Aspect Oriented Programming
Aspect J

By Gian Ricardo Berkenbrock
& Eduardo Dockhorn da Costa

Roteiro

- Problema;
- AOP;
- Aspect J;
- Proposta ao Problema;
- Conclusões;
- Referências.

Problema Proposto

- Desenvolver os tipos abstratos de dados: lista, fila, pilha e deque. Estes devem ser implementados de forma genérica.
- Desenvolver aspecto de fila ordenada.

Porque surgiu?

- Na OOP existe grande dificuldade em se modelar certas decisões de projeto fazendo com que a implementação destas decisões fiquem distribuídas no programa, resultando em um espalhamento e entrelaçamento de código com propósitos diferentes;
- Isto torna o desenvolvimento e a manutenção do código tarefas extremante difíceis de serem realizadas;

O que é?

- A Programação Orientada a Aspectos (AOP) surge como uma complementação a OOP;
- AOP envolve 3 passos distintos na sua implementação:
  - Decomposição dos Aspectos → decompõe os requisitos para identificar os interesses ortogonais e os crosscutings;
  - Implementação dos Interesses → implementa cada interesse separadamente;
  - Recomposição do Aspecto →Nesta etapa, um integrador do aspecto especifica regras para a recomposição criando unidades de modularização – os aspectos.

O que faz?

- AOP consegue separar códigos de funções específicas que afetam diferentes partes do sistema, chamadas de preocupações ortogonais (crosscuting concern);
- Os Crosscuting Concern surgem da preocupação que possui o desenvolvedor em dar suporte a todos os requisitos necessários do software. Entre eles podemos citar: persistências, tratamento de exceções, controle de concorrência, depuração entre outras.

Explicando melhor…

- Ao desenvolver um sistema este é projetado como uma resposta a várias exigências. Estas podem ser classificadas como “requisitos do núcleo do módulo” e “requisitos do sistema”.
- Muitos requisitos do sistema tendem a ser mutuamente independente (ortogonais) entre si e aos requisitos do núcleo do módulo.
- Eles tendem também a fazer parte da preocupação de vários módulos (crosscut).
- As técnicas de implementação atuais forçam o desenvolvedor a usar metodologias unidimensionais, enquanto este problema é de complexidade n-dimensional, prejudicando assim a modularização.

Hã... E daí?!?!

- Em suma, como conseqüência da impossibilidade de modularização, normalmente tem-se como resultado um emaranhamento de código (*code tangling*), onde o código referente à responsabilidade principal da classe está misturado com o código referente as responsabilidades ortogonais desta. Existe ainda a dispersão de código já que pode-se ter em diversos módulos a mesma característica sendo tratada;
- Os aspectos permitem que definamos modificações no software (tanto no estado do software como no seu comportamento) que se aplicam a várias partes do sistema ortogonalmente fornecendo ainda a modularização necessárias para tais modificações.

A Linguagem Aspect J

- Extensão à linguagem Java que inclui suporte à Programação Orientada a Aspectos, proposta pelo *Palo Alto Research Center*, tradicional centro de pesquisas da Xerox;
- Implementa alguns poucos conceitos associados a aspectos. Seu compilador suporta ainda a linguagem Java tradicional;
- Um *aspecto* é um módulo de código-fonte AspectJ (assim como as classes são em Java) que inclui a definição de *pointcuts*, *advices* e *joinpoints*. Um aspecto também possui um nome e uma designação de pacote. Pode ainda conter atributos e métodos.

Joinpoint, Pointcuts, Advices...

- *Joinpoint* - define em quais pontos do programa o código do aspecto será combinado com o código da aplicação;
- *Pointcut* – é uma construção de linguagem para a qual migram um ou mais Joinpoints, baseado em critérios pré-definidos.
- *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*.

