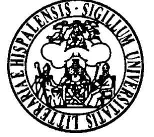
Departamento de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadística
Universidad de Sevilla

Avenida de la Reina Mercedes S/n. 41012 SEVILLA
Fax/Tlf: 954557139



Visión General de la Programación Orientada a Aspectos

Antonia Mª Reina Quintero

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla

Diciembre, 2000

Índice

1. Objetivos del documento ... 3
2. Introducción ... 4
3. Un poco de historia ... 9
4. ¿Qué es un aspecto? ... 10
5. Fundamentos de la programación orientada a aspectos ... 13
6. Tejiendo clases y aspectos ... 15
7. Estado del Arte en el Diseño de Lenguajes de Aspectos ... 18
   Lenguajes de Aspectos de Propósito General vs. Dominio Específico. ... 18
   Un lenguaje de dominio específico: COOL ... 19
   Un lenguaje de propósito general: AspectJ ... 21
   Un ejemplo: La gestión de una cola circular ... 24
   Una cola circular usando COOL ... 25
   Una cola circular usando AspectJ ... 29
   El problema de los lenguajes base ... 31
8. Los aspectos en el diseño ... 33
9. Disciplinas relacionadas con POA ... 38
   Reflexión y protocolos de metaobjetos ... 38
   Transformación de programas ... 38
   Programación subjetiva ... 38
10. Conclusiones ... 39
11. Listado de Figuras ... 40
12. Referencias ... 41

1. Objetivos del documento

La ingeniería del software, y en general, la informática es una disciplina que está en constante evolución. Cada día surgen nuevas técnicas y metodologías que intentan mejorar la calidad y la eficiencia de los productos software.

En los últimos tiempos ha surgido con fuerza una nueva forma de descomponer los sistemas: la orientación a aspectos. Este informe trata dar una visión general de la programación orientada a aspectos, desde sus comienzos, hasta llegar su estado actual. Se sentarán las bases sobre las que se apoya está tecnología, y se definirán los conceptos que maneja, sirviendo de presentación a los investigadores que quieran introducirse en este campo.

2. Introducción

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

En los primeros estadios de desarrollo de los lenguajes de programación se tenía un código en el que no había separación de conceptos, datos y funcionalidad se mezclaban sin una línea divisoria clara. A esta etapa se la conoce como la del *código spaghetti*, ya que se tenía una maraña entre datos y funcionalidad que recuerda a la que se forma cuando comemos un plato de esta pasta italiana.

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

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

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

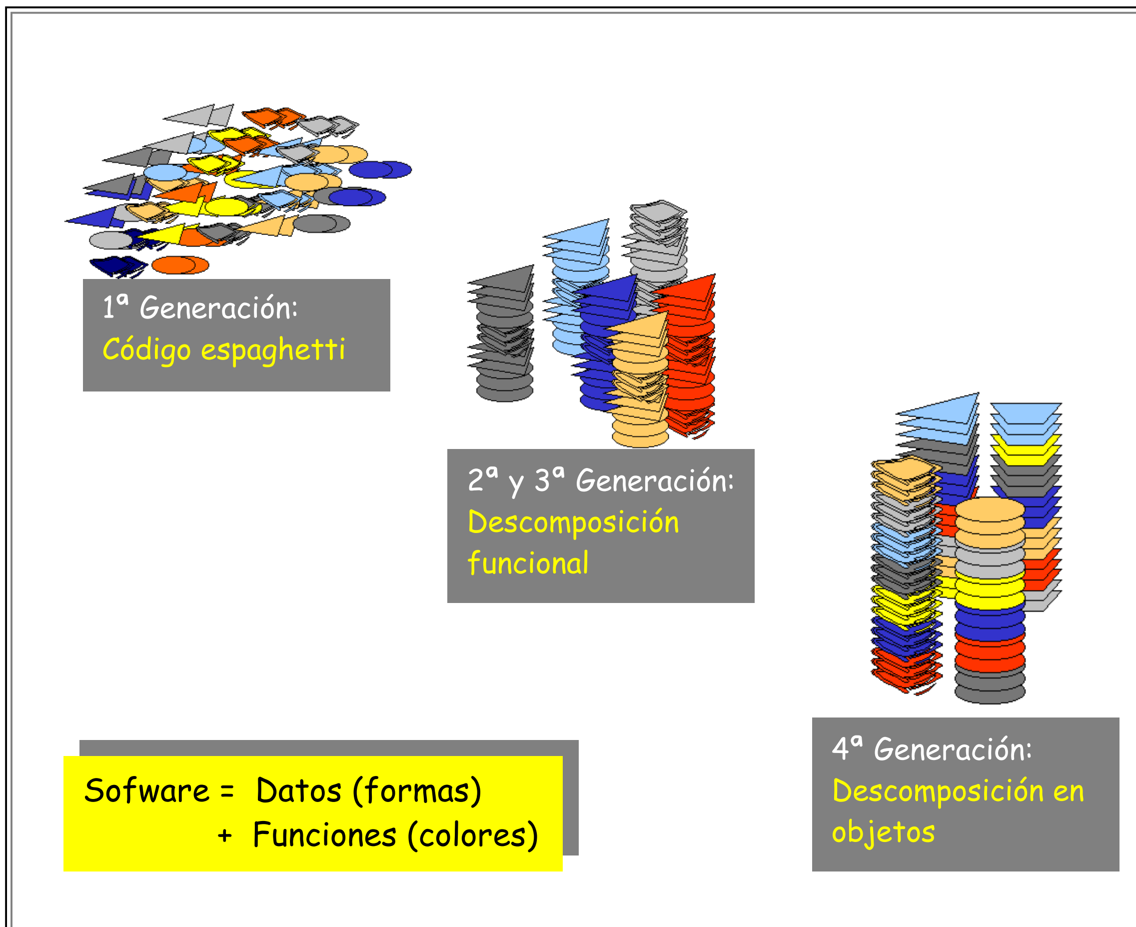


Figura 1 Esquema de la evolución de la ingeniería del software

En la segunda y tercera generación se tienen conjuntos de elementos agrupados por tonalidades, y en cada conjunto conviven distintas formas, lo que nos indica que la descomposición del sistema se hace en base a las funciones (tonalidades), y que los datos quedan esparcidos por todos los conjuntos.

En la cuarta generación, sin embargo, esta agrupación se realiza en base a las formas (datos), pero como se puede observar, en cada conjunto tenemos representadas distintas funcionalidades (tonalidades), con lo cual éstas también se nos dispersan por todo el sistema.

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

En el desarrollo de un sistema software además del diseño y la implementación de la funcionalidad básica, se recogen otros aspectos tales como la sincronización, la distribución, el manejo de errores, la optimización de la memoria, la gestión de

seguridad, etc. Mientras que las descomposiciones funcional y orientada a objetos no nos plantean ningún problema con respecto al diseño y la implementación de la funcionalidad básica, estas técnicas no se comportan bien con los otros aspectos. Es decir, que nos encontramos con problemas de programación en los cuales ni las técnicas funcionales, ni las orientadas a objeto son suficientes para capturar todas las decisiones de diseño que el programa debe implementar.

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

La *programación orientada a aspectos* (*POA*) es una nueva metodología de programación que aspira a soportar la separación de competencias para los aspectos antes mencionados. Es decir, que intenta separar los componentes y los aspectos unos de otros, proporcionando mecanismos que hagan posible abstraerlos y componerlos para formar todo el sistema. En definitiva, lo que se persigue es implementar una aplicación de forma eficiente y fácil de entender.

POA es un desarrollo que sigue al paradigma de la orientación a objetos, y como tal, soporta la descomposición orientada a objetos, además de la procedimental y la descomposición funcional. Pero, a pesar de esto, POA no se puede considerar como una extensión de la POO, ya que puede utilizarse con los diferentes estilos de programación ya mencionados.

El estado actual de la investigación en POA es análogo al que había hace veinte años en la programación orientada a objetos.

En la *Figura 2* se representa la forma en que la programación orientada a aspectos descompone los sistemas. Se sigue el razonamiento empleado en la *Figura 1*, donde los datos se identificaban con la forma de las figuras y las funcionalidades con el color o la tonalidad.

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

Por último, vista la evolución de la ingeniería del software, y comprobando que las formas de descomponer los sistemas han conducido a nuevas generaciones de sistemas, cabe plantearnos la siguiente pregunta ¿estaremos en la etapa incipiente de una nueva generación de sistemas software?

