Join Points / Point Cuts

Decisiones de diseño

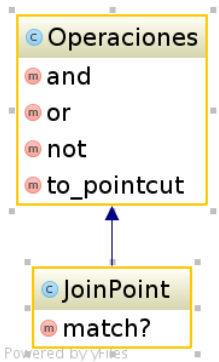
- Una clase para cada tipo: clase **JoinPoint** y clase **Pointcut.** Esto permite la separación de responsabilidades y también permitirá polimorfismo (logrado tanto por herencia simple como por redefinición).
- La clase **JoinPoint** será como una clase abstracta para que sus subclases redefinan los métodos necesarios y que luego cada una implemente su propio comportamiento **[*1]**. Brinda extensibilidad. Evita duplicación de código.
- "Conversión implícita de tipos" **[*2]**: cualquier operación entre Join Points da como resultado un nuevo objeto instancia de la clase **Pointcut** (luego, las operaciones entre Point Cuts, van a resultar en otro nuevo Point Cut). Esto hace que exista polimorfismo dado por herencia simple: las clases **JoinPoint** y **Pointcut** comparten el comportamiento de las 3 operaciones (and, or, not) el cual estará en la clase **Operaciones**, logrando que la lógica en común exista en un único lugar (mantenibilidad, extensibilidad).
- "Evaluación diferida" **[*3]**: las operaciones entre dichos objetos no se realizan de manera instantánea en el momento de la invocación de su método correspondiente, sino que la última operación invocada "es guardada" en un nuevo objeto creado ("objeto resultante") **[*4]**. Gracias a ello, más tarde podrá evaluarse el resultado de una solo operación, o de N operaciones consecutivas, en el momento que el programador lo indique. Esto permite expresiones "en serie" del tipo (jp1 & (~jp2)) | pc2.
- "Cadena de operaciones" **[*5]**: todos los objetos que la componen entienden el mismo mensaje (llamando *match*), el cual resuelve todas las operaciones que existan en esa cadena (partiendo desde un "nodo raíz" del tipo **Pointcut**).
- Los objetos **Joinpoint** hacen intersección, unión y complemento entre arrays de clases (arrays de objetos del tipo :Class). Esto se implementó con un hash que mapea cada símbolo de la operación (el cual se obtiene por el nombre del selector del método de la operación) al símbolo del método correspondiente en la clase Joinpoint. Ese hash es fácil de leer, fácil de modificar y fácil de extender y es conocido por todos los objetos Pointcut **[*6]**.

Observaciones

[*1]: El patrón de diseño que más se le parece es el *Strategy*, pero no es exactamente lo mismo.
[*2]: Esto es análogo a lo que hace la mayoría de los lenguajes de programación cuando se suma un número entero con un número real, el resultado es un nuevo número real (es decir que el resultado será del tipo más general).
[*3]: Casi como si fuese *Lazy Evaluation*.
[*4]: A su vez dicho objeto puede realizar las mismas operaciones con otros objetos polimórficos, volviendo a guardarse los detalles específicos de dicha operación, de esa manera se forma una "cadena de operaciones".
[*5]: La consecuencia es que existirá una estructura con forma de árbol binario, donde los nodos terminales serán objetos de la clase **JoinPoint** y los nodos no-terminales serán objetos de la clase **Pointcut**.
[*6]: Si uno quisiera agregar una nueva operación, habría que definir su método en la clase **Operaciones** y luego agregar un elemento al hash de **Pointcut** para mapear el símbolo de la 'operación' (clase Operaciones) al símbolo de 'resolución' (clase Pointcut) correspondiente.

Join Points

Diagrama de Clases

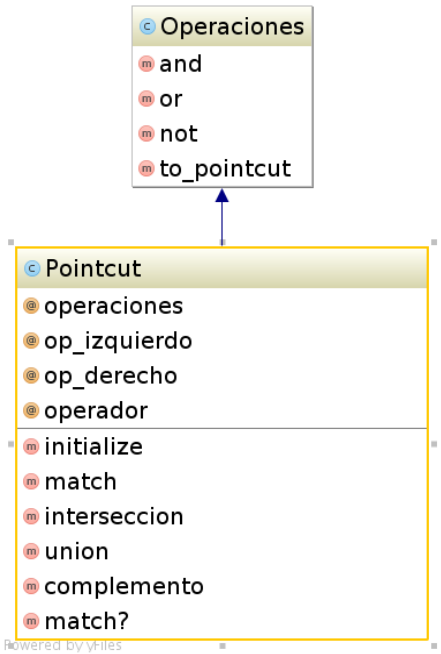


Point Cuts

La clase **Pointcut** se ocupa de hacer las operaciones de intersección, unión y complemento entre clases conocidas por cada pointcut, a través de un array de clases que puede ser provisto al pointcut en cualquier momento (con el setter correspondiente).

Las operaciones de and, or y not (compartidas por las clases JoinPoint y Pointcut, necesario para polimorfismo) no se realizan en el momento de las llamadas a sus métodos, sino que se van "guardando" y manteniendo el orden con el que fueron creadas (para luego poder evaluar todo el conjunto de operaciones obteniendo un resultado en forma diferida).

