UNIVERSIDAD PONTIFICIA DE SALAMANCA

FACULTAD DE INFORMÁTICA

Programa de Doctorado en Ingeniería de Software

Bienio 2003-2005

Software de Comunicaciones: Modelos y Estándares Avanzados

Trabajo de Investigación:

"Estudio de la Programación Orientada a Aspectos (AOP).

Caso Práctico: Diseño de Aspectos en la Sincronización de Agendas para dispositivos móviles"

Profesor:

Dr. D. Alvaro Suárez Sarmiento (ULPGC)

Dra. Elsa María Macías López (ULPGC)

Alumno: César Parejas Llanovarced

Madrid, Septiembre 2004

ÍNDICE

Capítulos

INTRODUCCIÓN

La programación orientada al Objetivo (OOP, por sus siglas en inglés) ha cosechado éxitos como un medio de manejar la complejidad del software a través de las estructuras jerárquicas de tipos de objeto cuidadosamente diseñadas (clases). Cada clase encapsula un conjunto de funciones comerciales reutilizables y reutiliza funciones ofrecidas por su clase o clases. La reutilización del código orientado al objeto elimina gran parte del código con tendencia a los errores técnicos o bugs que se repite en los proyectos de software complejos.

Pero un objeto sólo puede reutilizar el código en su ruta de herencia, Los lenguajes OOP como Java y C# sólo dan soporte a la herencia sencilla. En las aplicaciones Java, la jerarquía de herencia ha de ser desarrollada en torno a las funciones comerciales clave para permitir la máxima reutilización del complejo código de lógica comercial. Los elementos secundarios de los procesos comerciales –y de desarrollo– que cruzan transversalmente el árbol de herencia – la seguridad, la conexión a una página determinada, la gestión de recursos, la gestión de errores y otras restricciones del diseño, las propiedades o comportamientos que afectan a todo el sistema– han de repetirse en cada objeto afectado. Conforme los proveedores de software se van haciendo más complejos, va resultando más difícil mantener actualizados los segmentos de códigos repetitivos.

La programación orientada a aspectos (AOP, por sus siglas en inglés) es una nueva metodología de programación cuyo objetivo es facilitar la modularización de estas incumbencias dispersas. AOP persigue la implementación de aplicaciones de fácil diseño, entendibles y fáciles de mantener. Además, AOP promete una productividad más alta, de mejor calidad y mejor capacidad para implementar nuevas características en las aplicaciones.

En este trabajo se ofrece una breve introducción a los fundamentos técnicos de esta nueva tecnología de desarrollo y se analizan los enfoques actuales que pretenden resolver el problema de las incumbencias cruzadas. De igual modo, profundizamos en el análisis de AspectJ como lenguaje orientado a aspectos de mayor relevancia en la actualidad. Finalmente estudiamos un caso práctico de sincronización de agendas para dispositivos móviles, enfocándolo desde su etapa de diseño de su arquitectura y realizamos una comparativa en su diseño orientado a objetos y su diseño orientado a Aspectos con UML.

CAPITULO 1. FUNDAMENTOS DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS

EVOLUCIÓN DE LAS METODOLOGÍAS DE PROGRAMACIÓN DE SOFTWARE

En los inicios de la informática, los desarrolladores escribían programas a nivel de código de máquina. Desafortunadamente, los programadores pasaron más tiempo pensando en un conjunto particular de instrucciones de máquina que en el problema que nos concierne. Lentamente, migramos a los lenguajes de alto nivel que nos permitieron una cierta abstracción de la máquina subyacente. Entonces llegaron los lenguajes estructurados; pudimos descomponer el ámbito del problema en un conjunto de procedimientos necesarios para desarrollar las tareas requeridas. Sin embargo, como la complejidad creció, necesitamos técnicas mejores. La programación orientada a objetos (OOP) nos dejó ver al sistema como un conjunto de objetos colaborando. Las clases nos permitieron ocultar los detalles de implementación bajo las interfases. El polimorfismo proporcionó un comportamiento y una interfaz común para los conceptos relacionados, y además, permitieron crear componentes más especializados para cambiar un comportamiento particular sin necesidad de acceder a la implementación de los conceptos de base.

Las metodologías y los lenguajes de programación definen la manera de comunicarnos con las máquinas. Cada nueva metodología presenta nuevas maneras de descomponer el ámbito del problema: código de máquina, código independiente de la máquina, procedimientos, clases, etc. Cada nueva metodología permite una trazabilidad más natural de los requerimientos del sistema con las construcciones de programación. La evolución de estas metodologías de programación nos permitió crear sistemas de complejidad cada vez mayor. Lo inverso de este hecho puede ser igualmente cierto: permitimos la existencia de sistemas cada vez más complejos porque la evolución de las técnicas nos permitieron tratar esa complejidad.

Actualmente, OOP se presenta como la metodología elegida por la mayoría de los nuevos proyectos de desarrollo de software. De hecho, OOP ha demostrado su fortaleza cuando se trata de modelar comportamientos comunes. Sin embargo, no soporta adecuadamente la separación de incumbencias que se extienden –muchas veces de forma inconexa- en los distintos módulos del sistema. En contraste, la metodología AOP completa este vacío. AOP representa posiblemente el siguiente gran paso en la evolución de las metodologías de programación.

FUNDAMENTOS Y CARACTERÍSTICAS DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS

La programación orientada a aspectos amplía el paradigma de la orientación a objetos al permitirnos escribir código más mantenible mediante unidades de modularización de software llamadas "aspectos". Los aspectos encapsulan elementos como la optimización del rendimiento, la sincronización, la comprobación/tratamiento de errores, la supervisión/registro y la depuración, que

trascienden los límites tradicionales del módulo o clase. Los aspectos se separan (modularizan) de las clases y métodos que constituyen los componentes en tiempo de diseño, luego los compiladores o intérpretes crean clases extendidas que combinan las funcionalidad del aspecto con los componentes de la aplicación.

Hay varias herramientas y lenguajes que admiten el concepto de programación orientada a aspectos (AOP). TransWarp (http://www.zope.org/Members/pje/TransWarp) y Pythius (http://pythius.sourceforge.net), por ejemplo, proporcionan compatibilidad con AOP basada en Phyton, mietras que AspectC++ (http://www.aspectc.org) y AspectC (http://www.cs.ubc.ca/labs/spl/projects/aspect.html) admiten C++ y C, respectivamente. Java Aspect Components o JAC (http://jac.aopsys.com) es un marco de trabajo AOP para Java, mietras que HyperJ (http://HyperJ/HyperJ.htm) proporciona extensiones Java para AOP.

Visión del sistema de software como un conjunto de incumbencias

Una incumbencia es un objetivo, un concepto, o un campo de interés particular. En términos de tecnología, un sistema de software típico abarca varias incumbencias a nivel funcional y a nivel sistema. Por ejemplo, una incumbencia de nivel funcional de una aplicación de tarjetas de crédito, sería el proceso de pagos, mientras que una incumbencia a nivel sistema sería el registro, integridad de la transacción, autenticación, seguridad, desempeño, etc. Muchas de estas incumbencias –conocidas como incumbencias transversales (crosscutting concerns) – tienden a afectar múltiples módulos en la implementación. Empleando las metodologías actuales de programación, las incumbencias transversales se extienden sobre múltiples módulos, dando como resultado sistemas complicados de diseñar, entender, implementar y mantener.

Podemos ver un sistema complejo de software como la realización mancomunada de múltiples incumbencias. Un sistema típico puede verse como un conjunto de incumbencias, incluyendo lógica de negocio, desempeño, persistencia de datos, registro y control de errores, autenticación, seguridad, etc.

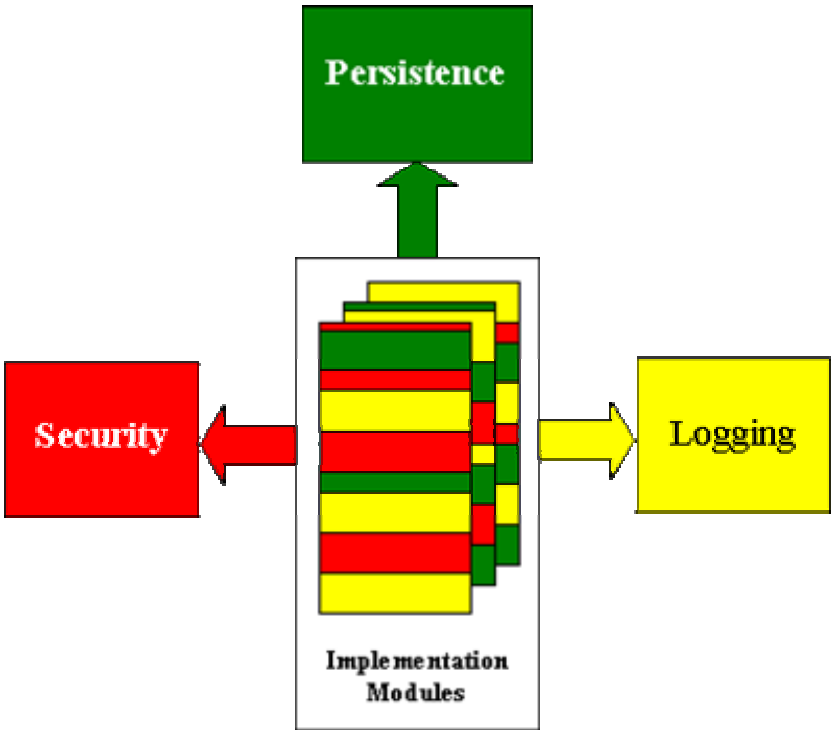


Figura 1. Ilustración de un sistema visto como un conjunto de incumbencias implementadas en diferentes módulos.

Incumbencias transversales en un Sistema

El desarrollo de un sistema se inicia a partir de un conjunto de requerimientos. Estos requisitos podríamos clasificarlos como requerimientos funcionales y requerimientos de nivel de sistema. Muchos requisitos de nivel de sistema tienden a ser ortogonales (mutuamente independientes) el uno del otro y de los requisitos funcionales. Las incumbencias de nivel sistema también tienden a dispersarse en muchos módulos funcionales. Por ejemplo, una aplicación comercial típica está compuesta de incumbencias transversales tales como autenticación, registro, acceso a recursos compartidos, administración, desempeño, y gestión de almacenamiento. Cada uno de estos puede dispersarse en muchos subsistemas. Por ejemplo, la incumbencia de gestión de almacenamiento afecta al estado de muchos objetos de negocio.

El diagrama de clases de “Vista del Run-Time” de la **Figura 2** ilustra cómo todas esas incumbencias que cruzan transversalmente se ven enredadas en la estructura principal de la clase.

