Programação Orientada a Aspectos

O paradigma da orientação a objetos trouxe grandes avanços para o processo de desenvolvimento de software como um todo, permitindo a construção de sistemas mais fáceis de serem projetados, mantidos, evoluídos e reusados. Porém, existem certas propriedades que se pretende implementar em um sistema que não são adequadamente encapsuladas em classes, seja porque se aplicam a diversas classes de um sistema simultaneamente, seja porque não pertencem à natureza intrínseca da classe a qual se aplicam. O resultado é que as classes onde tais propriedades precisam ser implementadas, além de lidar com as responsabilidades para as quais foram projetadas, acabam acumulando responsabilidades adicionais, o que vai contra o princípio da coerência, um dos pilares do paradigma orientado a objetos.

A orientação a aspectos provê meios para que propriedades relativas a preocupações que se aplicam a diversas partes do sistema simultaneamente sejam adequadamente implementadas em separado, preservando a coesão e reduzindo a complexidade. Com a separação destas propriedades em "Aspectos" torna-se possível modularizar o código referente a cada preocupação adequadamente, e então compor o software resultante através de um processo de combinação.

Os aspectos consistem de características estruturais e/ou comportamentais que devam ser adicionadas a diversas partes de um sistema, mas que são projetadas e implementadas em separado. A orientação a aspectos possui mecanismos para que, quando o sistema estiver em execução, as características implementadas pelos aspectos estejam adequadamente combinadas com aquelas fornecidas por cada módulo afetado, de modo que o sistema resultante apresente todas as propriedades pretendidas, mas sem comprometimento da coerência de cada módulo.

É muito importante salientar que a programação orientada a aspectos surgiu como uma complementação das demais técnicas de programação existentes no momento, principalmente a Orientação a Objetos. Assim como a AOP existem outras técnicas que surgem com propósitos muito semelhantes e que também possuem o mesmo caráter.

As preocupações que não são relativas a classe em questão, são conhecidas como preocupações ortogonais. São ortogonais não somente porque são inerentes a classe, mas também por serem preocupações de cunho secundário à esta, assim como podem ser ainda para outras classes do software. Os exemplos mais comuns de tais responsabilidades ortogonais são aquelas referentes a requisitos não-funcionais, tais como a persistência, o paralelismo, a concorrência, a segurança, a tolerância a falhas, entre outras.

Os maiores problemas causados por estas preocupações ortogonais estão no fato de que ela é responsável por duas sérias conseqüências para o desenvolvimento de softwares: a primeira é conhecido como "Code Tangling" ou emaranhamento de código que é justamente o fato de se misturar código referente a classe com código referente ao sistema. Outra conseqüência não grata ao desenvolvimento é conhecido como "Code Scattering"ou espalhamento de código que é o fato de tratarmos enumeras vezes em casos diferentes o mesmo problema.

Todas estas características influem negativamente na construção do software causando diversos problemas como os que vemos a seguir:

- Fraco Mapeamento – como existem características cujas responsabilidades não pertencem a classe, as características da classe acabam ficando obscurecidas. Em suma, fica difícil interpretar o propósito de construção da referida classe;
- Baixa Produtividade – a implementação simultânea de vários interesses faz com que o desenvolvedor mude seu foca dos objetivos primários da classe para os objetivos periféricos a ela;
- Menos Reuso de Código – como o módulo implementa vários conceitos diferentes, outros sistemas que requeiram funcionalidades similares podem não serem capaz de lidar com as funções projetadas no módulo em questão, descartando-se assim a possibilidade de reuso;
- Código com Qualidade Inferior – o emaranhamento de código pode causar problemas que não são facilmente vistos pelo desenvolvedor. Há ainda o problema de se tratar vários interesses simultaneamente pode fazer com que um certo interesse não receba toda a atenção que lhe é devida;
- Maior Dificuldade na Evolução – sempre que tentarmos fazer reuso do módulo em questão haveremos de ter que adaptá-lo ao nosso novo uso, gerando um grande e dispendioso re-trabalho.