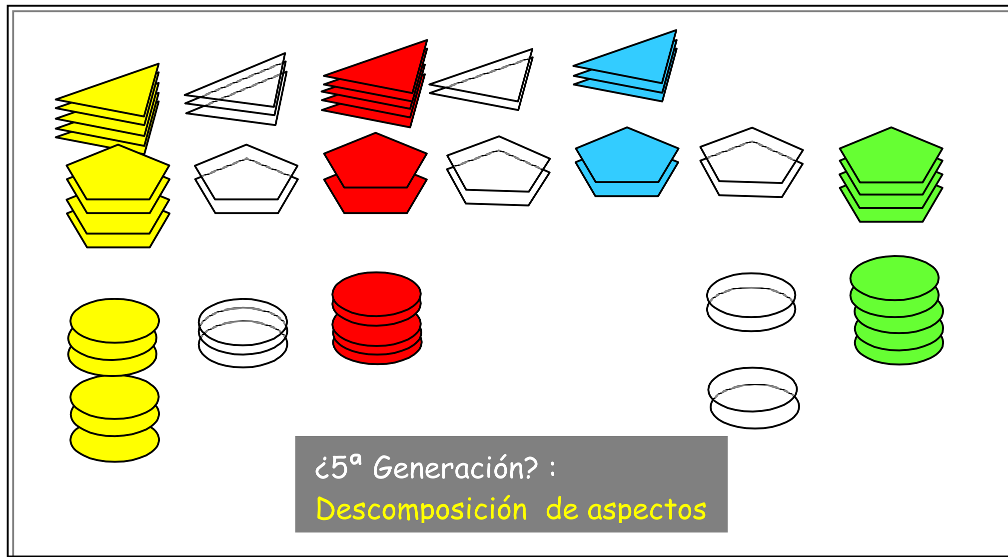


Figura 2. Descomposición en aspectos ¿Una quinta generación?

Este trabajo se distribuye de la siguiente manera: En el tercer apartado se explica brevemente cómo surge la idea de la orientación a aspectos, y aunque nos encontramos con una tecnología bastante joven, se ha creído conveniente dar una breve introducción a sus orígenes para resolver una de las preguntas básicas y más repetidas a lo largo de la humanidad. ¿De dónde venimos?. En el resto de los apartados se responderá a otra de las preguntas pilares: ¿Qué somos?.

Algunas veces, el responder a qué somos nos puede ayudar a conocer hacia dónde vamos, pero esta última pregunta ayudarán a responderla los investigadores que decidan estudiar disciplina, y algunos de ellos, ayudarán a elegir un camino para decidir hacia dónde vamos.

En el cuarto apartado se intenta dar una definición más o menos formal, más o menos intuitiva de el concepto básico que maneja la POA, el aspecto. El quinto intenta definir algunos conceptos nuevos que se introducen al aplicar esta nueva tecnología, así como sentar las bases para que cualquier investigador se pueda poner a trabajar.

Una parte importante de la programación orientada a aspectos es que las clases y los aspectos se han de mezclar. Existen dos enfoques principalmente para resolver el modo de realizar este entrelazado. Éstos se comentan en el apartado seis.

Si no existen herramientas, no se puede trabajar, y los lenguajes orientados a aspectos son una herramienta importante a la hora de trabajar con aspectos. Aquí también hay distintas perspectivas a la hora de diseñar estos lenguajes. Esta problemática se aborda en el apartado siete, donde se comenta someramente la estructuras de dos lenguajes de aspectos, uno de dominio específico (COOL), y otro de propósito general (AspectJ).

En este mismo apartado se incluye un ejemplo de implementación. Se escribe el código de una aplicación para gestionar una cola circular con un aspecto de

sincronización, utilizando los lenguajes tradicionales (Java), uno de aspectos de dominio específico (COOL), y uno de aspectos de propósito general (AspectJ). Así se podrán comparar los tres enfoques y ver las ventajas y los inconvenientes de cada uno.

Los aspectos no se han de quedar sólo en la implementación, sino que se han de llevar hasta la fase de diseño, y en el apartado ocho se comenta la propuesta de Yamamoto, que se basa en la extensión de UML para dar este paso.

En el apartado nueve, se comentan algunas de las disciplinas que han tenido una relación más estrecha con la orientación a aspectos. Y, por último, en el apartado diez se obtienen algunas conclusiones obtenidas durante la realización de este informe.

3. *Un poco de historia*

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

El trabajo del grupo Demeter [7] estaba centrado en la programación adaptativa, que no es más que una instancia temprana de la programación orientada a aspectos. La programación adaptativa se introdujo alrededor de 1991. Aquí los programas se dividían en varios bloques de cortes. Inicialmente, se separaban la representación de los objetos del sistema de cortes. Luego se añadieron comportamientos de estructuras y estructuras de clases como bloques constructores de cortes. Cristina Lopes [2] propuso la sincronización y la invocación remota como nuevos bloques.

No fue hasta 1995 cuando se publicó la primera definición temprana del concepto de aspecto [7], realizada también por el grupo Demeter, a la que se hace referencia en el apartado siguiente.

Gracias a la colaboración de Cristina Lopes y Karl J. Lieberherr con Gregor Kiczales y su grupo se introdujo el término de programación orientada a aspectos

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

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

- Un código menos enmarañado, más natural y más reducido.
- Una mayor facilidad para razonar sobre las materias, ya que están separadas y tienen una dependencia mínima.
- Más facilidad para depurar y hacer modificaciones en el código.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.

4. ¿Qué es un aspecto?

El nuevo paradigma de la programación orientada a aspectos es soportado por los llamados *lenguajes de aspectos*, que proporcionan constructores para capturar los elementos que se diseminan por todo el sistema. Estos elementos se llaman *aspectos.*

Una de las primeras definiciones que aparecieron del concepto de aspecto fue publicada en 1995 [7], y se describía de la siguiente manera: *Un aspecto es una unidad que se define en términos de información parcial de otras unidades*.

La definición de aspecto ha evolucionado a lo largo del tiempo, pero con la que se trabaja actualmente es la siguiente: *Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa* (G. Kiczales).

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

Se puede diferenciar entre un componente y un aspecto viendo al primero como aquella propiedad que se puede encapsular claramente en un procedimiento, mientras que un aspecto no se puede encapsular en un procedimiento con los lenguajes tradicionales [1].

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

En la *Figura 3*, se muestra un programa como un todo formado por un conjunto de aspectos más un modelo de objetos. Con el modelo de objetos se objetos se recoge la funcionalidad básica, mientras que el resto de aspectos recogen características de rendimiento y otras no relacionadas con la funcionalidad esencial del mismo.

Teniendo esto en cuenta, podemos realizar una comparativa de la apariencia que presenta un programa tradicional con la que muestra un programa orientado a aspectos. Esta equiparación se puede apreciar en la *Figura 4,* en la que se confrontan la estructura de un programa tradicional (parte izquierda) con la de uno orientado a aspectos (parte derecha).

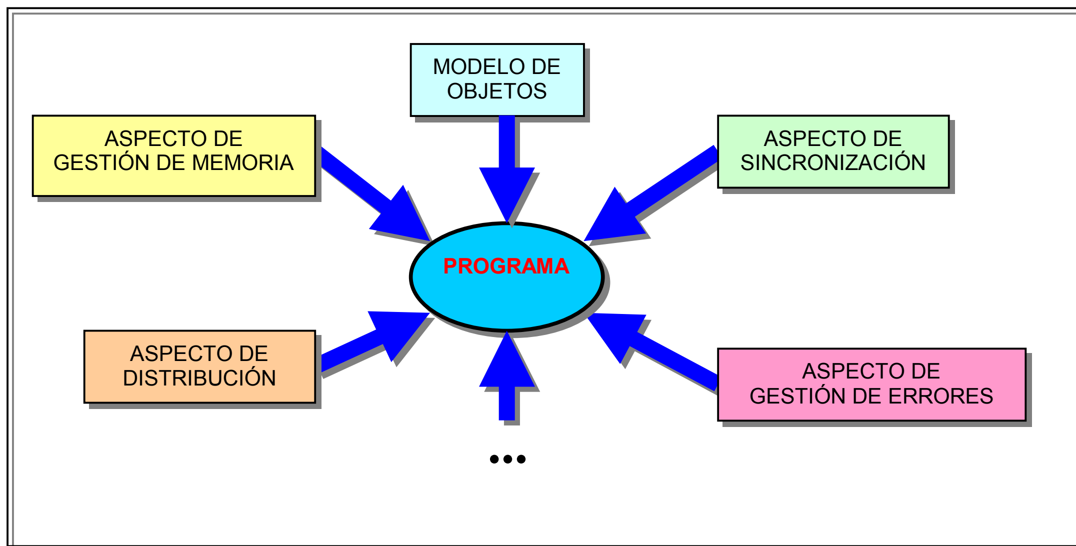


Figura 3. Estructura de un programa orientado a aspectos.

