Programação Orientada a Aspectos Aplicada.

Charles Wellington de Oliveira Fortes
chalkmaster@gmail.com

Resumo:

Demonstrar de forma clara e prática como a Programação Orientada a Aspectos pode ajudar a melhorar a qualidade do desenvolvimento de softwares, melhorando a modularidade e abstratividade do código e reduzindo a complexidade das classes.

Para tal, iremos expor através de modelos de código utilizados no desenvolvimento de aplicações JAVA e AspectJ.

Palavras - chave: Aspectos, POA, AspectJ, Java, Framework, Engenharia de Software.

1. Introdução:

O universo da programação busca a cada dia o aperfeiçoamento dos códigos (*source codes*), melhorando o aproveitamento de código, reduzindo o tempo de manutenção, facilitando novas implementações, preservando os sistemas contra erros e facilitando o reaproveitamento do código de forma concisa e simplificada.

Um dos maiores desafios encontrados até hoje é a criação de códigos que sejam simples de serem compreendidos e sejam de fácil manutenção e reaproveitamento. A busca desta aproximação da linguagem do homem com a linguagem da máquina, que chegamos a paradigmas de programação como a orientação a objetos, orientação a serviços e a própria orientação a aspectos dentre outros que buscam um alto nível de modularização e de abstração.

Mesmo com o surgimento da Orientação a Objetos nos meados dos anos 70 com a linguagem *Smaltalk*, nem todas as necessidades da abstração e da modularização do código foram supridas, pois ainda temos o que chamamos de *crosscutting* (“Elementos Entrecortantes”).

Elementos Entrecortantes são trechos de código que coexistem por diversos pontos do sistema e que compõem uma camada intrafuncional da aplicação na qual muitas vezes mistura funções coerentes à regra de negócio com as do próprio sistema, como pode ser observado na figura 1.

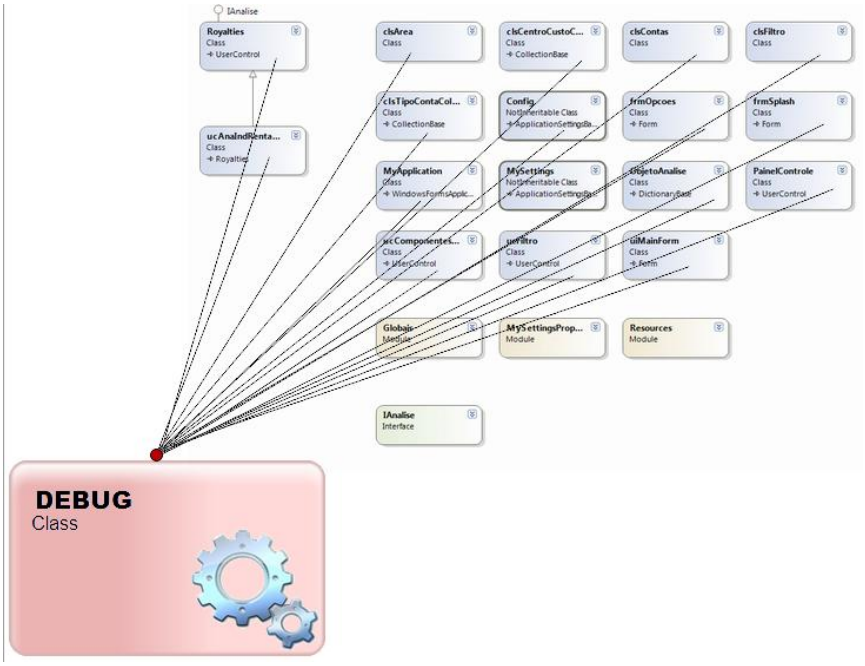

Figura 1 – Pontos onde funções “entrecortastes” atuando no sistema

Como vemos na figura 1, funções e métodos para tratamento de exceções e log de auditoria, por exemplo, se encontram espalhados por diversas classes e camadas da aplicação, mesmo que com a orientação a objetos o log seja inteiramente programado em uma única classe de log, esta ainda assim necessita ser instanciado em todos os

métodos nos quais se deseja que o log seja gravado, além de haver a necessidade de se chamar o método que dispara o log.