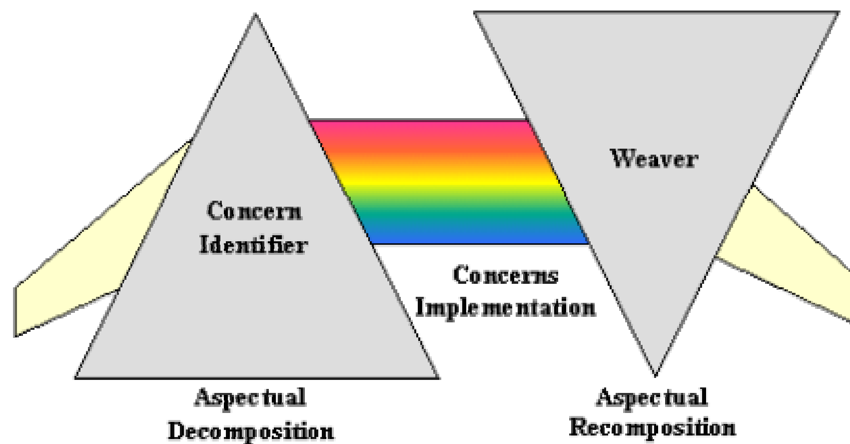
Fundamentos da Orientação a Aspectos

A Programação Orientada a Aspectos envolve três passos distintos na sua implementação:

1 – Decomposição em Aspectos – decompõe os requisitos para identificar os interesses ortogonais e os crosscutings;

2 – Implementação dos Interesses - implementa cada interesse separadamente;

3 – Recomposição do Aspecto - Nesta etapa, um integrador do aspecto especifica regras para a recomposição criando unidades de modularização – os aspectos. O processo de recomposição, também conhecido como Weaving, usa esta informação para compor o sistema final.



A figura ilustra os requisitos sendo filtrados pelo "Identificador de Interesses" durante a "Decomposição dos Aspectos". A parte colorida representa a "Implementação dos Interesses". E finalmente o "Weaver" é responsável pela "Recomposição de Aspectos".

Composição de um Sistema Baseado em Aspectos

Um sistema baseado em aspectos é composto de:

i) uma linguagem de componentes;
ii) uma (ou mais) linguagem(ns) de aspectos;
iii) um programa de componentes;
iv) um ou mais programas de aspectos;
v) um combinador de aspectos, que é capaz de combinar os programas de componentes e de aspectos de forma a gerar o programa final.

A linguagem de componentes normalmente é uma linguagem que conforma-se a um dos paradigmas dominantes, com Java, Smalltalk, C++, Pascal, C, Lisp, etc...

Algumas linguagens de aspectos são genéricas quanto à sua aplicação, como AspectJ e AspectC. Outras são voltadas a preocupações específicas, e nesses casos devem ser usadas múltiplas linguagens de aspectos, uma para cada preocupação.

O código do programa de componentes deve ser escrito tendo-se em mente que as responsabilidades ortogonais devam ser deixadas para os aspectos. Dessa forma, os benefícios da orientação a aspectos ficam mais claros.

Os programas de aspectos, que podem estar implementados em múltiplas linguagens de aspectos, implementam as responsabilidades ortogonais. O código do aspecto torna explícito o comportamento que será integrado ao código dos componentes, e em que contextos tal integração ocorre: são os chamados *pontos de junção*, que são elementos semânticos da linguagem de componentes com as quais os programas de aspectos se coordenam. Exemplos de pontos de junção comuns são: invocações de métodos (chamadas ou recebimentos), geração de exceções, criação de objetos, entre outros.

O combinador de aspectos identifica nos componentes pontos de junção onde os aspectos se aplicam, produzindo assim o código final da aplicação, que implementa tanto as propriedades definidas pelos componentes como aquelas definidas pelos aspectos. Combinadores podem atuar em tempo de compilação ou de execução.

Problemas da composição de aspectos

Existe um problema intrínseco da orientação a aspectos, relacionado à interação entre dois ou mais aspectos diferentes que se aplicam a um mesmo ponto de junção. São identificadas algumas diferentes categorias de problemas de composição de aspectos, das quais duas foram selecionadas:

- Problemas de Compatibilidade – os aspectos que se compõem podem ser incompatíveis. Por exemplo, os aspectos implementam propriedades relacionadas, o que pode gerar redundância (os aspectos implementam a mesma funcionalidade desnecessariamente) ou inconsistência (a funcionalidade de um aspecto invalida a do outro);
- Problemas de Ordenamento – os aspectos implementam propriedades que têm restrições temporais entre si.

