Aspect J v/s Javassist

Javier Bustos Jiménez

Departamento de Ciencias de la Computación (DCC)
Universidad de Chile.
jbustos@dcc.uchile.cl

1. Introducción

Como "separation of concerns" se conoce la idea que es posible trabajar en el diseño o implementación de un sistema en el sentido natural de las unidades de negocio: concepto, meta, estructuras de equipos, etc. Se desea que la modularidad de un sistema refleje "lo que pensamos de él" y no lo que el lenguaje o herramientas de desarrollo quieren que pensemos sobre él.

En otras palabras, se desea que todas las preocupaciones transversales de un sistema sean abstraídas del diseño original y tratadas de un modo especial, como por ejemplo: patrones de acceso a la memoria, sincronización de procesos concurrentes, manejo de errores, etc. Para ello se puede utilizan técnicas como Programación Orientada a Aspectos (AOP), Reflexión (aunque la primera es subconjunto de la segunda) y otras no abarcadas en este estudio.

La definición formal dice: "*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 de programa*" (G.Kiczales).

En este informe, se explicarán las nociones básicas de una herramienta AOP: **AspectJ** [1] y se comparará con otra herramienta reflexiva: **Javassist**. Finalmente, se entregará una comparación experimental entre ambas herramientas utilizando como métricas el tiempo de compilación, carga y ejecución.

2. AspectJ

Los lenguajes orientados a aspectos definen una nueva unidad de programación para encapsular las preocupaciones transversales. Aun así, es claro que existe una estrecha relación entre los componentes y los aspectos, y por lo tanto, el código de los componentes y de estas nuevas unidades de programación tienen que interactuar de alguna manera. Para que ambos se puedan mezclar, deben existir puntos comunes entre ellos (conocidos como `crosspoints`) y algún modo de mezclarlos.

Los `crosspoints` son una interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que a éste se puede adicionar comportamiento. Estos comportamientos se especifican en los aspectos.

El encargado de realizar este proceso de mezcla se conoce como *weaver*. Él 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 o `crosspoints` (figura 1).

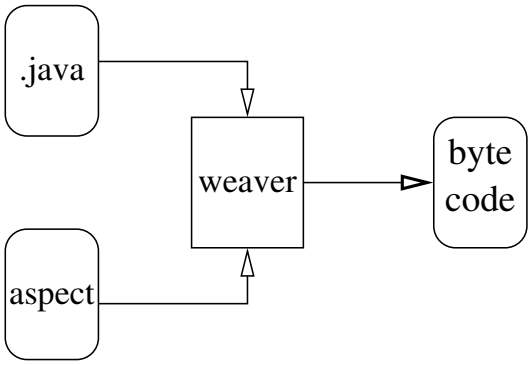

Figura 1. Tejedor de aspectos.

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 (métodos, asignaciones de variables). La forma de realizar ese entrelazado puede ser estático (modificando el código fuente de una clase) o dinámico (en tiempo de ejecución).

AspectJ [1] es una extensión a Java orientada a aspectos diseñado para ser utilizado con cualquier clase de aspecto (distribución, coordinación, manejo de errores). La extensión se realiza mediante nuevos módulos llamados **aspecto**.

En AspectJ, un aspecto es una clase, similares a las clases Java en lo que se refiere a reutilización de métodos (o advices en el caso de Aspect J) y uso estático de recursos, pero con la particularidad que puede contener constructores de corte (o **pointcut**) que no existen en Java. Estos cortes capturan colecciones de eventos en la ejecución de un programa, no definen acciones sino que describen dichos eventos.

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

- `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. Un `pointcut` está formado por dos partes separadas por dos puntos, su nombre y contexto (a la izquierda) y la definición de los eventos (a la derecha).

- `introduction`: Utilizados para introducir elementos completamente nuevos a las clases dadas, como por ejemplo: métodos, constructores, atributos, y varios de los anteriores a la vez.

- `advice`: Definen partes de la implementación del aspectos que se ejecutan en los `pointcuts` definidos. El cuerpo del aviso puede añadir distintos puntos del código mediante una palabra clave: `before`, `after`, `catch` (al recibir una excepción), `finally` (justo al terminar la ejecución, aún arrojando una excepción) y `around` (atrapa la ejecución de los métodos designados por el evento).

AspectJ permite una separación de las preocupaciones transversales de una forma limpia y poderosa, sin embargo no es conveniente utilizarla si se necesita mayor reflexión de clases habiendo herramientas más poderosas para ello (como Javassist).

3. Javassist

Javaassist (Java Programming Assistant) es un sistema de carga dinámica de clases Java mediante reflección. Las clases se cargan en tiempo de carga (figura 2), valga la redundancia, lo que permite crear nuevas clases o modificar nuestras propias clases (incluso llamadas internas a otros métodos) en tiempo real.

