Los aspectos en el diseño de aplicaciones orientadas a aspectos

Juan Manuel Nieto Moreno *
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT

La Programación Orientada a Aspectos (POA) [1] es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Una de las dificultades que está encontrando esta tecnología para tener una mayor difusión es la falta de estándares para representar los aspectos en las fases tempranas del ciclo de vida del software. En este documento nos centramos en la fase de diseño, para la que existen diferentes propuestas para tener en cuenta la influencia de los aspectos en el sistema, aunque dichas propuestas están aún en fase de maduración.

1. INTRODUCCIÓN

Algunas de las propuestas que presentamos han merecido especial atención, aunque no hayan satisfecho de forma total las necesidades semánticas de las actuales implementaciones de POA. La mayor parte de ellas proponen extensiones a las técnicas de modelado para sistemas orientados a objetos. La más extendida de estas es el lenguaje de modelado UML [8] y, en ello, se pueden distinguir dos líneas generales diferentes. Primero, las extensiones de UML con estereotipos específicos para aspectos concretos. En estos, los constructores requeridos por cada aspecto en particular están estereotipados de modo que la herramienta encargada del proceso de fusión de aspectos y clases bases pueda determinar los elementos a los que afecta el aspecto en concreto. Así, muchos de los detalles de comportamiento de los aspectos no se reflejan en el diseño en UML, sino que son conocidos por la herramienta de entrelazado.

El otro tipo de intentos opta por una manera más generalizada de soportar los aspectos en UML. Así por ejemplo, hay quien define un metaconstructor, de nombre `Aspect`, y definen estereotipos para el comportamiento de los avisos. Las operaciones afectadas por los avisos son referenciadas por dichos estereotipos. Una de las propuestas analizadas, patrones de composición, se diferencia de estas dos generalizaciones en su intento de diseñar comportamientos de corte reusables de forma independiente de la metodología de programación.

Cada una de las propuestas analizadas en este documento, así como muchas otras que aquí no aparecen, tienen sus ventajas y desventajas a la hora de describir los comportamientos de corte. Será el objetivo de este documento dar una vista general de tres de ellas que hasta la fecha han tenido una buena aceptación entre la comunidad investigadora y que enfocan la solución del problema de maneras diferentes. Como se ha dicho, la mayor parte de las propuestas explotan las posibilidades de UML para la especificación de los aspectos. Existen varias razones de peso para usar UML: primero por ser éste el lenguaje estándar de modelado más aceptado para la especificación, visualización, construcción y documentación del software entre la comunidad informática; segundo, porque UML es un lenguaje de modelado de propósito general, usable en casi cualquier dominio de problemas; y tercero, porque dispone de mecanismos de extensión, lo que lo convierte en un lenguaje de modelado extensible que le permite adaptarse a dominios específicos.

A continuación se estudiarán las propuestas de Suzuki y Yamamoto [4], Herrero et al. [5] y Composition Patterns [6] (Patrones de composición) de Siobhan Clarke. Tras su estudio y como conclusión de las observaciones realizadas en cada propuesta, se verá cuáles son las que consideramos características deseables para un buen modelo de diseño.

* Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

2. PROPUESTA DE SUZUKI Y YAMAMOTO

Suzuki y Yamamoto proponen [4] la introducción de una nueva meta clase, en el metamodelo de UML, llamada "aspecto". Reconocen los aspectos como "elementos que describen características de comportamiento y estructurales" y por ello los identifican como una clase especial de clasificadores, subclase de la metaclase clasificador de UML como muestra la figura 1:

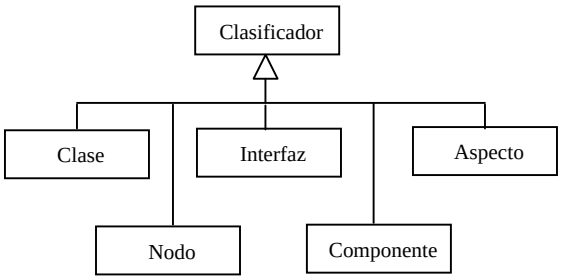

Figura 1 Nuevas metaclases de UML

De esta forma un aspecto puede tener cualquier número de atributos y operaciones y puede participar en múltiples asociaciones, generalizaciones y relaciones de dependencia. Para Suzuki y Yamamoto, las introducciones y avisos son consideradas como declaraciones de entrelazado, que se representan como operaciones especiales estereotipadas como "*weave*", indicando además el tipo de esta, por ejemplo, `{introduce}`. Cada operación "*weave*" debe indicar los elementos de la estructura a los que afecta (clases, métodos o variables). Esto se hace anteponiendo al nombre de la operación, un patrón de tipos que coincida con los elementos a los que afecta. La relación entre los aspectos y las clases a las que afectan se realiza mediante una relación de dependencia con el estereotipo "*realize*" de UML (véase [8] 2-18). En la figura 2 se ejemplifica lo dicho con un aspecto que implementa el patrón Observador y las dos clases a las que afecta, `Subject` y `Observer`.