A problemática da composição de aspectos é maximizada quando se considera um ambiente onde diversos aspectos de diferentes fornecedores são combinados em uma aplicação. Idealmente, deve existir um mecanismo que identifique e proíba a combinação de

aspectos incompatíveis, e que possibilite a definição de regras de precedência entre aspectos que apresentam restrições temporais entre si.

A Linguagem AspectJ

AspectJ é uma extensão da linguagem JAVA que dá ao programador suporte ao uso da orientação a aspectos. Ela implementa os requisitos necessários para uma perfeita implementação de aspectos e seu compilador é capaz ainda de compilar código Java puro. Um aspecto, como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, através da definição de aspectos especializados.

Aspectos podem alterar a estrutura estática de um sistema adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia do sistema, e convertendo uma exceção checada por uma não checada (exceção de *runtime*). Esta característica de alterar a estrutura estática de um programa é chamada *static crosscutting*.

Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos no fluxo de execução, chamados *join points*, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução. É possível definir um *join point* como resultado da composição de vários *join points*.

Normalmente um aspecto define *pointcuts*, os quais selecionam *join points* e valores nestes *join points* e *advices* que definem o comportamento a ser tomado ao alcançar os *join points* definidos pelo *pointcut*.

Joinpoints, Pointcuts, Advices...

Joinpoint – define em quais pontos do programa o código do aspecto será combinado com o código da aplicação. O joinpoint está em um ponto na execução do programa que ao ser "alcançado", dado o cenário ao qual ele foi determinado, executa um código específico pré-programado pelo desenvolvedor;

Pointcut – é uma construção de linguagem para qual migram um ou mais joinpoint baseados em critérios definidos anteriormente. Os pointcuts são responsáveis pela captura do joinpoint para que então seja feita a avaliação do código a ser executado neste ponto específico. Podem ser composto através dos operadores &&(e), ||(ou), e ! (não);

Advices – são trechos de código (similares a métodos) que estão associados aos pointcuts. O código definido nos advices é executado no momento da execução do joinpoint.

Joinpoint

Pointcut

Advice

```
package pacote;

aspect UmAspecto {

pointcut invocacoes() : call(* *..*(..));

 before (): invocacoes() {
  System.out.println("Invocando método.");
 }
}
```

Em suma, o advice é o código que é executado a cada joinpoint capturado por um pointcut.

Para definir pointcuts, identificando os joinpoints a serem afetados, utilizamos construtores de AspectJ chamados designadores de pointcut (pointcut designators), como os apresentados a seguir:

Call (X)	Invocação de método identificado por X
Execution (X)	Execução de método identificado por X
Get (X)	Acesso a atributo identificado por X
Set (X)	Atribuição de atributo identificado por X
This (Y)	O objeto em execução é a instância de Y
Target (Y)	O objeto de destino é instância de Y
Args (Y, ...)	Os argumentos são instâncias de Y
Within	O código em execução está definido em Y
Class (Y)	Quando o código executado pertence a classe Y

Um advices é o código executado durante um joinpoint. Os advices de AspectJ são apresentados a seguir:

Before	Executa quando o joinpoint é alcançado, mas antes de sua computação
After Returning	Executa após a computação **com** sucesso do joinpoint
After Throwing	Executa após a computação **sem** sucesso do joinpoint
After	Executa após a computação do joinpoint, em qualquer situação
Around	Executa quando o joinpoint é alcançado e tem total controle sobre a sua computação

Implementação de TAD usando Java e AspectJ

Proposta para TAD

Os tipos abstratos de dados (TAD) que foram desenvolvidos são: lista (simplesmente encadeada e duplamente encadeada), pilha, fila e deque. Estes três últimos foram desenvolvidos utilizando uma lista duplamente encadeada (DE) para armazenar os dados por eles manipulados. Essas implementações foram realizadas com a linguagem Java. Como esta apresentado na figura 1, a lista simplesmente encadeada (SlistImpl.java na figura 1a) compõe-se de nós simples (SNode.java na figura 1b e SNodeImpl.java na figura 1c). E a lista duplamente encadeada (DlistImpl.java na figura 1d) compõe-se de nós duplos (DNode.java na figura 1e e DNodeImpl.java na figura 1f).