El sistema está diseñado para que las clases se puedan modificar de manera sencilla incluso sin conocer los mecanismos internos de carga de clases de Java. Esta facilidad de uso es un aspecto muy importante y que le da ventaja sobre otras soluciones. Por ejemplo:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("test.Rectangle");
cc.setSuperclass(pool.get("test.Point"));
pool.writeFile("test.Rectangle");
```

Con este trozo de código podríamos hacer que nuestra clase `test.Rectangle` pase a tener como clase padre a `test.Point`. Todo eso en tiempo real.

Javaassist no permite cambiar las clases del sistema en tiempo real ya que eso sólo lo puede hacer el System ClassLoader (sino sería un gravísimo fallo de seguridad) pero lo que si que permite es crear nuevas clases.

Javassist provee una API para realizar reificación, reflección utilizando los métodos de la clase `CtClass`; para realizar instrospección utilizando los métodos de las clases `CtField` y `CtMethod` (comparables a la API tradicional de Java, pero sin la posibilidad de invocar los métodos). También la API provee de herramientas para realizar

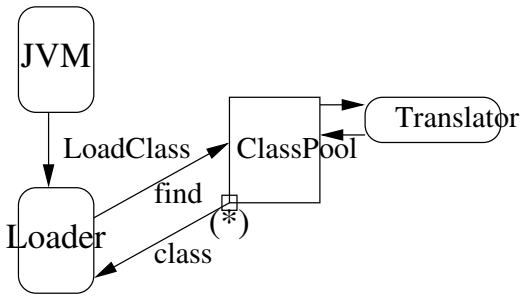

Figura 2. Javassist: en el punto marcado con (*) se puede notificar a un *Translator* que realiza la reflexión.

alteración del `bytecode` y para agregar nuevos miembros a una clase (`addMethod`, `addWrapper`); para más información sobre el uso de estas herramientas refierase a [2].

Para AOP, javassist puede ser una buena herramienta introduciendo nuevos métodos en las clases y para insertar avisos *before/after/around* tanto en los objetos que producen mensajes como en los que los reciben.

4. Comparación Teórica

Para la separación de preocupaciones transversales, Javassist provee de herramientas que permiten modificar el código fuente de cualquier clase de manera fácil usando los métodos de la metaclase `CtMethod` llamados `insertBefore` e `insertAfter`, equivalentes la utilización de `before` y `after` en `advices` de AspectJ.

Tanto Javassist como AspectJ pueden introspectar mensajes enviados y recibidos, y verificar el flujo de control de la ejecución. Sin embargo, dada la generalidad de Javassist (porque su interés es la reflección estructural, no sólo la separación de preocupaciones transversales) lo hace menos utilizable que AspectJ en ese campo, puesto que con este último se tiene total independencia entre lo que es código base y lo que es preocupación transversal. El gran problema que se tiene con javassist es que al agregar un nuevo método/campo/clase uno no tiene conocimiento de la correctitud de dicho código hasta el tiempo de ejecución. No así Aspect J, que conoce dicha correctitud en tiempo de compilación.

Javassist permite solamente una orientación débil a separación de aspectos (no como AspectJ). A excepción de un programa principal de Java (un programa del base-nivel en la terminología de la reflexión), otros aspectos (programas del metanivel) son los programas que describen cómo se altera el código de Java transversal. Aunque esos aspectos se escriben en Java regular, se requiere que los programadores explícitamente tengan un punto de vista del metanivel para escribir esos aspectos. Por otra parte, AspectJ permite que los programadores escriban programas del aspecto en una manera declarativa

con una sintaxis mejor. No es necesario un punto de vista explícito del metanivel.

