Programação Orientada a Aspectos Abordando Java e aspectJ

Vicente Goetten de Souza Junior

Diogo Vinícius Winck (Orientador)

Caio de Andrade Penteado e Machado (Colaborador)

União de Tecnologia e Escolas de Santa Catarina (UTESC)
Rua Visconde de Taunay, 166 – 89.201-420 – Joinville – SC – Brazil

vicente@decisao.inf.br

diogo@utesc.br

caioapmachado@terra.com.br

Abstract. *Object Oriented Programming can be defined as the paradigm that improves in the abstraction of data from the systems, as well as its modularization, the latter being increasingly desirable in current systems. Despite all the advantages that the guided programming of the objects offers, the increasing complexity of the systems presupposes several implementations that would make systems really modular, making the separation of the referring code to the business of the cross-cut concerns possible, e.g., the treatment of exceptions, log systems and competition. This article proposes the application of java and the language of aspects in a totally free environment.*

Resumo. *A programação Orientada a Objetos define-se como o paradigma que possibilita uma melhora na abstração de dados nos sistemas, bem como a sua modularização, aspecto cada vez mais desejável nos sitemas atuais. A despeito de todas as vantagens já oferecidas pela programação orientada a objetos, a crescente complexidade dos sistemas criou a necessidade de implementações que os tornassem realmente modulares, possibilitando a separação do código referente ao negócio dos interesses ortogonais, e.g., o tratamento de exceções, sistema de log e concorrência. Este artigo propõe a utilização da linguagem de aplicação java e a linguagem de aspectos aspectJ, em um ambiente totalmente gratuito.*

1. Programação Orientada a Aspectos

A programação orientada a aspectos (AOP – *Aspect-Oriented Programming*) foi criada há 7 anos, em Palo Alto, nos laboratórios da Xerox (Kiczales, 1997). A programação AOP não trabalha isoladamente, ao contrário, é um paradigma que extende outros paradigmas de programação, como a programação orientada a objetos e a estruturada, por exemplo.

Segundo Lopes (1997), "as aplicações estão ampliando os limites das técnicas de programação atuais, de modo que certas características de um sistema afetam seu comportamento de tal forma que as técnicas atuais não conseguem capturar essas propriedades de forma satisfatória." Na concepção de Soares e Borba (2002), "a programação orientada a aspectos propõe não apenas uma decomposição funcional, mas também ortogonal do problema. AOP permite que a implementação de um sistema seja separada em requisitos funcionais e não–funcionais."

As linguagens de programação atuais como Java, por exemplo, fornecem condições para representação abstrata dos elementos em um sistema como funções, métodos, classes, etc. Mas como tratar os aspectos que não são inerentes ao negócio no sistema? Como tratamento de exceções, log do sistema e concorrência? Somente utilizando a linguagem OO Java, não seria possível abstrair estes elementos. Segundo Becker e Geihs (1998), existem problemas de programação que as técnicas de programação orientada a objetos ou de programação estruturada não são suficientes para separar claramente todas as decisões de projeto que o programa deve implementar. Portanto, interesses (*concerns*), não pertencem ao domínio de negócio modelado por OOP, podendo se referir a requisitos não funcionais. Os interesses também podem ser definidos como aspectos.

Segundo Piveta (2001), "Quando duas propriedades sendo programadas devem ser compostas de maneira diferente e ainda se coordenarem é dito que elas são ortogonais entre si", o que são conhecidos como interesses ortogonais (*crosscut-concerns*). Ao tentar-se implantar estes interesses ortogonais pelo sistema, obtem-se código emaranhado, *Tangled code* (Kiczales, 1997). A figura 1 ilustra a implementação dos interesses ortogonais.

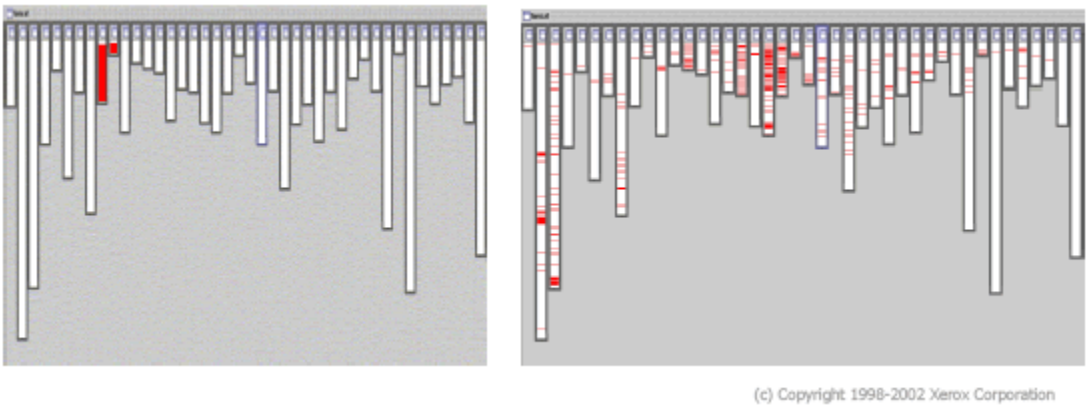

Figura1 – Implementação de Interesses ortogonais gerando código emaranhado.