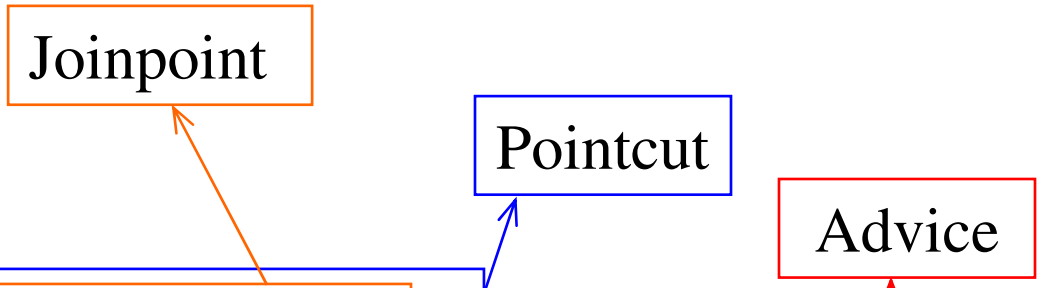
Ufa... Um Exemplo!!!

```
package pacote;

aspect UmAspecto {

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

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



RESUMINDO.....

O *advice* é o código que é executado a cada *join point* capturado por um *pointcut*.

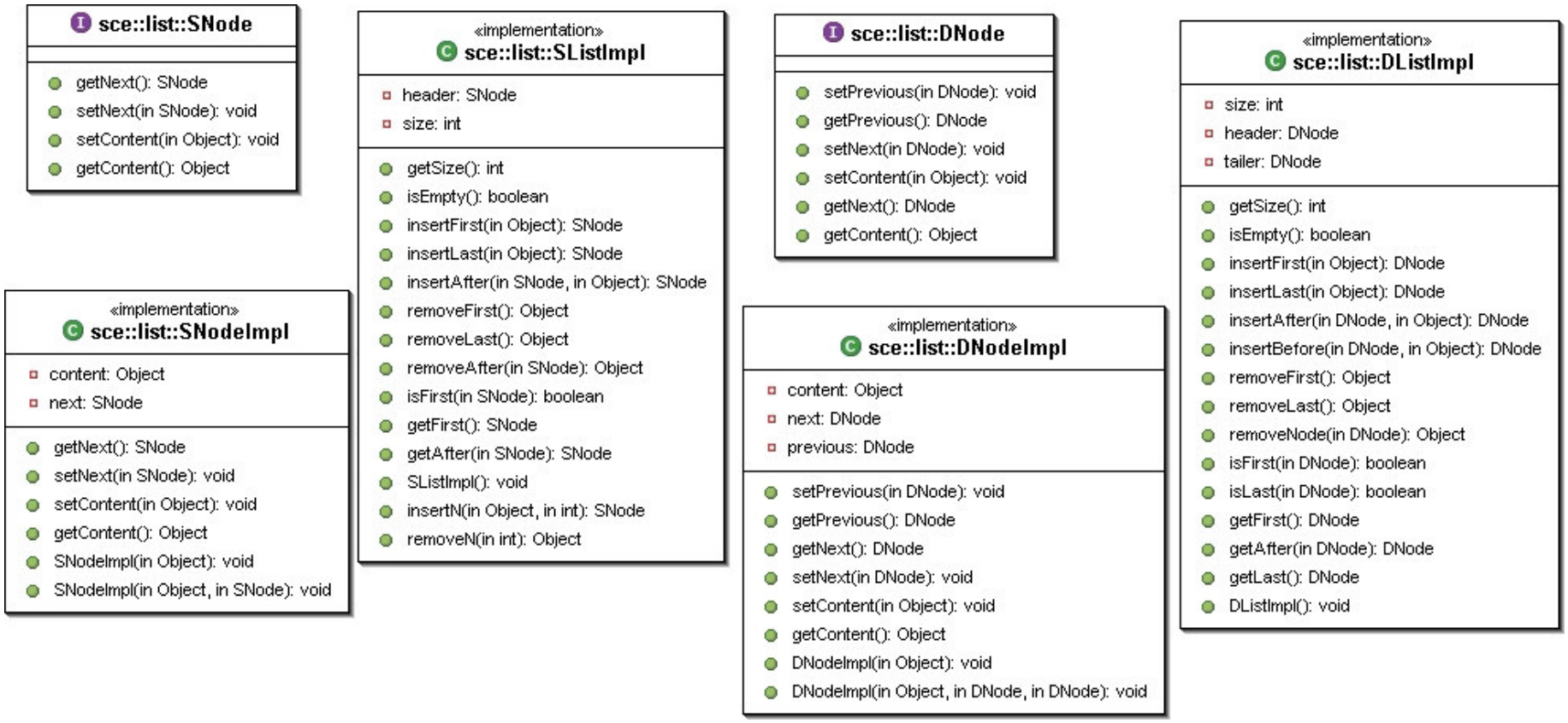
Designadores de Pointcut

- Quais são os eventos que podem ser capturados e como são descritos no AspecJ?
  - Execution – quando o corpo de um método particular executa;
  - Call – quando o método é chamado;
  - Handler – quando um manipulador de exceções é acionado;
  - This(X) – quando o objeto que esta em execução é do tipo ‘X’;
  - Target(X) – quando o objeto alvo da execução é do tipo ‘X’;
  - Class(Y) – quando o código executado pertence a classe Y;
  - CFlow(Test.Main()) – quando o joinpoint esta no fluxo de controle da chamada de procedimento principal da classe Test (sem argumentos);

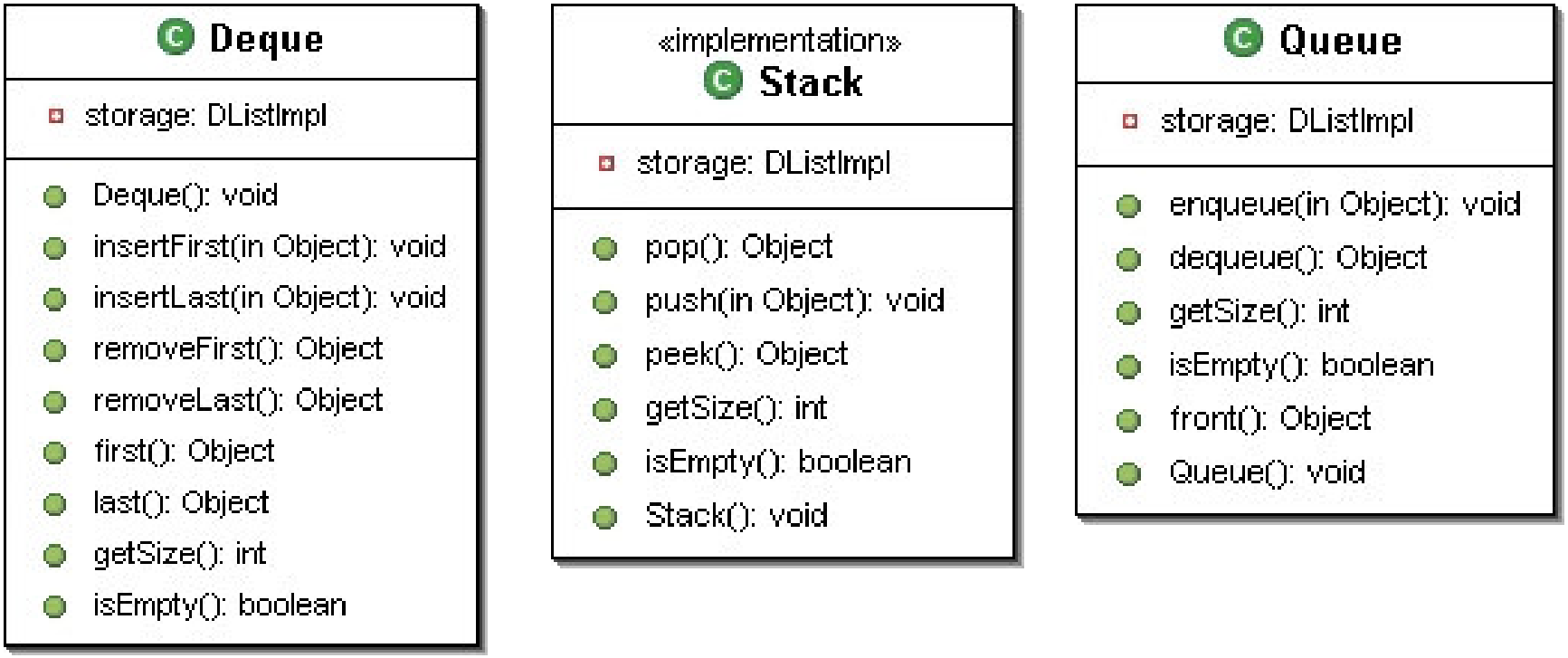
Advices

- Existem três tipos de Advices:
  - Before() → o trecho de código e executado antes do joinpoint;
  - After() → o trecho de código e executado depois do joinpoint;
  - Around() → o trecho de código e executado no momento da execução do joinpoint;
    - Proceed → Usado somente combinado com o Advice do tipo Around. Permite executar o código original dentro do escopo deste advice.

Containers – Lista



Containers – Deque, Stack & Queue



Comparação Entre Filas

```
package sce.queue;

public class App {

  public static void main(String[] args) {
    Queue fila = new Queue();
    fila.enqueue("Primeiro");
    fila.enqueue("Segundo");
    fila.enqueue("Terceiro");
    fila.enqueue("Quarto");
    fila.enqueue("Quinto");
    fila.enqueue("Sexto");
    fila.enqueue("Sétimo");
    fila.enqueue("Oitavo");
    fila.enqueue("Nono");
    fila.enqueue("Décimo");
    while(!fila.isEmpty()){
       System.out.println((String)fila.dequeue());
    }
  }
}
```

Fila sem Prioridade

```
package sce.queue;

public class OrderApp {

  public static void main(String[] args) {
    Queue fila = new Queue();
    fila.enqueue("Primeiro",0);
    fila.enqueue("Segundo",1);
    fila.enqueue("Terceiro",2);
    fila.enqueue("Quarto",3);
    fila.enqueue("Quinto",4);
    fila.enqueue("Sexto",4);
    fila.enqueue("Sétimo",4);
    fila.enqueue("Oitavo",7);
    fila.enqueue("Nono",8);
    fila.enqueue("Décimo",9);
    while(!fila.isEmpty()){
       System.out.println((String)fila.dequeue());
    }
  }
}
```

Fila com Prioridade

Resultado do Processamento

1. Primeiro
2. Segundo
3. Terceiro
4. Quarto
5. Quinto
6. Sexto
7. Sétimo
8. Oitavo
9. Nono
10. Décimo

Fila sem Prioridade

1. Décimo
2. Nono
3. Oitavo
4. Quinto
5. Sexto
6. Sétimo
7. Quarto
8. Terceiro
9. Segundo
10. Primeiro

Fila com Prioridade

NodeOrderAspect.java

```
package sce.list;

public aspect NodeOrderAspect {

  public interface DNodeOrderInterface{
    public int getOrder();
    public void setOrder(int newOrder);
}

  declare parents: DNode extends DNodeOrderInterface;

  private int DNodeImpl.order = 0;

  public int DNodeImpl.getOrder(){
    return this.order;
  }
  public void DNodeImpl.setOrder(int newOrder){
    this.order=newOrder;
  }
}
```

OrderingDListImplAspect.java

```
package sce.list;

public aspect OrderingDListImplAspect {

  pointcut ordem():execution(DNode DListImpl.insertLast(Object));

  DNode around() : ordem() {
    Object[] argm = thisJoinPoint.getArgs();
    return ((DListImpl)thisJoinPoint.getThis()).insertLast(argm[0],0);
  }

  public DNode DListImpl.insertLast(Object parObject,int order) {
    int position = 0;
    DNodeImpl toInsert=null;
    DNode navigator=null;
    if(!this.isEmpty()){
      navigator=this.header;
    }
    toInsert = new DNodeImpl(parObject,null,null);
    toInsert.setOrder(order);
...
```

Conclusões

- Os principais benefícios:
  - clareza de código;
  - facilidade de manutenção;
  - maior flexibilidade de configuração;
  - maior possibilidade de reuso.

Referências

- http://www.eclipse.org/aspectj
- http://www.aosd.net
- Soares, Sérgio & Borba, Paulo – AspectJ - Programação Orientada a Aspectos em Java
- http://java.sun.com
- http://www.javaworld.com