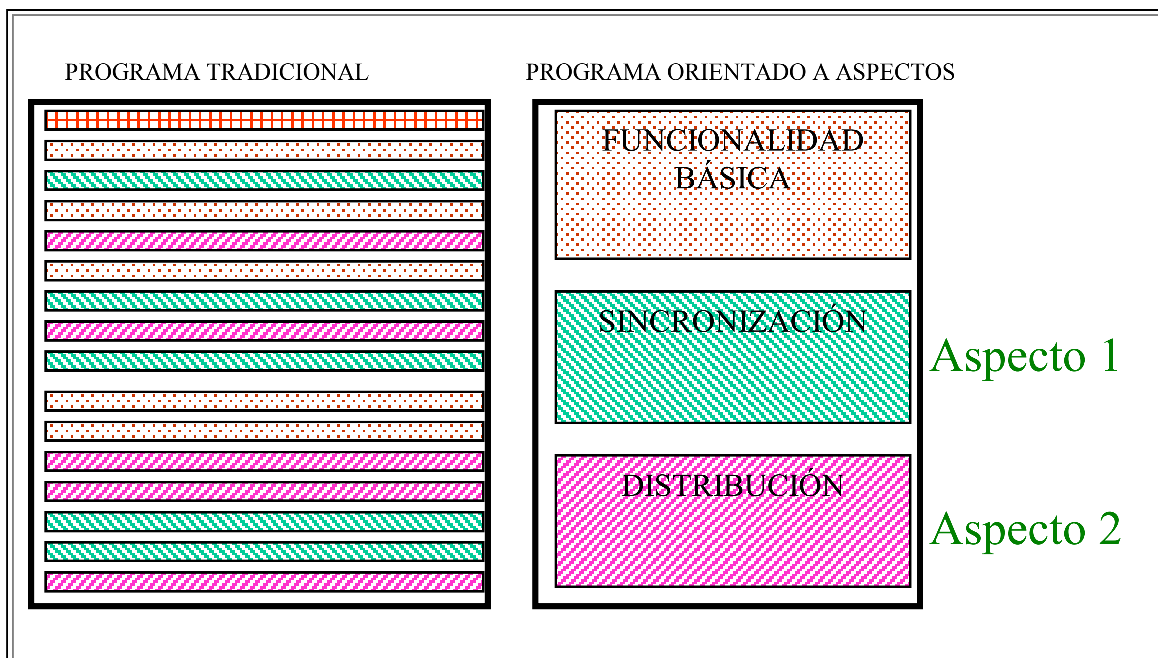


Figura 4 Comparativa de la forma de un programa tradicional con uno orientado a aspectos

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

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

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

5. Fundamentos de la programación orientada a aspectos

Con todo lo visto en los apartados anteriores, se puede decir que con las clases se implementa la funcionalidad principal de una aplicación (como por ejemplo, la gestión de un almacén), mientras que con los aspectos se capturan conceptos técnicos tales como la persistencia, la gestión de errores, la sincronización o la comunicación de procesos. Estos aspectos se escriben utilizando lenguajes de descripción de aspectos especiales [1].

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular las funcionalidades que cruzan todo el código. Además, estos lenguajes deben soportar la separación de aspectos como la sincronización, la distribución, el manejo de errores, la optimización de memoria, la gestión de seguridad, la persistencia. De todas formas, estos conceptos no son totalmente independientes, y está claro que hay una relación entre los componentes y los aspectos, y que por lo tanto, el código de los componentes y de estas nuevas unidades de programación tienen que interactuar de alguna manera. Para que ambos (aspectos y componentes) se puedan mezclar, deben tener algunos puntos comunes, que son los que se conocen como *puntos de enlace,* y debe haber algún modo de mezclarlos.

Los *puntos de enlace* son una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que éste se puede aumentar con comportamientos adicionales. Estos comportamientos se especifican en los aspectos.

El encargado de realizar este proceso de mezcla se conoce como *tejedor* (del término inglés *weaver)*. El tejedor se encarga de mezclar los diferentes mecanismos de abstracción y composición que aparecen en los lenguajes de aspectos y componentes ayudándose de los puntos de enlace.

Para tener un programa orientado a aspectos necesitamos definir los siguientes elementos:

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

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

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

En la *Figura 5* se aprecia la forma en la que se trabaja con las aplicaciones tradicionales, y cómo será esta forma de operar en una aplicación orientada a aspectos en la *Figura 6*.

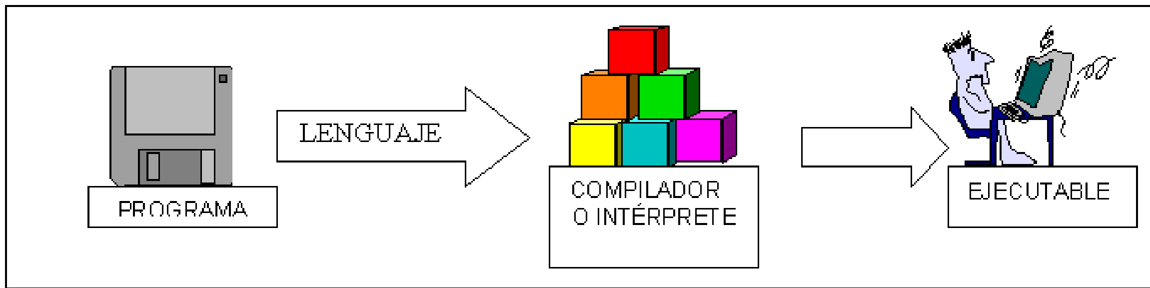


Figura 5. Estructura de una implementación en los lenguajes tradicionales.

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

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

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 tiene una opción para generar archivos .class, encargándose él de llamar al compilador Java.

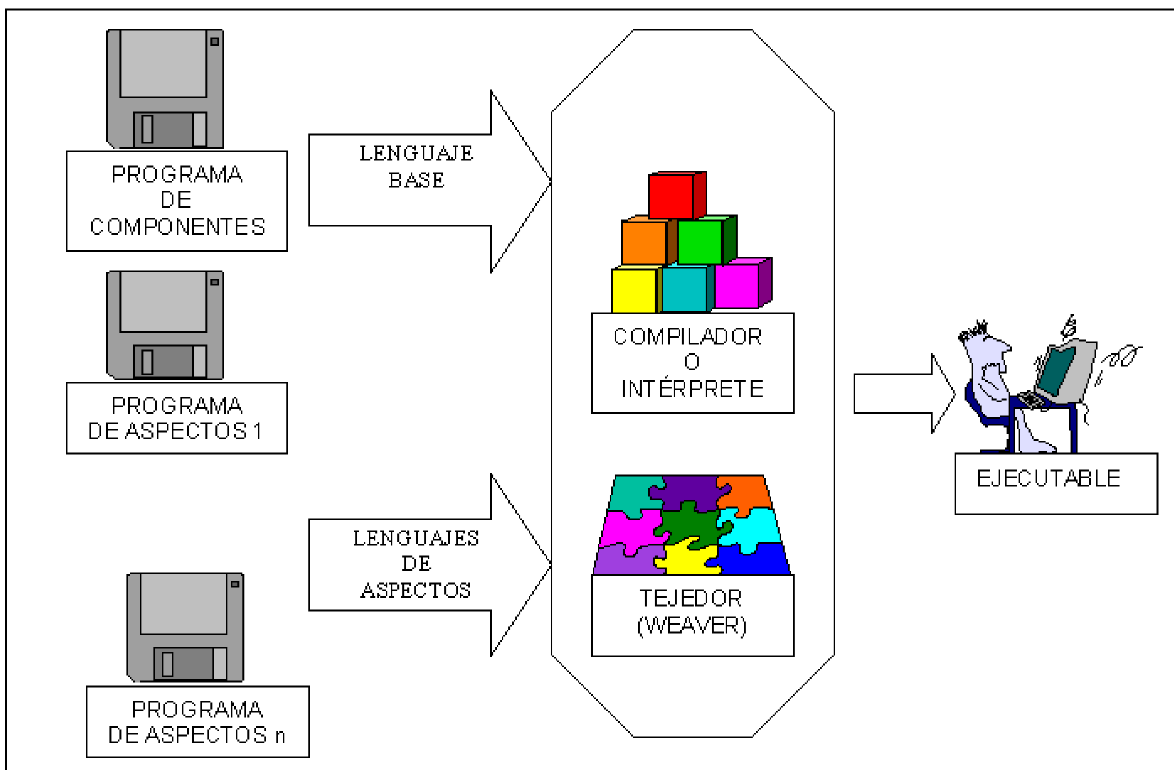


Figura 6. Estructura de una implementación en los lenguajes de aspectos.

6. Tejiendo clases y aspectos

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

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 de una clase insertando sentencias en estos puntos de enlace. Es decir, que el código del aspecto se introduce en el de la clase. Un ejemplo de este tipo de tejedor es el Tejedor de Aspectos de AspectJ [12].

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

- Entrelazado dinámico.

Una precodición o requisito para que se pueda realizar un entrelazado dinámico es que los aspectos existan de forma explícita tanto en tiempo de compilación como en tiempo de ejecución [3] .

Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Dado un interfaz de reflexión, el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica, si así se desea, durante la ejecución. Un ejemplo de este tipo de tejedores es el Tejedor AOP/ST [16], que no modifica el código fuente de las clases mientras se entrelazan los aspectos. En su lugar utiliza la herencia para añadir el código específico del aspecto a sus clases.

La forma en la que trabaja este tejedor se muestra en la *Figura 7*, que enseña la estructura de clases UML resultante de tejer dos aspectos, sincronización y traza, a la clase Rectángulo. La idea que se aplica [11] es que cada aspecto se representa por su propia subclase Rectángulo. Los métodos a los que les afectan los aspectos se sobrescriben en las subclases utilizando los *métodos tejidos.* Un método tejido envuelve la invocación de la implementación del método heredada . La clase vacía *clase tejida* finaliza la cadena de subclases.

Las subclases generadas por el tejedor de aspectos no tienen ningún efecto en la ejecución del programa. Si se le envía un mensaje a un objeto Rectángulo, se buscará el

correspondiente método de la clase del objeto y ejecutará la implementación que tenga. Para hacer que el código tejido tenga efecto, el tejedor cambia la clase de todos los objetos Rectángulo a la clase tejida. También instala un mecanismo para que esto ocurra con todos los objetos rectángulo que se añadan en el futuro.