Esta prática ainda que seja um grande avanço sobre a programação estruturada e outros paradigmas, ainda assim não é a solução para todos os casos encontrados, pois se necessitarmos mudar a assinatura do método de log será necessário varrer todo o programa alterando as chamadas do método, além de estarmos delegando obrigação para a classe e para o desenvolvedor que a programa.

A Orientação a Aspectos, assim como outros paradigmas, surgiram para resolver estas e outras deficiências que possamos encontrar na hora de buscar a abstração das camadas de negócios das de sistema, tornado o código ainda mais modularizado, limpo e livrando o desenvolvedor e a classe das responsabilidades inerentes de rotinas paralelas às de negócio, porém de igual valor para a aplicação.

2. Orientação a Aspectos:

A Programação Orientada a Aspectos (POA) foi criada no ano de 1997, em Palo Alto, nos laboratórios da Xerox, por Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier e John Irwin, com o objetivo de construir um paradigma que permitisse à linguagem de componentes modularizar melhor as unidades que compõem a camada sistêmica do programa.

Com a POA, o desenvolvedor precisa preocupar-se apenas em desenvolver códigos referentes às regras de negócios inerentes aos métodos e às classes do programa, não se preocupando com outras funções básicas e de igual importância ao

funcionamento do sistema, como o tratamento de exceções, permissões de acessos às rotinas, auditoria, concorrência, dentre muitas outras.

É comum observar nos ambientes organizacionais que adotam este paradigma, a existência de duas equipes de desenvolvimento, uma voltada para a criação da aplicação voltada às suas regras de negócios, e outra voltada à criação de aspectos eficientes e alto desempenho que serão utilizados nestes e em outros projetos que possam vir a ser realizados.

Assim, é importante salientar que a Programação Orientada a Aspectos (POA), não vem propor um novo paradigma de programação para substituir os atuais, mas sim um parceiro que trabalhará em conjunto com os mesmo de forma a suprir suas necessidades.

2.1. Composição de um sistema orientado a aspectos:

2.1.1. Linguagem de componentes:

Segundo Irwin et al. (1997), "a linguagem de componente deve permitir ao programador escrever programas que implementem as funcionalidades básicas do sistema, ao mesmo tempo em que não provêem nada a respeito do que deve ser implementado na linguagem de aspecto". Ex.: C#, JAVA, PHP, etc.

2.1.2. Linguagem de aspecto:

A linguagem de aspecto 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).

2.1.3. Programas de componente:

Os programas de componente são arquivos de código (*source code*) desenvolvido na linguagem de componente e que contem as codificações das regras de negócio em módulos/classes.

2.1.4. Um ou mais programas de aspectos

Os programas de Aspecto são os arquivos fonte (*source code*) implementados em linguagem de aspecto, como AspectJ, AspectC, AspectWerkz, entre outras. Algumas linguagens são voltadas a interesses específicos da aplicação, e neste caso devem ser utilizadas múltiplas linguagens de aspectos, uma para cada interesse. Um exemplo de uso é uma aplicação na qual utilizamos o Ridl para programar os aspectos de distribuição e o COOL para gerenciamento de concorrência.

2.1.5. Combinador de aspectos

O Combinador de aspectos (*Aspect Weaver*) é o responsável por combinar o código desenvolvido na linguagem de componentes com o desenvolvido na linguagem de aspectos. O Combinador de aspectos age em um momento antes da compilação do código em *bitecode*, inserindo os *Advices* que serão definidos untos aos *pointcus* aos *Join Points* que correspondentes.

O *Aspect Weaver* pode atuar em tempo de compilação ou execução, que por sua vez permite a adição/exclusão de aspectos enquanto a aplicação está em execução, dando um amplo campo de aplicação para o paradigma.