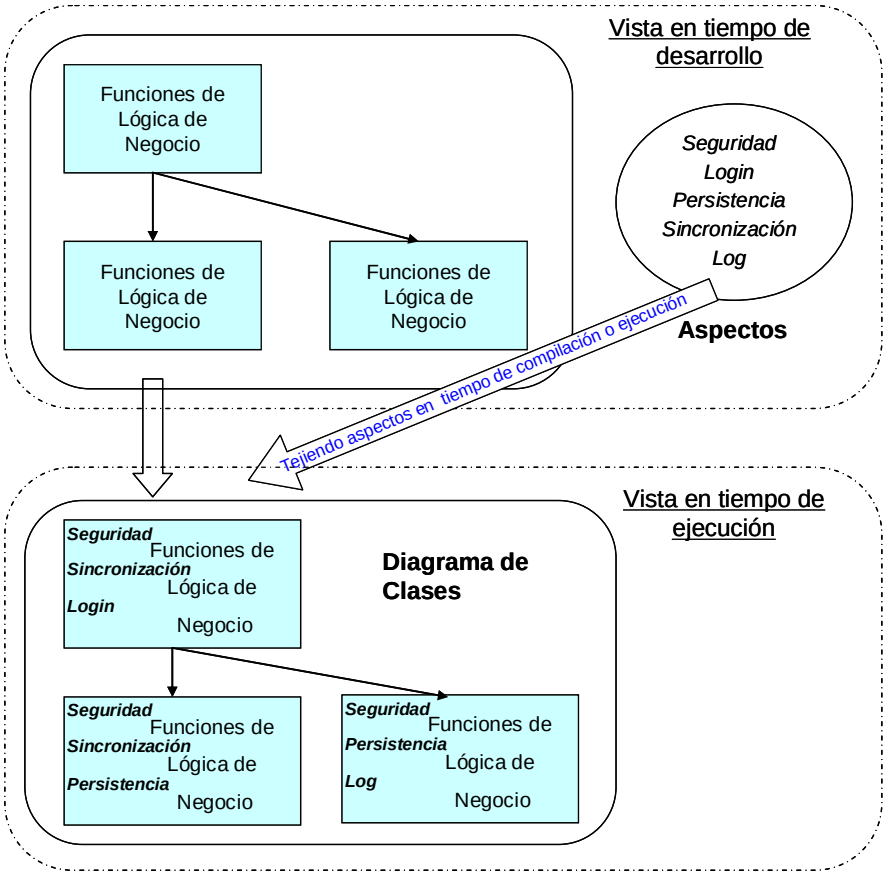


Figura 2. AOP nos permite trabajar con componentes reutilizables y permite que el run time OOP funcione con el código generado entrelazado con elementos de cruce transversal.

Consideremos un ejemplo simple y concreto del esqueleto de una clase que encapsula alguna lógica de negocio.

```
Public class AlgunaClaseLogicaNegocio extend OtraClaseLogicaNegocio {
        // Datos de la lógica de negocio

      // Otros datos de nivel de sistema: Log stream, indicador de consistencia //
      de  datos

        // Sobrescribir métodos de la clase base

        public void algunaOperacionNegocio (InformacionOperacion info) {
              // Asegurar la utenticación

              // Asegurar que info satisface el contrato

              // Bloquear el objeto para asegurar la consistencia de datos en el
              // caso que otros hilos traten de acceder a el

              // Asegurar que el caché está actualizado

              // Registrar el inicio de la operación

              // === Implementar la funcionalidad de la operación ===

              // Registrar la conclusión de la operación

              // Desbloquear el objeto
      }

      // Más operaciones similares a la anterior

      public void guardar (AlmacenamientoPersistente ps) {

      }

      public void cargar (AlmacenamientoPersistente ps) {

      }
}
```

En el código anterior, podemos considerar algunas cuestiones. Primero, *Otros datos de nivel de sistema* no pertenecen al objetivo principal de esta clase. Segundo, la implementación de *algunaOperacionNegocio()* parece que realiza más funciones que el objetivo central de la operación; como la gestión de incumbencias de registro, autenticación, operación multi-hilo, validación de la información contractual, y gestión del caché. Además, muchas de estas incumbencias secundarias podrían aplicarse de la misma manera a otras clases. Tercero, no queda claro si la implementación de las incumbencias de gestión de almacenamiento persistente de datos -*guardar()* y *cargar()*- deberían formar parte del objetivo central de la clase.

CAPITULO 2. ENFOQUES ACTUALES PARA LA GESTIÓN DE LAS INCUMBENCIAS TRANSVERSALES

Aunque las investigaciones han desarrollado formas de enfocar este problema de reutilización del código que cruza transversalmente, aún seguimos necesitando formas compatibles con el retroceso para proteger las enormes inversiones realizadas en el software del que disponemos y en la formación de desarrolladores. Muchas importantes tecnologías Java para empresas como la Enterprise JavaBeans (EJB) y los Java DataObjects(JDO), se han desarrollado para que se encarguen de los problemas del cruce transversal en los sistemas para empresas complejos basados en Java. Actualmente existen cuatro maneras de enfocar este problema, todas ellas basadas en sistemas OOP.

PRIMER ENFOQUE: HERENCIA MÚLTIPLE

En el mundo tradicional orientado al objeto, podemos mezclar distintas ramas de árboles de herencia mediante la herencia múltiple. Mientras que los lenguajes OOP del tipo Smalltalk y C++ soportan herencia múltiple, la técnica es difícil de usar y necesita para empezar de importantes esfuerzos de diseño. Los OOP como Java y C#, por otro lado, reducen la complejidad (y el abuso potencial) mediante la total eliminación de la herencia múltiple.

Por otro lado, la herencia múltiple no resuelve todos los problemas generados por la mezcla de cruces transversales. En la combinación de varios elementos, a menudo el sistema tipo se encuentra con casos ambiguos e incluso, insolubles. Por ejemplo, el problema de la “anomalía de herencia” demuestra que mezclar cuestiones de sincronización con otras cuestiones mediante la herencia es, simplemente, no factible. Resumiendo, la herencia múltiple ha demostrado ser demasiado compleja para ser útil.

SEGUNDO ENFOQUE: AOP DE TIEMPO DE COMPILACIÓN Y ASPECTJ

Que se estudia con mayor profundidad en los capítulos 1, 3 y 4.

Aunque los lenguajes AOP genéricos pueden tener grandes capacidades, tienen un aprendizaje laborioso. Otra importante limitación de AspectJ es el inflexible entrelazado de aspectos en tiempo de compilación. Por útlimo, este enfoque de AspectJ para AOP ha sido patentado por Xerox, lo que deja en situación incierta el futuro de AOP.

TERCER ENFOQUE: INSERCIÓN DINÁMICA DE ASPECTOS

Una tercera manera de acometer los problemas de reutilización del código de cruce transversal es insertar el código de aspecto directamente en los *bytecodes* durante el tiempo de ejecución. La manipulación de los *bytecodes* se hace normalmente mediante una plataforma especial, como la plataforma *jAdvise* de Bob Lee (http://crazy-bob.org/downloads.htm) o un contenedor especial de ejecución, del tipo Java Aspect Components (JAC, http://jac.aopsys.com/), desarrollado por Renaud Pawlak. Ambas herramientas lanzadas bajo licencias Open-Source.

Este enfoque nos permite mezclar elementos de cruce transversal en cualquier momento, en cualquier punto de unión, durante el flujo de ejecución. Comparado con el enfoque de AOP en tiempo de compilación, el entrelazado dinámico de aspectos es más sencillo de usar y mucho más flexible. Por ejemplo, podemos cambiar el comportamiento de AOP mediante los ficheros de configuración de run-time o de una interfaz de gestión sin tener que realizar una compilación general de la aplicación. No obstante , ni jAdvise ni JAC son actualmente estándares, y para utilizar cualquiera de los dos hay que aprender conceptos y API subyacentes. Es difícil la depuración del *bytecode* instrumentado de forma dinámica con herramientas que no son estándar. Y no siempre es posible utilizar plataformas o contenedores que no pertenecen a ninguna de las dos partes en cuestión en aplicaciones personalizadas. Además, las técnicas de programación para implementar los tejedores de aspecto de bytecodes dinámicos están aún en la infancia.

CUARTO ENFOQUE: EL MODELO DEL INTERCEPTOR

El cuarto enfoque lleva modelos de diseño. Mientras que AOP sigue necesitando lenguajes nuevos o plataformas de terceros, los modelos de diseño son corrientes. Por lo que se refiere a los elementos de cruce transversal, el modelo de diseño más importante es el modelo de Interceptor. En Pattern-oriented Software Architecture (John Wiley & Sons, 2000), Douglas Schmidt define el modelo de Interceptor como una técnica que “permite añadir servicios de forma transparente a una plataforma, y dispararlos automáticamente cuando se producen unos eventos específicos”. El modelo de interceptor es fácil de usar, es original de Java y soporta el entrelazado de aspectos durante el tiempo de ejecución. El desarrollador principal de *JBoss* y de *Xdoclet*, *Richard Vberg* (http://roller.ant-honyeden.com/page/rickard/) fomenta el entrelazado dinámico de aspectos basado en el enfoque del interceptor. Una de las ventajas de este enfoque es que sólo depende de las características del lenguaje estándar de Java y no modifica el bytecode.

Aunque los interceptores ofrecen la ruta de migración más corta para añadir elementos que utilizan cruces transversales al software que tenemos, hay limitaciones cuando lo comparamos con el enfoque AOP puro. No se pueden implementar fácilmente todas las características de AOP utilizando los interceptores run-time. Por ejemplo, los interceptores basados en proxy dinámicos de Java sólo soportan directamente los puntos de unión de invocación de método. Pero las invocaciones de métodos son los puntos de unión más populares porque mantienen la encapsulación de los objetos de forma segura. Como mostró *Renaud Pawlak* en su tesis doctoral, a diferencia de otros puntos de unión, las inserciones de aspecto de invocación de método son componibles.

Para la mayoría de los proyectos, es probable que la intercepción del método basado en modelos corrientes de diseño sean alternativas suficientes a los compiladores AOP o potenciadores de bytecode.

CAPITULO 3. METODOLOGIA AOP

Los gestores de software hasta ahora plantean la necesidad de modularizar la implementación de las incumbencias transversales. Los investigadores han estudiado varias maneras de realizar esta tarea bajo el tópico general de "separación de incumbencias". AOP representa uno de estos métodos. AOP se orienta hacia una separación limpia de las incumbencias para superar los problemas descritos anteriormente.

AOP se centra en la complejidad de las vías de cruce transversal y en los retos de diseño asociado a dicha complejidad. AOP nos proporciona segmentos de código reutilizables llamados "aspectos" que están agrupados de forma modular en componentes AOP centralizados. Los aspectos se pueden mantener desde un único punto y se entrelazan en el código de un objeto durante el tiempo de compilación. Los aspectos dan carácter de módulo a clases que no tienen ninguna otra relación dentro de otra aplicación. AOP añade dimensiones adicionales de reutilización de código, además de las jerarquías de clase OOP. Ello nos permite diseñar clases que se centran en la lógica comercial clave, mezclando después otros aspectos sin el código duplicado. AOP no es un sustituto de OOP, sino una extensión natural que le añade más capacidad.