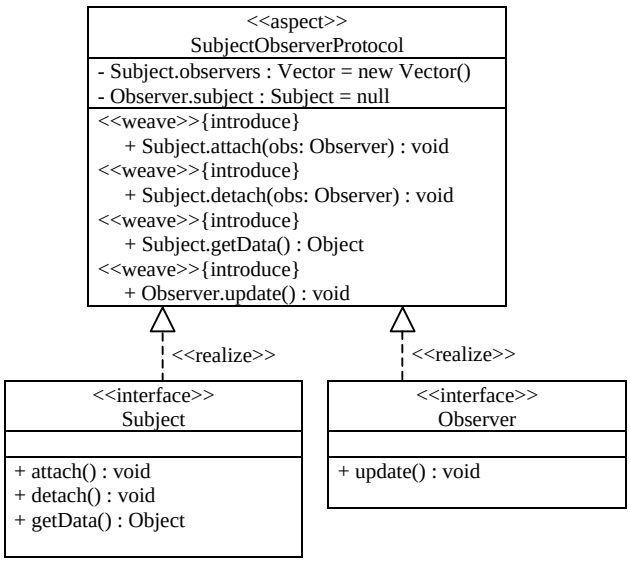

Figura 2 Ejemplo del aspecto SubjectObserverProtocol

Finalmente, como se ve en la figura 3, la estructura resultante, después de enlazar los aspectos con las clases afectadas, se representa en un paquete. Cada clase que se haya visto modificada durante el proceso de entrelazado se estereotipa con `<<wovenclass>>`. Suzuki y Yamamoto recomiendan que se especifique en la clase resultante, la clase origen y el aspecto que han intervenido.

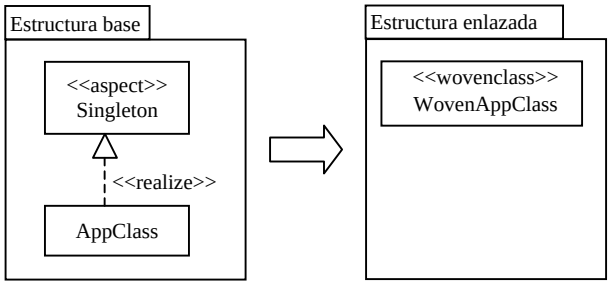

Figura 3 Resultado de enlazar clases y aspectos

En esta propuesta, donde se hace referencia a AspectJ [2] como lenguaje de aspectos, no se dice nada acerca de cómo se especifican los puntos de cortes, los avisos o la introducción de atributos. En cuanto a la relación `<<realize>>`, de la que se afirma que "es una relación entre un modelo de especificación y un modelo que lo implementa", en el caso de AspectJ, los aspectos realizan ambas cosas, tanto la declaración del concepto entrelazado, como la implementación del mismo. Analizar las reglas de entrelazado es parte del proceso de entrelazado, con objeto de identificar cuáles son las clases afectadas y generar las clases resultantes finales.

Esta propuesta se puede considerar un primer intento de extender UML para soportar los conceptos de POA, pero que, debido a los problemas comentados, es inapropiada para representar los conceptos tal como se han identificado en la aplicación que acompaña a este proyecto. En el siguiente apartado, se analiza otra propuesta, la de José L. Herrero y sus colaboradores.

3. PROPUESTA DE HERRERO ET AL.

José L. Herrero, profesor del departamento de informática de la Universidad de Extremadura, junto con M. Sánchez y F. Sánchez, presentó en el ECOOP'2000 otra interesante propuesta [5]. En ella las funcionalidades básicas de los objetos se definen en los objetos mismos, mientras que, otras propiedades de carácter no funcional se definen en entidades separadas. Para clasificar dichas otras entidades se utilizan estereotipos. Así la propiedad de sincronización, por ejemplo, se puede modelar independientemente usando los mecanismos estándar de UML. Una vez que las características estructurales y de comportamiento de las propiedades no funcionales se han definido con los modelos

adecuados, su sintaxis y semántica se captura en estereotipos especiales, que serán los que después se relacionen con cuantas clases sea necesario. Dicha relación se representa, también, mediante un estereotipo especial, que provee una meta propiedad, la cual especifica la correspondencia entre las clases bases y las estereotipadas definidas.