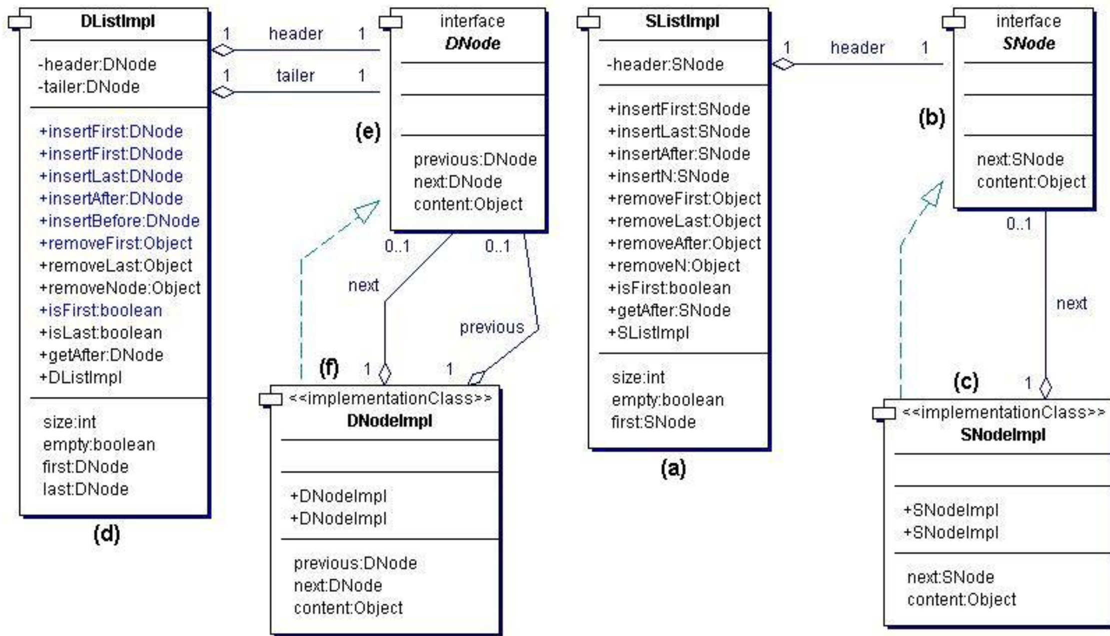


Figura 1: Diagrama das implementações das Listas

A implementação do TAD deque segue apresentada na figura 2, assim como os TAD queue na figura 3 e o TAD stack na figura 4.