En muchas maneras, el desarrollo de un sistema usando AOP es similar al otro empleando otra metodología: identificando las incumbencias

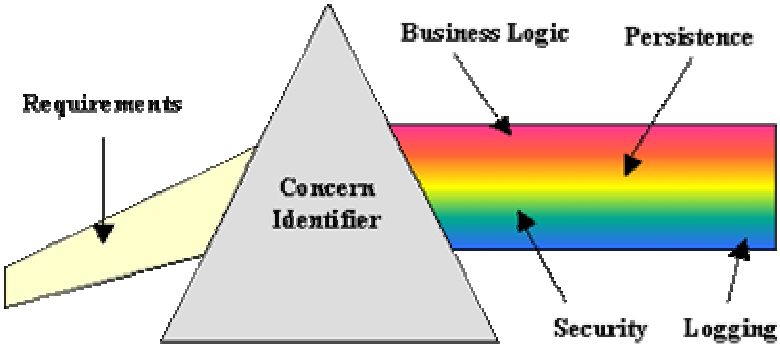


Figura 3. Descomposición de aspectos: Analogía del prisma

La figura 3 presenta un conjunto de requerimientos visto como un haz de luz atravesando un prisma, el cual separa cada incumbencia.

AOP está compuesta de tres pasos distintos en su realización:

1. **Identificación de aspectos:** Descompone los requerimientos para identificar incumbencias comunes y transversales. En esta etapa se separan las incumbencias de nivel funcional de las de nivel de sistema. Por ejemplo, para el módulo de consulta de la tarjeta de crédito, podríamos identificar tres incumbencias: la consulta de la tarjeta en sí, registro, y autenticación.

2. **Implementación de la incumbencia:** Se implementa cada incumbencia de forma separada. Para el ejemplo de la consulta de la tarjeta de crédito, se

implementaría la unidad de consulta en sí, la unidad de registro, y la de autenticación en formas separadas.

3. **Recomposición de Aspectos:** En este paso, un aspecto integrador especifica las reglas de recomposición mediante la creación de unidades modulares – aspectos. El proceso de recomposición, también conocido como *tejedor* o *integrador,* usa esta información para componer el sistema final. Para el ejemplo de la consulta de la tarjeta de crédito, se especificaría, en un lenguaje orientado a AOP, que se debe registrar el inicio y la conclusión de cada operación. También se podría especificar que cada operación debería validar la autenticación del usuario antes de proceder con la lógica de negocio.

ANATOMÍA DE UN LENGUAJE AOP

La implementación de una aplicación siguiendo la metodología AOP, como cualquier otra, se apoya en un conjunto de herramientas para su consecución. Por ejemplo, la implementación de una aplicación con metodología OOP requiere de un lenguaje de programación con ciertas especificaciones –como java- y herramientas tales como el compilador. Sin embargo, una implementación basada en AOP, consiste en:

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

- Uno o más lenguajes de aspectos. El lenguaje de aspectos define la forma de los aspectos – por ejemplo, los aspectos de Aspect J se programan de forma muy parecida a las clases.

- Un tejedor de aspectos para la combinación de los lenguajes. El proceso de mezcla se puede retrasar para hacerse en tiempo de ejecución, o hacerse en tiempo de compilación.

- Un compilador o intérprete. Que normalmente está embebido en el tejedor.

- El programa escrito en el lenguaje base que implementa los componentes.

- Uno o más programas de aspectos que implementan los aspectos.

Para el lenguaje de aspectos AspectJ, por ejemplo, hay un compilador llamado ajc, que tiene una opción de preprocesado que permite generar código java, para ser utilizado directamente por un compilador Java compatible con JDK, y también tienen una opción para generar archivos .class, encargándose él de llamar al compilador Java.

ENTRELAZADO ESTÁTICO VERSUS DINÁMICO

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

Entrelazado Estático

El entrelazado estático implica modificar el código fuente escrito en el lenguaje base, insertando sentencias en los puntos de enlace. Es decir, que el código de aspectos se introduce en el código fuente. Un ejemplo de este tipo de tejedor, es el tejedor de aspectos de AspectJ.

Entrelazado Dinámico

El entrelazado dinámico requiere que los aspectos existan y estén presentes de forma explícita tanto en tiempo de compilación como en tiempo de ejecución. Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Un tejedor dinámico será capaz de añadir, adaptar y remover aspectos de forma dinámica durante la ejecución. Como ejemplo, el tejedor dinámico AOP/ST utiliza la herencia para añadir el código específico del aspecto a las clases, evitando modificar así el código fuente de las clases al entrelazar los aspectos.

Otro ejemplo más completo sería el Junction Point Aspect Language (JPAL), basado en los conceptos de protocolos de meta objetos (metaobject protocols MOP) y meta-programación. En JPAL existe una entidad llamada Administrador de Programas de Aspectos el cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una librería dinámica que almacena los aspectos y permite dinámicamente agregar, quitar o modificar los aspectos, y mandar mensajes a dichos aspectos.

El tejido estático evita que el nivel de abstracción introducido por AOP derive de un impacto negativo en la eficiencia de la aplicación, ya que todo el trabajo se realiza en tiempo de compilación, y no existe sobrecarga en la ejecución. Si bien esto es deseable, el costo es una menor flexibilidad: los aspectos quedan fijos, no pueden ser modificados en tiempo de ejecución, ni existe la posibilidad de agregar o remover nuevos aspectos. Otra ventaja que surge es la mayor seguridad que se obtiene efectuando controles en la compilación, evitando que surjan errores catastróficos o fatales en ejecución. Podemos agregar también que los tejedores estáticos resultan más fáciles de implementar y consumen menor cantidad de recursos.

El tejido dinámico implica que el proceso de composición se realiza en tiempo de ejecución, decrementando la eficiencia de la aplicación. Por otro lado, el postergar el proceso de composición deriva en una mayor flexibilidad y libertad al programador, ya que cuenta con la posibilidad de modificar un aspecto según información generada en ejecución, como también introducir o remover dinámicamente aspectos. Sin embargo, la característica dinámica de los aspectos pone en riesgo la seguridad de la aplicación, ya que se puede alterar dinámicamente el comportamiento de un aspecto o remover un aspecto que posteriormente sea requerido. El tener que llevar mayor información a tiempo de ejecución, y tener que considerar más detalles, hace que la implementación de los tejedores dinámicos sea más compleja.

CAPITULO 4. PROGRAMACION ORIENTADA A ASPECTOS Y ASPECTJ

Desarrollado en Xerox PARC, AspectJ (http://aspectj.org). es una extensión a Java orientada a aspectos de origen abierto y disponible gratuitamente que permite la modularización de recursos compartidos, comprobación de errores, modelos de diseño y distribución Java. Como ilustra la **Figura 1,** AspectJ hace posible la modularización al tomar el código que aparece en varios lugares en un lenguaje base y colocarlo en un solo lugar, junto con instrucciones acerca de cómo insertarlo en los lugares correctos.

El ejemplo más común de código susceptible de este tipo de "separación" es el código de registro, dado que:

- Aparece en muchos lugares en el lenguaje base y no forma parte intrínsecamente de ninguna clase.
- Es conceptualmente distinto de las clases en las que aparece.
- No es una parte intrínseca de la funcionalidad del programa.
- Se inserta y elimina con frecuencia durante el desarrollo.
- A menudo se elimina del lenguaje base final.

Las dos primeras propiedades son importantes, y el código que las exhibe se conoce como crosscuting code o código cruzado. El objetivo de AOP es separar el código cruzado y reunir todas las parte relacionadas en un solo lugar, junto con instrucciones acerca de cómo volver a insertar dicho código en el programa (de forma estática en tiempo de compilación o dinámica en tiempo de ejecución).

INTRODUCCIÓN A ASPECTJ

AspectJ es tanto un lenguaje para la programación orientada a aspectos como un conjunto de herramientas. Los programas son programas AspectJ válidos y pueden tener acceso a todas las bibliotecas estándar Java. El conjunto de herramientas AspectJ incluye un compilador, depurador, tarea ant (para builds automatizados) y plug-ins de varios IDE Java (como JBuilder).

El lenguaje AspectJ se centra en torno a tres conceptos principales:

Puntos de unión (Jointpoints), que definen un punto en el flujo de ejecución de un programa (como por ejemplo, la invocación a un método), el acceso a un campo de datos, a la instanciación de un objeto, o la gestión de un error. Se pueden emplear comodines para hacer que las operaciones se correspondan según su modelo. Los puntos de enlace son mayormente una abstracción; cuando se utiliza AspectJ, empleamos pointcuts.

Cortes de punto (Pointcuts), que contienen uno o más puntos de unión y los enlazan a un "consejo" ("advice"). Cuando el flujo de ejecución llega a un punto de unión, se invoca al pointcut asociado con dicho punto de unión y el punto asociado pasa el contexto a su elemento de consejo asociado. La información del contexto es, fundamentalmente, datos de aplicación tales como argumentos a la invocación de método del punto de unión. Comprender como escribir pointcuts, y cuando se aplica uno determinado, es la parte más difícil del uso de AspectJ.

Consejos (advice), que especifican un segmento de código a ejecutar en un pointcut mediante la información del contexto reunida. Por ejemplo, los pointcuts se utilizan en AspectJ para especificar un conjunto de puntos de unión en un programa donde deseamos que algo pase. El consejo lo asociamos entonces a los pointcuts para decir lo que debe pasar. El consejo suele ser casi siempre código bastante simple. Lo importante del consejo no es que sea especialmente difícil o complejo, sino que se ejecuta cuando se aplica un determinado pointcut.

AspectJ proporciona un compilador que identifica lo puntos de unión en las aplicaciones, e inserta código de pointcut y de consejo directamente a los ficheros de clase Java. AspectJ también puede actuar como pre-compilador y mezclar elementos en los ficheros de código fuente de Java. En cualquiera de los dos casos, AspectJ genera clases totalmente compiladas con Java2.

Pointcuts de tipo call y pointcuts de tipo execution

Los dos pointcuts más comúnmente utilizados son el pointcut de tipo *execution* y el pointcut de tipo *call*. Su sintaxis básica es:

nombre del pointcut():call([firma del método])
nombre del pointcut():execution([firma del método])

donde nombre es el *nombre del pointcut* (y debe ser único) y [*firma del método*] es una expresión que captura un conjunto de métodos. AspectJ tienen un lenguaje entero, basado en coincidencias de cadenas, para especificar firmas. Por ejemplo, *com.world..*.*(..) coincide con todos los métodos en todas las clases que están en un paquete cuyo nombre comienza con com.world. El **listado 1** presenta algunas firmas junto con su explicación.

```
/* Invoca llamadas a cualquier método público que lance una excepción
RemoteException */
pointcut ejemplo1():call(public * * (..) throws RemoteException);

/* Invoca llamadas al constructor de cualquier subclase de la clase
UnicastRemoteObject. El símbolo '+' indica "cualquier subclase". */
pointcut ejemplo2():call(UnicastRemoteObject+.new(..));

/*Invoca la ejecución de cualquier método público de cualquier clase que
pertenezca a cualquier paquete bajo com.world(incluyendo clases en com.world).
El '..' es lo que indica cualquier paquete. */
pointcut ejemplo3():execution(public * com.world..*.*(..));

/* Invoca la ejecución de cualquier método público de cualquier clase que
pertenezca a cualquier paquete bajo com.world (incluyendo clases en com.world).
Los métodos tienen que ser métodos "setter" y retornar void */
pointcut ejemplo4():execution(public void com.world..*.set*(..));
```

Listado 1

Como muestra la **figura 4**, podemos considerar una llamada a un método como el límite de dos instancias. Cuando se invoca un método, un hilo sale del ámbito de una instancia y entra en el ámbito de la otra. Al hacerlo, ha atravesado un punto de unión. Desde el punto de vista de AspectJ, un trazado de pila no es sino una lista de puntos de unión que el hilo actual ha atravesado y que volverá a atravesar conforme la pila se vaya desenvolviendo.

