Programação Orientada aos Aspectos com *AspectJ*

Rui Carlos A. Gonçalves

29 de Agosto de 2008

Introdução

Durante os últimos anos assistimos a uma grande divulgação da programação orientada aos objectos (POO). Esta veio responder a requisitos essenciais no desenvolvimento de *software*, nomeadamente no que diz respeito às facilidades de manutenção e reutilização do código.

Mas apesar de todas estas evoluções, nem tudo aquilo que a POO se propunha conseguir foi alcançado. É neste contexto que surge a programação orientada aos aspectos (POA).

Este novo paradigma de programação surgiu em 1996 por Greger Kiczales e pela sua equipa no Xerox PARC, que também foram responsáveis pelo desenvolvimento do AspectJ, a linguagem POA mais usada.

Quando vamos desenvolver uma aplicação, é conveniente dividir o problema que nos foi proposto em partes. A separação do problema de acordo com os dados e as funcionalidades a eles associadas, facilita o estudo dos requisitos da aplicação. Isto é o que se designa por *separação de interesses*.

O paradigma OO estabelece uma separação de interesses tendo em conta os tipos de dados, representados pelos objectos, e as funções que utilizam cada tipo de dados, ou seja, os métodos a que um objecto responde.

Mas, como já foi referido anteriormente, apesar da orientação aos objectos ter trazido melhorias na forma de programar, não conseguiu resolver alguns problemas. Exemplos clássicos são o registos das operações em ficheiros de *log* que encontramos em muitos *softwares*, ou programação concorrente/distribuída. O código associado a este tipo de funcionalidades encontra-se normalmente espalhado por todos os módulos, o que dificulta a manutenção e a evolução do código. Em POA dizemos que isto são *interesses entrecortantes*, pois *cortam* transversalmente todos os módulos. A POA vem resolver este problema introduzindo um novo nível de separação de interesses, os *aspectos*.

Basicamente estes interesses transversais à aplicação são encapsulados nestas novas unidades modulares e posteriormente fundidas com as classes num único sistema. Isto não só aumenta a facilidade de manutenção destas funcionalidades da aplicação, visto que tudo se encontra num único local, como aumenta as possibilidades de reutilização das classes noutros contextos, pois estas encontram-se *limpas* destas funcionalidades que normalmente variam muito de caso para caso.

O AspectJ

O AspectJ (`http://www.eclipse.org/aspectj`) foi a primeira linguagem OA desenvolvida e é, provavelmente, a mais usada. Esta linguagem permite acrescentar aspectos a programas feitos em Java.

Para compreender o AspectJ é necessário conhecer três conceitos fundamentais presentes na POA:

advices são fragmentos de código com as acções referentes aos interesses entrecortantes;

join points são locais de um programa onde os *advices* serão executados, que podem ser a chamada/execução de um método, o acesso a membros de uma classe, a criação de um objecto, o lançamento de excepções, etc.;

pointcuts são um conjunto de *join points*, definidos segundo um determinado critério, que identificam os locais onde um determinado *advice* será executado.

Pointcuts

Genericamente, a definição de um *pointcut* obedece à seguinte sintaxe:

```
pointcut <nome>() :  <tipo>(<padrão>);
```

Onde:

- <nome> é o identificador deste *pointcut*;
- <tipo> indica a que tipo de *join point* nos referimos (chamada de uma função, acesso a um atributo, etc.);
- <padrão> é uma expressão que restringe os *join points* a serem considerados àqueles que fazem *matching* com esta expressão.

De seguida indicam-se os tipos de *join points* mais importantes:

- `call` - chamada de uma função;
- `execution` - execução de uma função;
- `get` - consulta dos atributos de um objecto;
- `set` - alteração dos atributos de um objecto;
- `within` - *join points* que ocorrem dentro de determinadas classes;
- `target` - envio de mensagens a um objecto de um determinado tipo;
- `args` - envio de mensagens em que os argumentos são de um determinado tipo.

A expressão pode ser uma palavra (nome de uma classe, nome de um método, etc.), mas também temos caracteres especiais:

- `*` - representa uma sequência de caracteres qualquer, desde que não contenha pontos;
- `..` - representa uma sequência de caracteres qualquer, mesmo contendo pontos;
- `+` - representa as subclasses de uma dada classe.