Este tejedor también tiene en cuenta el orden en el que se entremezclan los aspectos. Esto lo resuelve asignando una prioridad al aspecto. El aspecto que tenga asignado un número menor es el que se teje primero, y por lo tanto, aparecerá antes en la jerarquía de herencia. Esta prioridad se ha representado en la estructura UML de las clases como un número entre paréntesis después de la clase estereotipada.

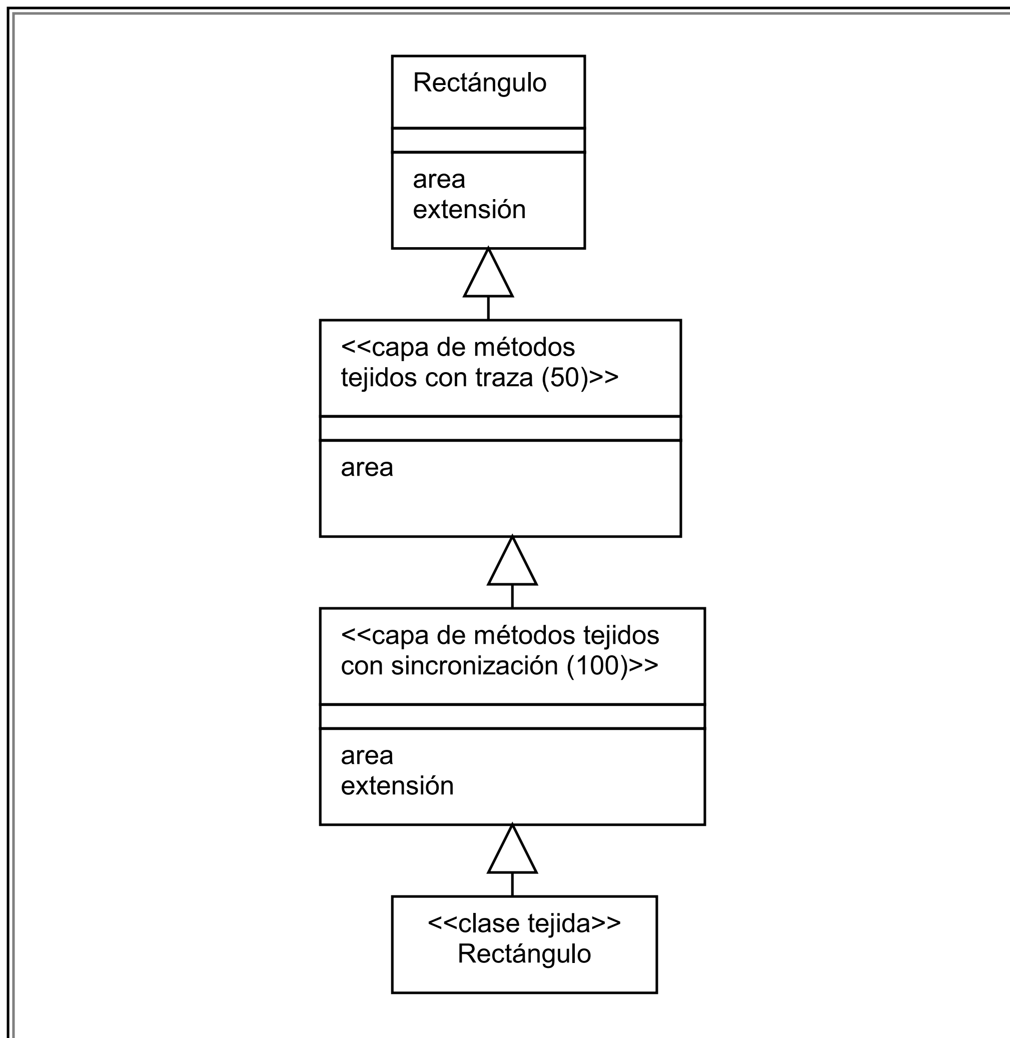


Figura 7. Estructura de clases resultado de tejer dos aspectos y una clase con el Tejedor AOP/ST.

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

Una de las primeras clasificaciones de las formas de combinar el comportamiento de los componentes y los aspectos fue dada por John Lamping [6]:

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

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

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

7. Estado del Arte en el Diseño de Lenguajes de Aspectos

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

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

Lenguajes de Aspectos de Propósito General vs. Dominio Específico.

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

Estos lenguajes normalmente imponen restricciones en la utilización del lenguaje base. Esto se hace para garantizar que los conceptos del dominio del aspecto se programen utilizando el lenguaje diseñado para este fin y evitar así interferencias entre ambos. Se quiere evitar que los aspectos se programen en ambos lenguajes lo cual podría conducir a un conflicto. Como ejemplos de lenguajes de dominio específico están COOL [2], que trata el aspecto de sincronización, y RIDL [2], para el aspecto de distribución.

Los *lenguajes de aspectos de propósito general* se diseñaron para ser utilizados con cualquier clase de aspecto, no solamente con aspectos específicos. Por lo tanto, no pueden imponer restricciones en el lenguaje base. Principalmente soportan la definición separada de los aspectos proporcionando unidades de aspectos. Normalmente tienen el mismo nivel de abstracción que el lenguaje base y también el mismo conjunto de instrucciones, ya que debería ser posible expresar cualquier código en las unidades de aspectos. Un ejemplo de este tipo de lenguajes es AspectJ, que utiliza Java como base, y las instrucciones de los aspectos también se escriben en Java.

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

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

Un lenguaje de dominio específico: COOL

COOL es un lenguaje de dominio específico creado por Xerox [2] cuya finalidad es la sincronización de hilos concurrentes. El lenguaje base que utiliza es una versión restringida de Java, ya que se han de eliminar los métodos `wait`, `notify` y `notifyAll`, y la palabra clave `synchronized` para evitar que se produzcan situaciones de duplicidad al intentar sincronizar los hilos en el aspecto y en la clase.

En COOL, la sincronización de los hilos se especifica de forma declarativa y, por lo tanto, más abstracta que la correspondiente codificación en Java.

COOL proporciona mecanismos para trabajar con la exclusión mutua de hilos de ejecución, el estado de sincronización, la suspensión con guardas, y la notificación de forma separada de las clases.

Un programa COOL está formado por un conjunto de módulos coordinadores. En cada coordinador se define una estrategia de sincronización, en la cual pueden intervenir varias clases. Aunque estén asociados con las clases, los coordinadores no son clases.

La unidad mínima de sincronización que se define en COOL es un método.

Los coordinadores no se pueden instanciar directamente, sino que se asocian con las instancias de las clases a las que coordinan en tiempo de instanciación. Esta relación está vigente durante toda la vida de los objetos y tiene un protocolo perfectamente definido [2].

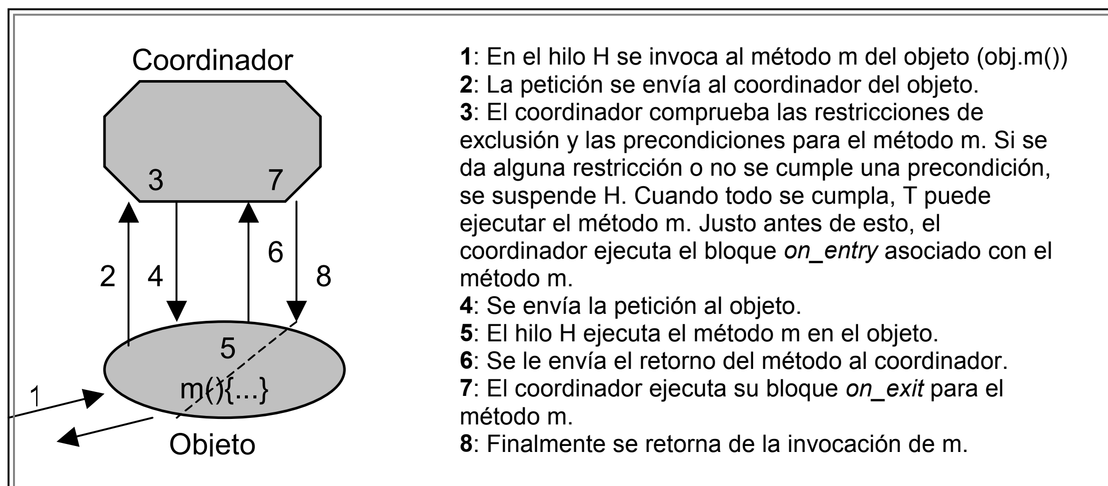


Figura 8 Protocolo entre un objeto y su coordinador en COOL.

Los coordinadores se escriben sabiendo perfectamente las clases a las que coordinan, sin embargo, las clases no tienen conocimiento de los coordinadores.

Para tener una idea general de cómo trabaja el coordinador COOL, qué forma tiene un coordinador. El cuerpo de un coordinador puede esta formado por :

- Variables de condición.

  Las variables de condición se utilizan para controlar el estado de sincronización, con el propósito de utilizarlas en la suspensión con guarda y en la notificación de los hilos de ejecución.

  Se declaran utilizando la palabra clave `condition`.
- Variables ordinarias.

  Las variables ordinarias mantienen la parte del estado del coordinador que no conduce directamente a la suspensión con guarda y a la notificación de hilos, pero que pueden afectar al estado de sincronización.