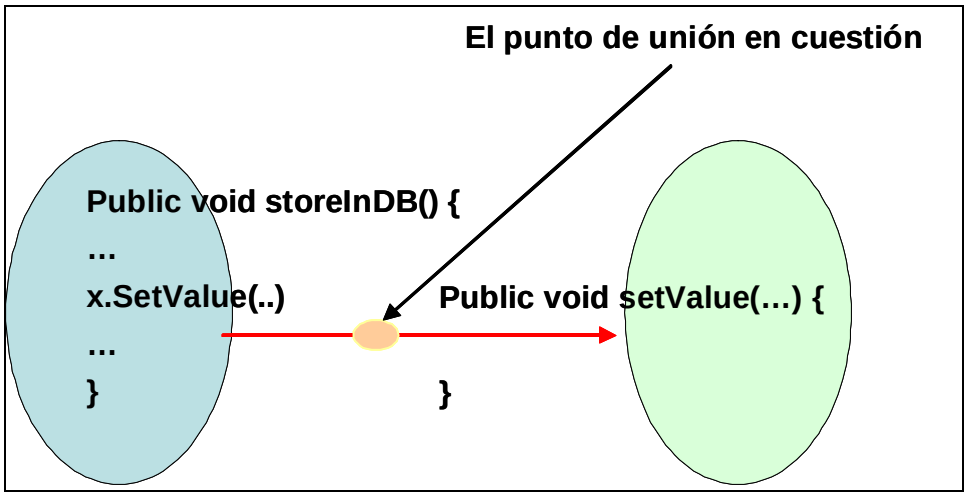


Figura 4. Hilo que sale del objeto de la izquierda y entra en el objeto de la derecha (es decir, atraviesa un punto de enlace)

Un pointcut de tipo *call* especifica un conjunto de puntos de unión justo antes (o después) de la llamada a los métodos. Es decir, si asociamos un consejo a un pointcut de tipo *call* y un hilo está a punto de llamar a un método que coincide con la firma del pointcut, el consejo se ejecutará. Por otra parte, los pointucts de tipo *execution* especifican un conjunto de puntos de unión justo después de la llamada al método (o justo antes que regrese la llamada). En resumen, los pointcuts de tipo *execution* tienen lugar dentro del objeto de destino. En la mayoría de los casos, no importa si el consejo se basa en puntos de enlace *call* o *execution*.

No obstante, existen diferencias entre los pointcuts de tipo *call* y *exection*. Para empezar, *call* y *execution* se ejecutan en momentos diferentes. Lo que significa que si se asocia más de una pieza de consejo utilizando un pointcut de tipo *call* o *execution* puede suponer una diferencia de cómo se ejecuta el programa.

Además, los pointcuts de tipo *call* se basan en los tipos de objetos de tiempo de compilación, según entiende la clase invocadora, mientras que los pointcuts de tipo *execute* se basan en el tipo de tiempo de ejecución del destino de la llamada del método. Por lo tanto, si el objeto que llama no conoce el tipo exacto del destino, sí podría ser importante que el consejo se base en un pointcut de tipo *call* o *execution*. En el **listado 2**, la instancia stringWrapper se declara que es una instancia de Object, pero en tiempo de ejecución, es una instancia de StringWrapper. Incluso aunque tengan las mismas firmas, el pointcut de tipo *call* del listado número 2 nunca es aplicable, mientras que el pointcut de tipo *execution* sí lo es.

```
/* El pointcut call no es aplicable, porque stringWrapper es declarado de tipo
Object (por lo tanto no concuerda con la firma). Pero StringWrapper está en el
paquete com.world, por lo que el pointcut execution si aplica */
pointcut callExample():call(* com.world..*.*(..));
pointcut executionexample():execution(* com.world..**(..));

/* El código */
public static void main(String[] args) {
        if ((args==null) || (args.length==0)) {
                System.out.println("No hay argumentos");
        }
        else {
                System.out.println("Argumentos:\n");
                for (int i=0; i<args.length; i++) {
                        Object stringWrapper = new StringWrapper(args[i]);
                        System.out.println("\t" + stringWrapper.toString() + "\n");
                }
        }
}
```

Listado 2

La diferencia final entre los pointcuts de tipo *call* y *execution* tiene que ver con una deficiencia en la versión actual de AspectJ. Como AspectJ se implementa en el compilador y requiere que dispongamos del código fuente de todas las clases, no se pueden especificar pointcuts de tipo *execution* para clases paras las que no se disponga del código fuente.

Pointcuts cflow

La idea que subyace a un pointcut de tipo *cflow* (flujo de control) es que con frecuencia deseamos insertar un consejo basándonos en algo parecido a un trazado de la pila, y no solo en la línea de código actualmente en ejecución.

Los pointcuts de tipo cflow se definen en términos de otro pointcuts. La sintaxis básica de un pointcut *cflow* es:

nombre de poincut():cfow([nombre de un pointcut diferente])
nombre de poincut():cfow([nombre de un pointcut diferente])

El punto actual de ejecución está en el *cflow* de un pointcut si éste se encuentra por debajo suyo en el trazado de la pila por ejemplo, si al desenvolver la pila el hilo de ejecución actual atraviesa el pointcut).

La diferencia entre *cflow* y *cflowbelow* es que un pointcut está dentro de su propio *cflow*, pero no está dentro de su *cflowbelow* (*cflow* y *cflowbelow* es lo mismo que <= y <).

En sí mismos, los pointcuts de tipo *cflow* no son muy útiles. Lo que los hace útiles es un álgebra de pointcuts. Se pueden utilizar todos los operadores boléanos

estándar (¡, &&, ||, etc.) con los pointcuts. La combinación de pointcuts permite crear instrucciones realmente sofisticadas.

Por ejemplo, supongamos que hemos escrito una biblioteca y que, en otro programa, deseamos registrar todas las llamadas en la biblioteca. Es decir, deseamos registrar las llamadas que entran en la biblioteca desde otro código, no las llamadas de un objeto de la biblioteca a otro ni las que se originan en el código de la biblioteca (incluso si algunas clases que no son de biblioteca aparecen en el medio del trazado de la pila. Esto se puede hacer en tan solo dos pasos: primero escribimos el pointcut de tipo *call* que especifique los métodos públicos de la biblioteca, lo llamamos "pointcut A". Luego, el pointcut definido por la expresión *A && ¡cflowbelow(A)* se captura exactamente de la forma deseada (distingue en tipo de ejecución, todas las llamadas a un método en la biblioteca que no están en el *cflow* de una llamada a un método en la biblioteca. El **listado número 3** es un ejemplo completo de este tipo de pointcut.

```
/* El pointcut enteringFromExternalSource  se asocia al punto de unión que
implica a los métodos públicos y no están en el cflowbelow de la invocación a un
método público. Se necesita usar && y ! para definer este pointcut. */
pointcut comWGrossoPublicMethods():call(public * com.wgrosso..*.*(..));
pointcut enteringFromExternalSource():comWGrossoPublicMethods() &&
!cflowbelow(comWGrossoPublicMethods());
```

Listado 3

Agregar Consejos

Existen tres tipos de consejos: consejo before, consejo after y consejo around. Los consejos before y after son sencillos. El consejo before se ejecuta justo antes de que lo haga el punto de que lo haga el punto de código indicado por el punto de enlace. Por su parte, el consejo after se ejecuta justo después de que lo haya hecho el punto del código indicado por el punto de enlace. El **listado número 4** imprime la secuencia "12345", que ilustra la sintaxis básica de los consejos before y after. Como representa la **figura 5**, el motivo de que el listado número 5 imprima "12345" es que las instrucciones se ejecutan en el siguiente orden:

1. El consejo before asociado al pointcut de tipo call.
2. El consejo before asociado al pointcut de tipo execution.
3. El consejo after asociado al pointcut de tipo execution.
4. El consejo after asociado al pointcut de tipo call.

```
/* The entire aspect. */
package com.wgrosso.aspectUpsam;
import org.aspect.lang.*;
public aspect Aspect_PrintThree{
        pointcut CallStringWrapper(): call(* com.wgross..*.*(..));
        pointcut ExecuteStringWrapper(): execution(* com.wgross..*.*(..));

        before():CallStringWrapper() {
                System.out.println("1");
        }
        before():ExecuteStringWrapper () {
                System.out.println("2");
        }
        after():CallStringWrapper() {
                System.out.println("4");
        }
        after():ExecuteStringWrapper () {
                System.out.println("5");
        }
}

/* The entire class */
Package com.wgrosso.aspectUpsam;
Public class PrintThree {
        Public static void main(String[] args) {
                System.out.println("3");
        }
}
```

Listado 4

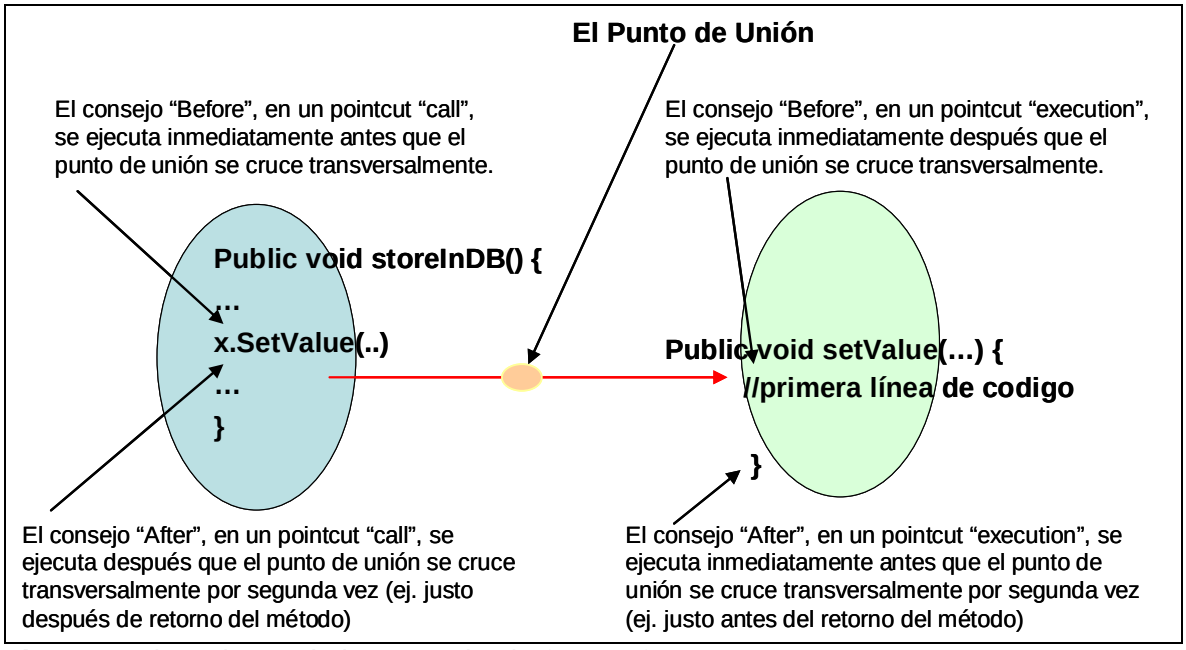


Figura 5. Ejecuciones de los consejos befote y after

El consejo around es más complicado que el consejo before o after. El consejo around funciona de forma que parte del código del consejo se ejecuta antes que el pointcut, y dicho código debe llamar al proceso para que el contenido del método se ejecute; véase el **listado número 5**.

```
/* Necesitamos definir un pointcut. */
pointcut callExample():call(public void com.wgrosso..*.*(..));

/* El consejo Around debe proceder a llamar cuando ocurra la invocación al
método original. Nótese que el consejo Around necesita un tipo de retorno que
concuerde con el pointcut. */
void around(): callExample() {
        /* Any code befote the proceed ejecutes befote the method body. */
        Proceed(); //proceed executes the method body
        /* Any code after the proceed executes after the method body. */
}
```