Vejamos agora alguns exemplos:

- ```
  pointcut tostr() : execution(String toString());
  ```

  Representa todas as execuções do método `toString`. Podemos restringir apenas à execução do método a objectos da classe `Xpto` substituindo `String toString()` por `String Xpto.toString()`, ou a objectos da classe `Xpto` e todas as suas subclasses fazendo `String Xpto+.toString()`;

- ```
  pointcut sets() : call(* Xpto*.set*(..));
  ```

  Representa a chamada de métodos começados pela palavra `set` de classes começadas pela palavra `Xpto`, com um número qualquer de argumentos. Podemos limitar apenas a métodos que recebam argumentos de um determinada tipo, por exemplo, fazendo `* Xpto*.set*(int,String)` limitaríamos apenas aos métodos cujo o primeiro argumento fosse um inteiro e o segundo uma *string* (para além de verificar todas as outras condições já indicadas).

- ```
  pointcut gets() : get(private String abc*);
  ```

  Representa todos os acessos (para consulta) a atributos *private* do tipo `String`, começados por `abc`.

Para definirmos *pointcuts* também temos à nossa disposição operadores lógicos (`!`, `||` e `&&`, cujo significado é o habitual).

```
pointcut gets() : get(* *) && within(Xpto);
```

Este *pointcut* representa um acesso a qualquer atributo, desde que seja da classe `Xpto`.

Agora vamos ver como utilizar o `target` e o `args` para restringir o tipo dos objectos a que o método é enviado, assim como o tipo dos argumentos com os quais o método foi invocado.

```
pointcut p1(Xpto x,int y,float z) : execution(* *(..))
                                  && target(x)
                                  && args(i,j);
```

Desta forma restringimos os *join points* aos métodos enviados a um objecto do tipo `Xpto` e que recebem dois argumentos (o primeiro do tipo `int` e o segundo do tipo `float`). É claro que isto poderia ter sido conseguido com a expressão `* Xpto.*(int,float)`, mas, como veremos de seguida, esta opção permitirá usar o objecto ao qual foi enviado o método, assim como os argumentos recebidos na especificação de *advices*.
```

Advices

Se os *pointcuts* indicavam o conjunto de pontos de uma aplicação onde queríamos realizar outras operações, os *advices* permitem-nos definir quais são essas operações.

Temos 3 (ou 5) tipos de *advices*. Usamos o:

- `before`, quando queremos executar acções antes de um *join point*;
- `after`, quando queremos executar acções depois de um *join point*, existindo três alternativas:
  - `after()`, que é sempre executado;
  - `after() returning`, que apenas é executado quando o método associado ao *join point* retorna;
  - `after() throwing`, que apenas é executado quando o método associado ao *join point* lança uma excepção.
- `around`, quando queremos executar acções em vez de um método (sendo que dentro do *advice* podemos executar o método original).

Vamos agora analisar alguns exemplos concretos. Para tal vamos considerar as seguintes classes Java:

```
1  public class Classe1 {
2    private int x;
3    private boolean a;
4
5    public Classe1(int x,boolean a) {
6      this.x = x;
7      this.a = a;
8    }
9
10   public void setX(int x) {this.x=x;}
11
12   public void setA(boolean a) {this.a=a;}
13
14   public String toString() {
15     return ("x=" + this.x + " / a=" + this.a);
16   }
17 }
```

```
1  public class Classe2 {
2    private int y;
3
4    public Classe2(int y) {
5      this.y=y;
6    }
7
8    public void setY(int y) {this.y=y;}
9
10   public String toString() {
11     return ("y=" + this.y);
12   }
13 }
```

O primeiro problema que vamos resolver consiste em alterar os métodos `toString()`. Vamos alterá-los de forma a que passem a criar uma *string* que contenha também o nome da classe. Tal pode ser conseguido do seguinte modo:

```
1  pointcut tostr(Object obj) : execution(String toString())
2                            && target(obj);
3
4  String around(Object o) : tostr(o) {
5    String s = proceed(o);
6    return ("<" + o.getClass().getName() + " @ " + s + ">");
7  }
```

Primeiro definimos o *pointcut* que, para além de restringir os métodos à execução do `toString()`, também *capta* o objecto ao qual o método é enviado através do `target`. Desta forma foi possível utilizar o objecto no *advice*, como se pode ver nas linhas 5 e 6. Foi também usado `proceed`, que executa o método original. Depois é só acrescentar o nome da classe e devolver o resultado.

No próximo exemplo vamos ver como usar o `args`, que é semelhante ao `target`, mas este *capta* os argumentos do método (ou construtor, como será o caso). Este aspecto irá imprimir uma mensagem sempre que um objecto de uma classe começada por `Classe` for inicializado através de um construtor que receba um argumento do tipo inteiro.