  Se declaran igual que en Java.

- Un conjunto de métodos autoexcluyentes.

  En esta sección se identifican los métodos que solamente pueden ser ejecutados por un hilo a la vez.

  Se identifican con la palabra clave `selfex`.

- Varios conjuntos de métodos de exclusión mutua.

  Un conjunto de exclusión mutua identifica una serie de métodos que no se pueden ejecutar concurrentemente por distintos hilos. Es decir, que la ejecución de un hilo H de uno de los métodos del conjunto evita que otro hilo H' ejecute cualquier otro método del mismo conjunto.

  El conjunto de exclusión mutua se declara con `mutex`.

- Gestores de métodos.

  Estos gestores se encargan de la suspensión con guarda y de la notificación de los hilos utilizando al estilo de las precondiciones utilizando la cláusula `requires` (a modo de precondición) y las sentencias `on_entry` y `on_exit`.

  La semántica de la suspensión con guarda es la siguiente: Cuando un hilo H quiere ejecutar un método M que tiene una precondición, definida en una cláusula `requires`, y se cumplen las restricciones de exclusión puede suceder lo siguiente:

1. Si la condición definida en `requires` se cumple, entonces el método M puede ser ejecutado por H.

2. Si no el hilo H no puede ejecutar M, y se suspende. El hilo permanecerá suspendido hasta que se cumpla la precondición. Cuando esto ocurra, se le notificará a H, y si la restricción de exclusión aún se mantiene, H podrá ejecutar M, pero si no se mantiene, se volverá a suspender H.

Las sentencias `on_entry` y `on_exit` se ejecutan cuando un hilo tiene permiso para ejecutar un método. Justo antes de ejecutarlo, se ejecutan las sentencias incluidas en el bloque `on_entry`, y justo después las del bloque `on_exit`.

Un lenguaje de propósito general: AspectJ

AspectJ es una extensión a Java™ orientada a aspectos y de propósito general [14]. AspectJ extiende Java con una nueva clase de módulos llamado aspecto. Los aspectos cortan las clases, las interfaces y a otros aspectos. Los aspectos mejoran la separación de competencias haciendo posible localizar de forma limpia los conceptos de diseño del corte.

En AspectJ, un aspecto es una clase, exactamente igual que las clases Java, pero con una particularidad, que pueden contener unos constructores de corte, que no existen en Java. Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y excepciones de señales y gestión. Los cortes no definen acciones, sino que describen eventos.

En defnitiva, en AspectJ los aspectos son constructores que trabajan al cortar de forma transversal la modularidad de las clases de forma limpia y cuidadosamente diseñada [18]. Por lo tanto, un aspecto puede afectar a la implementación de un número de métodos en un número de clases, lo que permite capturar la estructura de corte modular de este tipo de conceptos de forma limpia.

En general, un aspecto en AspectJ está formado por una serie de elementos:

- Cortes.

  Los cortes (`pointcut`) capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y señalización y gestión de excepciones. Los cortes no definen acciones, simplemente describen eventos.

  Un corte está formado por una parte izquierda y una parte derecha, separadas ambas por dos puntos. En la parte izquierda se define el nombre del corte y el contexto del corte. La parte derecha define los eventos del corte.

Los cortes se utilizan para definir el código de los aspectos utilizando avisos.

A los descriptores de eventos de la parte derecha de la definición del corte se les llama designadores. Un designador puede ser:

- Un método.
- Un constructor.
- Un manejador de excepciones.

Se pueden componer utilizando los operadores o ("|"), y ("&") y no ("!"). También se pueden utilizar caracteres comodines en la descripción de los eventos.

- Introducciones.

Las introducciones (`introduction`) se utilizan para introducir elementos completamente nuevos en las clases dadas. Entre estos elementos podemos añadir:

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

- Avisos.

Las declaraciones de avisos (`advice`) definen partes de la implementación del aspecto que se ejecutan en puntos bien definidos. Estos puntos pueden venir dados bien por cortes con nombre, bien por cortes anónimos. En las dos siguientes figuras se puede ver el mismo aviso, primero definido mediante un corte con nombre, y después mediante uno anónimo.

```
pointcut emisores (Point p1, int newval): p1 & (void setX (newval)|void setY(newval);
advice (Point p1, int newval): emisores (p1, newval){
   before {System.out.println ("P1:" + p1);
}
```

Figura 9. Aviso definido mediante un corte con nombre.

```
advice (Point p1, int newval): p1 & (void setX (newval)  | void setY(newval) {
   before {System.out.println ("P1:" + p1);
}
```

Figura 10. Aviso definido mediante un corte anónimo.

El cuerpo de un aviso puede añadir en distintos puntos del código, cada uno de los cuales se define mediante una palabra clave:

- Aviso `before`.- Se ejecuta justo antes de que lo hagan las acciones asociadas con los eventos del corte.

- Aviso `after`.- Se ejecuta justo después de que lo hayan hecho las acciones asociadas con los eventos del corte.

- Aviso `catch`.- Se ejecuta cuando durante la ejecución de las acciones asociadas con los eventos definidos en el corte se ha elevado una excepción del tipo definido en la propia cláusula `catch`.

- Aviso `finally`.- Se ejecuta justo después de la ejecución de las acciones asociadas con los eventos del corte, incluso aunque se haya producido una excepción durante la misma.

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

Los avisos se pueden declarar con el modificador static, indicado esto que se ejecuta el mismo aviso para todas las instancias de las clases designadas. Si se omite indica que solamente se ejecuta para ciertas instancias.

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

Esta composición se puede realizar de dos formas:

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

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

Un ejemplo: La gestión de una cola circular

Una vez que se han dado pinceladas de los dos enfoques diferentes existentes a la hora de diseñar los lenguajes de aspectos, se verá con un ejemplo práctico de cómo se reflejan estas características en el código de las aplicaciones.

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

En la *Figura 11*, se muestra el código Java de una lista circular, en principio, sin restricciones de sincronización, adaptado de [8]. La clase `ColaCircular` se representa con un array de elementos. Los elementos se van añadiendo en la posición `ptrCola` (por atrás), y se van borrando en la posición apuntada por `ptrCabeza` (la parte de delante). Estos dos índices se ponen a cero al alcanzar la capacidad máxima del array. Esta inicialización se consigue utilizando el resto obtenido al dividir la posición a tratar entre el número total de elementos del array. Esta clase solamente contiene un constructor y dos métodos: `Insertar` para introducir elementos en la cola, y `Extraer`, para sacarlos.

El problema del código de la *Figura 11* es que si varios hilos solicitan ejecutar métodos del objeto `ColaCircular`, éstos no están sincronizados, pudiéndose dar el caso de que la cola haya cubierto su capacidad, y se sigan haciendo peticiones de inserción de elementos, y al contrario, es decir, que la cola esté vacía y se hagan peticiones de extracción de elementos. En estos casos, el hilo que hace la solicitud debería esperar a que se extraiga algún elemento, en el primer caso, o a que se inserte alguno en el segundo. Es decir, que se necesita código adicional para eliminar estas inconsistencias.

Java permite coordinar las acciones de hilos de ejecución utilizando *métodos e instrucciones sincronizadas*.

A los objetos a los que se debe coordinar el acceso se le incluyen métodos sincronizados. Estos métodos se declaran con la palabra reservada `synchronized`. Un objeto solamente puede invocar a un método sincronizado al mismo tiempo, lo que impide que los métodos sincronizados en hilos de ejecución entren en conflicto.

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

espera que el tiempo de espera se ha acabado. En Java, por defecto, no se define ninguna estrategia de sincronización.

Una cola circular usando COOL

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

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

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

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];

          array[ptrCabeza] = null;
          ptrCabeza = (ptrCabeza + 1) % array.length;
          eltosRellenos--;

      return obj;
     }
 }
```

Figura 11 Código de la clase cola circular sin restricciones de sincronización.

La versión sincronizada del ejemplo de la *Figura 11* se muestra en la *Figura 12* . Aquí se ha introducido la exclusión mutua y condiciones con guardas. El código que se introduce para la sincronización es el sombreado.

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

    public ColaCircular (int capacidad)
     {
          array = new Object [capacidad];
     }
    public synchronized void Insertar (Object o) {

      while (eltosRellenos == array.length){
           try {
              wait ();
           }catch (InterruptedException e) {}
      }

      array[ptrCola] = o;
      ptrCola = (ptrCola + 1) % array.length;
      eltosRellenos++;

      notifyAll();

     }

    public synchronized Object Extraer () {

     while (eltosRellenos == 0){
          try {
             wait ();
          }catch (InterruptedException e) {}
     }
     Object obj = array[ptrCabeza];

     array[ptrCabeza] = null;
     ptrCabeza = (ptrCabeza + 1) % array.length;
     eltosRellenos--;

     notifyAll();


     return  obj;
     }
 }
```

Figura 12. Objeto cola circular con restricciones de sincronización.

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

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

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

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

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];

          array[ptrCabeza] = null;
          ptrCabeza = (ptrCabeza + 1) % array.length;
          eltosRellenos--;

      return obj;
     }
 }
```

Figura 13. Clase ColaCircular definida en el lenguaje base de COOL.

Si el mismo ejemplo de la cola circular con sincronización se implementa en COOL (*Figura 13* y *Figura 14*) se tiene que lo único que hay que añadirle es el módulo coordinador. La clase `ColaCircular` se escribe en el lenguaje base (Java), y lo único que habría que hacer para poner en funcionamiento la sincronización sería entrelazar los dos módulos juntos (el coordinador y la clase).