Listado 5

CAPITULO 5. CASO PRÁCTICO: SINCRONIZACIÓN DE AGENDAS PARA DISPOSITIVOS MÓVILES

En vista de la gran variedad de dispositivos informáticos y de comunicación existentes actualmente; por ejemplo, teléfonos celulares, asistentes digitales personales (PDA) y ordenadores portátiles; la posibilidad de sincronizarlos es de gran interés para el usuario final, debido a que la mayoría de los dispositivos comparte aplicaciones similares. En este capítulo analizamos brevemente los requerimientos funcionales necesarios en la sincronización de agendas para dispositivos móviles, y posteriormente planteamos un diseño orientado a objetos con la finalidad de identificar las incumbencias transversales en tiempo de diseño y presentamos parte del código de los aspectos de “sincronización” y “acceso a datos” en AspectJ.

REQUERIMIENTOS FUNCIONALES

- Sincronización
- Sincronización Selectiva
- Resolución de Conflictos Inteligente
- Consistencia Basada en bloqueo exclusivo de la agenda
- Consistencia Basada en privilegios de autorización
- Consistencia Basada en restricciones de acceso

REQUERIMIENTOS DE ANÁLISIS

Se pretende que los usuarios a través de sus dispositivos portátiles (PDA, teléfonos móviles, portátiles, etc), puedan acceder a información de las agendas de otros usuarios independientemente de la plataforma, el fabricante o la aplicación.

La sincronización de agendas consiste en que dos o más personas enfrenten sus dispositivos personales (estos descubren que otros dispositivos están pidiendo sincronizar sus agendas) y entonces se intercambian: rango de fechas, y propuesta de día y hora de reunión o bien un parámetro que indique que los dispositivos busquen posibles días y horas.

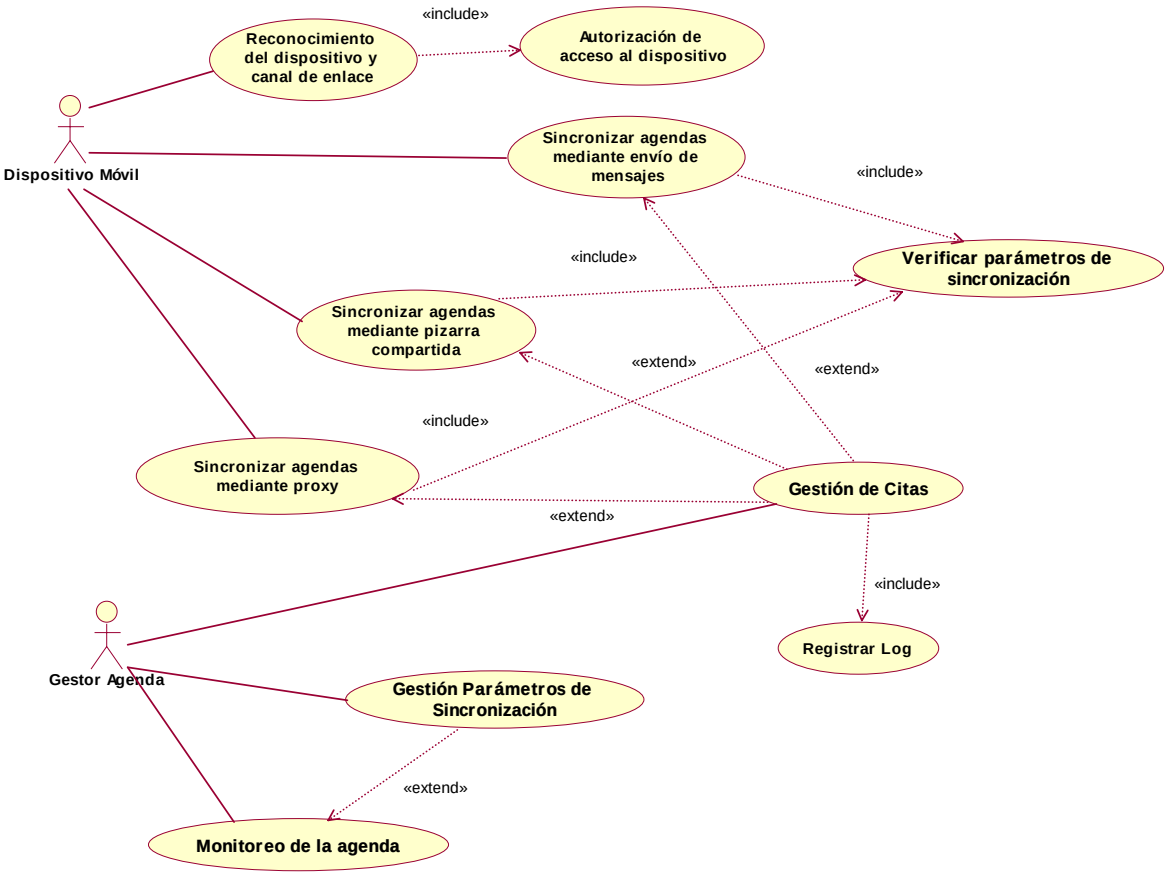
Casos de Uso

Actores

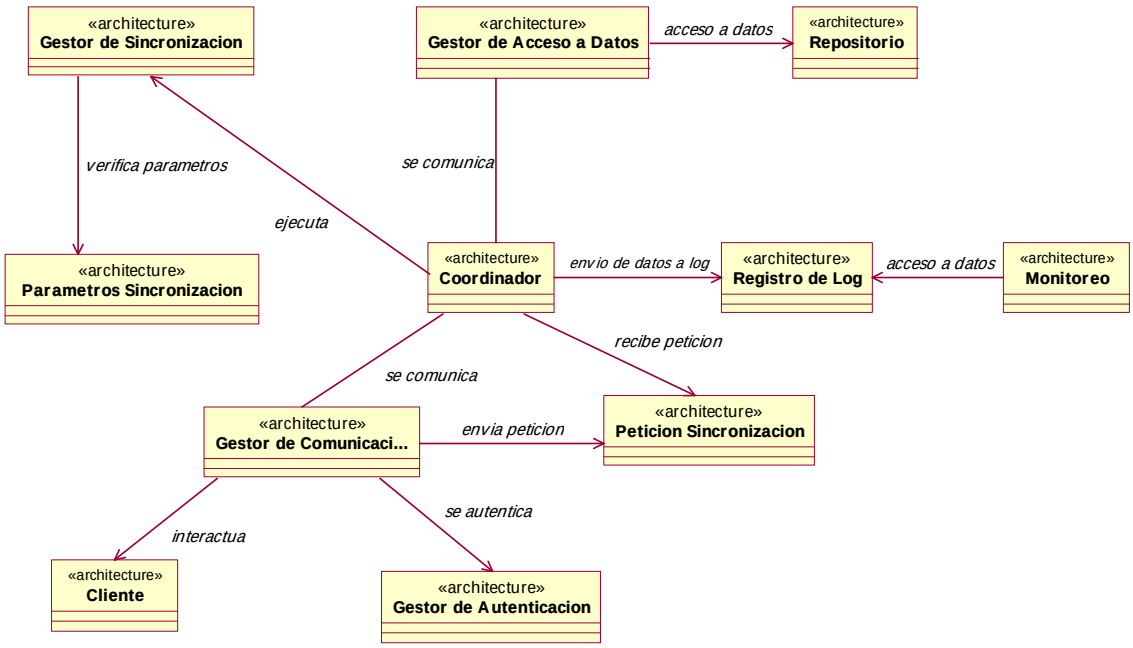
Para esta aplicación, se han identificado dos actores

- **Dispositivo móvil**. Quien puede realizar las siguientes tareas:
    - Reconocimiento de dispositivo y canal de enlace
    - Sincronización de agendas mediante pizarra compartida (WiFi)
    - Sincronización de agendas mediante Proxy (UMTS)
    - Añadir Cita
    - Eliminar Cita
    - Actualizar Cita
    - Generar Log
    - Autorización de acceso a dispositivo
    - Sincronización de agendas según el canal mediante envío de mensajes (Bluetooth, WiFi o UMTS)

- **Gestor de la Agenda**. Quien puede realiza las siguientes tareas:
  - Añadir Cita
  - Actualizar Cita
  - Eliminar Cita
  - Gestión de parámetros de sincronización



DISEÑO DE LA ARQUITECTURA DE LA APLICACIÓN



Identificación de Incumbencias

- *Incumbencia de almacenamiento de datos*
  Responde a la pregunta: ¿Como serán almacenados los datos en el repositorio?
  - usando DBMS
  - usando sistema de archivos XML
  - otros

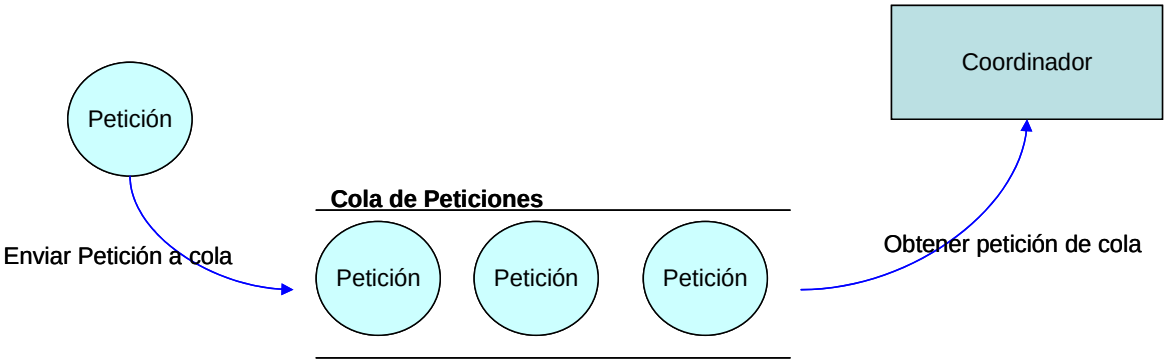
- *Incumbencia de acceso a los datos.* Abarca las siguientes operaciones:
  - Añadir cita al repositorio
  - Eliminar cita del repositorio
  - Actualizar cita en el repositorio
  - Verificar existencia de cita
  - Comprobar restricción de acceso