Pode-se afirmar, que, o principal objetivo da programação orientada a aspectos é separar o código referente ao negócio do sistema, dos interesses não inerentes ao negócio, de uma forma centralizada, como exemplifica a figura 2.

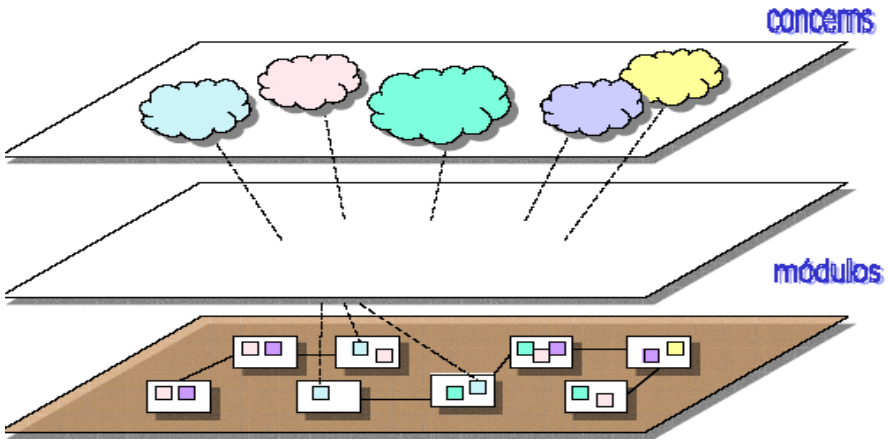

Figura 2 – Separação e centralização dos interesses ortogonais no sistema

2. Composição do sistema orientado a aspectos

Um sistema que utiliza a programação orientada a aspectos é composto pelos seguintes componentes:

- *Linguagem de componentes.* Segundo Irwin (1997), "a linguagem de componentes deve permitir ao programador escrever programas que implementem as funcionalidades básicas do sistema, ao mesmo tempo em que não prevêem nada a respeito do que deve ser implementado na linguagem de aspectos."

- *Linguagem de aspectos.* A linguagem de aspectos deve suportar a implementação das propriedades desejadas de forma clara e concisa, fornecendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem (IRWIN et al.,1997).

- *Combinador de aspectos.* A tarefa do combinador de aspectos (*aspect weaver*) é combinar os programas escritos em linguagem de componentes com os escritos em linguagem de aspectos. Neste artigo será utilizada a linguagem Java como linguagem de componentes e o *aspectJ* como linguagem de aspectos.

- *Programas escritos em linguagem de componentes.*

- *Um ou mais programas escritos em linguagem de aspectos.*

Na figura 3, é exemplificada a composição de um sistema utilizando o paradigma da programação orientada a aspectos.