```
coordinator ColaCircular
 {
     selfex Insertar, Extraer;
     mutex  {Insertar, Extraer};
     cond lleno = false, vacio = true;
     Insertar : requires !lleno;
        on_exit {
          empty = false;
          if (eltosRellenos == array.length)
               lleno = true;
          }
     Extraer: requires !vacio;
          on_exit {
               lleno = false;
          if (eltosRellenos == 0) vacio = true;
 }
```

Figura 14. Implementación de la cola circular utilizando un coordinador COOL.

Si se compara el código sombreado de la *Figura 12* con el coordinador que se ha añadido en COOL (también sombreado en la *Figura 14*), se puede ver claramente, que en el caso Java, el tratamiento de la sincronización queda distribuido por todo el código de la clase, mientras que con COOL, el aspecto de sincronización está perfectamente localizado. También se puede apreciar que aunque ambos ejemplos persiguen el mismo objetivo, en el de la *Figura 14* queda definida de forma mucho más clara la estrategia de sincronización que se sigue.

En el coordinador se indica (`selfex`) que los métodos `Insertar` y `Extraer` son autoexcluyentes, es decir, que como mucho los estará ejecutando un hilo. Es decir, que si hay un hilo que esté ejecutando el método Insertar y ser recibe una petición para ejecutar este mismo método, el hilo que haga esta petición tendrá que esperar a que el primero acabe con la suya. Lo mismo ocurriría para el método Extraer. Hay que tener en cuenta también que esta opción no exime de que un hilo esté realizando una inserción y otro una extracción. Por ello se ha de añadir una sección mutua.

Los métodos que acompañan a la sentencia `mutex` no se podrán ejecutar concurrentemente por distintos hilos. Al ejecutar un hilo uno de los métodos del conjunto, hace que los demás hilos no puedan ejecutar ningún otro métodos del mismo conjunto, con lo cual, mientras se está insertando un elemento en la cola, no se puede extraer otro, y viceversa. Es decir, se está definiendo la sección mutua que nos hacía falta.

En la última parte del coordinador se representa una condición con un guarda. La sentencia `requires` implica que el método `Insertar` sólo se podrá ejecutar

cuando la cola circular no esté llena, y que el método `Extraer` sólo se ejecutará cuando la cola esté vacía. Las variables que intervienen en el estado del coordinador se definen con `cond`.

Una cola circular usando AspectJ

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

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

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

 protected synchronized void antesInsertar
                                  (ColaCircular c){
   while (eltosRellenos == c.getCapacidad()) {
     try { wait(); } catch (InterruptedException ex) {};
   }
}

after(ColaCircular c):insertar(c) { despuesInsertar();}
protected synchronized void despuesInsertar (){
     eltosRellenos++;
     notifyAll();
}

before(ColaCircular c):extraer(c) {antesExtraer();}
protected synchronized void antesExtraer (){
   while (eltosRellenos == 0) {
     try { wait(); } catch (InterruptedException ex) {};
   }

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

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

Figura 15 Aspecto para definir estrategia de sincronización mediante AspectJ.

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


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

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

     }

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];

          array[ptrCabeza] = null;
          ptrCabeza = (ptrCabeza + 1) % array.length;


      return obj;
     }

   public int getCapacidad(){
      return capacidad;
     }

 }
```

Figura 16. Clase ColaCircular para un aspecto en AspectJ

Para escribir estos ejemplos se ha utilizado la versión de 0.7 de AspectJ. Debido a que este compilador ha pasado por varias versiones y la compatibilidad hacia atrás de alguna de las mismas no es demasiado buena, hay que dejar este punto claro.

En la *Figura 15* se presenta un aspecto para tratar la sincronización de la clase `ColaCircular`. En la *Figura 16* se representa la clase a la que está asociada este aspecto. Nótese que esta versión difiere un poco de la de COOL, ya que el atributo `eltosRellenos` se ha pasado al aspecto, ya que es una información que solamente interesa para sincronizar. La `ColaCircular` solamente ha de conocer por dónde se inserta y por dónde se extrae.

La idea que se ha seguido es bastante simple. Se crean dos cortes, uno para capturar el caso de que se produzca una llamada al método `Insertar`, y otro para el método `Extraer`.

Con cada corte se definen dos avisos, uno de tipo `before` y otro de tipo `after`. En el de tipo `before` se comprueba si se dan las condiciones necesarias para que se ejecute el método. En el caso de la inserción, la comprobación de que el buffer no esté lleno, y en el caso de la extracción que no esté vacío. Si alguna de estas condiciones no se cumple, el hilo que ejecuta el objeto se pone en espera hasta que se cumpla la condición.

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

Al dispararse el evento asociado a los cortes se ejecuta el código definido en el bloque asociado al aviso.

Como se puede comprobar con este ejemplo, la estrategia de sincronización seguida queda mucho más clara y más intuitiva en el caso de COOL que con AspectJ, ya que con el primero se indica de forma declarativa. Sin embargo, con COOL no podemos definir otro aspecto que no sea la sincronización. Esto quiere decir que con AspectJ tenemos una mayor flexibilidad.

De todos modos se puede crear una librería para conseguir la misma funcionalidad que los coordinadores COOL, y tener la estrategia de sincronización más clara en el código AspectJ. De hecho, esta librería viene incorporada junto con los ejemplos que se proporcionan en el compilador `ajc` de AspectJ.

También se podrían crear librerías de aspectos, que recojieran las distintas funcionalidades de cada uno de los distintos aspectos del sistema, del mismo modo que la librería propuesta por los creadores de AspectJ para la sincronización.

El problema de los lenguajes base

Para el diseño de los lenguajes de aspectos hay dos alternativas relativas al lenguaje base. Una sería el diseñar un nuevo lenguaje base junto con el lenguaje de aspectos y la otra sería tomar un lenguaje ya existente como base, lo cual es posible, ya que la base ha de ser un lenguaje de propósito general. Esta última opción tiene la ventaja de que se tiene que trabajar menos en el diseño e implementación de lenguajes para los entornos orientados a aspectos, y se pueden utilizar lenguajes ya trillados, y segundo, que el programador solamente tendrá que aprender el lenguaje de aspectos, pero no el lenguaje para la funcionalidad básica, que todavía constituye la mayor parte de los programas orientados a aspectos.

Hasta ahora, ambos, los lenguajes de dominio específicos y los de propósito general se han diseñado para utilizarse con lenguajes base existentes. COOL y AspectJ utilizan Java como base. Sin embargo, aparte de las ventajas ya mencionadas, esta decisión también conlleva algunos problemas.

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

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

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

8. Los aspectos en el diseño

En las primeras fases, la orientación a aspectos se centró principalmente en el nivel de implementación y codificación, pero en los últimos tiempos cada vez se elevan más voces y surgen más trabajos para llevar la separación de funcionalidades a nivel de diseño.

Esta separación se hace necesaria, sobre todo, cuando las aplicaciones necesitan un alto grado adaptabilidad o que se puedan reutilizar. Los trabajos surgidos hasta el momento proponen utilizar UML como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar el diseño funcional de los objetos separado del diseño no funcional del mismo, o lo que es lo mismo, representar la funcionalidad básica separada de los otros aspectos.

Las ventajas que tiene el capturar los aspectos ya desde la fase de diseño son claras [19]:

- Facilita la creación de documentación y el aprendizaje.

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

- Facilita la reutilización de los aspectos.

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

La propuesta que se realiza en [19] de extensión del metamodelo de UML para tener en cuenta a los aspectos en la fase de diseño consiste en los siguiente:

Añadir nuevos elementos al metamodelo para el aspecto y la clase "tejida", y se reutilizar un elemento ya existente para la relación clase-aspecto.

- Aspecto.

  El aspecto se añade como un nuevo constructor derivado del elemento `Clasificador` (*Figura 17*) que describe características estructurales y de comportamiento [20] .