- *Incumbencia de Sincronización.* El objetivo de esta incumbencia es el de conservar la consistencia de información de la agenda, para ello se deben tener el cuenta las siguientes operaciones:
  - Resolución de conflictos de acceso
  - Definir el flujo de mensajes entre el dispositivo cliente y servidor

- *Incumbencia de Registro de Log*
  Tiene por objetivo registrar la ejecución de las operaciones de la aplicación

- *Incumbencia de Monitoreo y registro de parámetros de sincronización.*

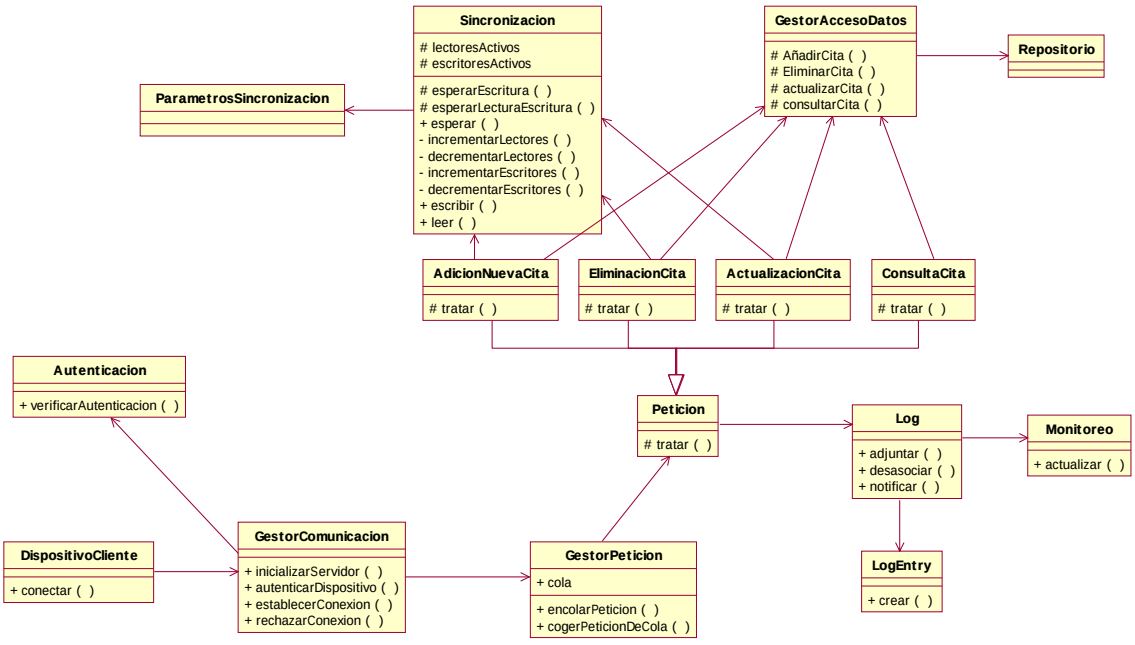
- *Gestión de cola de petición de sincronización.* Se pretende ordenar las peticiones de sincronización mediante una estructura de cola.



- *Incumbencia de conexión*
  - Estableciendo conexión entre los dispositivos servidor y cliente
  - Enviando petición de sincronización a la “cola de peticiones”

- *Incumbencia de Autenticación/Autorización*
  - Determina si el dispositivo tiene permiso de acceso a la cola de sincronización.
  - En base a los parámetros de sincronización, determina si la petición de sincronización es permitida.

DISEÑO ORIENTADO A OBJETOS

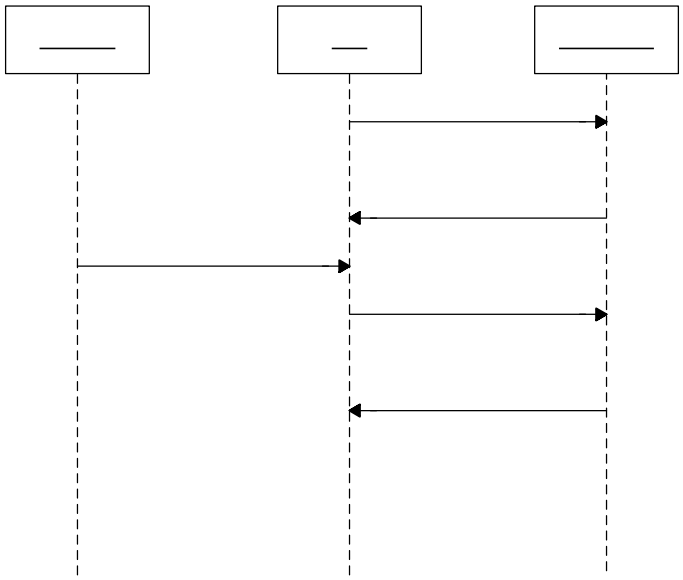


Patrones de Diseño Aplicados

- *Command Pattern (patrón orden)*
  Empleamos este patrón para encapsular una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer la operación.

Debido a lo anterior, el componente “Coordinador” es eliminado.

- *Observer Pattern (patrón observador)*
  Definimos una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.
  - Sujeto → Log
  - Observador → Monitoreo



Peticion Log M

Identificando Aspectos Mediante la Metodología ASAAM

ASAAM(Aspectual Software Architecture Analysis Method) es una metodología propuesta con la finalidad de identificar y especificar aspectos en la etapa de análisis de la arquitectura del software. Nace como extensión de la metodología SAAM (Software Architecture Analysis Method) e introduce una serie de reglas heurísticas que ayudan a deducir aspectos de la arquitectura del software con sus correspondientes componentes enredados (tangled) a partir de la evaluación de sus escenarios. De acuerdo a esta propuesta, existen muchas similitudes entre escenarios y aspectos. En primer lugar, los aspectos son incumbencias transversales, esto es, incumbencias que interactúan sobre diferentes componentes. De igual modo, los escenarios (indirectos) también requieren de cambios en muchos componentes y pueden ser llamados componentes transversales. En segundo lugar, los aspectos se derivan de la descripción del problema y de la descomposición de su diseño. De manera similar, la categorización de los escenarios en directos o indirectos es completamente dependiente de la descripción del problema y de su diseño de arquitectura. Basados en estas observaciones, se plantea que el análisis de los escenarios (SAAM) provee aspectos de arquitectura del software potenciales.

ribir()

lizar()

car()

lizar()

Esta metodología plantea cinco pasos para la identificación de aspectos:

- *Desarrollo de la arquitectura del software candidato*

- *Desarrollo de escenarios*
- *Evaluación de escenarios en forma individual e identificación de aspectos en cada uno de ellos*
- *Valoración de interacción con el escenario y clasificación de componentes*
- *Refactorización de la arquitectura*

Se plantean los siguientes escenarios:

Escenario 1: Sincronización de agendas independiente del canal de enlace
Se pretende sincronizar varios tipos de dispositivos (PDA, Teléfonos móviles, Portátiles) con interfaces de comunicación inalámbrica.
La sincronización podría hacerse de varias formas diferentes:

- Si usamos Bluetooth, por ejemplo, mediante una conexión cliente/servidor P2P instalado en los dispositivos que se intercambien mensajes.
- Si usamos WiFi se podría usar TCP(IP para intercambiar mensajes o usar una pizarra compartida para especificar cosas más complicadas que mensajes de texto.
- Si usamos UMTS se podría utilizar un Proxy en el que primero se envía el rango de fechas que los usuarios establecieron y allí se ejecuta el servidor que envía de vuelta a los móviles la posible fecha de reunión.

Implicaciones
Si analizamos este escenario, identificaremos claramente el aspecto de sincronización, que si se programaría bajo OOP, produciría una gran cantidad de código repetitivo en diferentes clases de implementación. Sin embargo, mediante AspectJ, podemos aislar dicho código en un sólo lugar y hacer referencia al mismo en tiempo de compilación mediante pointcuts.

Escenario 2: Cambio de la representación del repositorio de datos
Se requiere cambiar de tipo de repositorio de datos, por ejemplo de Sistema de gestión de base de datos relacional a ficheros XML.

Fecha	Hora Inicio	Actividad	Descripcion	Duracion (mins)	Lugar	Estado	Origen
05/05/2004	9:00	Cita	Cita médica con el otorrino	30	millenium	pasado	personal
06/05/2004	15:00	Reunión	Revisión plan acción marketing	120	sala cristal	cancelado	sync
09/05/2004	8:00	Recordatorio	Cumpleaños Pepito Perez	5		pendiente	personal
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

Tabla: Movimientos de la Agenda

Implicaciones
La realización de éste escenario requiere de cambios en la arquitectura. El impacto de estos cambios influye sobre muchas clases del diseño, como consecuencia de ello se obtendría código repetitivo y disperso.

Escenario 3: Extender el ámbito de monitorización
Si pretendemos por ejemplo, monitorizar el estado de la cola de peticiones de sincronización, monitorizar el proceso de sincronización y/o monitorizar el estado de conexión de los dispositivos, tendríamos necesariamente que pensar en muchos cambios de diseño de la arquitectura actual.

Solución Basada en Objetos e Implicaciones

Una posible solución es la de extender las clases de registro de la actividad de la aplicación (Logging). Sin embargo, esto conlleva a entradas de registro redundantes y en muchos casos irrelevantes.

Otra solución podría ser la de asociar una nueva clase Monitor a cada clase que se requiera. Sin embargo, esto traería consigo un alto acoplamiento de código disperso y enredado.