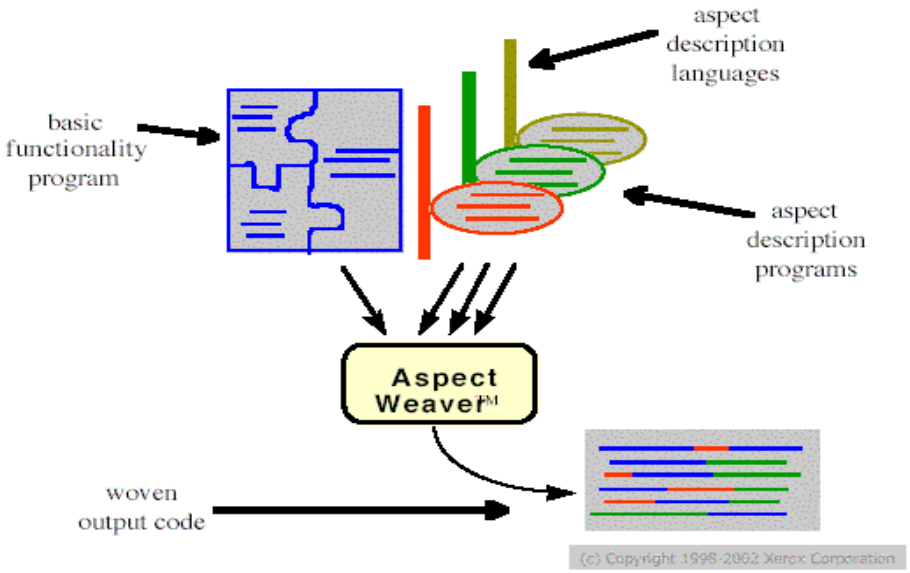

Figura 3 – Composição dos sistemas utilizando programação orientada a aspectos

3. Combinadores (weavers)

As classes referentes ao código do negócio nos sistemas não sofrem qualquer alteração para suportar a programação orientada a aspectos, isto é feito no momento da combinação entre os programas escritos em linguagem de componentes e os programas escritos em linguagem de aspectos, este processo pode-se definir como recompilação aspectual.

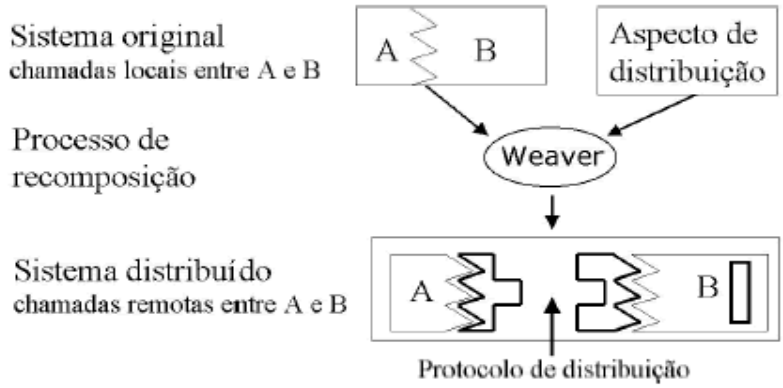

Figura 4 – Combinador aspectual (*weaver*)

3.1 Combinação estática

Um sistema orientado a aspectos utilizando combinação estática pode trazer agilidade ao mesmo, já que não há necessidade que os aspectos existam em tempo de compilação e execução.

O uso de uma combinação estática previne que um nível adicional de abstração cause um impacto negativo na performance do sistema (BÖLLERT, 1998a).

3.2 Combinação dinâmica

Segundo Piveta (2001), “Para que a combinação possa ser dinâmica é indispensável que os aspectos existam tanto em tempo de compilação quanto em tempo de execução. Através de uma interface reflexiva, o combinador de aspectos tem a possibilidade de adicionar, adaptar e remover aspectos em tempo de execução.”

4. Extendendo o Java a Programação Orientada a Aspectos – *AspectJ*

Atualmente existem diversas opções para utilização da programação orientada a aspectos, este artigo abordará a utilização do *aspectJ* como linguagem de aspectos, a linguagem Java como linguagem de componentes e a plataforma Eclipse como ambiente de desenvolvimento. Ambos, são distribuídos gratuitamente e estão disponíveis em diversas plataformas. Segundo Kiczales (2001), "O *AspectJ* é uma extensão simples e prática da AOP para Java (Kiczales, 2001)".

4.1 Exemplo de aplicação com *aspectJ*

O primeiro passo é a criação do programa propriamente dito, feita segundo o modelo tradicional de orientação a objeto, com a criação das classes, com seus respectivos métodos e atributos, e sem necessidade de qualquer preocupação com a orientação a aspectos. O exemplo conterá o código a seguir (Figura 5).

```
public class Principal
{
  public static void main(String[] args)
  {
        Principal hello = new Principal();
        hello.Mensagem();
        System.exit(0);
  }

  public void Mensagem()
  {
        System.out.println("Olá mundo !");
  }
}
```

Figura 5 - Um programa simples

Este programa instancia um objeto da própria classe "Principal", e faz uma chamada ao seu único método ("Mensagem"), que por sua vez exibe na tela uma mensagem de saudação.

4.1.2 Criando os Aspectos

Os aspectos de um programa são definidos através de comandos armazenados em um arquivo específico para esta finalidade. Para criar este arquivo no eclipse, pode-se proceder de forma semelhante à criação de uma classe, selecionando a opção "*File > New > Other" e em seguida "AspectJ > Aspect*" na janela que irá se abrir (seletor de projetos), pressionando por fim o botão "*next*". Novamente uma caixa de diálogo surgirá, onde se deve informar o nome desejado para o arquivo de aspectos, e então pressionar o botão *finish*. Supondo que se tenha dado o nome de "Aspectos" ao aspecto recém criado, o novo arquivo conterá inicialmente as seguintes linhas de código:

```
public aspect Aspectos
{
}
```

Dentro destas chaves serão inseridas posteriormente as declarações dos *pointcuts* e dos *advices* que compõem os aspectos. Antes porém é necessária a definição de alguns destes termos (Chaves, 2003):

Join points: são pontos bem definidos da execução de um programa, por exemplo, uma chamada a um método ou a ocorrência de uma exceção, entre muitos outros.

