturan los *join points* de un conjunto de interfaces que deben ser implementadas por las clases. Esto quiere decir que todas las adaptaciones pueden ser usadas mediante la implementación de ciertas interfaces.

Imaginemos que la interfaz `MobileApplication` define los mensajes `sendMessage()`, `receiveMessage()` y `contactPeople()`. También existen conectores y aspectos que adaptan el comportamiento de estos mensajes de acuerdo al contexto, como se muestra en la Figura 5.3

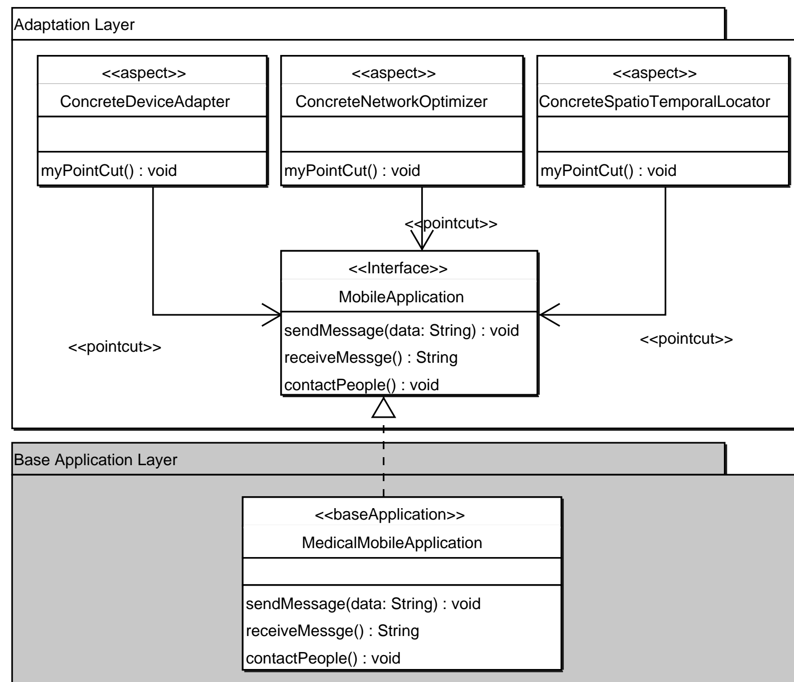

FIGURA 5.3. Estructura de reuso mediante interfaces

Si alguna clase de nuestra nueva aplicación implementa la interfaz antes mencionada obtendrá automáticamente la adaptación de su comportamiento.

En el caso de necesitar eliminar ciertas adaptaciones que no son deseadas, es posible hacerlo excluyendo los aspectos que las implementan del proceso de *weaving*.

Solución

Para la definición de estos puntos de enlace existen dos posibilidades.

1. Definir en la capa de enlace (*Linkage Layer*) un conjunto de interfaces, las cuales son afectadas por los conectores. De esta manera la nueva aplicación

deberá implementar las interfaces para lograr el comportamiento enriquecido con sensibilidad al contexto.

2. Definir ciertos estándares en lo que respecta a la codificación y la manera de nombrar los métodos.

Consecuencias

Una de las ventajas de la primera opción es que se establece un **contrato** entre los aspectos y los objetos afectados. Las interfaces definen un protocolo que debe ser respetado por los objetos y asegura a los aspectos la existencia de los *join points* necesarios. Por otro lado, esto puede representar una desventaja para el desarrollador de la aplicación ya que se verá forzado a definir muchos métodos que originalmente no existirían, lo cual es objetable. Para minimizar este efecto indeseado, es importante definir interfaces "pequeñas" y muy específicas, de manera tal que el desarrollador puede seleccionar aquellas en las que está interesado.

Una variante de esta primera opción consiste en definir un conjunto de clases abstractas implementando las interfaces predefinidas. Estas clases proveerían implementaciones por defecto para todos los métodos de la interfaces; y el desarrollador de la nueva aplicación debería subclasificar de dichas clases abstractas redefiniendo aquellos métodos donde le interese se aplique la adaptación provista por la capa aspectual.

La definición de estándares para la codificación libera el desarrollador de la obligación de definir todos los métodos declarados en las interfaces. Estos estándares definen el nombrado de los métodos siguiendo ciertos patrones que también son utilizados para definición de los *poincuts*. Estándares de este tipo existen desde el comienzo de la comunidad Java

Por ejemplo, en el estándar JavaBeans [52], los métodos que permiten recuperar o especificar el contenido de las propiedades (*getters* y *setters*) de un *java bean* se deben nombrar respetando la siguiente convención:

- `SomeType getSomething()` para los métodos que permiten recuperar el valor de una propiedad.

- `void setSomething(SomeType st)` para los métodos que establecen el valor de una propiedad.

Los prefijos `get` y `set` son parte de la convención para el nombrado de métodos de los *java beans*. Siguiendo esta línea podríamos definir reglas para el nombrado de los

métodos a los cuales queremos que se les adapte el comportamiento. El punto débil de este mecanismo de implementación es que pueden existir casos donde un mismo método debe ser adaptado de varias manera a la vez, y generar reglas para estos casos es inherentemente complicado. Y, aunque pudiera hacerse, el código resultante sería difícilmente legible.

En el tercer nivel de reuso creamos una aplicación utilizando el modelo de contexto, adaptaciones y conectores existentes, esto se logra implementando interfaces o siguiendo convenciones en el código de la aplicación base.

5.4. Conclusiones

En este capítulo se han analizado varias situaciones donde se agrega sensibilidad al contexto a aplicaciones móviles usando orientación a aspectos. Estas situaciones caracterizan escenarios de reuso que surgen usualmente cuando se aplica AOP.

En este trabajo los hemos llamado niveles de reuso. Cada nivel de reuso estipula en qué condiciones puede darse, y los cambios que deben realizarse en el framework de sensibilidad al contexto o los lineamientos a seguir en la construcción de la aplicación.

La tabla 5.1 muestra para cada uno de los niveles, las capas afectadas por los cambios.

	Nivel 1	Nivel 2	Nivel 3
BMAL	□	□	■
LL	■	■	□
AAL	□	■	□
CAL	□	□	□

Cuadro 5.1. Comparasión entre los niveles de reuso. Negro indica cambio en la capa.

En este capítulo se han presentado casos de reuso de manera bastante informal. Es necesario desarrollar formalismos que permitan discernir fácilmente en qué casos se puede reusar el comportamiento definido en los aspectos y a qué costo. Una importante mejora consistiría en definir mecanismos que permitan la notación del flujo

del control de manera tal que podamos clasificar aplicaciones por su similitud en este sentido. De la misma manera estos descriptores podrían aplicarse a los aspectos indicando en que categoría de aplicaciones éstos son reusables.

6. Conflictos entre Aspectos Sensibles al Contexto

Hasta aquí hemos visto que es posible utilizar AOP para separar los *concerns* relacionados con la sensibilidad al contexto del resto de la aplicación.

En este capítulo estudiaremos casos en los que se presentan conflictos entre aspectos debido a su accionar descoordinado.

Al estructurar una aplicación móvil con *middleware* basado en aspectos corremos el riesgo de que ellos compitan por los ciertos recursos compartidos. Aunque estos recursos no sean de uso exclusivo, su utilización descoordinada puede llevar a la aplicación a situaciones de inestabilidad.

Los conflictos son comunes en el dominio de *resource-awareness* [53]. En el caso de *aspect oriented programming* estos conflictos implican interacciones que tienen su origen en la semántica de los aspectos, es decir, sus objetivos y la forma en la consumen los recursos necesarios para lograrlos. Estas interacciones no pueden ser detectadas mediante análisis sintáctico de los *pointcuts*, por lo tanto no pueden determinarse en tiempo de compilación. En muchos casos dependen de situaciones en tiempo de ejecución que hasta podrían no darse en varias corridas de la aplicación.

En este capítulo presentaremos ejemplos de estas situaciones y un enfoque para resolverlas en el marco de este trabajo.

6.1. Conflictos Aspectuales

En el contexto de un *middleware* para aplicaciones móviles sensibles al contexto, varios aspectos adaptan el comportamiento de la aplicación a las condiciones del ambiente para asegurar el uso eficiente de los recursos disponibles. Este enfoque trae aparejados los beneficios mencionados en el capítulo 4.

Teniendo en cuenta que diversos aspectos modelan e implementan el comportamiento referido a la optimización de diferentes recursos es esperable que ellos sean desarrollados de forma independiente. De la misma manera, siguiendo el principio de *obliviousness* [33], podemos inclinarnos a pensar que el programa base es completamente independiente de los aspectos y estos, a su vez, son completamente independientes entre sí, asumiendo que al integrarlos en una aplicación no deberían surgir interacciones entre ellos. Analizando la situación podemos intuir que cualquier aspec-

to afecta a los recursos del sistema de una u otra manera. Aún un concern "inocente" como el *logging* podría afectar negativamente a la aplicación base u otros aspectos, si consume grandes cantidades de memoria o ciclos de CPU.