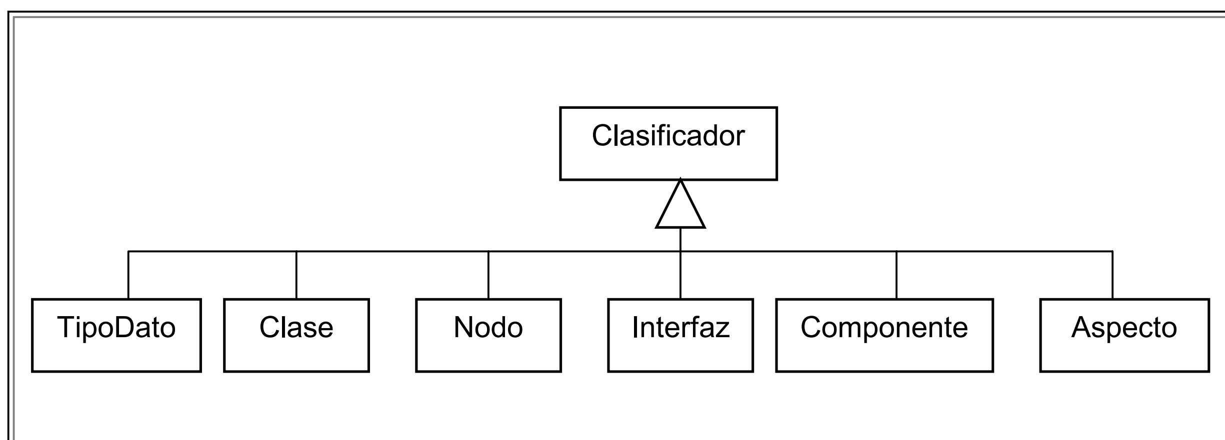


Figura 17. El Aspecto como un elemento del metamodelo UML derivado de Clasificador

Un aspecto así modelado puede tener atributos, operaciones y relaciones. Los atributos son utilizados por su conjunto de definiciones tejidas. Las operaciones se consideran como sus declaraciones tejidas. Las relaciones incluyen la generalización, la asociación y la dependencia. En el caso de que el tejedor de aspectos utilice el tipo de tejidos, es decir, introduzca un aviso tejido en AspectJ, éste se especifica como una restricción para la correspondiente declaración tejida.

El aspecto se representa como un rectángulo de clase con el estereotipo `<<aspecto>>` (*Figura 18*) . En el apartado de la lista de operaciones se muestran las declaraciones tejidas. Cada tejido se muestra como una operación con el estereotipo `<<tejido>>`. Una signatura de la declaración tejida representa un designador cuyos elementos están afectados por el aspecto.

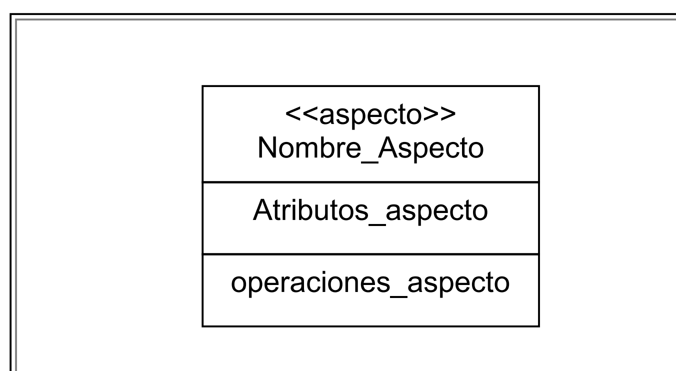


Figura 18. Representación de un aspecto en UML extendido

- Relación aspecto-clase.

En el metamodelo de UML se definen tres relaciones básicas, que se derivan del elemento `Relación` del metamodelo: `Asociación`, `Generalización` y `Dependencia`.[20] .

La relación entre un aspecto y su clase es un tipo de relación de dependencia. La relación de dependencia modela aquellos casos en que la implementación o el funcionamiento de un elemento necesita la presencia de otro u otros elementos.

En el metamodelo UML del elemento Dependencia se derivan otros cuatro que son: `Abstracción`, `Ligadura`, `Permiso` y `Uso` (*Figura 19*). La relación aspecto-clase se clasifica como una relación de dependencia del tipo abstracción.

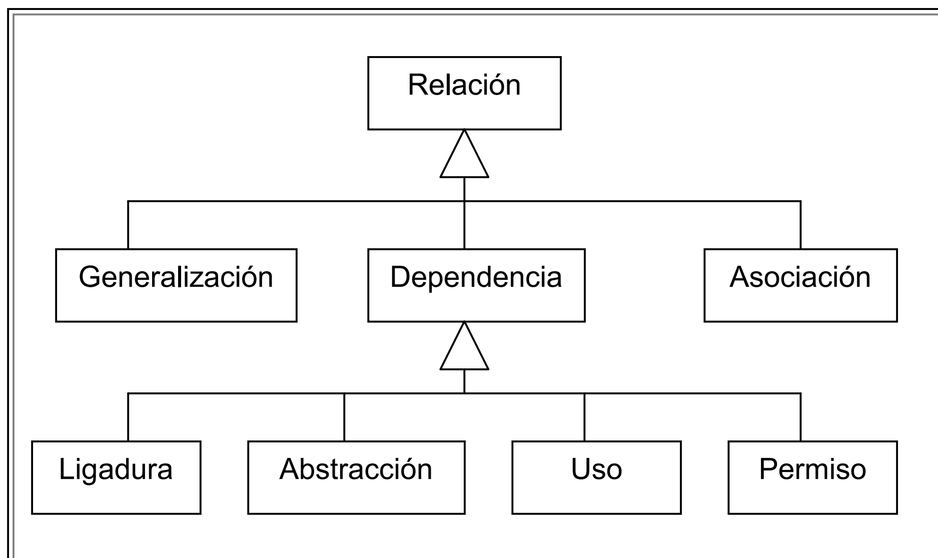


Figura 19. Tipos de relaciones del metamodelo de UML

Una relación de dependencia de abstracción relaciona dos elementos que se refieren al mismo concepto pero desde diferentes puntos de vista, o aplicando diferentes niveles de abstracción. El metamodelo UML define también cuatro estereotipos [20] para la dependencia de abstracción: derivación, realización, refinamiento y traza.

Con el estereotipo `<<realiza>>` se especifica una relación de realización entre un elemento o elementos del modelo de especificación y un elemento o elementos del modelo que lo implementa. El elemento del modelo de implementación ha de soportar la declaración del elemento del modelo de especificación.

La relación aspecto-clase se recoge en la dependencia de abstracción con el estereotipo realización (`<<realiza>>`). Esta relación se representa en la *Figura 20*.

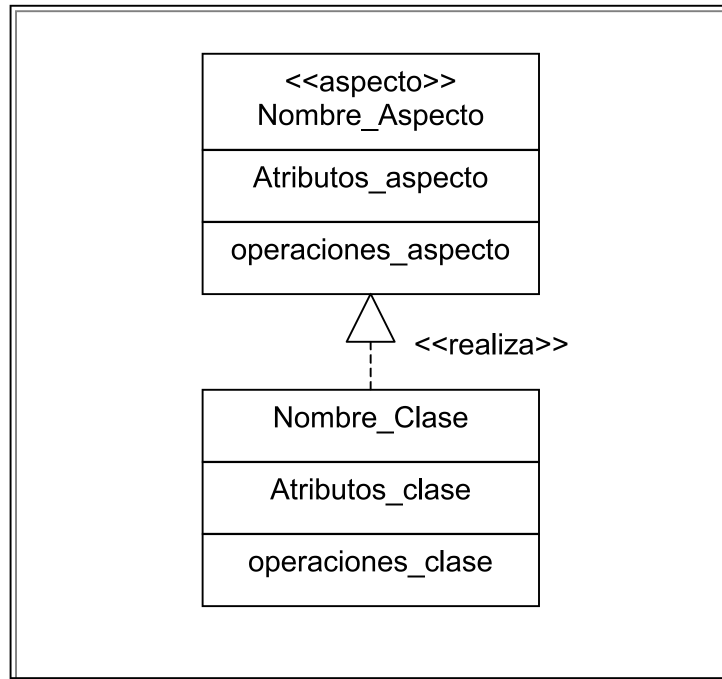


Figura 20. Notación de la relación aspecto-clase

- Clase tejida.

Al utilizar un tejedor de aspectos, el código de las clases y los aspectos se mezclan y se genera como resultado una clase tejida. La estructura de la clase tejida depende del tejedor de aspectos y del lenguaje de programación que se haya utilizado. Por ejemplo, como se ha visto en el apartado *Tejiendo clases y aspectos,* AspectJ reemplaza la clase original por la clase tejida, mientras que AOP/ST genera una clase tejida que se deriva de la clase original.

La solución que proponen Suzuki y Yamamoto para modelar esto es introducir el estereotipo `<<clase tejida>>` en el elemento Clase para representar una clase tejida. Se recomienda que en la clase tejida se especifique con una etiqueta la clase y el aspecto que se utilizaron para generarla.

En la *Figura 21* se representa la estructura de las clases tejidas, utilizando tanto el tejedor AspectJ como el tejedor AOP/ST.