Diagrama de Clases

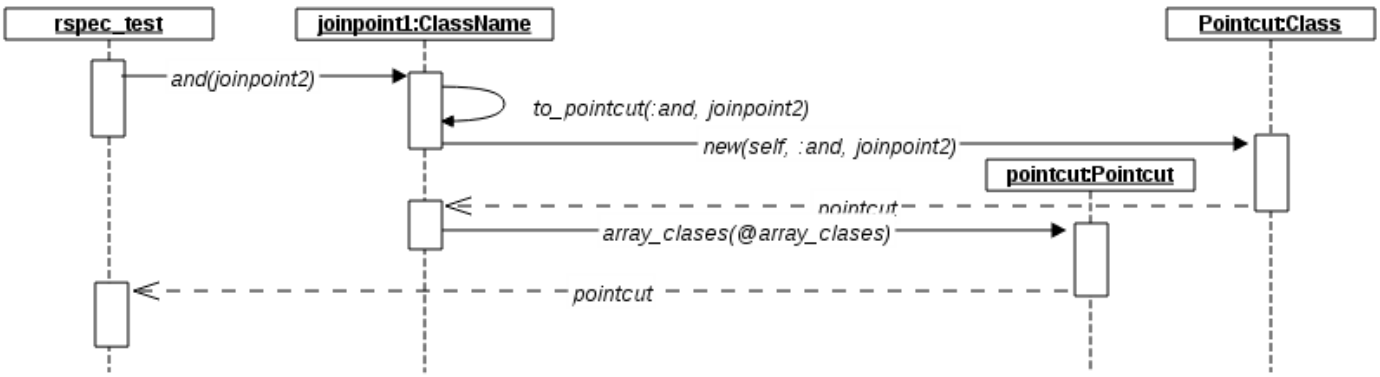


Ejemplos

Ejemplo 1: Operación *and* entre dos objetos subtipos de **JoinPoint** (devuelve un **Pointcut**)

- joinpoint1.and(joinpoint2): Hace que el receptor del mensaje "and" ejecute un método que creará un nuevo **Pointcut** pasándole 3 parámetros:
    - A si mismo (objeto en joinpoint1)
    - El símbolo del selector del mensaje (en este caso es :and)
    - El operando (objeto en joinpoint2).

Diagrama de Secuencia



(las demás operaciones tendrán el mismo diagrama de secuencia, pero modificando los parámetros de los mensajes que sean necesarios modificar para poder describir la operación correspondiente)

Ejemplo 2: Varias operaciones posibles (and, or, not) polimórficas entre objetos subtipo de **JoinPoint** y objetos del tipo **Pointcut**, cuyos resultados también pueden hacer las mismas operaciones.

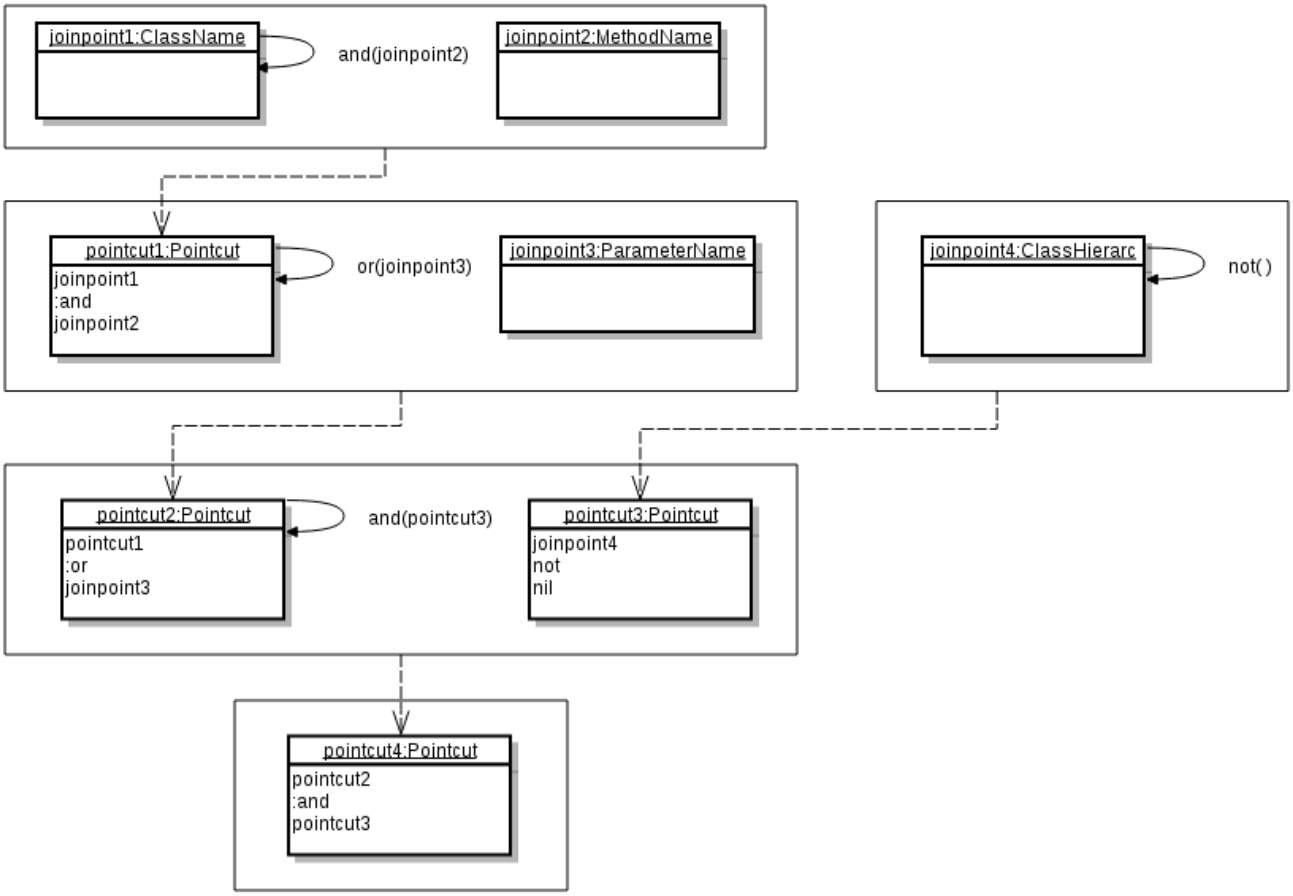
El siguiente ejemplo muestra cómo se realizan las operaciones “compuestas” usando algunos objetos **JoinPoint** y algunos objetos **Pointcut**, usando las tres operaciones posibles (and, or, not) entre todos ellos.