Podemos afirmar que cualquier aspecto va a consumir recursos, al menos los ciclos de procesador necesarios para su ejecución.

En ciertos casos, esto no sería un problema, pero en el contexto de la computación móvil, donde los recursos son sumamente escasos, la utilización descoordinada de los mismos puede convertirse en un asunto serio.

La figura 6.1 muestra un esquema abstracto donde dos aspectos tienen efectos inversos sobre los recursos, como ilustraremos más adelante, este comportamiento descoordinado puede llevar al sistema a una situación peligrosa. Las flechas sólidas indican el efecto deseado del aspecto sobre un recurso, las flechas punteadas indican un efecto lateral de la aplicación del aspecto sobre un recurso.

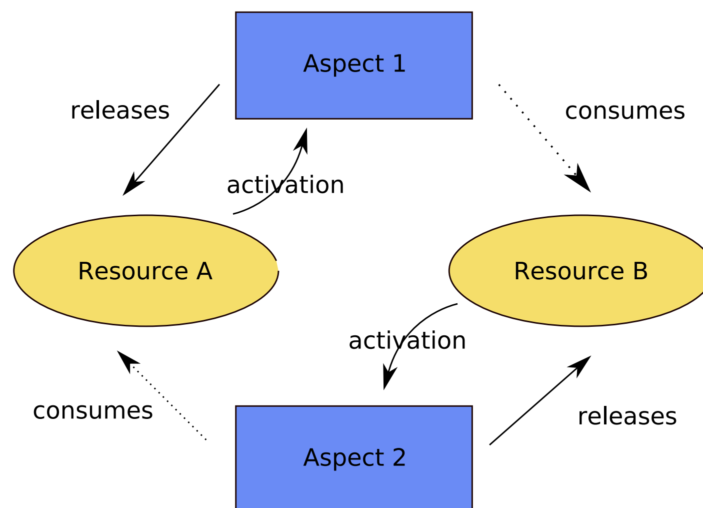

Figura 6.1. Esquema abstracto de conflictos entre aspectos.

6.1.1. Ejemplos

En esta sección presentaremos ejemplos que ilustrarán el tipo de conflictos que se desea resolver.

Imaginemos un sistema móvil que se ejecuta en una PDA en el contexto de una red inalámbrica. Como se mencionó en el capítulo 2, estos dispositivos permiten ejecutar

aplicaciones relativamente complejas, pero tienen considerables limitaciones en lo que a recursos de *hardware* se refiere.

Memory Saver Vs Battery Optimiser

Supongamos que existe un aspecto llamado *Memory Saver* que monitorea la cantidad de memoria libre en el sistema. Cuando detecta que esa cantidad es menor que el límite aceptable se encarga de vaciar todas las *caches* del sistema. Esta modularización del sistema es útil, ya que libera a la aplicación del código relacionado a la administración y cuidado de los niveles de memoria libre aceptables. De otra manera, en todos aquellos lugares donde se utiliza memoria con fines no centrales a la funcionalidad principal de la aplicación, seria necesario chequear el límite de memoria libre y disparar las acciones que fueran necesarias para mantener aceptable el estado del sistema.

Dado que se trata de un sistema en una PDA conectada a una red inalámbrica, es importante cuidar la cantidad y el tiempo de las conexiones, pues una placa de red inalámbrica en general consume mucha energía. Si una conexión inalámbrica se utilizara continuamente impactaría negativamente en la autonomía del sistema. Asumimos entonces que existe un aspecto que trata de optimizar la utilización de las conexiones de red con el fin de maximizar la duración de la energía disponible. Para esto el aspecto *Battery Optimizer* utiliza la siguiente estrategia: almacenar temporalmente la información de salida y, cuando hay suficiente cantidad de información como para justificar la utilización la red, dispara la conexión real.

Es evidente que en ciertas situaciones estos aspectos podrían entrar en conflicto. El aspecto que optimiza la utilización de energía consume memoria, que a su vez está bajo el control del aspecto *Memory Saver*. Si el dispositivo posee suficiente memoria como para satisfacer las necesidad de *buffering* de datos sin traspasar el límite aceptable de memoria libre, este conflicto nunca existirá. Pero el consumo real de memoria puede variar entre una ejecución y otra. No es posible predecir durante el momento del desarrollo del sistema si esta situación existirá o no. Este conflicto no puede ser detectado automáticamente mediante análisis estático del código fuente[1], ya que está ligado al estado del sistema en una ejecución en particular.

En la figura 6.2 podemos ver esquematizada esta situación conflictiva. El aspecto *Battery Optimizer* consume memoria como efecto lateral de su ejecución, esto tiene

[1]La manera habitual de detectar automáticamente posibles puntos de conflicto consiste en encontrar intersecciones entre los conjuntos de *join points* afectados por ambos aspectos, pero dado que se trata de diferentes *concerns* no hay razón para pensar que dichas intersecciones existirán.

un efecto negativo cuando la memoria disponible traspasa el límite aceptable. Por otro lado el aspecto *Memory Saver*, al intentar liberar memoria, afectará el recurso "conexión de red" y en consecuencia también generará consumo de energía. Esto puede llevar a un círculo vicioso de activación de estos aspectos.

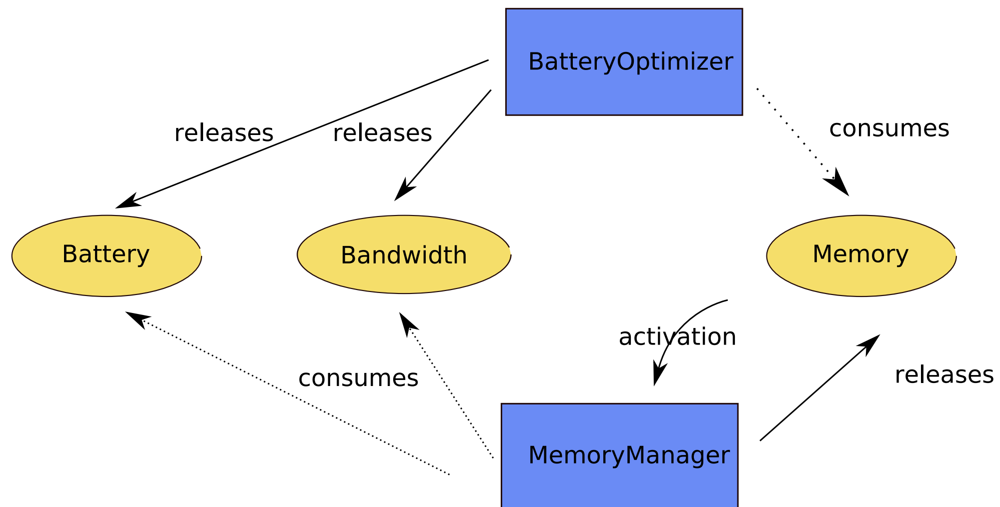

FIGURA 6.2. Esquema de conflictos entre aspectos.

Network Optimiser Vs Security Vs Processing Time

Continuando con el mismo escenario supongamos ahora que nuestro sistema sufre de un tiempo de repuesta pobre en las conexiones ya que la red se encuentra sobrecargada. Además, dado que se trata de una red inalámbrica decidimos utilizar encriptación en las comunicaciones para proveer privacidad. La inclusión de encriptación saturará aún más a la red, ya que usualmente los mensajes encriptados son más grandes que los no encriptados. Para aliviar el problema de sobrecarga de la red decidimos comprimir la información que entra en la red y descomprimirla cuando llega a un dispositivo. Este procesamiento extra deriva en un uso mayor de CPU, que no estará disponible para la aplicación.

Podemos ver que se trata de un delicado equilibrio donde la estrategia más adecuada debería ser adaptativa. Si la seguridad es un requerimiento de alta prioridad podemos aplicar encriptación siempre, pero el algoritmo de compresión podría utilizarse únicamente cuando la red se encuentra sobrecargada. De esta manera haríamos un menor uso del procesador cuando la red esta libre y la compresión no es necesaria.

Discusión

De los ejemplos mostrados es claro que los conflictos por recursos entre aspectos existen, aún cuando estos no afecten los mismos *join points*.

Al pensar en mecanismos de resolución de conflictos, es importante no perder de vista el principio de *obliviousness* [33]. Aunque esta propiedad no sea obligatoria para considerar a un sistema como orientado a aspectos, es muy deseable ya que asegura un grado de desacoplamiento ideal, donde la aplicación base es independiente de los aspectos. Extendiendo esta noción podríamos hablar de *obliviousness* entre aspectos, para indicar que cada aspecto no es consciente de la existencia de los demás. Esta independencia entre aspectos es sumamente deseable, porque permite reutilizarlos por separado. Por lo tanto, es importante que el mecanismo de resolución preserve esta propiedad.