Herrero et al. ejemplifican su propuesta con el diseño del aspecto de sincronización (véase figura 4), entendiéndose por sincronización "un conjunto de restricciones impuestas a un objeto que, para poder preservar su integridad, aplazan o aceptan ciertas acciones". Para el diseño necesitan dos nuevos estereotipos. El primero de ellos, `<<Synchronization>>`, se introduce para las clases. Sus atributos incluyen información referente al estado actual de sincronización y sus métodos permiten alterar dicho estado. Véase la figura 4 donde se implementa este aspecto.

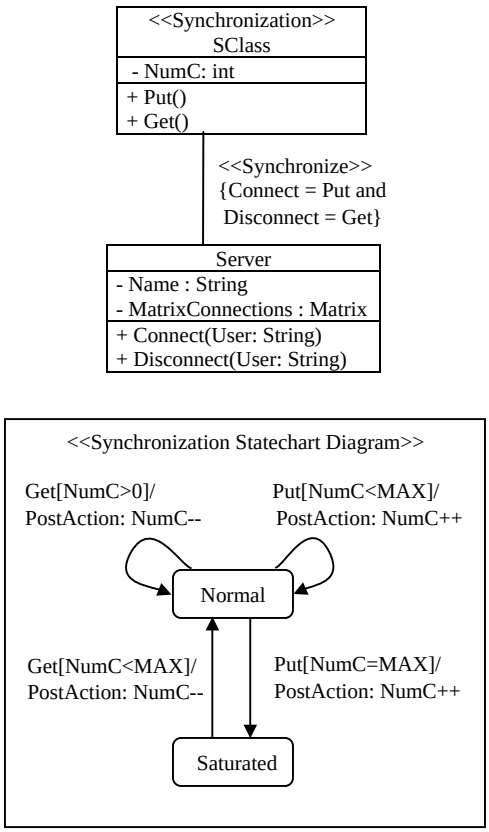

Figura 4 Aspecto de sincronización (usando estereotipos)

En el ejemplo, el atributo `NumC` representa dicha información del estado y las operaciones `Put()` y `Get()` las acciones que lo alteran. El otro estereotipo, `<<Synchronization Statechart Diagram>>`, definido para los diagramas de estado, describe el comportamiento dinámico del nuevo estereotipo anterior, de tal forma que cada clase `<<Synchronization>>` está asociada con un diagrama de estados del estereotipo `<<Synchronization Statechart Diagram>>`. Como restricción, las transiciones del diagrama de estados sólo se pueden disparar con operaciones contenidas en la clase del estereotipo asociado.

Como se ve en la figura 5, en las transiciones se puede especificar el evento de llamada, la condición de disparo, precondiciones, la acción a realizar y poscondiciones si las hubiera. Estas guardas y pre-poscondiciones se corresponderían en AspectJ con los avisos `before`, `after` y `around`.

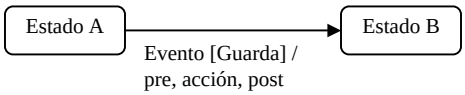

Figura 5 Transiciones en <<Synchronization Statechart Diagram>>.

Una vez definidos estos estereotipos, se puede aplicar el aspecto a las clases bases deseadas. Para ello se usa una relación de asociación también estereotipada como `<<Synchronize>>` (véase el ejemplo de la figura 4). Dicho estereotipo requiere una expresión para indicar los elementos que, en cada entidad, representan el mismo concepto. En el ejemplo, dicha expresión indica que las operaciones `Connect` y `Disconnect` de la clase base se corresponden con `Put` y `Get`, respectivamente, del aspecto. Desde el punto de vista de AspectJ, se puede entender esta técnica como la definición de los puntos de corte con las operaciones que se van a ver afectadas por el aspecto.

Sin embargo, con esta propuesta tampoco es posible representar fielmente la semántica de AspectJ. En UML una asociación se usa para definir una relación semántica entre dos entidades. En el caso del estereotipo `<<Synchronized>>` la relación se puede interpretar como de "es parte de" o "tiene", lo que significaría que la clase base asociada es complementada con las propiedades y estados de la clase `<<Synchronized>>`, por lo que dichas propiedades y estados serían subpropiedades y subestados, respectivamente, de la clase base. Sin embargo, AspectJ no está sólo limitado a la adición de subpropiedades y subestados, sino que también permite definir elementos que afectan a los ya existentes en la clase base, o bien, añadirle nuevos, de tal forma que pasan a ser miembros como los otros.