Responde al siguiente snippet de código:

```
pointcut1 = joinpoint1.and(joinpoint2)  # and entre 2 join points
pointcut2 = pointcut1.or(joinpoint3)    # or entre 1 point cut y 1 join point
pointcut3 = joinpoint4.not( )           # not de 1 join point
pointcut4 = pointcut2.and(pointcut3)    # and entre 2 point cuts -> “nodo raíz”
```

Diagrama de Objetos

(adaptado, no se garantiza que sea "UML compatible")



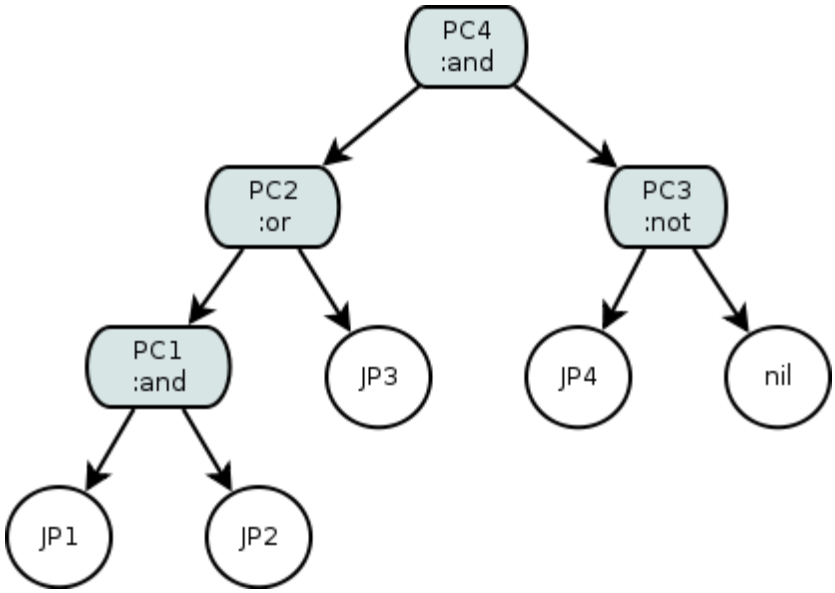
Podría resumirse con esta expresión similar (notar que en el siguiente ejemplo solo se opera entre join points y no mezclado entre point cuts y join points como en el snippet anterior):

((joinpoint1 & joinpoint2) | joinpoint3) & (~joinpoint4)

Estructura de datos generada: Árbol binario

Si se hace un seguimiento de los atributos “operando_izquierdo” y “operando_derecho” de cada Pointcut, empezando desde el pointcut “más externo” (en el ejemplo es el pointcut4), entonces el pointcut4 vendría a ser el nodo raíz de un árbol binario, y cada “nodo” que sea Pointcut va a tener referencias a otros 2 nodos (cualquiera de ellos puede ser del tipo **Pointcut** o de un subtipo del tipo **JoinPoint**).

Esta estructura correspondería al snippet de código antes mencionado, siendo la 1er linea de código para el PC1, la 2da línea para el PC2, la 3er línea para el PC3 y la 4ta línea para el PC4 y a su vez, la referencia guardada en PC4 sería equivalente a esta expresión: **((jp1 & jp2) | jp3) & (~jp4)**



(Diagrama no standard: es solo para explicar este árbol)