En los ejemplos se ha visto que los aspectos afectan a los recursos generando efectos laterales. En parte esto se debe a que los aspectos usan los recursos a su manera y no tienen forma de conocer o controlar los efectos laterales que podrían ocasionar. Aunque la tuvieran, este conocimiento atentaría contra el principio de *obliviousness*.

Si ni los aspectos ni la aplicación base pueden actuar como coordinadores de la utilización de recursos un tercer módulo deberá desempeñar ese rol.

Este módulo necesitará de alguna manera saber qué es lo que hacen los aspectos con los recursos administrados. Es necesario, entonces, que los aspectos expongan la forma en la que afectan a los recursos. Esto es meta-información de los aspectos que puede ser expresada utilizando diferentes medios. Los lenguajes más modernos permiten expresar esta metainformación directamente en el código fuente. Este mecanismo se conoce con el nombre de anotaciones en el caso de Java y atributos en C#. De aquí en más nos referiremos genéricamente a ellos con el nombre de anotaciones.

6.2. Trabajos Relacionados

Esta sección trata de listar algunos de los trabajos existentes en el área de detección y resolución de conflictos entre aspectos. El objetivo es presentar un ejemplo de cada enfoque; para tener una lista completa de los trabajos publicados al respecto referirse a [54].

Uno de los trabajos más significativos en área es el de Remi Dounce [55] donde se propone un enfoque fuertemente teórico en cual se define un lenguaje formal de aspectos y reglas de composición. Al componer los aspectos es posible detectar

interacciones (gracias a las características del lenguaje). Estas interacciones se convierten en conflictos cuando no son deseables. Los conflictos deben ser resueltos por el programador reescribiendo la composición de los aspectos.

En [56] se presenta una aplicación que permite detectar conflictos entre aspectos a nivel de código fuente. En este trabajo se referencia una taxonomía de conflictos entre aspectos y estrategias para resolverlos. Dicha taxonomía esta basada en los problemas que surgen al aplicar dos o más aspectos sobre el mismo *join point*. La detección de conflictos se basa en una categorización, en 2 niveles, de la similitud de los *pointcuts* en AspectJ. La herramienta presenta todos los posibles conflictos al programador para que éste determine si realmente se trata de un conflicto y en tal caso reprogramar lo que sea necesario para eliminarlo.

El trabajo de Moreira et al. (*Concern Oriented Requirement Engineering* [57]) encara el problema de conflictos en la etapa de requerimientos. En este trabajo los requerimientos son agrupados en *concerns*. A partir de las relaciones entre los *concerns* se calcula el **impacto** de unos contra otros. Este **impacto** se usa para determinar posteriormente la existencia de conflictos que pueden ser resueltos por priorización o renegociación de los requerimientos. Esta priorización de requerimientos es fija para todo el sistema, es decir, si un *concern* es más importante que otro, lo será en todo el sistema. No existen situaciones especiales en las que las prioridades puedan cambiar. Esto es una consecuencia lógica de atacar los conflictos tempranamente.

Tessier presenta en [58] una metodología basada en modelos que permite la detección de conflictos entre aspectos. En ese trabajo se presente auna taxonomía de conflictos. Las categorías incluyen:

- *Crosscutting Specifications*
- *Aspect-Aspect Conflicts*,
- *Base-Aspect Conflicts*
- *Concern-Concern Conflict*

El tipo de conflictos que tratamos de resolver en esta sección puede encuadrarse en la categoría *Concern-Concern*, y dentro de ella en la subcategoría *Inconsistent Behaviour*. Esta subcategoría se refiere a los conflictos donde un aspecto puede alterar el est ado usado por otros aspectos, causando efectos laterales indeseables.

En [59] Bergmans propone el uso de anotaciones como medio para detectar conflictos entre *crosscutting concerns*. En su enfoque los conflictos pueden ser detectados

cuando varios aspectos trabajan sobre el mismo *join point*. Como mencionamos anteriormente, intentamos resolver conflictos entre aspectos aún cuando afecten diferentes *join points*.

6.3. Resolución de conflictos para *Resource-Awareness*

A pesar de la imposibilidad de detectar mediante análisis estático los conflictos aspectuales que se ejemplificaron, es posible estudiar sus causas y razonar acerca de las situaciones peligrosas que los aspectos podrían ocasionar.

En el el contexto de *resource-awareness*, la situaciones conflictivas pueden ser definidas como un estado *no deseable* del sistema. Este, a su vez, puede caracterizarse por el estado de cada recurso del sistema. En nuestro primer ejemplo, un conflicto surge cuando el sistema tiene poca memoria disponible y los *buffers* de datos almacenados por el aspecto *Battery Optimiser* deben ser vaciados muy frecuentemente. En un caso así, la estrategia más apropiada sería desactivar completamente el aspecto *Battery Optimiser*. Es evidente que si existiera suficiente memoria libre todos los aspectos podrían ejecutarse sin problemas, en ese caso no existirían conflictos.

Estudiando el dominio y generando los posibles casos de peligro es posible construir una lista de situaciones conflictivas. Cada una de éstas debe ser asociada a un conjunto de operaciones que permitan salir de la situación y llevar al sistema nuevamente a una situación estable. Cómo encontrar estas situaciones es parte del trabajo futuro de investigación. En lo que resta de esta sección describiremos cómo implementar la resolución de conflictos.

6.3.1. *Semantic Labels*

Las anotaciones son poderosos mecanismos que permiten la extensión (en cierta medida) de los lenguajes de programación. Ellas nos proveen un medio para expresar propiedades que de otra manera deberían estar separadas del código. Las anotaciones han sido objeto de diferentes usos, por ejemplo marcar las clases para ser manejadas dentro de un contenedor de aplicaciones, como es el caso de las anotaciones de EJB3 [60] y Hibernate [61]. Últimamente se han utilizado, junto a los aspectos, para demarcar y modularizar *crosscuting-concerns* [62].

Nosotros utilizaremos las anotaciones para indicar la forma en que los aspectos utilizan los recursos del sistema. Para distinguir estas anotaciones de las de propósito general las llamaremos *semantic labels* [63].

Las *semantic labels* permiten resolver o evitar los conflictos entre aspectos en tiempo de ejecución. Estos descriptores indican la manera en que los recursos son afectados durante la ejecución de ciertos aspectos[2]. En otras palabras las *semantic labels* denotan el rol que juega un aspecto respecto de cada recurso con el cual se relaciona.

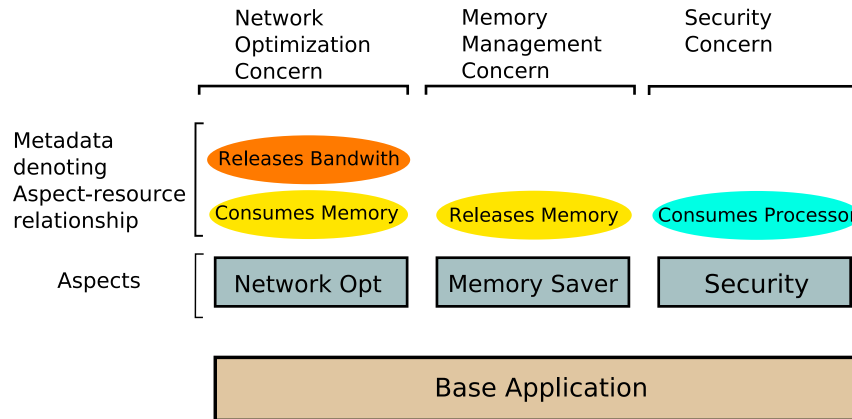

FIGURA 6.3. Aspectos con metadatos relacionados a los recursos.

La figura 6.3 muestra esquemáticamente diferentes *concerns* implementados mediante aspectos donde cada uno de éstos es enriquecido con metadatos que indican cómo afecta a los recursos.

Las *semantic labels* proveen un mecanismo abstracto para hablar de los aspectos, ya no nombrándolos explícitamente sino a través de los roles que juegan en el sistema. El objetivo es poder escribir sentencias como:

Realizar X acción sobre los aspectos con el rol "Consumidor De Memoria".

6.3.2. La Coordinación de los Aspectos

Teniendo a los aspectos anotados es necesario que exista un módulo encargado de consumir esos metadatos y responsable de llevar a cabo la coordinación de los aspectos de acuerdo al estado del sistema. Este módulo será otro aspecto, llamado *Coordinator*, ya que la responsabilidad de monitorear los recursos atraviesa a los demás aspectos.

Este aspecto monitorea los cambios en los estados de los recursos en busca de patrones que definen situaciones conflictivas para el sistema (definidas al comienzo de la sección 6.3).