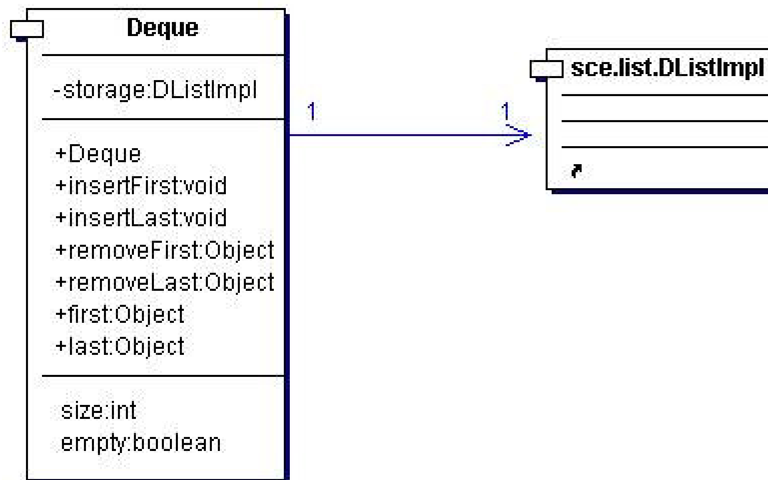


Figura 2: Diagrama do TAD Deque

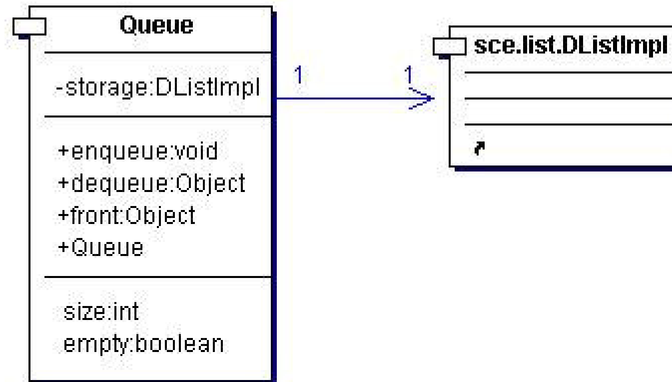


Figura 3: Diagrama do TAD Queue

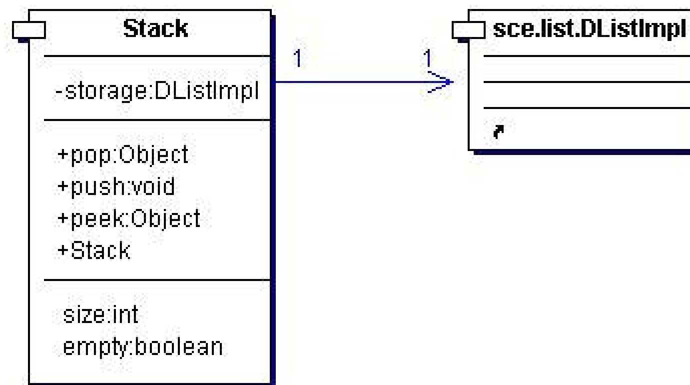


Figura 4: Diagrama do TAD Stack

Proposta para ordenação da fila usando o AspectJ

Para realizar a ordenação da fila alguns aspectos foram implementados através da ferramenta AspectJ. A ordenação implementada funciona com o maior valor na frente da fila e o menor ao final. Então, com o intuito de manter a genericidade da fila, foi implementado um aspecto (código 1) que adiciona um atributo inteiro no nó (linha 12 do código 1), o qual será utilizado para a ordenação. Além do atributo ele implementa também os métodos de manipulação dele (linha 14 e 15 do código 1) e também publica esses métodos na interface, fazendo a interface DNode estender a interface declarada nesse aspecto (linha 5-10 do código 1).

```
1 package sce.list;
2
3 public aspect NodeOrderAspect {
4
5 public interface DNodeOrderInterface{
6 public int getOrder();
7 public void setOrder(int newOrder);
```

```
8 }
9
10 declare parents: DNode extends DNodeOrderInterface;
11
12 private int DNodeImpl.order = 0;
13
14 public int DNodeImpl.getOrder(){return this.order;}
15 public void DNodeImpl.setOrder(int newOrder){this.order=newOrder;}
16
17 }
```

Código 1: Aspecto de alteração do nó

No código 2, o aspecto de ordem da lista DE foi definido. Neste aspecto foi definidos um pointcut denominado ordem e um advice para esse pointcut. Esse pointcut acontece sempre que encontra o método insertLast que recebe um Object como parâmetro e que retorna um DNode da classe DListImpl. Para esse pointcut, o advice around desconsidera o código original e utiliza o método insertLast que aceita o parâmetro de ordem junto com o Object, fixando a ordem para todos em 0. Da linha 12 até 33 do código 2 está o código que realiza a inserção em ordem na lista.

```
1 package sce.list;
2
3 public aspect OrderingDListImplAspect {
4
5 pointcut ordem():execution(DNode DListImpl.insertLast(Object));
6
7 DNode around() : ordem() {
8 Object[] argm = thisJoinPoint.getArgs();
9 return ((DListImpl)thisJoinPoint.getThis()).insertLast(argm[0],0);
10 }
11
12 public DNode DListImpl.insertLast(Object parObject,int order) {
13 int position = 0;
14 DNodeImpl toInsert=null;
15 DNode navigator=null;
16 if(!this.isEmpty()){
17 navigator=this.header;
18 }
19 toInsert = new DNodeImpl(parObject,null,null);
20 toInsert.setOrder(order);
21 if(this.isEmpty()){
22 this.insertFirst(toInsert);
23 }else if((order > navigator.getNext().getOrder())){
24 toInsert.setNext(navigator.getNext());
25 toInsert.setPrevious(navigator);
26 navigator.getNext().setPrevious(toInsert);
27 navigator.setNext(toInsert);
28 this.size++;
```

```
29 }else{
30 ...
31 }
32 return(toInsert);
33 }
34 }
```