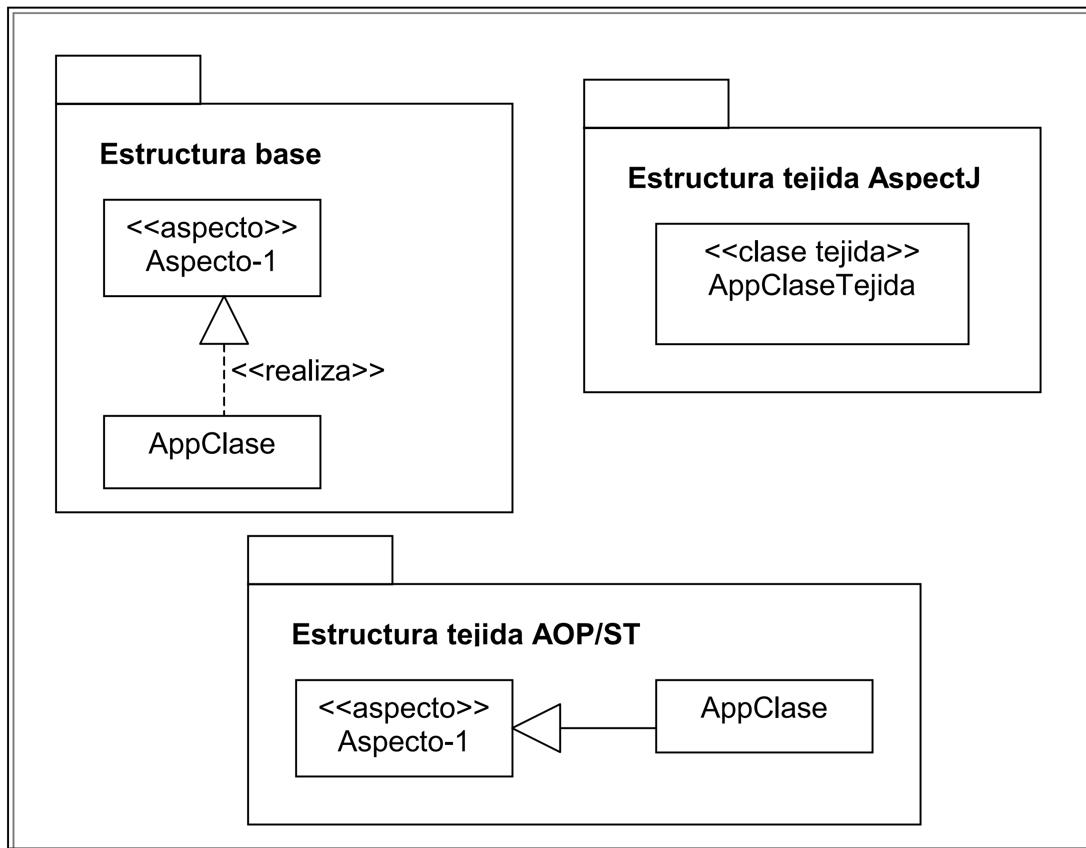


Figura 21. Estructura de las clases tejidas.

9. Disciplinas relacionadas con POA.

Por último, hay muchas disciplinas y campos de investigación relacionados de alguna manera con la orientación a aspectos, algunos persiguen los mismos objetivos y algunos otros. En este apartado se hará mención a algunos de ellos, de manera que se tenga una visión más amplia de por dónde se mueve la investigación en esta disciplina.

Reflexión y protocolos de metaobjetos

Un sistema reflectivo proporciona un lenguaje base y uno o más metalenguajes que proporcionan control sobre la semántica del lenguaje base y la implementación. Los metalenguajes proporcionan vistas de la computación que ningún componente del lenguaje base puede percibir.

Los metalenguajes son de más bajo nivel que los lenguajes de aspectos en los que los puntos de enlace son equivalentes a los "ganchos" de la programación reflectiva. Estos sistemas se pueden utilizar como herramienta para la programación orientada a aspectos.

Transformación de programas

El objetivo que persigue la transformación de programas es similar al de la orientación a aspectos. Busca el poder escribir programas correctos en un lenguaje de más alto nivel, y luego, transformarlos de forma mecánica en otros que tengan un comportamiento idéntico, pero cuya eficiencia sea mucho mejor. Con este estilo de programación algunas propiedades de las que el programador quiere implementar se escriben en un programa inicial. Otras se añaden gracias a que el programa inicial pasa a través de varios programas de transformación. Esta separación es similar a la que se produce entre los componentes y los aspectos.

Programación subjetiva

La programación subjetiva y la orientada a aspectos son muy parecidas, pero no son exactamente lo mismo. La programación subjetiva soporta la combinación automática de métodos para un mensaje dado a partir de diferentes sujetos. Estos métodos serían equivalentes a los componentes tal y como se perciben desde la POA.

Hay una diferencia clave entre ambas disciplinas, y es que mientras que los aspectos de la POA tienden a ser propiedades que afectan al rendimiento o la semántica de los componentes, los sujetos de la programación subjetiva son características adicionales que se añaden a otros sujetos.

10. Conclusiones

La separación de los aspectos a todos los niveles (diseño, codificación y ejecutable) es un paso importante que se ha comenzado a dar, pero que aún hay que refinar, sobre todo en cuestiones de eficiencia.

Los lenguajes de aspectos de dominio específico juegan un papel crucial en la programación orientada a aspectos y que son preferibles a los lenguajes de aspectos de propósito general, porque soportan mejor la separación de funcionalidades. Pero aún no tenemos experiencia suficiente con los lenguajes de aspectos.

Los entornos de programación orientada a aspectos que han habido hasta ahora no soportan una separación de funcionalidades suficientemente amplia, o bien el lenguaje de aspecto soporta el dominio de aspecto parcialmente, o bien no se sostiene bien la restricción del lenguaje base.

El campo de la orientación a aspectos es un campo de investigación muy joven, en el que se abre un horizonte bastante amplio. Desde la definición de los distintos aspectos mediante lenguajes de propósito específico, hasta middlewares orientados a aspectos, pasando por las etapas de análisis y diseño orientadas a aspectos, aún quedan muchos campos que abonar, por lo que se cree interesante el abrir nuevas líneas de investigación en esta parcela.

11. Listado de Figuras

Figura 1 Esquema de la evolución de la ingeniería del software 5
Figura 2. Descomposición en aspectos ¿Una quinta generación? 7
Figura 3. Estructura de un programa orientado a aspectos. 11
Figura 4 Comparativa de la forma de un programa tradicional con uno orientado a aspectos 11
Figura 5. Estructura de una implementación en los lenguajes tradicionales. 14
Figura 6. Estructura de una implementación en los lenguajes de aspectos. 14
Figura 7. Estructura de clases resultado de tejer dos aspectos y una clase con el Tejedor AOP/ST. 16
Figura 8 Protocolo entre un objeto y su coordinador en COOL. 19
Figura 9. Aviso definido mediante un corte con nombre. 22
Figura 10. Aviso definido mediante un corte anónimo. 22
Figura 11 Código de la clase cola circular sin restricciones de sincronización. 25
Figura 12. Objeto cola circular con restricciones de sincronización. 26
Figura 13. Clase ColaCircular definida en el lenguaje base de COOL. 27
Figura 14. Implementación de la cola circular utilizando un coordinador COOL. 28
Figura 15 Aspecto para definir estrategia de sincronización mediante AspectJ. 29
Figura 16. Clase ColaCircular para un aspecto en AspectJ 30
Figura 17. El Aspecto como un elemento del metamodelo UML derivado de Clasificador 34
Figura 18. Representación de un aspecto en UML extendido 34
Figura 19. Tipos de relaciones del metamodelo de UML 35
Figura 20. Notación de la relación aspecto-clase 36
Figura 21. Estructura de las clases tejidas. 37

12. Referencias

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

[2] C. I. V. Lopes, *D: A Language Framework for Distributed Programming*, Ph.D. thesis, Northeastern University, Nov. 1997

[3] F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, P. Verbaeten. "*Aspects should not die*." In European Conference on Object-Oriented Programming, Workshop on Aspect-Oriented Programming. 1997.

[4] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. *Aspect-oriented programming workshop report*, 1997.

[5] K. Mehner, A. Wagner, *An Assesment of Aspect Language Design,K* University of Paderborn. First International Symposium on Generative and Component-Based Software Engineering (GSCE' 99) Young Researchers Workshop Abstracts

[6] J. Lamping. *The Interaction of Components and Aspects.* Position paper at the ECOOP'97 workshop on Aspect Oriented Programming, 1997.

[7] http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html

[8] Lea D. *Concurrent Programming in Java^TM: design principles and patterns.* Addison-Wesley, 1996.

[9] J. Jaworski. *Java^TM. Guía de Desarrollo.* Prentice Hall, 1996.

[10] J. Lamping. *The role of the base in the aspect oriented programming.* Position paper in the European Conference on Object-Oriented Programming Workshop, 1999.

[11] K. Boellert. *On Weaving Aspect.* Position papers in the European Conference on Object-Oriented Programming Workshop, 1999.

[12] *AspectJ* Home Page: http://www.parc.xerox.com/aop/aspectj/

[13] G. Kiczales, C. Lopes. *Aspect-Oriented Programming with AspectJ^TM –Tutorial-.* Xerox Parc. http://www.aspectj.org.

[14] *AspectJ^TM : User's Guide.* Xerox Parc Corporation. http://www.aspectj.org.

[15] *AspectJ^TM . Código de ejemplo.* Xerox Parc Corporation. http://www.aspectj.org.

[16] *AOP/ST* Home Page: http://www.germany.net/teilnehmer/101,199268/

[17] Kai Böllert. *Aspect-Oriented Programming. Case Study: System Management Application.* Graduation thesis, Fachhochschule Flensburg, 1998.

[18] C. Videira Lopez, G. Kiczales. *Recent Developments in AspectJ^TM*. Position paper at the ECOOP'98 workshop on Aspect Oriented Programming, 1998.

[19] J. Suzuki, Y. Yamamoto. *Extending UML with Aspect: Aspect Support in the Design Phase*. 3er Aspect-Oriented Programming (AOP) Workshop at ECOOP'99.

[20] OMG. *Unified Modeling Language Specification*. *v. 1.3*, *1ª Edición*. Marzo, 2000. http://www.omg.org/technology/documents/formal/unified_modeling_language.htm