---

[2]Asumiendo que estos poseen sólo un *advice* o que todos los *advices* afectan los recursos de la misma manera. Alternativamente podrían utilizarse anotaciones a nivel de *advice*.

El coordinador conoce las estrategias para resolver cada situación conflictiva. Cada una de estas estrategias se define como un conjunto de operaciones sobre los aspectos presentes en el sistema. Por ejemplo, una estrategia podría ser "*desactivar todos los aspectos que **consumen memoria***". Es importante notar que estas operaciones hablan de los roles, no de los aspectos concretos. Por lo tanto, el nivel de abstracción es mayor. La misma estrategia en diferentes contextos puede afectar a diferentes aspectos. Asimismo, un rol puede afectar a más de un aspecto, debido a esto nuestro enfoque permite efectuar un tipo limitado de cuantificación [33].

Cuando el coordinador encuentra una situación conflictiva aplica la estrategia correspondiente. Esto es, ejecuta cada operación declarada en la estrategia. Para ello busca los aspectos que juegan el rol especificado y ejecuta la acción que corresponda. De esta manera el comportamiento del sistema es afectado para salir de la situación conflictiva.

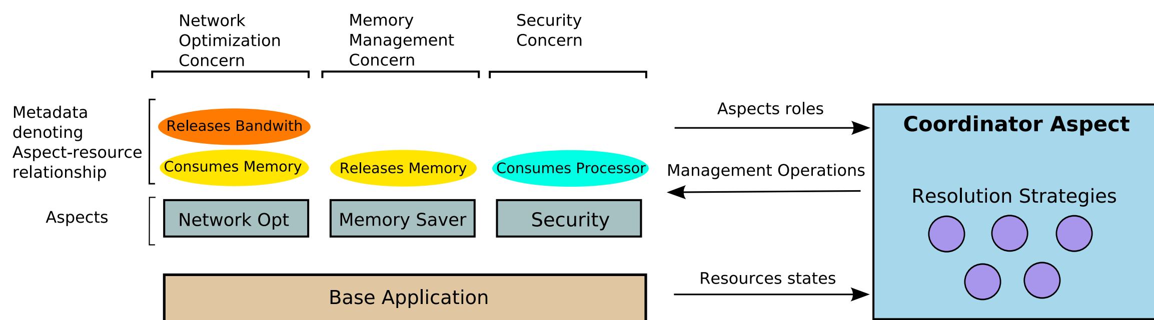

FIGURA 6.4. Esquema completo de resolución de conflictos: metadatos, coordinador y estrategias.

La figura 6.4 muestra el esquema completo donde el coordinador consume los metadatos de los aspectos y monitorea los recursos. Este coordinador contiene las estrategias de resolución de conflictos que se traducen en operaciones de administración de los aspectos coordinados.

6.3.3. Diseño

La figura 6.5 muestra un diagrama de clases UML del diseño preliminar de la solución propuesta en 6.3.2.

La administración de los aspectos requiere cierta infraestructura en los aspectos que será coordinados, por ejemplo las operaciones genéricas como desactivar, activar o suspender temporalmente al aspecto. Estas operaciones son provistas por el aspecto abstracto *ResourceHandler*. Los aspectos que heredan de *ResourceHandler* son los

```
1     @AffectsMemory("consumed")
2     @AffectsBattery("released")
3      public aspect BatteryOptimiser{ ...
```

Listing 6.1. Aspecto anotado.

encargados de implementar los diversos *concerns* para la optimización en el uso de los recursos. Estos aspectos deben declarar qué recursos afectan y en qué forma lo hacen. En el diagrama se ilustran las anotaciones en los comentarios de los aspectos[3].

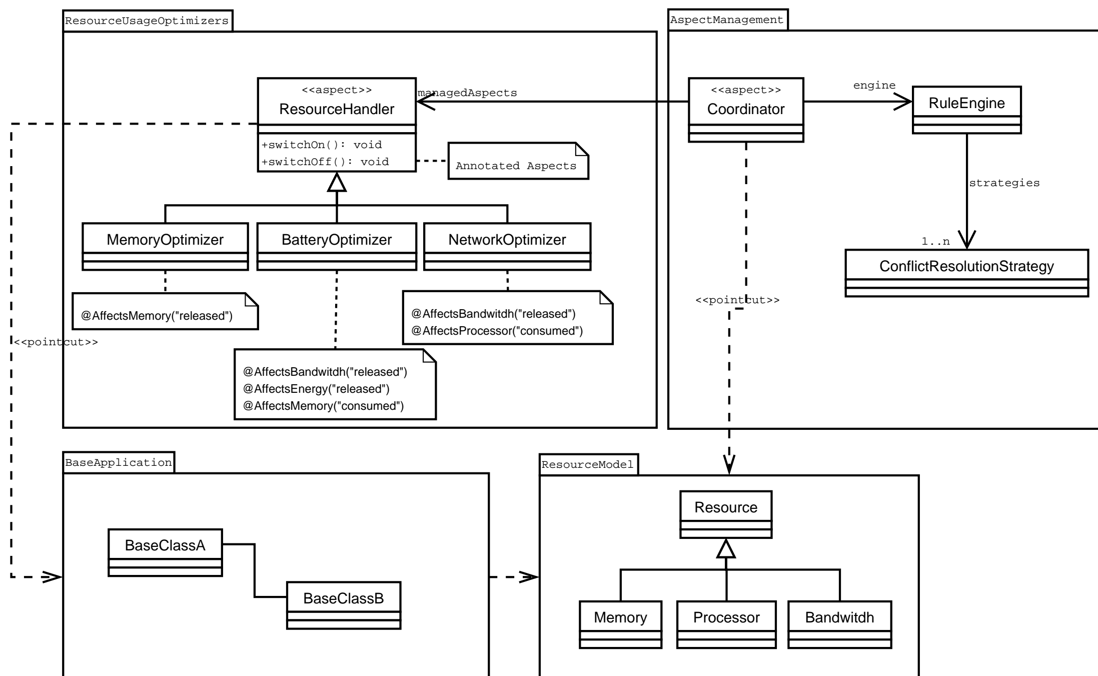

Figura 6.5. Diagrama de clases para la resolución de conflictos.

Las declaraciones se realizan anotando el código de los aspectos como se muestra en el listado de código 6.1. Como podemos ver allí cada anotación describe un recurso distinto, y el argumento de la anotación especifica la forma en la que es usado el recurso. El ejemplo mostrado en 6.1 se puede leer como "*el aspecto BatteryOptimiser consume memoria y libera (ahorra) energía*".

El coordinador intercepta los cambios de estado en los recursos. El estado actual de los recursos es introducido en un motor de reglas donde cada regla identifica un

[3]Dado que no hay una notación unificada para las anotaciones en los diagramas UML, se ha seguido una de las notaciones propuestas en [64].

```
1  @AffectsMemory(ResourceUsageType.CONSUMED)
2  @AffectsBandwidth(ResourceUsageType.RELEASED)
3
4   public aspect BandwidthOptimizer extends ResourceOptimizer{
5
6   pointcut usage(BaseSystem system):  target(system)
7         && call(public void BaseSystem.systemUse(..));
```

LISTING 6.2. Apariencia final del código de un aspecto anotado

```
1  pointcut optimizersConstructor(ResourceOptimizer optimizer):
2       execution (ResourceOptimizer+.new(..))
3       && target(optimizer);
```

LISTING 6.3. Pointcut para intercepción durante la creación de los aspectos que manejan recursos.

conflicto. El motor de reglas activa las reglas que coinciden con el estado actual. Las reglas ejecutan las estrategias de resolución de conflictos. Los aspectos coordinados se encuentran organizados de acuerdo a sus roles, para que sean fácilmente accesibles a las estrategias. Para ello existe un mecanismo de registración que permite mantener referencias a todos los aspectos.

6.3.4. Implementación

Como prueba de concepto se desarrolló un prototipo que demuestra que el enfoque propuesto es factible de ser implementado. El prototipo utiliza AspectJ para implementar los aspectos. Estos aspectos son enriquecidos usando el mecanismo de anotaciones provisto en Java5. Cada anotación hace referencia a un recurso y es parametrizada con el rol que cumple el aspecto respecto al recurso indicado. Para denotar los roles se utilizaron tipos enumerativos, como se muestra en listado 6.2.

Para registrar los aspectos a ser manejados se interceptó la creación de los mismos (ver listado 6.3). Dado que nuestros aspectos, que optimizan la utilización de recursos, heredan del aspecto abstracto `ResourceOptimizer` es posible capturar todas las instanciaciones usando el constructor `''+''` de AspectJ. Dentro del *advice* correspondiente se examinan los metadatos de cada aspecto mediante la API de reflexión de Java y se los organiza adecuadamente, de manera tal que sean fácilmente recuperables mediante sus roles.