Una buena característica de Javassist es que permite control fino de cómo se tejen los aspectos, porque la descripción de aspectos con Javassist no es declarativa (pero sí procesal). Además, Javassist proporciona una gran capacidad de introspección, que es casi igual a la capacidad de la reflexión API de Java.

Si un aspecto pide al compilador tejer un cierto código en un punto solamente si el programa destino satisface algunas condiciones, por ejemplo, sólo si una clase hereda de otra cierta clase específica. Entonces Javassist sí podría hacerlo pero AspectJ no [3].

5. Comparación Experimental

Para la realización de este experimento se usó un programa sencillo de figuras en el cual cada vez que se realizaba un llamado al mensaje `move(int, int)` se debía almacenar esa información para tener la posibilidad de realizar `undo` en un futuro cercano.

Se probaron tres estrategias de solución, programando normalmente en Java (sin tomar en cuenta que eso es una preocupación transversal) el método en la clase `Shape`, utilizando la reflexión en tiempo de carga de *Javassist* para agregar dicha actualización y, por último, separar definitivamente la preocupación transversal utilizando *Aspect J*.

Cuadro 1. Tiempo de compilación (en segundos)

Clase	Normal	Javassist	Aspect J
Shape	0.820 (javac)	0.890 (javac)	1.270 (ajc)
Box	0.850 (javac)	0.900 (javac)	0.890 (javac)
Circle	0.860 (javac)	0.900 (javac)	0.900 (javac)
Triangle	0.880 (javac)	0.890 (javac)	0.840 (javac)

Se midió el tiempo de compilación de cada estrategia (cuadro 1), para mostrar si las diferencias que se planteaban en la teoría ocurrían también en la práctica.

Además, se midió el tiempo que tomaba la creación de 1 a 10 objetos (del tipo `Circle`, `Box` o `Triangle`, que heredan de `Shape`): el objetivo de dicho experimento es notar las diferencias en tiempo de carga de las distintas estrategias (figura 3). Finalmente se midió el tiempo (en milisegundos) que tomó a cada estrategia realizar entre 10.000 y 100.000 llamados a mensajes `move(int,int)` divididos en 10 objetos (figura 4).

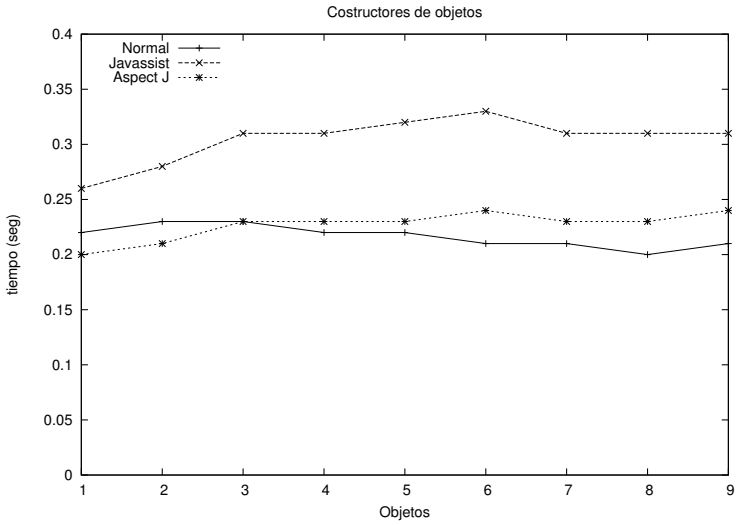

Figura 3. Creación de objetos.

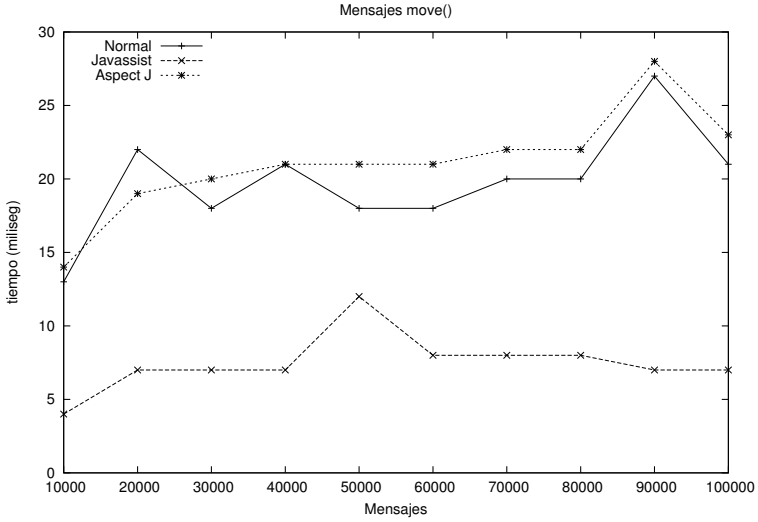

Figura 4. Llamados al mensaje `move(int,int)`.

6. Conclusiones

Como era de esperarse, el tiempo de "compilación + tejido" de *Aspect J* se hizo notorio en la experimentación (cuadro 1), al igual que el tiempo extra de carga de clases que posee *Javassist* (figura 3).

También se puede notar que no hay mayor diferencia en tiempos de ejecución entre una programación sin preocupaciones transversales y una totalmente separada como *Aspect J* (figura 4). Sin embargo, se puede notar claramente la optimización que realiza *javassist* al momento de ejecutar y realizar los saltos al nivel meta, siendo incluso mejor que una programación Java normal.

Lo anterior se debe a la simplicidad del ejemplo y a que se realiza el salto a nivel meta sólo para un tipo de mensaje (`move(int, int)`), para ejemplos de mayor complicación no se espera un comportamiento así teóricamente, sin embargo no deja de ser un caso curioso y propuesto para futuras investigaciones.

Referencias

1. Kiczales Gregor, et al.: "An Overview of Aspect J" In: Proc. of ECOOP 2001.
2. Shigeru Chiba: "Load-time Structural Reflection in Java" In: Proc. of ECOOP 2000. pp 313-336.
3. Shigeru Chiba: "What are the best join points?" In: Workshop. of OOPSLA 2001.