Otra deficiencia se deriva de la definición de las expresiones que establecen la correspondencia entre las operaciones de la clase base y el aspecto. Estas se pueden considerar como la declaración de los puntos de corte, sólo que, al hacerse en las relaciones, se está perdiendo la posibilidad de poder sobrescribir su definición en subaspectos, lo cual nos permite AspectJ. De acuerdo con el mecanismo de herencia de

UML, esto sólo sería posible si se declarasen como elementos del aspecto.

Finalmente, la propuesta analizada es un intento de extender UML para definir en entidades separadas los problemas horizontales en las aplicaciones. Sin embargo, sus capacidades de modelado están limitadas al diseño de propiedades y estados suplementarios que se asocian con la clase base. La siguiente propuesta surge de la metodología "*subject-oriented programming*", y veremos como sus proponentes justifican su aplicación a la metodología orientada a aspectos.

4. PROPUESTA DE CLARKE

Patrones de composición, del nombre inglés *Composition Patterns* es el modelo propuesto por Siobhan Clarke en su tesis [6], para enfrentar el problema de los requisitos que afectan a múltiples clases repartidas por el sistema. En principio, la propuesta está orientada al campo de la programación orientada a sujetos (*subject-oriented progamming*, *SOP*), pero, según sus autores, es aplicable a la POA [7]. La idea se basa en la observación de que existen patrones en la forma en la que los aspectos se relacionan con las clases bases, lo que permite el diseño independiente de unos y otros. Una vez se tengan estos diseños independientes, es posible combinarlos definiendo reglas de composición entre ellos.

Debido al mayor fundamento teórico subyacente, este apartado se subdivide en dos, introduciendo en el primero el modelo de patrones de composición y analizando en el segundo, cómo pueden las mismas ideas aplicarse a la POA y los inconvenientes que ello conlleva.

Los patrones de composición son plantillas UML para los sujetos de diseño, que toman clases (clases modelos) y operaciones (operaciones de conducta) como parámetros. Los sujetos de diseño son paquetes UML que se usan en el modelado de diseño de sujetos (SODM, del inglés *Subject-oriented design model*) para encapsular todos los elementos del modelo concernientes a un requisito. Por consiguiente, los patrones de composición son sujetos de diseño parametrizados, que encapsulan todos los elementos del modelo relativos a un requisito de tipo horizontal, es decir, lo que entendemos por aspecto.

Las clases modelos se usan para modelar conceptos estructurales horizontales, mientras que las operaciones de conducta se usan para los relativos al comportamiento. En un patrón de composición deberá existir al menos una clase modelo y, por cada una de las existentes, varias (posiblemente cero) operaciones de conducta. Por ello, cada conjunto de parámetros (una clase modelo y sus operaciones de conducta) se agrupan encerrados entre los símbolos de mayor y menor (`< >`). Véase, como ejemplo, la figura 5, que muestra un patrón de composición *Observer* que implementa el conocido protocolo *Subject/Observer*. Dicho patrón contiene dos clases modelo, que son `Subject` y `Observer`, y una que no lo es, de nombre `Vector`. La clase modelo `Subject` tiene una sola operación de conducta, `_aStateChange`, mientras que `Observer` tiene tres, `update`, `_start` y `_stop`. Véase en la figura 5 cómo están definidos en la caja de parámetros de la plantilla usando los símbolos <>.

Respecto a la signatura de las operaciones, se tienen las siguientes posibilidades:

- `opTemp()` Indica que la operación por la que será reemplazada no tiene parámetros.
- `opTemp(..)` Indica que la operación por la que será reemplazada puede tener cualquier número de parámetros.
- `opTemp(.., TypeName, ..)` Indica que la operación por la que será reemplazada puede tener cualquier número de parámetros, pero al menos uno del tipo indicado por `TypeName`.