Como motor de reglas se utilizó JBoss Rules [65], que es la continuación de un reconocido motor de reglas conocido previamente como Drools. Las reglas que

```
1  rule "MemoryVsBattery"
2   no-loop true
3   when
4    m: Resource(name=="memory",availability<MLIMIT)
5    b: Resource(name=="battery",availability<BLIMIT)
6   then
7    Coordinator.stop("AffectsMemory(CONSUMED)")
8    Coordinator.......
```

LISTING 6.4. Ejemplo de regla simple implementada en JBoss Rules.

```
1
2 pointcut resourceChanged(Resource resource):
3          target(resource) &&
4          (( call(public void Resource.setAvailability(..))
5              || call (public void Resource.consumed(..))
6              || call (public void Resource.released(..)))
7          ) && ! within (ResourceOptimizer+);
```

LISTING 6.5. Intercepción de los cambios de estado en los recursos del sistema

permiten administrar los aspectos y resolver los conflictos toman la forma general de *condición* $\rightarrow$ *acción*, como se muestra en el listado 6.4.

Finalmente fue necesario interceptar todos los cambios de estado en los recursos (listado 6.5). El motor de reglas de reglas es alimentado con la información referente al estado actual de los recursos del sistema. Una vez introducida esta información es posible disparar la evaluación de las reglas, de manera que se activen aquellas cuya condición coincide con el estado actual del sistema (ver listado 6.6).

6.4. Conclusiones

La separación de *concerns* puede traer como consecuencia problemas de interacción al momento de componer el comportamiento del sistema final. El desarrollo independiente de aspectos que manipulan recursos puede llevar a un uso descoordinado

```
1   void around(Resource resource): resourceChanged(resource)
2   {
3    engine.modifyObject(resource);
4    engine.fireAllRules();
5    }
```

LISTING 6.6. Aserción de cambios en el motor y evaluación de reglas

e inapropiado de los mismos. Por lo tanto, es necesario un mecanismo de coordinación entre los aspectos que al mismo tiempo permita mantener la independencia de éstos.

En este capítulo se ha mostrado cómo es posible evitar los conflictos aspectuales basándose en la administración los aspectos enriquecidos con metadatos.

Es importante recalcar que se conserva el principio de *obliviousness*, y que los aspectos permanecen independientes entre sí. A pesar de brindar coordinación respecto del manejo de recursos, la aplicación base desconoce la existencia de módulos implementando dichos *concerns*. A su vez, los aspectos permanecen independientes entre sí, ya que no se coordinan conociendo al resto, sino son coordinados por una tercera parte.

Podemos mencionar las siguientes ventajas del enfoque expuesto:

- Los aspectos pueden ser desarrollados independientemente y luego, en la etapa de integración, sus conflictos pueden ser estudiados y las estrategias para resolverlos implementadas.

- Dado que las estrategias no están ligadas a aspectos, sino a roles que pueden ser cumplidos por diversos aspectos, son reusables independientemente de los aspectos que controlan en un caso concreto. Como consecuencia de esto, nuevos aspectos incluídos en una aplicación pueden ser administrados por estrategias definidas con anterioridad.

- Las estrategias permiten definir comportamiento específico para resolver determinados conflictos. Esto puede ser visto como una priorización dinámica de los aspectos. Las condiciones del contexto y la estrategia determinan qué aspectos deben continuar funcionando y cuales deben ser desactivados.

7. Conclusiones

El paradigma de orientación a aspectos se ha mostrado como un medio eficaz para la separación de *concerns* (SOC) en las aplicaciones sensibles al contexto.

En este trabajo se ha utilizado la orientación a aspectos para modelar e introducir el comportamiento relacionado con el contexto de aplicación, generando de esa manera, aplicaciones sensibles al contexto.

La definición de contexto es muy amplia y abarca todo aquello que rodea a la aplicación, incluyendo al usuario y a la aplicación misma [10]. En este trabajo hemos ejemplificado como integrar los componentes principales del contexto de una aplicación:

- Posición geográfica [66].
- Perfil de usuario [67],[43].
- Recursos disponibles [66].

En todos los casos se observó una mejora substancial en la modularidad de los diseños resultantes.

Se ha logrado respetar el principio de *obliviousness* [33], generando aplicaciones base que son completamente independientes de los *concerns* referidos a la sensibilidad al contexto. Este resultado permite suponer que es posible tomar aplicaciones existentes y extenderlas de manera que sean sensibles al contexto.

Por otro lado, se ha estudiado el impacto de AOP en el reuso de las implementaciones para *concerns* de sensibilidad al contexto. Hemos visto que es posible caracterizar diferentes niveles de reuso de acuerdo a los cambios requeridos para la aplicación de los aspectos [68]. Para maximizar el reuso se separaron los aspectos que implementan la sensibilidad al contexto de las definiciones de los *pointcut* concretos que indican qué partes de las aplicaciones serán afectadas por el código aspectual.

Los *concerns* separados y modelados mediante aspectos son fácilmente modificables. La funcionalidad relacionada con la sensibilidad al contexto puede evolucionar independientemente de la aplicación base sobre la que es aplicada. Los aspectos aislan ambos mundos, el referido a la aplicación base y el referido a la sensibilidad al contexto. La conexión entre ambos se realiza mediante la definición de *poincuts* que

sí son sensibles a los cambios en las firmas de los métodos de la aplicación base. Estos *pointcuts* son la parte más difícil de reusar, pues expresan puntos en la ejecución del programa base. En ciertas situaciones pueden reusarse bajo condiciones específicas que se han descripto en el capítulo 5 y en [68].

En el campo de *resource-awareness* hemos notado la existencia de conflictos en tiempo de ejecución entre los diferentes aspectos que trabajan sobre una misma aplicación. Estos conflictos han sido tratados y se ha desarrollado un mecanismo [63] que permite, una vez identificados, definir estrategias de resolución de conflictos. Este mecanismo conserva la independencia de la aplicación base y de los aspectos entre sí.

7.1. Trabajo Futuro

Aspect Oriented Programming es un paradigma muy joven aún. Hoy en día aún se discute en su comunidad sobre qué técnicas o paradigmas son aplicaciones de AOP y cuales no, como por ejemplo la metaprogramación. Sin lugar a dudas queda un largo camino por recorrer para que la orientación a aspectos se convierta en una herramienta de uso habitual para los diseñadores y programadores. Si bien es ampliamente aceptado que los paradigmas actuales (en particular el más popular: la orientación a objetos) no permiten modularizar todos los *concerns* que hacen a un sistema de software, las técnicas de separación de *concerns* no han penetrado lo suficiente en la industria. Recién a partir de 2004 y 2005 se ha observado una utilización más fuerte y sistemática de este tipo de herramientas en aplicaciones no académicas.

De las varias técnicas de separación de *concerns* que existen, hemos analizado el impacto de una sola de ellas (AOP) en el desarrollo de aplicaciones sensibles al contexto. Es necesario evaluar los efectos de aplicar esquemas de SOC simétricos, como por ejemplo *Subject Oriented Programming*. A partir de la separación de concerns simétrica surgen nuevos interrogantes: ¿La sensibilidad al contexto es una dimensión (completa) del sistema? Los paradigmas simétricos sugieren componer las distintas dimensiones del sistema, pero antes debemos definir la dimensión de *context-awareness*. ¿Se trata de una o muchas dimensiones? Si modelamos con el mismo nivel de granularidad que en este trabajo, veremos que *concerns* muy acotados, como la optimización de un recurso en particular no parecen tener suficiente substancia como para definir una dimensión. Por otro lado, si reunimos todos esos *pequeños concerns* en uno solo de *context-awareness* estamos acoplándolos de manera tal que no pueden ser tratados independientemente.

Ya en los puntos atacados en este trabajo, es necesario profundizar la investiga-

ción en diversas áreas. En el campo de reuso de aspectos es necesario contrastar la aplicación de otras herramientas de AOP como Caesar [69], donde existe un conjunto más rico de abstracciones que permiten nuevas variantes en el reuso de aspectos. En lo que a interacción entre aspectos se refiere, se ha resuelto una categoría de conflictos, pero quedan por resolver otras. La interacción entre los aspectos a nivel de manipulación de recursos es acotada y centralizada en los mismos; esto facilitó el desarrollo del mecanismo de resolución de conflictos. Pero existen conflictos entre *concerns* de tipo funcional, como ser la posición del usuario, su perfil y el horario entre otros. Por ejemplo, un *concern* de localización podría intentar mostrar al usuario un servicio, pero dicho servicio no ser aplicable debido al perfil de usuario. Estos conflictos son más difíciles de resolver, pues requieren un estudio más profundo de los *concerns* y sus interacciones. La resolución de conflictos para estos casos probablemente involucre la definición de ontologías para cada *concern* y la derivación de los conflictos a partir de ese conocimiento formalizado.