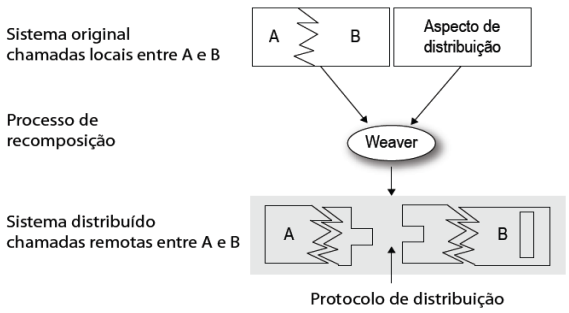

Figura 2 – Funcionamento do *Aspect Weaver*

2.1.6. Linguagens POA e suas características:

O desenvolvimento em POA pode ocorrer por linguagens de propósitos específicos ou em linguagens de propósito geral.

As linguagens de "propósito específico" são mais abstratas e eficientes. Estas têm a capacidade de restringir a utilização de algumas palavras chaves na linguagem de componente, de forma a impedir que funções cabíveis aos aspectos sejam feitas na camada de negócios.

Já as linguagens gerais, são mais abrangentes e ganham melhor aceitação junto aos desenvolvedores, pois além de não exigir a aprendizagem de uma nova linguagem para sua que o aspecto seja desenvolvido, e possibilita ao desenvolvedor utilizar a mesma IDE para desenvolver o programa de aspecto e o programa de componentes. Estas linguagens, porém, não permitem a reserva de palavras como as linguagens específicas.

3. Entendendo Aspectos com AspectJ

3.1. Elementos básicos

O AspectJ é uma das mais consolidadas ferramentas de desenvolvimento de aspectos de propósito geral que encontramos hoje, por isto vamos utilizá-la para demonstrar a Programação Orientada a Aspectos.

Os elementos básicos que encontramos na POA e presentes no AspectJ são: *Join Points, PointCuts, Advices, Introduction, Compile-time declaration* e *Aspects.*

Para melhor ilustrar nossas idéias, vamos tomar o seguinte programa Java (Figura 3) e o Aspecto abaixo (Figura 4) como base para a representação de alguns destes itens:

```
import java.util.ArrayList;

public class main {


    private static ArrayList Estoque = new ArrayList();

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        if (InicializaEstoque()){
            Iterator inter = Estoque.iterator();
            while (inter.hasNext()){
                System.out.println(inter.next());
            }

        }
    }

    @SuppressWarnings("finally")
    public static boolean InicializaEstoque(){
        boolean Status = false;
        try{
            Estoque.add("Lápis");
            Estoque.add("Caderno");
            Status = true;
        }
        catch (Exception e){
            Status = false;

        } finally {
            return Status;
        }

    }

}
```

Figura 3 – Modelo de Código Java que irá receber o aspecto da figura 4

```
public aspect debug {
    pointcut debug() : call (* *.*(..)) && !within(debug);

    before() : debug() {
        String nomeMetodo = thisJoinPointStaticPart.getSignature().getName();
        System.out.println("========== Debug AspectJ ==========");
        System.out.println(" Entrando no metodo -> '" + nomeMetodo + "'" );
        System.out.println("==================================");
    }

    after() : debug() {
        String nomeMetodo = thisJoinPointStaticPart.getSignature().getName();
        System.out.println("========== Debug AspectJ ==========");
        System.out.println(" Saindo do metodo -> '" + nomeMetodo + "'" );
        System.out.println("==================================");
    }


}
```

Figura 4 – Aspecto "Debug ()" que será aplicado a nossa aplicação (Figura 3)

Veja que nosso código fonte (*Source Code*) apenas instancia um objeto do tipo *ArrayList*, o inicializa incluindo dois itens e em seguida, caso os itens tenham sido incluídos com sucesso, os imprime no console. Em momento algum há chamadas ao método de debug(), por tanto se o programa da classe "main()" acima for executado ele apenas ira imprimir na tela:

"Lápis"

"Caderno"

O que determina onde, como quando nosso aspecto de debug() será executado é o próprio aspecto, não havendo a necessidade de instanciação de objetos, chamadas de métodos ou qualquer outro tipo de referência ao aspecto dentro de nosso fonte.