```
1  pointcut init(int x) : initialization(new(..)) && within(Classe*) && args(x);
2
3  after(int x) : init(x) {
4    System.out.println("Classe: " +
5      thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
6    System.out.println("Argumentos: " + x);
7  }
```

Neste exemplo foi também usada a variável `thisJoinPointStaticPart`, que contém uma série de informações sobre o *join point* em que o aspecto vai ser executado. Existe também a variável `thisJoinPoint`, que poderia ser usada neste caso para saber quais os argumentos recebidos. Tal será feito no próximo exemplo.

Suponhamos agora que queríamos registar todas as alterações realizadas em qualquer uma das classes. Para alterar o valor dos atributos é necessário recorrer aos métodos `set*`. Logo vamos criar um *pointcut* que agrupe todas as execuções destes métodos. Também vamos querer registar qual o objecto afectado, quais os argumentos usados no método (sendo que o seu tipo nem sempre é o mesmo) e qual o nome do método.

```
1  pointcut sets(Object obj) : call(* Classe*.set*(..)) && target(obj);
2
3  before(Object obj) : sets(obj) {
4    try {
5      BufferedWriter bw = new BufferedWriter(new FileWriter("history.txt",true));
6
7      Date d=(Calendar.getInstance()).getTime();
8      bw.write("Data: " + d + "\nObjecto: " + obj + "\nMetodo: "
9              + thisJoinPointStaticPart.getSignature().getName()
```

```
10              + "\nArgumentos:\n");
11      Object[] args = thisJoinPoint.getArgs();
12      String[] ids = ((CodeSignature)thisJoinPoint.getSignature())
13                      .getParameterNames();
14      for (int i = 0; i < args.length; i++) {
15        bw.write("  " + ids[i] + "=" + args[i] + "\n");
16      }
17      bw.close();
18    } catch(Exception e){System.out.println(e);}
19  }
```

Aqui já recorremos à variável `thisJoinPoint` para obter os argumentos usados no método, assim como os identificadores dos argumentos. O resultado obtido será algo como:

```
1  Data: Sat Dec 09 17:56:30 WET 2006
2  Objecto: <Classe1 @ x=12 / a=true>
3  Metodo: setX
4  Argumentos:
5    x=10
```

Conclusão

Ao longo deste artigo ficaram visíveis alguns dos benefícios que podemos obter com a POA. Contudo há alguns factos que é necessário realçar. Em primeiro, que uma abordagem dos problemas segundo esta metodologia, é bastante mais complicada do que a abordagem OO. É difícil definir a estrutura do *software*, identificando correctamente todas as partes que o compõem. Mesmo depois de definida a estrutura, cometemos facilmente erros na definição dos aspectos (acontece várias vezes definir aspectos que originam ciclos infinitos, pois ao executá-los criávamos situações onde outros aspectos poderiam ser executados), que por vezes não são muito visíveis. Sublinha-se ainda que não é objectivo da POA substituir a POO, tratam-se sim de dois paradigmas complementares.

Por último referia-se que este texto é apenas uma introdução à POA/AspectJ e, como tal, muitas das capacidades deste novo paradigma não foram aqui demonstradas. Sugere-se a quem quiser aprofundar os seus conhecimentos nesta área, a consulta da página *web* do *AspectJ*, onde poderá encontrar uma vasta documentação sobre este tema.

Referências

[1] `http://www.eclipse.org/aspectj`

[2] `http://en.wikipedia.org/wiki/Aspect-oriented_programming`

[3] `http://www.research.ibm.com/hyperspace`

[4] `http://www.devx.com/Java/Article/28422`