A. Referencia del Lenguaje AspectJ

En este capítulo se describe brevemente la sintaxis del lenguaje AspectJ.

A partir de 2005 AspectJ provee dos modelos de sintaxis, uno basado en palabras clave y otro basado en anotaciones. La notación basada en palabras clave es la más antigua y difundida. El segundo esquema de sintaxis proviene la unificación de AspectWerkz y AspectJ. En este capítulo nos referiremos únicamente a la sintaxis basada en palabras clave ya que es la utilizada a lo largo del presente trabajo.

Este apéndice es un una breve e incompleta referencia del lenguaje. El sitio de AspectJ [38] provee la documentación actualizada del lenguaje, además existen excelentes libros que introducen al lenguaje [17, 70].

A.1. Aspectos

Un aspecto en AspectJ se compone principalmente de (figura A.1):

- Declaraciones de *Pointcuts*.
- Declaraciones de *Advices*.
- Declaraciones de inter-tipo.

Los poincuts permiten capturar los *join points* sobre los cuales se desea aplicar el comportamiento de un aspecto. Este comportamiento esta definido en los *advices*. Cada *advice* se enlaza a un *poincut* determinado. Por otro lado, las declaraciones inter-tipo permiten agregar comportamiento y estructura directamente en las clases de los objetos afectados. También permiten alterar la jerarquía de tipos de la aplicación base. En las siguientes secciones revisaremos las principales características sintácticas y semánticas de los aspectos en AspectJ.

A.2. Pointcuts

Los *pointcuts* permiten capturar *join points* en el flujo del programa. Una vez capturados es posible definir acciones a ser realizadas antes o después de la ejecución

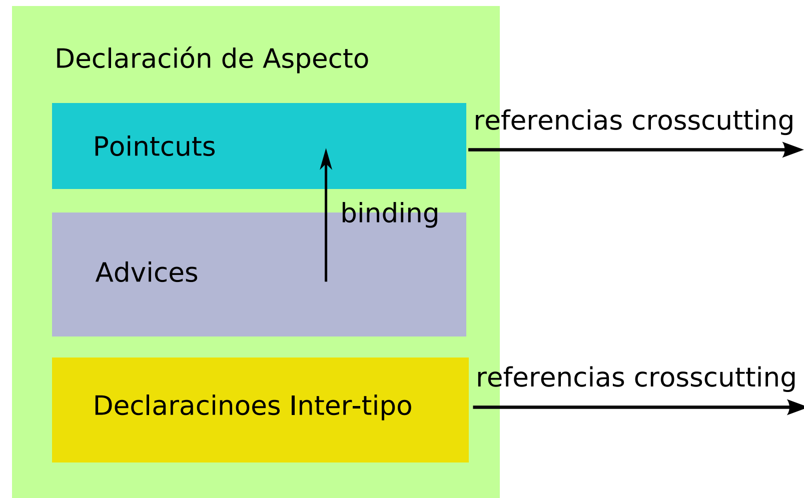

Figura A.1. Estructura de un aspecto

pointcut name — arguments — pointcut definition

```
pointcut schedEventConstruction(ScheduledEvent se): call( private void buildEvent(ScheduledEvent))  && args(se);
```

Figura A.2. Estructura de pointcut nombrado

del *join point*. Los *poincuts* permiten además exponer información de contexto del *join point* capturado.

Un *pointcut* puede ser nombrado, facilitando así su reuso. El nombre del *pointcut* con el que se referencia el pointcut es conocido como *pointcut designator*.

Los *pointcuts* sin nombre se denominan anónimos, y se declaran junto al *advice* que hace uso de ellos.

Los pointcuts nombrados adhieren a la siguiente sintaxis:

```
[acces level] pointcut pointcutName([args]): pointcutDefinition
```

Los niveles de acceso son los mismos que en Java (private, public, default, protected). `Pointcut` es palabra clave. `PointcutName` es el nombre con el que luego se referenciará al *poincut*. A la derecha de los dos puntos se encuentra la definición del *pointcut*, que es una expresión denota los *join points* capturados por el *pointcut* en cuestión.

En la figura A.2 vemos un ejemplo de *pointcut* utilizado en este trabajo donde se muestra su estructura.

La definición del *pointcut* (conocida como *pointcut expression*) es la que brinda la capacidad de cuantificar [33]. Esto significa que un *pointcut* puede capturar multiples *join points* mediante la utilización de comodines.

A.2.1. Expresiones de Pointcut

Existen diferentes tipos de expresiones de *pointcut*. Las más utilizadas son `call` y `execution`.

Es importante tener en mente la diferencia entre ambas, ya que a menudo se confunde su semántica.

Los *pointcuts* de tipo `call` permiten introducir el comportamiento en el contexto de la invocación (en el llamado) de un método. Es decir, el contexto que se captura es el del *sender* del mensaje. Por el contrario, los *pointcuts* de tipo `execution` referencian al contexto de ejecución del método invocado. Esto es, el contexto es del receptor del mensaje.

El parámetro del tipo de `pointcut` es, en general, la firma de un método. Por ejemplo la expresión:

```
call(public void  WindowAdapter.show(void))
```

Aquí `public void WindowAdapter.show(void)` es la firma del método que catura el *pointcut*. En este caso el *pointcut* sólo captura el método público `show` en la clase `WindowAdapter`, este método no tiene parámetros ni valor de retorno. Si cualquiera de estas condiciones no se cumpliera, el *pointcut* no capturaría ningún método.

Comodines y Operadores

Existen mecanismos sintácticos que premiten omitir parte de la firma, de manera tal que se capture más de un *join point* a la vez. Los comodines brindan la posibilidad de no especificar ciertas partes de la firma.

Por ejemplo, el siguiente *pointcut*:

```
call (public void  WindowAdapter.*(void))
```

captura todos los mensajes de la clase `WindowAdapter` que son públicos, no tienen parámetros, ni valor de retorno.

Es posible reemplazar cualquier parte de la firma del método por comodines. Existen 3 tipos de comodines en AspectJ:

- `*` Denota cualquier cadena de caracteres que no contenga el punto (“.”).

Patrón de Firma	Tipos Capturados
`lifia.*Class`	Captura todas los tipos dentro del paquete `lifia` cuyo nombre termina con la cadena “`Class`”
`lifia.gui.Window+`	Captura todos los subtipos de `Window`
`lifia.gui.*Window+`	Captura todos los subtipos de aquellos tipos con sufijo `Window` en su nombre

Cuadro A.1. Ejemplos de utilización de comodines.

Patrón de Firma	Tipos Capturados
`!Vector`	Captura todos los tipos distintos de `Vector`
`Vector || HashTable`	Captura `Vector` y `HashTable`
`java.util.RandomAccess && java.util.List`	Captura todos los subtipos de **ambos** tipos.

Cuadro A.2. Ejemplos de utilización de operadores.

+ Dado un tipo denota cualquiera de sus subtipos.

.. Denota cualquier cadena de caracteres, incluyendo al punto (“.”).

La tabla A.1 muestra ejemplos de utilización de los comodines.

Dentro de las expresiones formadas por firmas de mensajes es posible usar ciertos operadores:

Operadores Binarios En AspectJ están disponibles los operadores `&&` y `||`. Combinar pointcuts con `&&` obliga a que los *join points* capturados coincidan con ambas partes de la expresión. Por otro lado, el operador `||` permite capturar *join points* que coincidan con cualquiera (o ambas) de las partes de la expresión.

Operador Unario El operador `!` indica que se seleccionen todos aquellos *join points* que no coinciden con la expresión que se encuentra a su derecha.

La tabla A.2 muestra ejemplos de uso de estos operadores.

Tipos de Pointcuts

Existen distinos tipos de *pointcut* especializados en la captura de determinados *join points*. La tabla A.3 muestra la sintaxis para los principales tipos de *join points* expuestos en AspectJ.

Categoría de Joinpoints	Sintaxis
Method execution	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler	`execution handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(Const_Signature)`
Advice execution	`adviceexecution()`

Cuadro A.3. Categorías de *pointcuts*

Pointcuts Basados en el Flujo del Control

El flujo del control está dado por el orden de ejecución de las intrucciones a partir de un *join point*. Los *pointcuts* basados en el flujo del control capturan *join points* ya capturados por otros *pointcuts*. Por ejemplo, en la clase `CuentaCorriente` el método `getSaldo()` puede ejecutarse independientemente o como parte del método `debitar(long cantidad)`. En el segundo caso el método `getSaldo()` se ha ejecutado *dentro del flujo de control* de `debitar(long)`. En AspectJ es posible expresar este tipo de situaciones, de manera tal de capturar ciertos *join points* dentro de determinados contextos.