CODIFICANDO ASPECTOS CON ASPECTJ

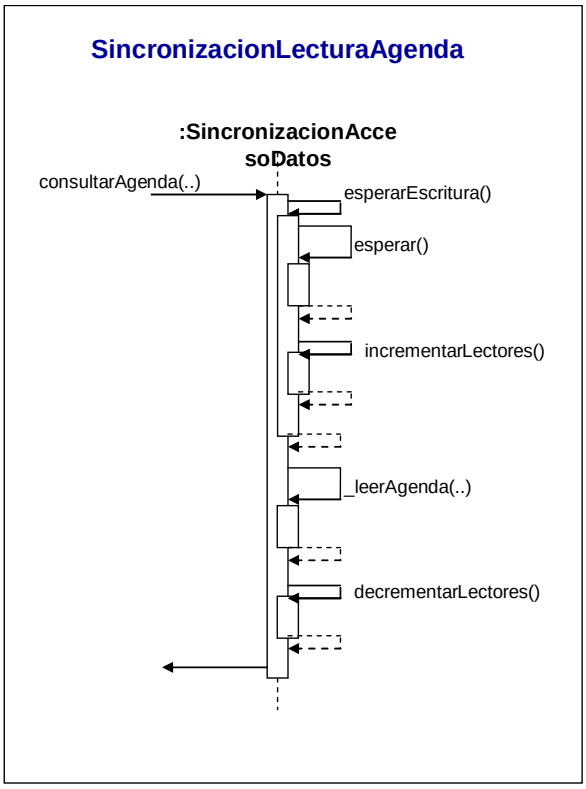
Implementación del aspecto de sincronización

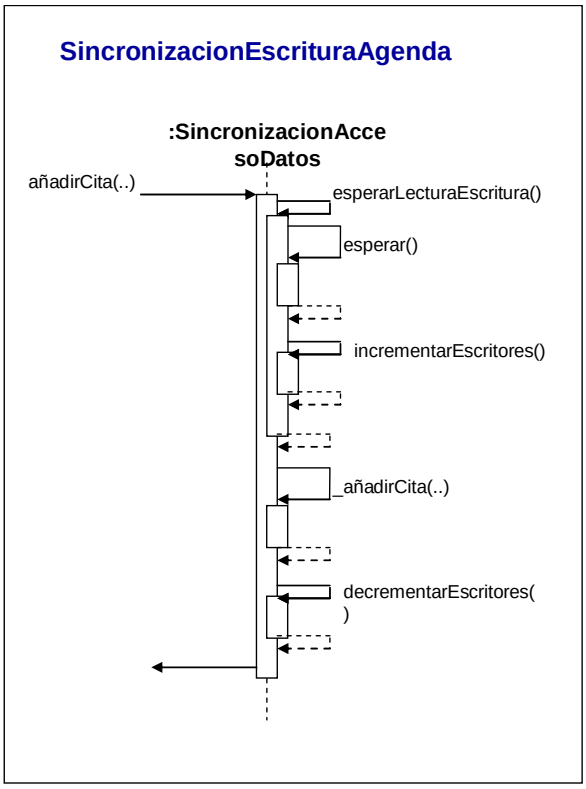
```
public aspect SincronizacionAccesoDatos {
        pointcut write(GestorAccesoDatos g): instanceof(g) & receptions(
                void anadirCita(AdicionNuevaCita),
                void eliminarCita(EliminacionCita));

        pointcut read(GestorAccesoDatos g): instanceof(g) & receptions(
                void consultarCita(ConsultaCita));

        before (GestorAccesoDatos g): write(g) {
                g.esperarLecturaEscritura();}
        after (GestorAccesoDatos g): write(g) {
                g.decrementarEscritores();}

        before (GestorAccesoDatos g): read(g) {
                g.esperarEscritura();}
        after (GestorAccesoDatos g): read(g) {
                g.decrementarLectores();}
}
```





Las ventajas de este código son la clara separación del comportamiento cruzado (crosscutting behaviour) que aparece en el diseño base y colocarlo en un solo lugar, junto con instrucciones acerca de cómo insertarlo en los lugares correctos. El diagrama de secuencias de arriba, muestra el orden de ejecución del aspecto de sincronización para los escenarios de lectura y escritura de la agenda.

Implementación del aspecto de Acceso a Datos

```
public aspect accesoDatos {
        pointcut datos(String param, Cita c):
                call(void GestorAccesoDatos.anadirCita(String, Cita)) &&
args(param, c) ||
                call(void GestorAccesoDatos.eliminarCita(String, Cita)) &&
args(param, c) ||
                call(void GestorAccesoDatos.actualizarCita(String, Cita) &&
args(param, c) ||
                call(void GestorAccesoDatos.consultarCita(String, Cita)) && args(s, c);

        before(String param, Cita c): datos(param, c) {

                if repositorio es base de datos relacional {
                        cita.name = Sql query 1
                        cita.actualizar = Sql query 2
                        cita.consultar = Sql query 3
                }

                If repositorio es XML
                ….
        }
}
```

Una rutina java que detecta el tipo de repositorio y construye el objeto correspondiente en base a los campos obtenidos, el cual será empleado por el resto de los componentes.

DISEÑO UML ORIENTADO A ASPECTOS

Diagrama de Clases del aspecto de Sincronización

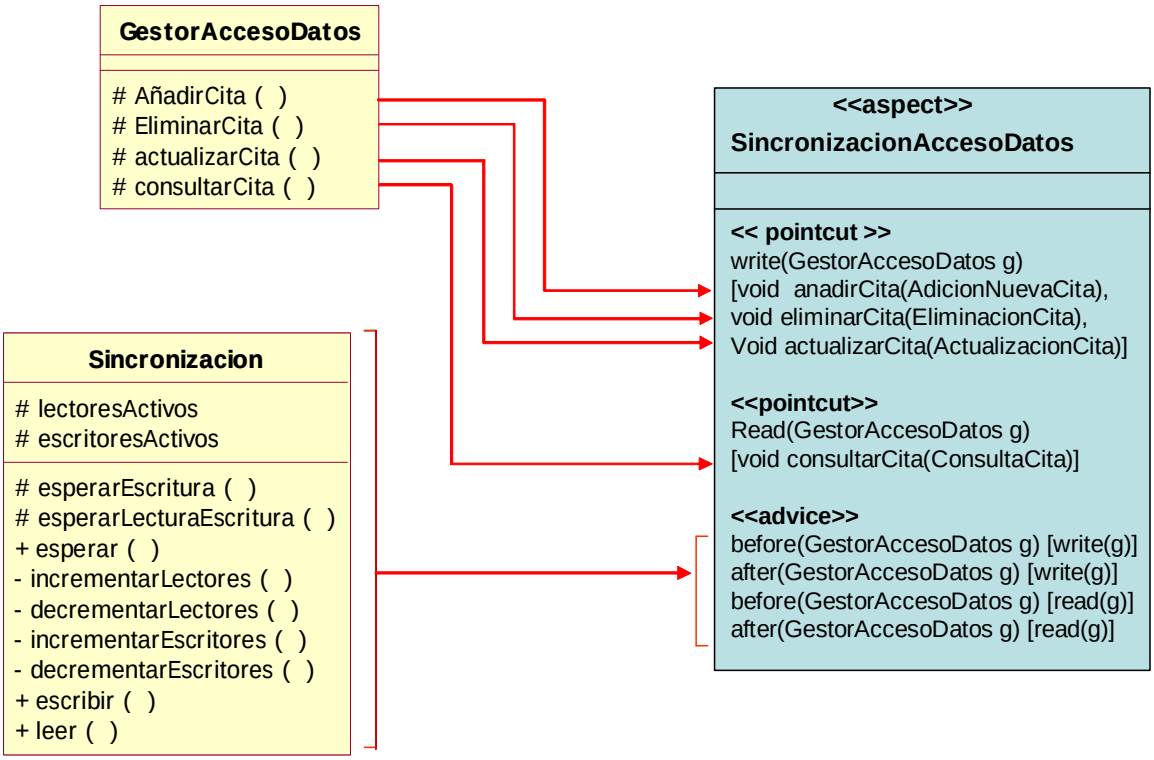
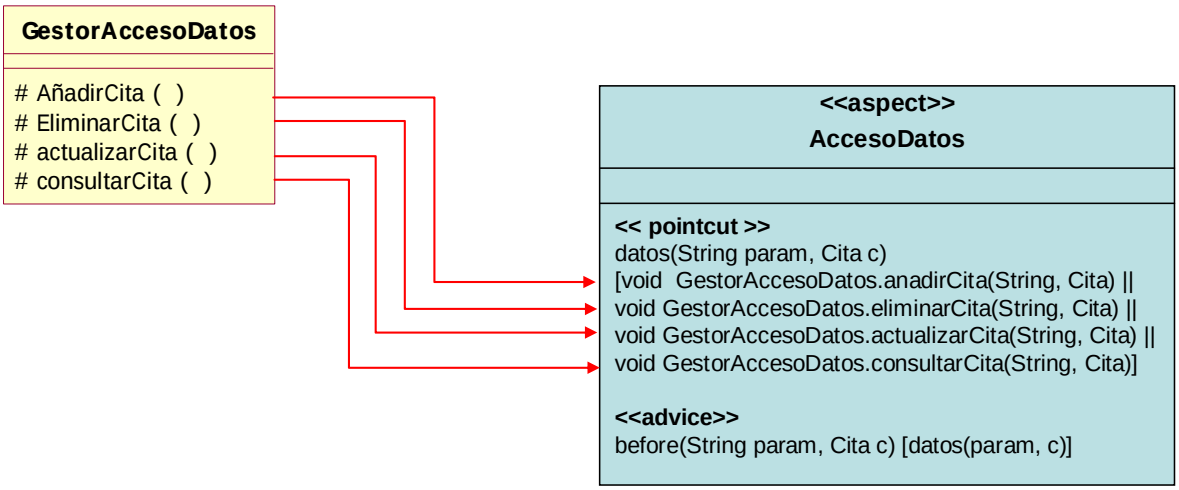


Diagrama de Clases del aspecto de Acceso a Datos



CONCLUSIONES

- Para este trabajo, hemos considerado la sincronización de las agendas en un solo sentido desde el cliente. El cliente siempre es el que inicia la sincronización enviando un mensaje de petición al servidor. El servidor procesa los datos que recibe en la petición y los unifica con los que posee, pero éste no envía las modificaciones que realiza de vuelta al cliente.

- Otros aspectos potenciales son los del manejo de errores, registro de las actividades del sistema, etc. En el trabajo de investigación del siguiente curso, pretendo continuar con la implementación de los mismos.

- Los patrones de diseño orientado a objetos presentan limitaciones para resolver eficazmente el problema de las incumbencias transversales que se disparsen a lo largo de muchas clases de negocio.

- Una de las dificultades que está encontrando la tecnología AOP para tener una mayor difusión es la falta de estándares para representar los aspectos en las fases tempranas del ciclo de vida del software. Para el presente trabajo, nos hemos basado en la representación UML de aspectos de *Siobhan Clarke*, la cual se basa en la observación de que existen patrones en la forma en la que los aspectos se relacionan con las clases bases (Composition Patterns), lo que permite el diseño independiente de unos y otros.

- La maduración de la programación orientada a aspectos como tecnología para el desarrollo software y el trabajo de los muchos grupos de investigación que se están dedicando a ella nos llevará a la obtención de mejores técnicas para la documentación de los sistemas software.

- En este trabajo se han descrito, desde un punto de vista técnico, las características de la tecnología AOP, así como los servicios subyacentes que proporciona como nueva metodología de desarrollo. Finalmente, se ha presentado un estudio práctico de una aplicación de sincronización de agendas en dispositivos móviles empleando AspectJ como una solución apropiada para la codificación de aspectos.

ANEXOS

A. RESUMEN Y EJEMPLOS DE SINTAXIS DE LA SENTENCIA "POINTCUT" EN ASPECTJ

Pointcuts call a métodos y constructores

Los pointcuts call a métodos y constructures captura el punto de ejecución inmediatamente después de evaluar los argumentos del método, pero antes de la ejecución del método mismo. Estos toman la forma de call(MethodOrConstructorSignature). La tabla 1 muestra ejemplos de estos pointcuts.