Pointcuts: são elementos do programa usados para definir um *join point*, como uma espécie de regra criada pelo programador para especificar eventos que serão atribuídos a *join points*. Os *pointcuts* existem para que possam ser criadas regras genéricas para definir os eventos que serão considerados "*join points*", para que não seja necessário definir cada *join point* individualmente (o que tornaria a POA quase sem sentido). Outra função dos *pointcuts*, além de definir os eventos considerados como *join points*, é apresentar dados do contexto de execução de cada *join point*, que serão utilizados pela rotina disparada pela ocorrência do *join point* em questão.

Advices são pedaços da implementação de um aspecto que são executados em pontos bem definidos da execução do programa principal (*join points*). Eles são compostos de duas partes: a primeira delas é o *pointcut*, que define as regras de captura dos *join points*, e a segunda delas são o código que será executado quando ocorrer um *join point* definido pela primeira parte.

Por fim, um aspecto é uma unidade de código (um “tipo”) que encapsula os diversos *pointcuts* e *advices* de um programa. Uma representação gráfica pode ser Observada na Figura 6.

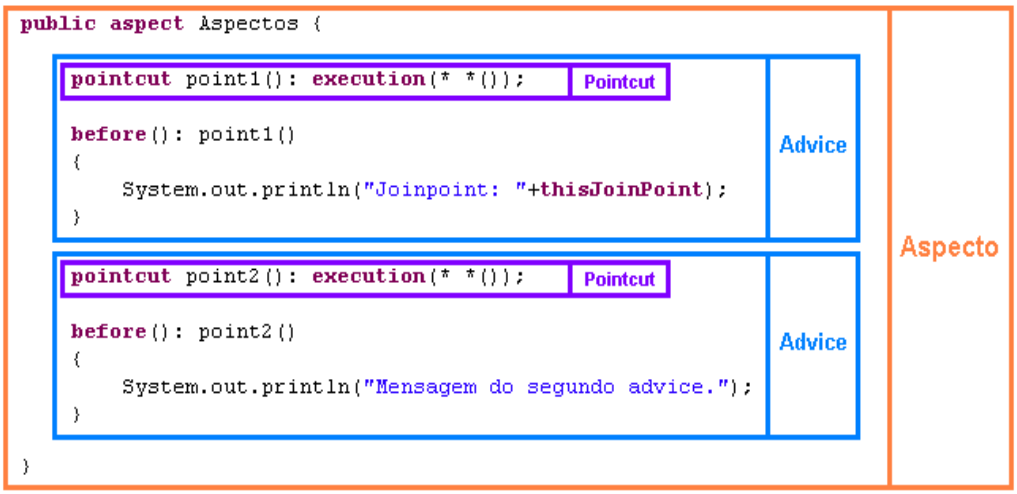

Figura 6.: Elementos de um Aspecto

É importante observar que os *pointcuts* e os *advices* são elementos reais, ou seja, são representados fisicamente por trechos de código criados pelo programador. Já os *join points* não existem fisicamente. Eles são apenas os “pontos de entrada” mentalizados pelo programador, como marcadores imaginários de pontos onde se deseja executar porções extras de código.

Agora, com estes conceitos, pode-se criar um aspecto que tem a função de gerar uma entrada de log para cada vez que o programa fizer qualquer chamada para qualquer método, independente do nome do método, ou dos parâmetros que lhe sejam passados, registrando ainda o nome do método que foi chamado, ou seja, do método que está em execução. Baseado neste enunciado pode-se mentalizar os *join points* necessários, neste caso, cada execução de um método. A regra que define cada execução de um método como um join point, ou seja, deve-se criar um *pointcut*, definição do *pointcut*:

*Pointcut point1( ): execution(* *( )) ;*

Está definido, neste trecho, o nome do *pointcut* (point1), e o tipo de evento que se deseja capturar (no caso a execução de um método: *execution*). Para cada tipo de evento será necessário fornecer parâmetros que definam a gama de abrangência desejada para os eventos capturados. No caso dos eventos de execução (*execution*), os parâmetros especificam as características dos métodos que são executados. Em outras palavras, todas as “execuções” de quaisquer métodos são capturadas, mas somente aquelas que atenderem aos

critérios definidos pelos parâmetros serão consideradas *"join points"*. No exemplo acima o primeiro elemento do parâmetro refere-se ao tipo de retorno do método (*int, float, void*), no caso qualquer tipo (*), seguido pelo nome do método, também qualquer nome (*), novamente qualquer assinatura (..). Existem diversos outros tipos primitivas de *pointcuts*, que são descritos em detalhe na documentação que acompanha o *AspectJ*.

Uma vez criado o *pointcut*, deve-se definir o que será executado no momento em que ocorrer um *join point* definido por ele. No exemplo será apresentada uma mensagem, relatando o nome do método que está sendo executado antes da execução do método:

```
before(): point1()
{
  System.out.println("Join Point: "+ thisJoinPoint);
}
```

No comando acima é criada a porção do *advice* que será executada sempre que um *join point* for capturado pelo *pointcut* "point1". O comando "*before*" que antecede o nome do *pointcut* (point1) especifica que.a porção de código que se segue será executada antes da execução do *join point* que a disparou. Ao invés de *"before*" pode-se usar "after" ou "around", para que o código seja executado depois ou durante o *join point* respectivamente. Em seguida, após especificar a qual *pointcut* o *advice* se refere e o seu tipo (*before, after* ou *around*), é inserido entre chaves ({}) a parte de código a ser executada: no exemplo acima usamos o comando "*println*" para fazer a saída via terminal (console) de um texto padrão, sendo acrescentado ao final o comando *thisJoinPoint*, que retorna uma *string* contendo o nome do join point que disparou sua execução.

Depois de criado este aspecto, ou tantos quantos forem necessários, para testar, é necessário primeiramente fazer *"rebuild all"*, onde ocorre automaticamente o *weaverirng* , e em seguida executar o programa principal como uma aplicação Java (*Run > Run As > Java Application)*, e o resultado obtido será:

Joinpoint: execution(void Principal.Mensagem())

Olá mundo !

Destas duas mensagens, como pode ser visto na classe principal e no *advice* do arquivo de aspectos, a primeira foi gerada pela ocorrência de um *join point* e a segunda pela execução do programa principal, demonstrando com isso o funcionamento do programa orientado a aspecto.

5. Conclusões

A engenharia de software e as linguagens de programação coexistem em um relacionamento de suporte mútuo. A maioria dos processos de projeto de software considera um sistema em unidades cada vez menores. As linguagens de programação, por sua vez, fornecem mecanismos que permitem a definição de abstrações de unidades do sistema e a composição destas de diversas formas possíveis para a produção do sistema como um todo (BECKER & GEIHS, 1998).

A programação orientada a aspectos possibilita que o programador separe o código referente ao negócio dos interesses ortogonais, centralizando estes interesses em aspectos bem definidos no sistema.

Esta separação propiciada pela AOP (programação orientada a aspectos) pode trazer diversos benefícios para uma equipe de desenvolvimento. Para todos os sistemas, poderia ser utilizado um mesmo programador especialista no tratamento de exceções, sistema de log, conexão com banco de dados, etc, já que não há necessidade que este programador conheça o código referente ao negócio do sistema.

A utilização da programação orientada a aspectos aumenta em larga escala a taxa de reutilização de código, já que as classes conterão código referente apenas ao negócio do sistema, evitando desta forma, a utilização de código espalhado pelo mesmo.

6. Referências

BECKER, Christian, GEIHS, Kurt. **Quality of Service - Aspects of Distributed Programs**. Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto (Japão), 1998.

BÖLLERT, Kai. **On Weaving Aspects.** In the Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

Chavez, Christina von Flach G. e Lucena, Carlos J. P.A (2003). Theory of Aspects for Aspect-Oriented Software Development.. UFBA.

IRWIN, John, KICKZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag, MAEDA, Chris, LOPES, Cristina Videira, LOINGTIER, Jean-Marc. **Aspect- Oriented Programming.** proceeding of ECOOP'97, Finland: Springer-Verlag, 1997.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. Proceedings European Conference on Object-Oriented Programming. Disponível em http://citeseer.nj.nec.com/63210.html

LOPES, Cristina Isabel Videira. **D: a language framework for distributed programming.** Ph. D. Thesis, Northeast University, 1997.

Piveta, Eduardo (2001). Um modelo de suporte a programação orientada a aspectos. UFSC. Dissertação submetida como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Soares, Sérgio e Borba, Paulo. (2002). AspectJ, Programação orientada a aspectos em Java. UFPE. Artigo remetido para SBPL.