Es posible además incluir o no los *join points* capturados por el *poincut* original. Por ejemplo, la expresión:

```
cflow(call(* CuentaCorriente.debitar(..))
```

utiliza la palabra clave `cflow` para indicar que abarca a los *join points* dentro del flujo de control del `pointcut` que comprende la llamada a los métodos `debitar` en clase `CuentaCorriente`; incluyendo la invocación a dicho método.

Con la palabra clave `cflowbelow` se indica el mismo conjunto de *join points* exceptuando aquellos capturados por la expresión principal. En nuestro ejemplo serían los todos los que están dentro del flujo de control de `debitar()`, excluyendo a este último.

A.2.2. Definición Formal

Una definición algo más formal de la sintaxis de los *pointcuts* es la siguiente:

```
<pointcut> ::= <access_type> <pointcut_name> ( { <parameters> } )
              :  { designator [ && | || ] };
<access_type> ::= public | private [abstract]
<pointcut_name> ::= { <identifier> }
<parameters> ::= { <identifier> <type> }
<designator> ::= [!] call | execution | target | args |
              cflow | cflowbelow | staticinitialization |
              within | if | adviceexecution |
              preinitialization
<identifier> ::= letter { letter | digit }
<type> ::= defined valid Java type
```

A.3. Advices

Los *advices* definen el comportamiento de los aspectos, así como los métodos definen comportamiento de los objetos.

Para que un *advice* se ejecute necesita estar enlazado a un *pointcut* nombrado o anónimo.

Los *advices* pueden acceder al contexto de ejecución del *join point* sobre el que están trabajando, tanto a los parámetros en la invocación a un método como el valor de retorno del mismo.

Existen tres tipos principales de *advice*:

before: permite ejecutar instrucciones **antes** de la ejecución de un *join point* dado.

after: permite ejecutar instrucciones **después** de la ejecución de un *join point* dado.

around: permite reemplazar la ejecución del *join point* y decidir cuando ejecutarlo.

A.3.1. Estructura

Un *advice* se compone de 3 partes:

Declaración del Advice que especifica cuando se ejecuta el *advice* con respecto al *join point* capturado (*before, after, around*). Aquí se declara también la información de contexto que estará disponible en el cuerpo del *advice*.

Especificación del Pointcut donde se declara el *pointcut* al que se enlaza el *advice*.

Cuerpo del Advice que contiene el comportamiento del *advice*. Dependiendo del tipo de *advice* aquí se puede invocar la ejecución del *join point*. Aquí también se tiene acceso a una variable especial llamada `thisjoinpoint`, que provee información reflexiva sobre el *join point* capturado.

A.3.2. Before advice

Como su nombre lo indica, las instrucciones de este tipo de *advice* se ejecutan antes que el *join point* capturado. El siguiente ejemplo de código muestra un *advice* que se ejecuta antes que cualquier método público de la clase `ByteArchiver`

```
 before() : call(public * ByteArchiver.*(..)) {
    // log the use of this class
}
```

Al finalizar la ejecución normal del *advice* se ejecutará el *join point*. Si por alguna razón se produce una excepción en el *advice*, el código del *join point* capturado no se ejecutará.

A.3.3. After advice

Es tipo de *advice* se ejecuta luego de finalizada la ejecución del *join point*. Dado que muchas veces es necesario diferenciar entre situaciones donde el control retorna normalmente y aquellas en las que se ha producido una excepción, AspectJ provee mecanismos para distinguirlas.

Existen tres variantes de `after` para discernir entre ellas:

- `after returning` que ejecuta luego de un retorno normal.

```
 after() returning : call(* Account.*(..)) {
 ... log the successful completion
}
```

- `after throwing` que ejecuta luego de producirse una excepción durante la ejecución del `join point`.

```
after() throwing : call(* Account.*(..)) {
```

- `after` que se ejecuta al finalizar el *join point*, sin importar la forma en la que terminó su ejecución.

```
after(): call(* Account.*(..)) {
```

Los *advice* de tipo *after* permiten además acceder a información de contexto como el valor de retorno y la excepción producida (esto último en los *advices* `after throwing`).

El siguiente ejemplo muestra cómo se accede a dicha información:

```
after() throwing (RemoteException ex)
                             :call(* *.*(..) throws RemoteException)
 {
   System.out.println("Captured exception:" + ex);
 }
```

A.3.4. Around advice

El *advice* de tipo *around* rodea la ejecución del *join point* afectado, el cual no se ejecuta a menos que en el cuerpo del *advice* se explicite su invocación. Esta característica permite a este tipo de *advice* ser usado para omitir la ejecución del *join point*. También es posible alterar el contexto de ejecución del *join point*, ejecutarlo repetidas veces o realizar operaciones luego de su ejecución.

La palabra clave utilizada para invocar al *join point* es `proceed`. Esta funciona como una invocación a un método. Cuando se usa `proceed` es posible pasarle al *join point* un contexto alterado. En este caso los parámetros de `proceed` dependen del *join point* que esta siendo capturado, ya que ambos deben ser concordantes.

A continuación se muestra un ejemplo de *around*:

```
void around():applicationStarted() {
 waba.sys.Vm.debug("LogAround: before starting the app");
```

```
  proceed();
  waba.sys.Vm.debug("LogAround: after starting the app");
 }
```

En este caso si no invocáramos a `proceed` el *join point* nunca se ejecutaría.

Un ejemplo de utilización más interesante consiste en usar `around` para hacer reintentos. Esto es, mientras el resultado de la ejecución del *join point* no es el esperado seguimos invocando a `proceed`. A continuación se muestra la forma que podría tomar un `advice` de este tipo, donde se asegura que el resultado de la ejecución del *join point* sea el esperado:

```
 Object around():somePointcut() {
  Object expectedResult = null;
  while (expectedResult == null)
       expectedResult = proceed();
  return expectedResult;
}
```

Como puede observarse, el *join point* se ejecuta tantas veces como sea necesario hasta que devuelva un valor no nulo.

Esta es la base para implementar otros *advices* más complejos donde, por ejemplo, es posible chequear si se produce una excepción durante la ejecución del *join point*, alterar el contexto si es necesario y reintentar.

A.3.5. Definición Formal

Una definción más formal para la sintaxis de los *advices* es la siguiente:

```
Advice ::= [ReturnType] TypeOfAdvice "("[Formals]")"
           [AfterQualifier] [throws TypeList] ":"
           Pointcut "{" [AdviceBody] "}"
TypeOfAdvice ::= before | after | around
ReturnType ::= TypeOrPrimitive ;(applies only to around advice)
AfterQualifier ::= ThrowsQualifier | ReturningQualifier
                   ;(applies only to after--defined later)
Formals ::= ;(as a Java parameter list)
```

```
Pointcut ::= ;
AdviceBody ::= ;(as a Java method )
```

A.4. Declaraciones Intertipo (Crosscutting Estático)

Si bien los *advices* permiten modificar el comportamiento de los *join points*, muchas veces es necesario alterar la estructura estática de los programas. Esto incluye alterar las jerarquías de tipos y clases, agregar métodos y atributos a las clases del programa base.

Estos cambios afectan el comportamiento en compilación del código base. A continuación veremos ejemplos de cada caso.

A.4.1. Introducción de Miembros

Para una correcta separación de *concerns* a veces es necesario que cierto comportamiento o cierta estructura de los objetos queden declarados en un aspecto, pero que al momento del *weaving* sean introducidos en alguna clase del sistema base.

Por ejemplo, si es necesario que para la implementación de un *concern* una clase implemente un nuevo mensaje, este puede declararse en el aspecto. Este fue el caso en la sección 4.3.4, donde se implementó personalización. En este caso se necesitaba que una clase exponga una variable de instancia mediante un *getter*. Dado que este *getter* es una necesidad surgida del nuevo *concern* se decidió no alterar la clase original. A continuacion vemos cómo se declaró ese aspecto.

```
 public privileged aspect CompletionAspect {
...
 public ScheduledEvent NewEventWindow.getScheduledEvent() {
  return scheduledEvent;
 }
...
}
```

Nótese que en este caso el aspecto fue declarado como privilegiado (palabra clave `privileged`) para tener acceso a una variable privada de la clase `NewEventWindow`.

A.4.2. Modificación de la Estructura de Tipos

En cierta ocasiones los *advices* de los aspectos necesitan trabajar sobre ciertos tipos base. Para asegurarse de eso necesitan afectar las clases en cuestión alterando su tipos base. Esto a hace los aspectos más reusables, ya que no se acoplan a clases específicas de la aplicación sino a un tipo más genérico y con una API controlada.

En AspectJ es posible alterar la jerarquía de clases de un sistema, siempre y cuando no se violen las reglas de herencia de Java.