Tabla 1. Pointcuts call a métodos y constructores

Pointcut	Descripción
call(public void MyClass.myMethod(String))	Llama al método myMethod() de la clase MyClass que toma un String como argumento, que retorna un dato de tipo void, y que es de acceso public
call(void MyClass.myMethod(..))	Llama al método myMethod() de la clase MyClass que toma cualquier tipo de argumento, que retorna un dato de tipo void, y cuyo tipo de acceso es cualquiera.
call(MyClass.myMethod(..))*	Llama al método myMethod() de la clase MyClass que toma cualquier tipo de argumento y retorna cualquier tipo de dato
call(MyClass.myMethod*(..))*	Llama a cualquier método cuyo nombre comience por "myMethod" de la clase MyClass
call(MyClass.myMethod*(String,..))*	Llama a cualquier método cuyo nombre comience por "myMethod" de la clase MyClass y que su primer argumento sea de tipo String
call(*.myMethod(..))*	Llama al método myMethod() de cualquier clase del paquete por defecto
call(MyClass.new())	Llama a cualquier constructor de la clase MyClass' que no tome ningún número de argumentos.
call(MyClass.new(..))	Llama a cualquier constructor de la clase MyClass' que tome cualquier tipo y número de argumentos.
call(MyClass+.new(..))	Llama a cualquier constructor de la clase o subclase de MyClass. (Las subclases son indicadas por el uso del comodín '+')
*call(public * com.mycompany..*.*(..))*	Llama a cualquier método público de cualquier clase y paquete contenida en el paquete com.mycompany

Pointcut execution de métodos y constructores

Los pointcuts execution de métodos y constructures captura la ejecución del método. En contraste con los pointcuts call, estos pointcuts representan el cuerpo mismo del método o constructor. Estos toman la forma de execution(MethodOrConstructorSignature). La tabla 2 muestra ejemplos de estos pointcuts.

Tabla 2. Pointcut execution de métodos y constructores

Pointcut	Descripción
execution(public void MyClass.myMethod(String))	Ejecución del método myMethod() de la clase MyClass que toma un String como argumento, que retorna un dato de tipo void, y que es de acceso public
execution(void MyClass.myMethod(..))	Ejecución del método myMethod() de la clase MyClass que toma cualquier tipo de argumento, que retorna un dato de tipo void, y cuyo tipo de acceso es cualquiera.
execution(MyClass.myMethod(..))*	Ejecución del método myMethod() de la clase MyClass que toma cualquier tipo de argumento y retorna cualquier tipo de dato
execution(MyClass.myMethod*(..))*	Ejecución de cualquier método cuyo nombre comience por "myMethod" de la clase MyClass
execution(MyClass.myMethod*(String,..))*	Ejecución de cualquier método cuyo nombre comience por "myMethod" de la clase MyClass y que su primer argumento sea de tipo String
execution(*.myMethod(..))*	Llama al método myMethod() de cualquier clase del paquete por defecto
execution(MyClass.new())	Ejecución de cualquier constructor de la clase MyClass' que no tome ningún número de argumentos.
execution(MyClass.new(..))	Ejecución de cualquier constructor de la clase MyClass' que tome cualquier tipo y número de argumentos.
execution(MyClass+.new(..))	ejecuciónde cualquier constructor de la clase o subclase de MyClass.
*execution(public * com.mycompany..*.*(..))*	Ejecución de cualquier método público de cualquier clase y paquete contenida en el paquete com.mycompany

Pointcuts de acceso al atributo

Los pointcuts de acceso al registro capturan el acceso de lectura y escritura a un atributo de la clase. Poe ejemplo, se podría capturar todos los acceso al atributo out de la clase System (como en System.out). Se podría de la misma forma capturar el acceso a la lectura o escritura. Por ejemplo, capturar la escritura en el atributo x de MiClase, como en MiClase.x = 5. El pointcut de acceso de lectura toma la forma get(FirmaDelAtributo), mientras que el pointcut de acceso de escritura toma la forma set(FirmaDelAtributo). FirmaDelAtributo puede usar comodines de la misma manera que MethodOrConstructor en los pointcuts call y execution.

Tabla 3. Pointcuts de acceso al atributo

Pointcut	Description
get(PrintStream System.out)	Ejecución de acceso de lectura al atributo out de tipo PrintStream de la clase System
set(int MyClass.x)	Ejecución de acceso de escrituta al atributo x de tipo int de la clase MyClass

Pointcuts de manejo de excepciones

Los pointcuts de manejo de excepciones, capturan la ejecución del manejo de excepciones de tipos específicos. Estos toman la forma handler(PatronDeTipoDeExcepcion).

Tabla 4. Pointcuts de manejo de Excepciones

Pointcut	Descripción
handler(RemoteException)	Ejecución del manejo del bloque try-catch de tipo RemoteException
handler(IOException+)	Ejecución del manejo del bloque try-catch de tipo IOException o de cualquiera de sus subclases.
handler(CreditCard)*	Ejecución del manejo del bloque try-catch de cualquier tipo cuyo nombre comienza por CreditCard

Pointcuts de inicialización de clases

Los pointcuts de inicialización de clases capturan la ejecución de la inicialización de clases estáticas, que se especifica en el bloque static en la definición de la clase. Estos toman la forma staticinitialization(TypePattern).

Tabla 5. pointcuts de inicialización de clases

Pointcut	Descripción
staticinitialization(MyClass)	Ejecución del bloque estático de la clase MyClass
Staticinitialization(MyClass+)	Ejecución del bloque estático de la clase MyClass o de cualquiera de sus subclases

Pointcuts basados en la estructura Léxica

Los pointcuts basados en la estructura Léxica capturan todos los puntos de unión que se encuentran dentro de la estructura léxica de una clase o un método. El pointcut que captura el código lexicográfico de una clase, incluyendo una clase interna, toma la forma within(TypePattern). El pointcut que captura el código

lexicofráfico de un método o constructor, incluyendo cualquier clase local, toma la forma withincode(MethodOrConstructorSignature).

Tabla 6. Pointcuts basados en la estructura Léxica

Pointcut	Descripción
within(MyClass)	Cualquier pointcut dentro del ámbito lexicográfico de la clase MyClass
within(MyClass)*	Cualquier poincut dentro del ámbito lexicográfico de las clases cuyo nombre comienza por "MyClass"
withincode(MyClass.myMethod(..))*	Cualquier pointcut dentro del ámbito lexicográfico de cualquier método de la clase MyClass

Pointcuts basados en flujo de control

Los pointcuts basados en flujo de control, capturan pointcuts basados en otros pointcuts de flujos de control (el flujo de las instrucciones de programa). Por ejemplo, si en una ejecución, el método a() llama al método b(), entonces se dice que b() está en el flujo de control de a(). Con los pointcuts basados en flujo de control, se puede, por ejemplo, capturar todos los métodos, accesos a atributos, y manejos de excepciones causados por invocación a un método. El pointcut que captura los pointcuts en el flujo de control de algún otro pointcut, incluyendo el del mismo pointcut especificado, toma la forma de cflow(pointcut), mientras que el que excluye al del mismo poitncut especificafo toma la forma de cflowbelow(pointcut).

Tabla 7. Pointcuts basados en flujo de control

Pointcut	Descripción
cflow(call(MyClass.myMethod(..))*	Todos los puntos de unión en el flujo de control de un pointcut call a cualquier método de nombre myMethod() de la clase MyClass, incluyendo las llamadas del propio método especificado.
cflowbelow(call(MyClass.myMethod(..))*	Todos los puntos de union en el flujo de control de un pointcut call a cualquier método de nombre myMethod() de la clase MyClass excluyendo las llamadas del propio método especificado.

Pointcuts self, target y de argumento de tipos

Los Pointcuts self, target y de argumento de tipos, capturan los puntos de unión basados en el propio objeto, en el objeto destino, y en los tipos de los argumentos. Estos son los únicos constructores que pueden capturar el contexto de los puntos de unión. El pointcut que captura los pointcuts basados en el propio objeto, toma la forma de this(TypePattern or ObjectIdentifier), mientras que el basado en el destino toma la forma target(TypePattern or ObjectIdentifier). Los

pointcuts basados en el argumento, toma la forma args(TypePattern or ObjectIdentifier, ..).

Tabla 8. Pointcuts self, target y de argumento de tipos

Pointcut	Description
this(JComponent+)	Todos los puntos de unión donde éste objeto es instancia de JComponent
Target(MyClass)	Todos los puntos de unión donde el objeto desde el cual el método es llamado es de tipo MyClass
args(String,..,int)	Todos los puntos de unión donde el primer argumento es de tipo String y el último es de tipo int
args(RemoteException)	Todos los puntos de unión donde el tipo de argumento o el tipo de manejo de excepción es RemoteException

Pointcuts de test-condicional

Los pointcut de test-condicional, capturan los puntos de unión basados en algunas comprobaciones condicionales en el punto de unión. Toma la forma de if(BooleanExpression).

Tabla 9. Pointcuts de test-condicional

Pointcut	Description
if(EventQueue.isDispatchThread())	Todos los puntos de unión donde la expresión EventQueue.isDispatchThread() es verdadera.

Usando operadores ||, &&, y ! con pointcuts

Se pueden usar operadores || y && para combinar pointcuts nomenclados y anónimos. Por ejemplo, para designar un pointcut para llamar al método m1() o m2() de la clase MyClass, se podría usar call(* MyClass.m1()) || call(* MyClass.m2()). Para designar un pointcut para llamar al método m1() de la clase MyClass que está en el flujo de control de la llamada al método m2 de la clase MyClass, se podría usar call(* MyClass.m1()) && cflow(call(* MyClass.m2()).

Igualmente, se podrían usar los pointcuts tales como this(), within(), y cflow() con otros pointcuts usando && para capturer un pequeño subconjunto del punto de unión.

El operador “!” se usa para especificar los pointcuts que no se desea capturer por el pointcut especificado. Por ejemplo, para designar una llamada a todos los métodos, excepto m1() de la clase MyClass, se usaría el pointcut !call(* MyClass.m1()).

BIBLIOGRAFÍA

1. Aspect Oriented Software Development.http://aosd.net

2. 2. The AspectJTM Programming Guide, theAspectJ Team, Xerox Parc Corporation (http://www.parc.xerox.com/csl/projects/AOP.html)

3. 3. E. Gamma, R Helm, R. Johnson y J. Vlissidess, Design patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

4. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm y W. Griswold. An Overview of AspectJ, ECOOP 2001

5. AspectJ in Action. Practical Aspect – Oriented Programming. Ramnivas Laddad. 2003

6. J. Suzuki y Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. 3rd AOP workshop at ECOOP'99 (Lisbon, Portugal, June 1999).

7. Links a documentos AOP http://www.vvs.newu.edu/home/lieber/AOP.html

8. J.L. Herrero, M. Sánchez y F. Sánchez. Changing UML metamodel in order to represent separation of concerns. ECOOP'2000 Workshop (Sophia Antipolis, France, June 2000).

9. SAAM: A Method for Analyzing the Properties of Software Architectures. http://www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html

10. Implementing distribution and persistence aspects with aspectJ. http://portal.acm.org

11. AspectJ, http://www.aspectj.org

12. Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns. Siobhan Clarke. Department of computer Science, Trinity Collegue. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/clarke.pdf

13. S. Clarke. Composition of Object-Oriented Software Design Models. Doctor of Philosophy in Computer Applications thesis. January, 2001, Dublin City University.

14. MSN –AOP. http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx