Adoptando el Paradigma de la Programación Orientada a Atributos

Raúl Marticorena Sánchez, Carlos López Nozal, Carlos Pardo Aguilar
Área de Lenguajes y Sistemas Informáticos
EPS, Edificio C. C/ Francisco de Vitoria, s/n. CP 09006, Burgos
Universidad de Burgos
email: {rmartico, clopezno, cpardo}@ubu.es

Resumen

A la sombra del paradigma de la programación orientada a objetos, surgen en la actualidad nuevos paradigmas, relacionados en mayor o menor medida. Dentro de ellos se pueden enmarcar la programación orientada a aspectos y la programación orientada a atributos. Dada la actual imposición curricular del paradigma orientado a objetos, resulta difícil escapar a la inclusión de alguno de estos nuevos paradigmas en el campo docente.

En particular este trabajo muestra los primeros resultados de la introducción del concepto de programación orientada a atributos: desde una visión superficial, en la que se usan marcos de trabajo simples, a un nivel profundo con el desarrollo de bibliotecas basadas en dicho concepto, y finalmente con su empleo en entornos más avanzados.

El trabajo pretende mostrar las primeras experiencias realizadas en este sentido, y los resultados observados, para ir mejorando en su progresiva incorporación, y ayudando a futuras planificaciones de su integración y uso en asignaturas.

1. Planteamiento del Problema

Desde hace ya muchos años se ha estado impartiendo en asignaturas de programación el paradigma de orientación a objetos (POO). En nuestro caso particular, con diferentes lenguajes como C++, Eiffel o Java, pero básicamente centrándonos en las características generales: abstracción, encapsulación, herencia, polimofismo, etc [1].

Durante varios años, esta situación se ha mantenido estable, incluso con la incorporación del concepto de genericidad en lenguajes muy extendidos (Java) o pujantes (lenguajes de la familia .NET).

Sin embargo, el propio paradigma ha originado problemas que han motivado nuevas soluciones. El paradigma de la orientación a aspectos surge como necesidad a la captura de partes de un sistema que los modelos de programación habituales obligan a repartir en distintos módulos del sistema. Estos fragmentos, que afectan a distintos módulos, son llamados aspectos, y los problemas que solucionan se denominan problemas cruzados (crosscutting concerns).

Por otro lado se observa que algunos aspectos semánticos pueden ser resueltos por medio de herramientas, aprovechando ciertas declaraciones en el código. Esta opción, menos compleja, resuelve problemas similares a la programación orientada a aspectos, y recibe el nombre de programación orientada a atributos (POA) [2][3][4].

Los programadores pueden señalar o marcar elementos de sus programas (clases, variables de instancia, métodos, etc.) con metadatos, para indicar que mantienen una semántica específica de la aplicación o del dominio. Otras herramientas procesan dichos atributos para entrelazar la lógica de negocio con las cuestiones semánticas marcadas.

Como ejemplo, en esta línea de trabajo, surgieron soluciones como la incorporación del diseño por contrato con etiquetas, siguiendo reglas de documentación (JML [5]), o la generación de código intermedio (XDoclet para EJB [6]). Estas soluciones siguen la idea previa de usar preprocesadores: tomando cierta información introducida por el programador (comentarios especiales) se genera código adicional.

Dicha solución ha sido adoptada por las dos plataformas de desarrollo dominantes, .NET y Java, incorporando la idea de atributo en las correspondientes especificaciones de sus lenguajes.

Este concepto también permite llevar a cabo la implementación directa de un elemento de UML [7] tan usado como son los estereotipos y valores etiquetados. El actual vacío que existía entre la solución de diseño e implementación viene a ser cubierto de una manera natural.

En la medida de nuestro conocimiento, no existen experiencias docentes en este campo. Por lo tanto, no cabe sino afrontar el reto de introducir dicho paradigma como una pieza más de aquellas asignaturas en las que se ha venido utilizando la POO y el modelado UML como solución instrumental.

A continuación este trabajo presenta el contexto docente particular en la Sec. 2, identificando el problema de partida. En la Sec. 3, se expone el uso de atributos dentro de un marco de trabajo simple, basado en pruebas. En la Sec. 4 se definen, desarrollan y usan bibliotecas semánticas para la identificación de patrones de diseño y generación de código. Como última experiencia, en la Sec. 5, se utilizan atributos para el desarrollo avanzado de componentes distribuidos. Se finaliza en la Sec. 6 con las conclusiones obtenidas y líneas de trabajo futuro.

2. Contexto Particular

El paradigma de POO se introduce en segundo curso de la titulación técnica utilizando Java como lenguaje de soporte. A partir de ese momento, se ve reforzado su uso en asignaturas relacionadas con el desarrollo de estructuras de datos (2º curso), programación avanzada usando patrones de diseño y la web (3er curso), y en trabajos final de carrera. En paralelo se imparte en segundo curso UML como lenguaje de modelado y se utiliza intensivamente en asignaturas de cursos posteriores.

En el segundo ciclo de la ingeniería se plantean prácticas en C, C++ y Java, tomando como base los lenguajes más utilizados en el ciclo anterior. Tanto en primer como segundo ciclo se muestra una cierta evolución a implantar .NET, particularmente en trabajos final de carrera.

La experiencia actual es positiva y la demanda por parte de las empresas circundantes confirma la adecuación de dichos contenidos. Pero debido precisamente a la elección de Java y el posible cambio hacia .NET, junto con el hecho de que ambos en sus últimas especificaciones incluyan atributos como parte inherente del lenguaje, disparan la necesidad de su estudio.

Sin embargo, en la actualidad surge el problema, desde un punto de vista docente, de cómo y cuándo incluir esta nueva solución. Básicamente motivado por una adaptación a los nuevos paradigmas, entornos de desarrollo y por la posibilidad de utilizar este nuevo concepto de manera natural.

3. Primera Experiencia: Uso de Atributos

Durante los últimos años se ha venido incorporando la inclusión de pruebas [8] [9] de forma obligatoria en asignaturas de programación de nivel II.

Dichas prácticas se han venido realizando utilizando un framework de pruebas como JUnit [1] [10] que se ha convertido en un estándar de facto.

Las versiones utilizadas hasta la fecha se basaban en un diseño basado en patrones [13] y la utilización de reflexión e introspección para su correcto uso. A los alumnos se les presentaba dicho concepto brevemente, explicando el patrón *Compuesto* y las reglas de nombres a utilizar para que la reflexión basada en dichas normas funcionase correctamente.

Como ejemplo, el Algoritmo 1 muestra un breve extracto de código escrito siguiendo dichas reglas. El método `setUp` se ejecuta antes de cada test para realizar las inicializaciones pertinentes. Los métodos cuyo nombre empieza por la palabra `test`, contienen un test individual e independiente. En particular el último test comprueba que una excepción es lanzada en tiempo de ejecución siguiendo las especificaciones dadas.

Debido a la complejidad y nivel de la resolución interna de dichos conceptos, simplemente se realizaba una presentación superficial, dando una breve guía de programación.

Curiosamente, la evolución actual de este framework de pruebas, ha sufrido un cambio radical. Se abandona la solución reflexiva, los patrones de diseño quedan ocultos, y todo su funcionamiento se basa en el uso de atributos (anotaciones en Java [11]). Se ha optado por permitir compatibilidad hacia atrás y hacia delante

en los tests construidos, aunque se recomienda cambiar a la nueva solución.

```
public void setUp(){
}

public void testComprobar(){
}

public void testExcepcion(){
   try{
   // código que genera excepción
   fail(); // provocar fallo
   }
   catch(ExcepcionEsperada es){
       // ok
   }
}
```

Algoritmo 1. Codificación en JUnit 3.8.1

A continuación se muestra un breve extracto del código con atributos (ver Algoritmo 2), equivalente al mostrado previamente en el Algoritmo 1. La convención de nombres se abandona y se sustituyen por anotaciones precedidas con el símbolo @.

```
@Before
public void inicializar(){
}

@Test
public void comprobarPrecio(){
}

@Test(
expected=ExcepcionEsperada.class)
public void exception(){
// código que genera excepción
}
```

Algoritmo 2. Codificación en JUnit 4.2

Aunque la solución parece intuitiva, el cambio de paradigma puede desorientar inicialmente. De hecho, en algunos casos, el planteamiento es completamente distinto, como se puede deducir del último test, donde toda la semántica ha sido sustituida básicamente por una anotación con valores asociados.

A la hora de afrontar el cambio en la docencia de dichos contenidos se introduce el concepto, las anotaciones y su semántica asociada, ilustrando su uso sobre una colección de tests de ejemplo. Se proporciona a los alumnos una plantilla base a partir de la cual construir nuevos tests, y se les aportan ejemplos de utilización.

La complejidad de la ejecución dentro del framework, así como el procesado de dichos atributos por el propio entorno de ejecución, se omiten, incidiendo en la semántica qué implica el atributo, pero no el cómo se resuelve internamente el problema de su procesado.

3.1. Práctica Propuesta

Se plantea una práctica obligatoria donde se propone un módulo a probar, utilizando pruebas de caja negra. Para ello los alumnos deben implementar el correspondiente módulo de prueba con JUnit, siguiendo las directrices marcadas anteriormente, utilizando obligatoriamente la nueva implementación del framework.

3.2. Evaluación

Se solicita a los alumnos que rellenen una encuesta en relación con dicha práctica, centrándose en aquellos alumnos repetidores que realizaron la práctica en años anteriores con la versión previa, frente a la solución presentada en este curso.

Las preguntas realizadas y resultados obtenidos, recogen la opinión sobre 38 alumnos (un 50% aproximado de la población total de alumnos que realizaron la práctica). En las Tabla 1 y Tabla 2 se presentan las preguntas realizadas y la frecuencia de las puntuaciones recibidas.

De estos resultados se pueden obtener algunas conclusiones previas. En relación a la primera pregunta, se observa que existe una cierta tendencia a solicitar un número de horas en el temario para explicar con mayor detalle el concepto de atributo. Inicialmente el concepto se usa, pero no parece estar del todo bien definido y aclarado por parte del docente.

Curiosamente, a partir de la segunda pregunta, se desprende que no existe esa misma tendencia en el alumnado en cuanto a comprender las ventajas del nuevo concepto y su uso. El interés del aprendizaje de usar atributos en el código, no ha quedado plasmado con esta primera experiencia.

Desde el punto de vista de alumnos que han cursado la asignatura en años previos, y que

realizaron la práctica con la antigua versión del framework, se les pregunta sobre el grado de dificultad ante la nueva versión y su preferencia al utilizar una u otra (ver Tabla 2).

Pregunta \ Puntuación	1	2	3	4	5
Es necesario dedicar alguna hora de teoría a explicar el concepto de anotaciones en Java y su uso	0	4	7	18	9
Es conveniente aprender el concepto de programación orientada a atributos usando anotaciones	0	2	20	10	5

Tabla 1. Evaluación de anotaciones

En este caso las puntuaciones reflejan el grado de dificultad, siendo una puntuación de 1 muy fácil y 5 muy difícil:

Pregunta \ Puntuación	1	2	3	4	5
Considera que el cambio realizado en la nueva versión de JUnit, con el uso de anotaciones, hace que la práctica respecto a otros años, en dificultad es:	1	8	6	5	1
Versión JUnit elegida	3.8.1			4.2	
Si se dejase la posibilidad de realizar la práctica con una de las dos versiones (3.8.1 o 4.2) ¿cuál elegirías?:	9			12	

Tabla 2. Valoración de dificultad y selección

Se observa una tendencia general a considerar más fácil el uso de la nueva versión, aunque existe un cierto número de respuestas que indican todavía dificultades y reticencias a aceptarla.

Es más curioso todavía, si se observa que casi existe un empate ante la posibilidad de realizar las prácticas con la antigua versión, incluso aunque se apoye en conceptos avanzados como la introspección y patrones de diseño, conceptos que los alumnos de estos cursos todavía no han visto.

4. Segunda Experiencia: Definición de Atributos

Esta experiencia ha sido llevada a cabo en la asignatura de Diseño y Mantenimiento del Software II, impartida como materia troncal en quinto curso de Ingeniería en Informática. El contenido temático de esta asignatura, se basa en el estudio de los conceptos y relaciones existentes entre atributos del software, métricas de producto, patrones de diseño, pruebas y refactorizaciones.

En la asignatura se ha llevado a la práctica dos experiencias relacionadas con programación orientada a atributos. Por un lado, definir los atributos necesarios que permitan reflejar la semántica de aplicación de patrones de diseño, en un sistema software. Por otro lado, aprender a traducir a código los conceptos de modelado UML: valores etiquetados y estereotipos.

4.1. Práctica Propuesta: Semántica de Patrones de Diseño

Antes de proponer la práctica a los alumnos, se parte de una sesión de prácticas guiadas donde se muestra a los alumnos la importancia que tiene el conocimiento previo de los patrones de diseño en la fase de mantenimiento. En esta sesión de entrenamiento, los alumnos se enfrentan de manera supervisada al conocimiento del diseño de tres componentes software.

El problema que ocurre en la mayoría de los casos, es que parte de los componentes no mantienen la documentación de diseño. Únicamente se proporciona código ejecutable o código fuente. Se les muestra que la identificación de patrones puede hacerse de manera automática, a través de herramientas, o a partir de la documentación de diseño.

Para este cometido, en la misma sesión se enseña a manejar la herramienta WOP [14]. Esta herramienta permite identificar patrones de diseño a partir de un repositorio de definición de éstos, basado en su estructura.

El componente JUnit lleva asociada documentación de diseño basada en patrones. Por tanto, es un candidato perfecto para probar la bondad de la herramienta en su funcionalidad de identificación. En la Tabla 3 se muestra el resultado de la comparativa, la primera columna indica los patrones que son aplicados. Esta información ha sido extraída de la documentación

de diseño del componente. En la segunda columna los patrones disponibles en la herramienta para su identificación. Y en la columna tercera los patrones que se logran identificar con la herramienta.

PD en documentación de diseño JUnit	WOP Repositorio	WOP Identificación
Comando	NO	NO
Método Plantilla	SI	NO
Recolección de parámetros	NO	NO
Adaptador (clase)	SI	NO
Compuesto	SI	SI

Tabla 3. Evaluación de WOP

Una vez vista la dificultad de la herramienta para extraer la información de diseño a partir de la definición estructural externa de un patrón, se propone una nueva alternativa de trabajo: incluir la semántica asociada a la aplicación de los patrones de diseño en el propio código. Se toma como base la notación definida por uno de los autores de GoF y recogida en [15]. En esta propuesta, se toma la representación de los patrones de diseño basada en estereotipos y valores etiquetados UML [7], en lugar de colaboraciones. La razón para usar esta alternativa es el mayor contenido semántico que puede aportar. En la Figura 1 y Figura 2 se puede observar las diferentes representaciones UML de una aplicación del patrón *Singleton* [13] sobre la clase `GestorImpresion`.

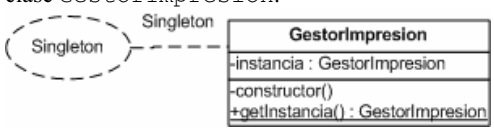

Figura 1. Representación de patrones con colaboraciones

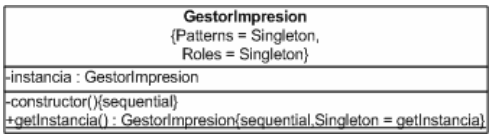

Figura 2. Representación de patrones con valores etiquetados y estereotipos

A partir de la semántica expuesta, se propone definir una anotación Java [11] que la represente. En el Algoritmo 3 se muestra el código de una posible anotación java para representar la semántica de aplicación de patrones de diseño. El tipo `PatternName` se corresponde con una enumeración de cadenas con los nombres de los patrones tratados.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface
 PatternAnnotation {
   PatternName[] patternName();
   String participantName();

}
```

Algoritmo 3. Anotación Java para patrones

4.2. Práctica Propuesta: Traductor UML a Java

La segunda experiencia se basa en añadir una nueva funcionalidad a un software existente, en concreto a una herramienta CASE UML desarrollada en varios proyectos final de carrera. Se pide construir un traductor de UML a código java que permita tratar los conceptos de estereotipos y valores etiquetados. En el desarrollo se deben aplicar patrones de diseño, refactorizaciones, pruebas y métricas de producto.

A modo de ejemplo, en la Figura 3 se muestra una clase generada con la herramienta (`PrinterPool`) donde se aplica el patrón de diseño *Singleton*. La funcionalidad pedida consiste en leer la información de las instancias del metamodelo UML y transformarlas a código java. En el Algoritmo 4 se puede observar el resultado de salida esperada.

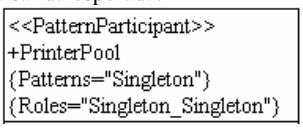

Figura 3. Entrada caso de prueba: mapeo estereotipo y valores etiquetados

```
import java.lang.annotation.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface
 PatternParticipant {
```

```
  String[] Patterns();
  String[] Roles();
}

@PatternParticipant (
 Patterns={"Singleton"},
 Roles={"Singleton_Singleton"})
public class PrinterPool{
}
```

Algoritmo 4. Salida caso de prueba: mapeo estereotipo y valores etiquetados

4.3. Evaluación

La encuesta fue realizada a 14 alumnos sobre un total de 31 alumnos matriculados en la asignatura. En la Tabla 4 se observan un resumen de los resultados.

Pregunta \ Puntuación	1	2	3	4	5
Piensas que la representación de patrones de diseño mediante anotaciones java o valores etiquetados clarifica tus diseños	0	2	5	7	0
La identificación de PD mejora la comprensión preliminar de un diseño	1	4	9	1	4

Tabla 4. Evaluación de patrones de diseño y anotaciones

Como conclusión, los alumnos muestran un interés elevado en añadir contenido semántico a su código para incorporar información explícita sobre sus diseños. Indirectamente, ven la mejor adecuación de la programación orientada a atributos para poder conseguir este propósito. Aunque como contradicción, no queda tan clara la utilidad de la semántica en la identificación de patrones.

5. Tercera Experiencia: Uso Avanzado y Evaluación de Solución con Atributos

En la asignatura optativa de quinto curso de Sistemas Distribuidos [16][17][18], se proponen prácticas utilizando componentes distribuidos Enterprise JavaBeans. Mientras que en cursos anteriores se ha venido utilizando una herramienta que sigue la especificación EJB 2.1 [19], basada en interfaces y ensamblado basado en ficheros de despliegue (utilizando XML), en el presente curso se ha avanzado hacia la siguiente especificación EJB 3.0 [20], dónde desaparece la necesidad de ficheros intermedios utilizando atributos.

Como ejemplo, mientras en la especificación 2.1 es necesario trabajar con un número de 3 a 5 clases/interfaces Java junto con 1 o 2 ficheros de despliegue XML, en la nueva especificación se puede ver reducido a una o dos clases/interfaces Java. En el Algoritmo 5 se puede ver un esquema de declaración de un EJB de mensaje con la especificación 2.1. Se omite el código de acceso a la cola de mensajes, por simplificación, y los ficheros adicionales para el despliegue del EJB.

```
public class MdbBean
       implements
   MessageDrivenBean,
   MessageListener {
   // lógica de negocio de acceso
   // a la cola de mensajes
}
```

Algoritmo 5. Esquema de EJB MDB 2.1

```
@MessageDriven(activationConfig =
{
@ActivationConfigProperty(propert
yName = "destination",
propertyValue = "ColaEjemplo"),

@ActivationConfigProperty(propert
yName = "destinationType",
propertyValue =
"javax.jms.Queue")
    }
)
public class MessageDrivenBean
implements MessageListener {
   // cuerpo de la clase
}
```

Algoritmo 6. Esquema de EJB MDB 3.0

Por otro lado, en la especificación EJB 3.0 [20], el uso de atributos (anotaciones) sustituye al código que enlaza el EJB con la cola de mensajes (inyección de recursos) y declaraciones adicionales. Se elimina la necesidad de ficheros externos XML y se simplifica el código ejecutable del EJB a partir de una solución declarativa que recoge dicha semántica. En el Algoritmo 6, se

muestra el código equivalente, utilizando atributos.

Dado que en este caso, el alumno sí dispone de una fuerte base, al tratarse de alumnos de último curso de la ingeniería, se pide una implementación siguiendo ambas especificaciones, pero además solicitando una evaluación comparativa entre ambas.

5.1. Enunciado de la Práctica

La práctica plantea el desarrollo de un Enterprise JavaBean orientado a mensajes (Message Driven Bean - MDB) para captar mensajes asíncronos. Los mensajes se envían a un tema de suscripción conteniendo un fichero a replicar en varios suscriptores, guardando un registro de la operación en base de datos.

El MDB debe ser desarrollado siguiendo ambas especificaciones:

- EJB 2.1 sobre JOnAS
- EJB 3.0 sobre EasyBeans (como plugin de JOnAS)

Se debe analizar y evaluar las correspondientes facilidades y dificultades asociadas a cada una de las soluciones. Para el desarrollo de ambos EJBs se les facilita código de ejemplo, que puede ser utilizado como plantilla.

5.2. Evaluación

Junto con el propio ejercicio práctico se ha realizado una evaluación docente de la asignatura donde se les ha planteado preguntas similares a las planteadas en primeros cursos. Al tratarse de una optativa el número de encuestados es mucho menor, recogiendo los resultados de aquellos alumnos que han seguido la asignatura con asiduidad y que han presentado las prácticas obligatorias. En la Tabla 5 se muestran los resultados recogidos.

Aunque el número de encuestados es pequeño, sí que permite recoger algunas conclusiones parciales. Al igual que ocurría en primeros cursos, los alumnos creen adecuado que se introduzca el concepto de atributos previamente.

Por otro lado un alto porcentaje ve conveniente aprender dicho concepto. En este caso particular los alumnos sí que han realizado el mismo desarrollo con ambas soluciones y la comparativa es objetiva.

Respecto a la última pregunta, en el caso particular del desarrollo de EJB con la especificación 2.1 o 3.0, parece que existe una tendencia generalizada a admitir que el uso de atributos les ha facilitado el desarrollo de la misma, pese a lo novedoso de la solución.

Pregunta \ Puntuación	1	2	3	4	5
Es necesario dedicar alguna hora de teoría a explicar el concepto de anotaciones en Java y su uso en esta asignatura	1	4	2	2	0
Es conveniente aprender el concepto de programación orientada a atributos usando anotaciones	1	2	3	3	0
	Muy fácil	Fácil	Igual	Difícil	Muy difícil
Consideras que las prácticas con anotaciones son:	0	6	2	1	0

Tabla 5. Evaluación docente en sistemas distribuidos

6. Conclusiones y Líneas de Trabajo Futura

Aunque se han mostrado conclusiones particulares sobre cada una de las experiencias, se da a continuación una visión global de nuestra primera experiencia con la inclusión de la programación orientada a atributos, durante el presente curso.

Se puede establecer la necesidad inicial de incluir en los temarios de programación orientada a objetos de primeros cursos, alguna sesión de teoría y práctica al concepto de atributo. La simple utilización del concepto, deja a los alumnos con una cierta indecisión y dudas al respecto de su uso y funcionamiento. Para apoyar esta idea se cree adecuado coordinar la inclusión del concepto en asignaturas de ingeniería del software.

También, a la vista de los resultados, parece que el realizar prácticas comparativas entre soluciones orientadas y no orientadas a atributos acaba aclarando el papel concreto de dicho paradigma para el alumnado.

En este punto, puede ser importante el resaltar la importancia de su uso práctico en la actualidad, y su posible entronque con el concepto de la

programación orientada a aspectos en futuros trabajos.

Esto permitiría en posteriores cursos donde se utiliza dicho paradigma, el poder focalizar sobre la semántica particular que cada conjunto de atributos recoge, y las particularidades dentro de cada framework o plataforma de desarrollo.

En próximos cursos se podrá comprobar el efecto de haber introducido el paradigma en asignaturas previas, permitiendo agilizar el proceso de aprendizaje o profundizar sobre cuestiones más avanzadas.

Como reto futuro, parece obvio el comenzar a introducir en un modo similar el concepto de aspecto, y su relación con el paradigma de programación orientada a objetos y a atributos. A partir de estos primeros resultados, se puede empezar a prever los problemas y soluciones que habrá que plantear y resolver ante la introducción de nuevos paradigmas.

Referencias

[1] Meyer, B. Construcción de Software Orientado a Objetos 2ª Edición. Prentice Hall, 1998.

[2] Pawlak, R. Spoon: Annotation –Driven Program Transformation – The AOP Case. 1st International Middleware Workshop on Aspect Oriented Middleware Development (AOMD) ser. AICPS. Vol. 118.Grenoble, France ACM, Nov. 2005. pp. 1—6

[3] Wada, H. and Suzuki, J. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In 8th International Conference on Model Driven Engineering Language and Systems (MoDELS), ser. LNCS, vol. 3713. Montego Bay, Jamaica: Springer, Oct. 2005, pp. 584—600

[4] Eichberg, M., Schäfer, T., and Mezini, M. Using Annotations oto Check Structural Properties of Classes. In 8th International Conference on Fundamental Approaches to Software Engineering (FASE). ser. LNCS, nº 3442. Edinburgh UK: Springer, Apr. 2005, pp. 237—252.

[5] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes, 31(3):1-38, March 2006.

[6] Walls, C. & Richards, N. XDoclet in Action. Manning Publications. 2003.

[7] OMG Object Management Group. Unified Modeling Language: Superstructure version 2.0. Revised Final Adopted Specification (ptc/04-10-02) October 8, 2004. Disponible en http://www.uml.org/.

[8] Myers, G.J. El Arte de Probar el Software. El Atenero, 1984.

[9] Binder, R.V. Testing Object-Oriented Systems. Models, patterns, and tools. Addison Wesley, 2000.

[10] Link, J. et al. Unit Testing in Java. How Tests Drive the Code. Morgan Kaufmann, 2003.

[11] McLaughlin, B. & Flanagan D. (2004). Java 1.5 Tiger: A Developer´s Notebok. Ed O'REILLY.

[12] JUnit. Testing Resources for Extreme Programming. http://www.junit.org. Última visita 26 de enero de 2004.

[13] Gamma, E. Helm, R., Jonhson R., y Vlissides, J. Patrones de Diseño. Addison Wesley, 2003.

[14] WOP Web Of Patterns. Ontology documents homepage. http://www-ist.massey.ac.nz/wop/

[15] Dong, J. (2002) UML Extensions for Design Pattern Compositions. Journal of Object Technology, vol.1, nº5, Noviembre-Diciembre de 2002, páginas 151-163. Disponible en http://www.jot.fm/issues/issue_2002_11/article3

[16] Coulouris, G. , Dollimore, J. y Kindberg, T. *Sistemas Distribuidos. Conceptos y Diseño.* Ed. Addison-Wesley. 3ªEdición, 2001.

[17] Liu, M.L. *Computación Distribuida. Fundamentos y Aplicaciones.* Ed. Addison-Wesley, 2004.

[18] Tanenbaum, A. & Van Steen, M. *Distributed Systems.* 1st Edition. Prentice-Hall. 2002.

[19] DeMichiel, L.G. Enterprise JavaBeansTM Specification, Version 2.1. Sun Microsystems. FCS. November, 2003.

[20] DeMichiel, L.G. & Keith, M. JSR 220: Enterprise JavaBeansTM,Version 3.0. EJB Core Contracts and Requirements. Sun Microsystems. May, 2006.