La forma de estas declaraciones es la siguiente:

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

Un ejemplo de uso podría ser un aspecto que debe persistir objetos en forma binaria. Sin duda este necesitará que dichos objetos implementen la interface `Serializable`. Si dichas clases se encuentran en un paquete conocido, podríamos usar la siguiente declaración:

```
aspect Serializer {
 declare parents : lifia..* implements Serializable;
  ...
}
```

De esta manera todas las clases de dentro nuestro paquete lifia pueden ser serializadas por el aspecto de persistencia.

A.4.3. Declaración de Warnings y Errores en Compilación

AspectJ provee un mecanismo que permite generar mensajes en tiempo de compilación. Estos mensajes pueden ser: *warnings* o errores.

Dado que este comportamiento es únicamente aplicable durante la compilación, los `pointcuts` permitidos deben ser puramente estáticos. Por lo tanto, no pueden usarse ninguno de los constructores dinámicos de *pointcuts*: `this()`, `target()`, `args()`, `if()`, `cflow()` y `cflowbelow()`.

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

Durante el proceso de compilación, cada vez que el *pointcut* capture algún *join point* se imprimirá el mensaje especificado. Si se trata de un error el compilador detendrá la compilación. Si se trata de un *warning* la compilación continuará luego de imprimir el correspondiente mensaje.

Bibliografía

[1] Siobhan Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, June 2001.

[2] AT & T. Friend finder, 1992. http://www.gismonitor.com/news/newsletter/archive/062702.php.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Krzysztof Czarnecki, Ulrich Eisenecker, and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[5] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1990.

[6] Kalle Lyytinen and Youngjin Yoo. Issues and challenges in ubiquitous computing. *Communications of the ACM*, 45, 2002.

[7] Mark Weiser. Some computer science problems in ubiquitous computing. *Communications of the ACM*, July:137–143, 1993.

[8] Mark Weiser. The computer for the twenty-first century. *Scientific American*, September(265(3)):94–104, 1991.

[9] A. Schmidt. Implicit human computer interaction through context. In *Personal Technologies, Vol 4(2)*, June 2000.

[10] A.K. Dey and G.D Abowd. Towards a better understanding of context and context-awareness. Technical report, Georgia Institute of Technology, 1999.

[11] A.K. Dey. Understanding and using context. In *Personal and Ubiquitous Computing Journal*, volume 5, pages 4–7, 2001.

[12] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[13] Tazari amd R. Grimm M. and Finke M. Modelling user context. In *The 10th International Conference on Human Computer Interaction*, 2003.

[14] Andrew Jagoe. *Mobile Location Based Services: Professional Developer Guide*. Pearson Education, 2002.

[15] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[16] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-awareness on mobile devices - the hydrogen approach. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 292.1, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[18] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[21] W. Harrison and H. Ossher. Subjectoriented programming: (a critique of pure objects). In ACM SIGPLAN Notices, editor, *Proceedings OOPSLA '93*, volume 28. ACM, 1993.

[22] M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O.Ñierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. 1993.

[23] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. Lehrmann Madsen, editor, *Proc. 7th European Conf. Object-Oriented Programming*, pages 372–395, 1992.

[24] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[25] Demeter Research Group. Online material on adaptive programming Demeter/Java, and APPCs.
http://www.ccs.neu.edu/research/demeter/.

[26] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[27] Karl Lieberherr and David H. Lorenz. Coupling aspect-oriented and adaptive programming. In Filman et al. [71], pages 145–164.

[28] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[29] Prose Dynamic AOP. The prose website.
http://prose.ethz.ch/.

[30] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[31] M. Haupt, C. Bockisch, M. Mezini, and K. Ostermann. Towards Aspect-Aware Execution Models. Technical Report TUD-ST-2003-01, Software Technology Group, Darmstadt University of Technology, Alexanderstrasse 10, 64289 Darmstadt, Germany, 2003.

[32] Jikes research virtual machine for java.
http://jikesrvm.sourceforge.net/.

[33] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [71], pages 21–35.

[34] M. Aksit and B. Tekinerdogan. Aspect-oriented programming using composition filters. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, page 435. Springer, 1998.

[35] Java 2 micro edition.
https://java.sun.com/javame/.

[36] Plam os. http://www.palmsource.com/palmos/.

[37] Microsoft windows.
http://www.microsoft.com/windows/.

[38] AspectJ project.
http://www.eclipse.org/aspectj/.

[39] Xerox PARC. AspectJ home page.
http://eclipse.org/aspectj/.

[40] Eclipse homepage.
http://www.eclipse.org.

[41] Extended markup languaje.
http://www.w3c.org/.

[42] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[43] Arturo Zambrano, Luis Polasek, and Silvia Gordillo. Decoupling personalization aspects in mobile applications. In *Selected and Revised Papers from Interacción 2004, Proceedings. Mayo*. Springer, 2005.

[44] J Blom. Personalization - a taxonomy. In *CHI 2000 Workshop on Designing Interactive Systems for 1-to-1 Ecommerce*, 2000.

[45] Luis Alonso Romero Rui Alexandre P. P. da Cruz, Francisco J. García Peñalvo. Perfiles de usuario: En la senda de la personalización. Technical report, Departamento de Informática y Automática. Universidad de Salamanca, 2003.

[46] W. Retschitzegger G. Kappel and W. Schwinger. Modeling ubiquitous web applications: The wuml approach. In *International Workshop on Data Semantics in Web Information Systems (DASWIS-2001)*. ACM-Press, 2001.

[47] World Wide Web Consortium. Composite capabilities/preference profiles, 2001.

[48] Superwaba development platform for mobile devices.
http://www.superwaba.org/.

[49] Ayla Dantas, Paulo Borba, and Vander Alves. Using aspects to structure small devices applications. In *First Workshop on Reuse in Constrained Environments (RICE'03), OOPSLA2003*, 2003.

[50] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 85–92. ACM Press, 2002.

[51] Iambic Inc. Agendus for palm o s. http://www.iambic.com/agendus/.

[52] Javabeans specifications.
http://java.sun.com/products/javabeans/docs/spec.html.

[53] Choonsung Shin and Woontack Wook. Conflict resolution method using context history for context-aware applications. In *First International Workshop on Exploiting Context History in Smart Environment. Pervasive 2005*, 2005.

[54] Robert E. Filman. A bibliography of aspect-oriented programming, September 2004.

[55] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, March 2004.

[56] Sandra Casas, Claudia Marcos, Veronica Vanoli, Hector Reinaga, Luis Sierpe, Jane Pryor, and Claudio Saldivia. Administración de conflictos entre aspectos en aspectj. In *Proceedings of the Fourth Argentine Symposium on Artificial Inteligence*, pages 1–11, 2005.

[57] Ana M. D. Moreira, João Araújo, and Awais Rashid. A concern-oriented requirements engineering model. In Oscar Pastor and João Falcão e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[58] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In Minhuan Huang, Hong Mei, and Jianjun Zhao, editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, September 2004.

[59] Lodewijk M. J. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[60] Ejb specification at the java community process. http://www.jcp.org/en/jsr/detail?id=220/.

[61] Hibernate. http://www.hibernate.org/.

[62] Ramnivas Laddad. Aop and metadata: A perfect match, March 2005. http://www-128.ibm.com/developerworks/java/library/j-aopwork3/.

[63] Arturo Zambrano, Tomas Vera, and Silvia Gordillo. Solving aspectual semantic conflicts in resource aware systems. In Walter Cazzola, Shigeru Chiba, Yvonne Coady, and Gunter Saake, editors, *Third ECOOP Workshop on Reflection, AOP and Metadata for Software Evolution*, Nantes, France, 2006.

[64] V. Cepa and S. Kloppenburg. Representing explicit attributes in uml. *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

[65] Jboss rules engine. http://www.jboss.com/products/rules.

[66] Arturo Zambrano, Silvia Gordillo, and Ignacio Jaureguiberry. Aspect-based adaptation for ubiquitous software. In Fabio Crestani, Mark D. Dunlop, and Stefano Mizzaro, editors, *Mobile HCI Workshop on Mobile and Ubiquitous Information Access*, volume 2954 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2003.

[67] Arturo Zambrano, Luis Polasek, and Silvia Gordillo. Desacoplando la personalización en las aplicaciones móviles. In *Interacción 2004, Proceedings. Mayo*, 2004.

[68] Arturo Zambrano, Silvia E. Gordillo, and Ignacio Jaureguiberry. Reusing context-awareness through aspects. In Jaelson Castro and Ernest Teniente, editors, *CAiSE Workshops (2)*, pages 717–729. FEUP Edições, Porto, 2005.

[69] Klaus Ostermann and Mira Mezini. Conquering aspects with Caesar. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 90–99. ACM Press, March 2003.

[70] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley and Sons, 2003.

[71] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.