Código 2: Aspecto de ordem na lista

E no código 3, foi definido o aspecto para a fila poder inserir ordenadamente, utilizando o método definido na linha 15 até 18 do código, através do pointcut adiciona, o qual encontrado sempre que executado o método enqueue da classe Queue. E também o advice around é definido para realizar a utilização do método recém inserido.

```
1 package sce.queue;
2
3 import sce.list.DNode;
4
5 public aspect OrderQueueAspect {
6
7 pointcut adiciona():execution(void Queue.enqueue(Object,int));
8
9 DNode around() : adiciona() {
10 Object[] argm = thisJoinPoint.getArgs();
11 int pos = ((Integer)argm[1]).intValue();
12 return ((Queue)thisJoinPoint.getThis()).enqueue(argm[0],pos);
13 }
14
15 public DNode Queue.enqueue(Object parObject,int order) {
16
17 return this.storage.insertLast(parObject,order);
18 }
19
20 }
```

Código 3: Aspecto de ordem da fila

Conclusão

A programação orientada a aspectos dá ao desenvolvedor diversas ferramentas para o aumento da modularidade dos programas. O objetivo da técnica é a separação das preocupações ortogonais (crosscutting concerns) de modo a evitar o entrelaçamento de código com o código pertencente a classe. Evita ainda o espalhamento de código, não especificando diversas vezes o mesmo código em partes diferentes do sistema.

Desta forma obtemos ganhos com manutenção e evolução do sistema além de favorecer o reuso de suas partes. Com a separação dos códigos, inerente a classe, aumenta-se a legibilidade deste, tornando a sua interpretação mais fácil.

Já linguagem AspectJ estende perfeitamente a orientação a aspectos na linguagem Java, permitindo assim a possibilidade de usar-se de tais artifícios no desenvolvimento.

Entre as desvantagens da linguagem estão a necessidade de se familiarizar com as novas construções de AspectJ.

Existe ainda um problema intrínseco da orientação a aspectos, relacionado à interação entre dois ou mais aspectos diferentes que se aplicam a um mesmo ponto de junção. Existem algumas diferentes categorias de problemas na composição de aspectosentre elas:

- problemas de compatibilidade – os aspectos que se compõem podem ser incompatíveis. Por exemplo, os aspectos implementam propriedades relacionadas, o que pode gerar redundância (os aspectos implementam a mesma funcionalidade desnecessariamente) ou inconsistência (a funcionalidade de um aspecto invalida a do outro);
- problemas de ordenamento – os aspectos implementam propriedades que têm restrições temporais entre si. A problemática da composição de aspectos é maximizada quando se considera um ambiente onde diversos aspectos de diferentes fornecedores são combinados em uma aplicação. Idealmente, deve existir um mecanismo que identifique e proíba a combinação de aspectos incompatíveis, e que possibilite a definição de regras de precedência entre aspectos que apresentam restrições temporais entre si.

Porém estes pequenos empecilhos não denigrem a qualidade que os aspectos dão aos códigos de desenvolvimento. É seguro acreditar que os méritos da programação a aspectos são suficientes para estabelecê-la como ferramenta auxiliar no desenvolvimento de softwares orientados a objetos, maximizando a abstração dos dados, a modularidade do programa e um maior reuso dos componentes.

Referências

http://www.eclipse.org/aspectj
http://www.aosd.net
http://java.sun.com
http://www.javaworld.com
Soares, Sérgio & Borba, Paulo – AspectJ - Programação Orientada a Aspectos em Java
Chaves, Rafael Alves - Aspectos e Middleware
Chaves, Rafael Alves - Aplicando o Desenvolvimento de Software Orientado a Aspectos às aplicações paralelas e distribuídas com AspectJ