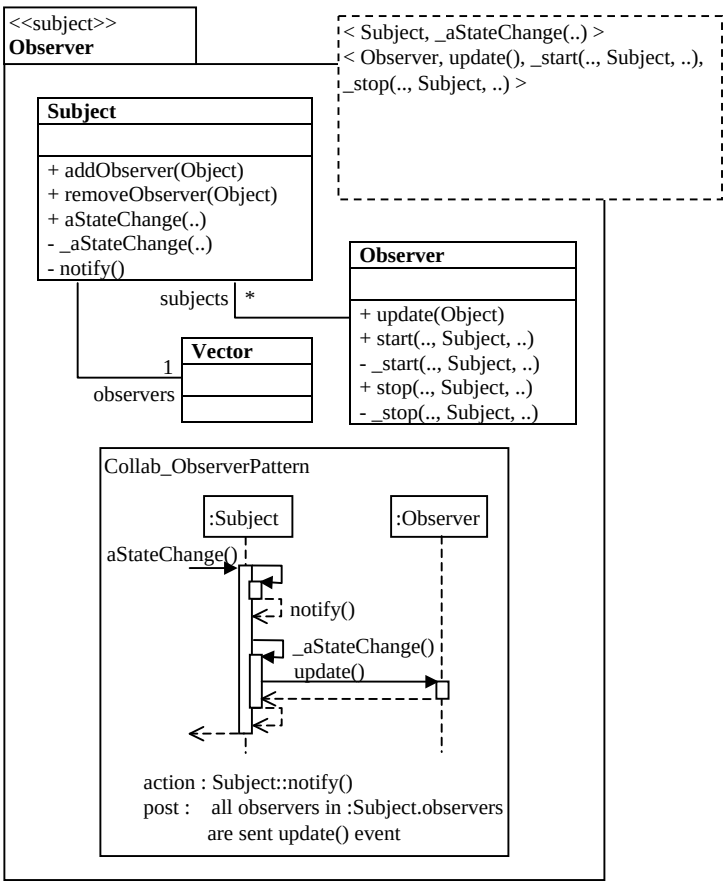

Figura 6 Ejemplo del aspecto `Observador`

EL SODM define una relación de composición para poder especificar cómo los diferentes conceptos han de ser integrados para poder tener el resultado combinado deseado. Los patrones de composición se vinculan con los conceptos de diseño a los que afectan

mediante dicha relación, que extiende las relaciones de enlace estándar definidas en UML entre las plantillas de especificación ([8] pag.3-54) y los elementos que han de reemplazarlas. Dicha extensión es necesaria porque en UML la correspondencia de parámetros está restringida a una relación uno-a-uno. De este modo, el atributo utilizado `bind[]`, define los, potencialmente múltiples, elementos que reemplazan las plantillas. La correspondencia entre los parámetros en `bind` y los parámetros en la plantilla es según el orden en el que están definidos.

En la figura 7, el patrón de composición `Observer` se combina con el concepto de diseño, o clase base, `Library`, haciendo uso de una relación de composición atribuida con `bind[]`. Ésta especifica las clases y operaciones que reemplazan los parámetros de la plantilla del patrón de composición.

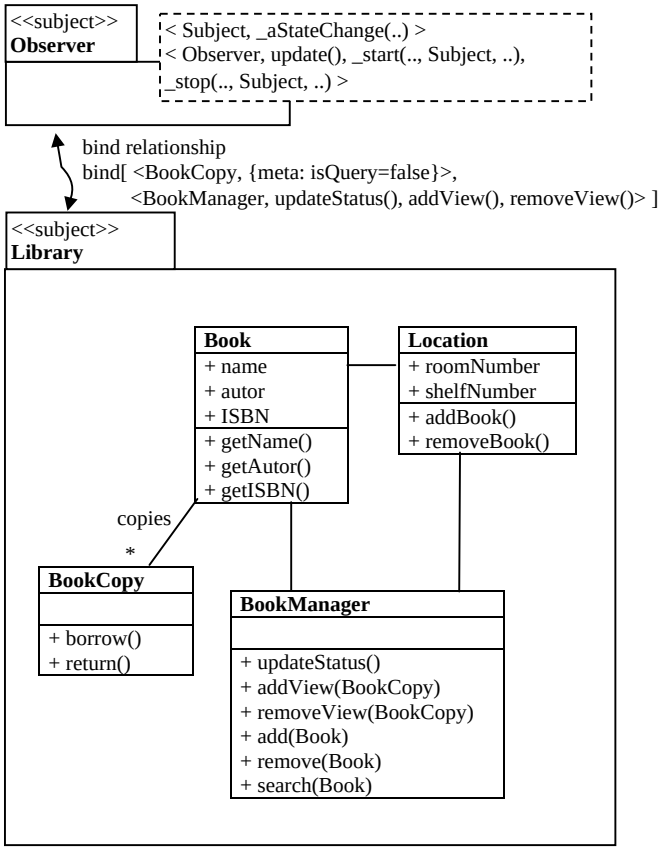

Figura 7 Componiendo el aspecto `Observer` y la clase base `Library`

De manera resumida, la sintaxis de `bind[]` es la siguiente (más detalles en [6] tabla 3):

```
bind [Body]
Body :: < PatternClass [, TempOp]* >
PatternClass :: className
             || { className [, className]* }
             || {*}
             || {meta: meta-test}
TempOp :: [className.]opName
       || { [className.]opName [,
[className.]opName]*] }
       || {*}
       || {meta: meta-test}
```

Los patrones de composición están vinculados con sus argumentos mediante una relación que incluye una especificación con `bind[]`. Las relaciones de composición con `bind[]` aplican siempre una estrategia de integración conocida como fusión de patrones, que indica como se combinan los elementos de diseño. El SODM propone dos estrategias, fusión y sobreescritura, que integran las propiedades de las clases modelo como los atributos, sus asociaciones, generalizaciones y relaciones de dependencia. Para indicar cómo se fusionan las operaciones se utilizan diagramas de colaboración, que se incluyen en los mismos patrones de composición. En estos, las operaciones originales de entrada, cuyo comportamiento se va a alterar, y las resultantes después del proceso de enlazado se referencian con el mismo nombre, salvo por un guión subrayado "_" que se antepone a las primeras (véase como ejemplo en la figura 6 *_aStateChange*, *_start*, *_stop* y *update*). Además sólo las operaciones resultantes (o las que no se alteran) se definen públicas, para asegurar que serán las únicas que puedan usarse.

La figura 8, en la siguiente página, muestra el resultado de enlazar el patrón de composición `Observer` con el sujeto `Library`, usando las relaciones de composición especificadas en la figura 9. Observe como cada operación de entrada y la correspondiente operación enlazada de salida (ejemplo, *_aStateChange* y *aStateChange*) han sido sustituidas por las operaciones resultantes (siguiendo con el mismo ejemplo, *Library_borrow* y *borrow*) en las clases finales (*BookCopy*), así como en los diagramas de colaboración finales (*Collab_ObserverPattern - borrow*). La clase `BookManager` como observadora que es, tiene ahora la operación `updateStatus()`, que es la operación que se llamará para las notificaciones. Las operaciones `addView` y `removeView` inicializan un objeto `BookCopy` añadiendo y eliminando, respectivamente, un objeto `BookManager` en su lista de observadores.

4.1. APLICACIÓN A LA POA

Conceptualmente, la programación orientada a aspectos y el diseño con patrones de composición persiguen los mimos fines. Los patrones de composición permiten la separación y el *diseño* de comportamientos de corte de modo que sean reusables, mientras que la POA permite la separación y la *programación* de los mismos. Además Clarke afirma que los patrones de composición constituyen un modelo de diseño independiente de cualquier modelo de programación y para demostrarlo, Clarke presenta en [7] un mecanismo de correspondencia entre los conceptos definidos para los patrones de composición y los conceptos de POA en AspectJ. Tal correspondencia se describe en la siguiente tabla:

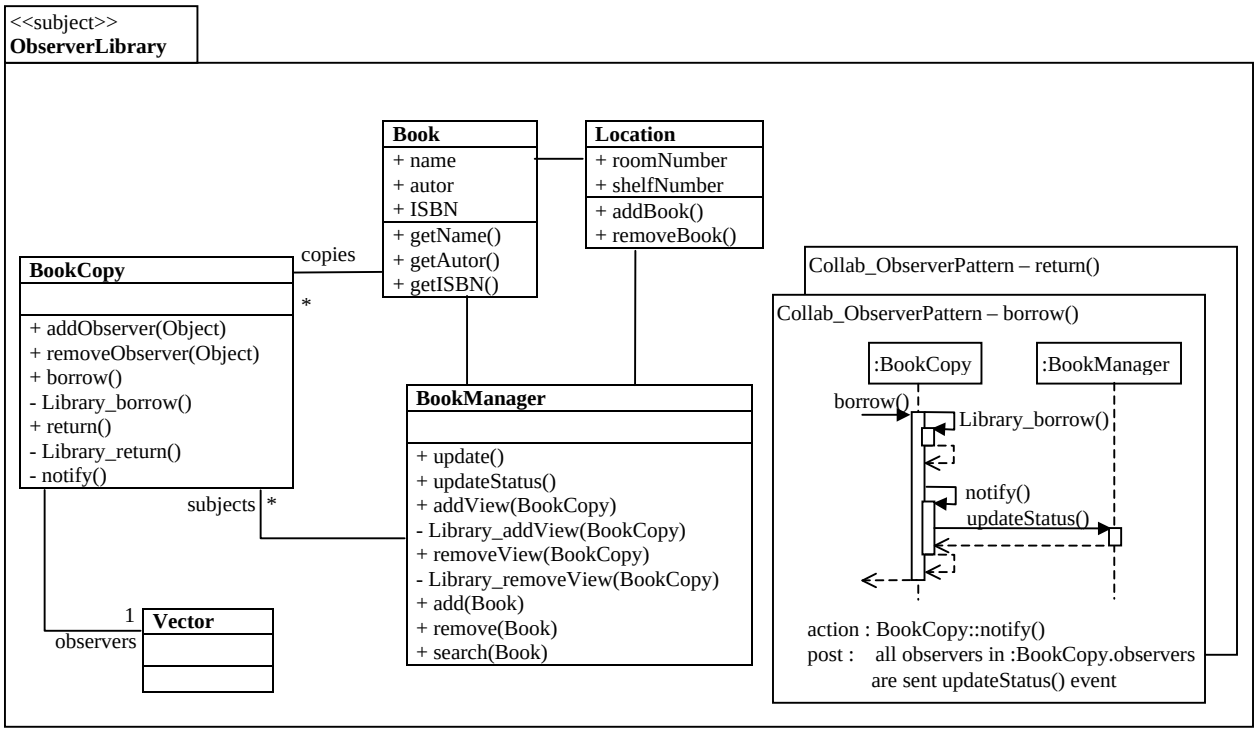

Figura 8 Fusión del aspecto `Observer` y la clase `Library`

Element	AspectJ	Composition Pattern
Aspect	Un aspecto es un constructor, con una interfaz bien definida, que se puede instanciar y manejar en tiempo de compilación.	Un patrón de composición es a nivel de diseño equivalente a un aspecto.
Pointcut	Los puntos de enlace son los puntos donde intervienen los comportamientos de corte. Un pointcut es un conjunto de puntos de enlace.	Las operaciones de conducta se pueden definir y referenciar en los diagramas de interacción, indicando que son puntos de enlace para los comportamientos de corte. Dichas operaciones pueden ser reemplazadas múltiples veces por operaciones finales y son por tanto equivalentes a los puntos de enlace.
Advice	Un advice es un fragmento de código que se ejecuta en un punto de corte.	En un diagrama de interacción, se puede indicar que se ejecute el comportamiento de corte cuando se llame a una operación. Ese comportamiento es equivalente al de los advices.
Introduction	Una introducción es un elemento de programación, tal como un atributo, un constructor o un método, que se le añade a un tipo.	Los elementos de diseño que no son elementos de plantilla pueden ser definidos en los patrones de composición. Estos pueden ser clases, atributos, operaciones o relaciones, y son equivalentes a las introducciones.

La propuesta de Clarke extiende UML para poder encapsular requisitos de corte en un modelo de diseño que denomina patrones de composición. Sin embargo, a pesar de las similitudes, el mismo modelo no es apropiado para diseñar programas orientados a aspectos en el propio AspectJ (a pesar de las correspondencias expuestas). Se explican a continuación los principales conflictos entre la tabla anterior y la semántica de los aspectos en el modelo de aspectos de AspectJ.

Los patrones de composición no diferencian apropiadamente los avisos de los aspectos. Los avisos deben extraerse de los diagramas de interacción, que describen de qué tipo son (`before`, `after` o `around`) y las acciones que lleva a cabo. Además los diagramas de interacción no establecen apropiadamente el campo de acción dónde se ejecutan los avisos. De acuerdo con la semántica del modelo de patrones de composición, las colaboraciones de los diagramas de interacción se encuentran en el ámbito de las clases bases (véase en la figura 6 el diagrama de interacción de `Collab_ObserverPattern`, que realiza la operación `aStateChange`; en la figura 8 aparece dos veces, una ligado a la operación `borrow` y otra a `return`). A diferencia, en AspectJ los avisos se ejecutan en los aspectos y no en las clases bases. De esta forma, en patrones de composición, el aviso podría acceder a cualquier miembro de la clase base, mientras que en AspectJ, el acceso a miembros por

parte de los avisos está limitado a aquellos que los puntos de corte expongan.

Los aspectos en AspectJ pueden contener miembros propios como atributos y operaciones. Los patrones de composición, al ser estereotipados como paquetes UML, no pueden. Introducirlos en las clases modelo, implicaría que terminarían en las clases bases.

Además, con los patrones de composición, sólo se pueden diseñar aspectos de corte estáticos. Los dinámicos, en los que, por ejemplo, existiera una dependencia con el contexto dinámico en el punto de enlace, no se consideran. En cuanto a los avisos, éstos no sólo afectan al comportamiento original de clases bases, sino que también puede afectar a los nuevos miembros añadidos con `introduction`. La semántica de los patrones de composición no soporta esta característica.

El último de los conflictos detectados resulta del hecho de que los patrones de composición cambian la definición estática de las clases. Esto ocurre al añadir nuevas operaciones que implementan los comportamientos de corte a la definición de las clases bases. Sin embargo, los avisos en AspectJ no modifican las operaciones existentes, ni les añaden ninguna nueva a las clases bases.

El SODM de Clarke es un buen modelo para la programación orientada a sujetos. Sin embargo, aunque ambas metodologías persigan fines similares, este modelo no es apropiado para representar conceptos básicos de la semántica de lenguajes orientados a aspectos como AspectJ. Como resultado de la observación de las particularidades acertadas de los modelos expuestos, así como de sus carencias y dificultades, el siguiente apartado propone las características deseables de un buen modelo apropiado para la POA.

5. COMO DEBE SER UN BUEN MODELO

Se han analizado tres propuestas para el diseño de aplicaciones orientadas a aspectos, una de ellas proveniente de la programación orientada a sujetos, y se ha visto, contrastando con la semántica de AspectJ, que no se adecuan suficientemente a la semántica de las aplicaciones orientadas a aspectos. Como conclusión del documento podemos esquematizar las que serían las características deseables que debería tener un modelo que nos permitiera de manera adecuada definir este tipo de aplicaciones. Se presentan a continuación a modo de enumeración:

1. Un lenguaje de modelado debe permitir la especificación tanto de los elementos estructurales como de comportamiento que se entrelacen en una determinada descomposición de un sistema.
2. Un lenguaje de modelado debe permitir la especificación del conjunto de puntos de enlace donde los conceptos de corte afectan a la aplicación.
3. Un lenguaje de modelado debe permitir que dichas especificaciones sean modeladas de manera independiente de modo que puedan ser reusadas y combinadas en diferentes contextos.
4. Un lenguaje de modelado debe ofrecer herramientas gráficas para designar los elementos (y sus interrelaciones) que son afectados por los conceptos de corte, tanto en la especificación de los conceptos como en la de los puntos donde estos repercuten.
5. Un lenguaje de modelado debe permitir atribuir de alguna manera las especificaciones de los conceptos de corte, permitiendo indicar la manera en la que dichos conceptos afectan a los otros y cómo habrán de ser combinados con los elementos afectados.
6. Un lenguaje de modelado debe permitir un nivel de abstracción adecuado para la especificación estructural y de comportamiento de los elementos de corte.
7. Un lenguaje de modelado debe permitir un nivel de abstracción adecuado para la especificación del conjunto de puntos de enlace.
8. Un lenguaje de modelado debe ofrecer a los diseñadores herramientas que permitan, de manera sencilla, obtener resultados e interpretarlos de manera inequívoca.
9. Un lenguaje de modelado debe permitir a los diseñadores identificar los elementos que están en uso en las especificaciones.
10. Un lenguaje de modelado debe permitir a los diseñadores definir cómo se componen los elementos base con los aspectos.

6. CONCLUSIONES

La fase de diseño es una actividad importante del ciclo de vida del software que a menudo queda olvidada. En el caso de la POA, la mayor parte del trabajo, y sobre todo en su nacimiento, se ha centrado en el desarrollo de los lenguajes de implementación, olvidando las fases iniciales. Quizás por ello no se dispone aún de un estándar, lo que dificulta el asentamiento de la tecnología y hace que aquellos que están interesados en estudiar los beneficios del uso de aspectos tengan que acudir a las actuales implementaciones en código para entender su utilidad. Hemos analizado tres interesantes propuestas y mediante ejemplos se ha

comprobado su utilidad para especificar ciertos aspectos. Sin embargo, a la vista de éstas –y otras propuestas que aquí no se han analizado– parece difícil llegar a un equilibrio entre un lenguaje suficientemente general que sea adecuado para cada una de las diferentes implementaciones que se han hecho de la POA y un lenguaje con la suficiente capacidad expresiva, de modo que nos permita modelar todo aquello que después nos permite hacer cada lenguaje concreto. 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.

7. REFERENCIAS

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

2. *The AspectJ™ Programming Guide*, the AspectJ Team, Xerox Parc Corporation

3. Peri Tarr, Harold Ossher. *Hyper/J™ User and Installation Manual*, IBM Research, 2000.

4. 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).

5. 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).

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

7. S. Clarke y Robert J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In proceedings of ICSE 2001.

8. OMG Unified Modeling Language Specification. v. 1.5 March 2003.