Esta prática permite ao desenvolvedor preocupar-se única e exclusivamente com o desenvolvimento de seu código, sem se preocupar com importâncias que não cabe a regra de negócio, pois estes serão tratados posteriormente pelo aspecto. Mesmo tratamento de exceções ou instanciações de objetos é possível através deles.

3.1.2. Join Points

Um Join Point é o local no código de seu programa onde os pontos de corte (*pointcuts*) de seu aspecto irão atuar.

Os Join Points são definidos pelo gerenciador de aspectos de acordo com as instruções contidas no *pointcut* e é neste ponto do código que atuará o(s) *Advice(s)* de seu aspecto.

No exemplo acima o programa irá executar o aspecto debug() sempre que houver uma chamada de método que não seja o próprio método debug() (abordaremos como isto ocorre logo adiante), executando *Advices* antes e depois da execução deles.

3.1.3. PointCuts

Um ponto de corte (*pointcut*) é o que define seu *Join Points*, que é onde o *Advice* será aplicado, ele pode ser referenciado a uma chamada de métodos, a instanciação de um objeto, uma exceção, e em diversas outras situações.

Em outras palavras, o *pointcut* é o que informa ao *Aspect Weaver* o local onde os *Advices* serão inseridos e sobre quais circunstâncias. No exemplo acima, definimos que nosso aspecto debug irá atuar em nossa solução, sempre que um método que não seja o próprio método debug() for chamado pela aplicação veja (Figura 5):

```
pointcut debug() : call (* *.*(..)) && !within(debug);
```

Figura 5 – definição do *pointcut*

Quando declaramos nosso *pointcut*, utilizamos a palavra reservada “call” para identificar que ele atuará na chamada dos métodos, e usamos “(* *.*(..))” para definir o escopo de atuação do aspecto, em quais métodos ele irá agir.

Neste exemplo dissemos ao aspecto para atuar em todos os projetos que compõem a solução, em todas as classes e em todos os métodos, com exceção ao próprio método debug() (definido após utilizar o operador lógico “&&” , negando o método reservado “*within*” que é usado para adicionar um método à condição).

3.1.4. Advice

Os *Advices* são os trechos de instruções de códigos que serão inseridos nos *Join Points*, podendo atuar antes, depois ou mesmo substituindo o conteúdo do *Join Point* pelo trecho descrito em si.

Esta atuação é definida pelas palavras reservadas informadas em sua declaração. No exemplo acima, estamos incluindo os trechos de código que irão informar em que passo do sistema estamos, antes e depois do *Join Point* através das palavras reservadas “*before*” (antes) e “*after*” (depois).

3.1.5. Introduction

Uma *Introduction* é uma declaração estática da POA que permite alterar classes, interfaces e até mesmo outros aspectos do sistema, adicionando, por exemplo, um método, uma variável ou outro objeto no ponto especificado.

3.1.6. Compile-time declaration

O *compile-time declaration* é uma declaração estática que permite disparar alertas ou mesmo erros quando certos padrões de utilização de classes são encontrados durante o desenvolvimento, ajudando o desenvolvedor desta forma, a não utilizar indevidamente um método ou uma propriedade de determinado objeto.

3.1.7. Aspect

Um *aspect* é um bloco de código composto por *Join Point, PointCuts, Advices* e outros que irão atuar conforme viemos demonstrando.

Um aspect no AspectJ, assim como em outras ferramentas que focam o desenvolvimento de Aspectos de propósito gerais, pode conter os mesmos elementos que qualquer classe da linguagem de componentes adotada (Java no nosso caso) contém, além de seus próprios elementos, dando ao desenvolvedor muita liberdade e conforto ao desenvolver, pois este se familiariza com a ferramenta.

3.2. A lógica de compilação

O compilador de aspectos atua instante antes do compilador da linguagem de componente produzir o *bitecode* que dará origem a aplicação propriamente dita, produzindo um código intermediário que mescla o código da IDE de componente com o código do aspecto traduzido.

Voltando ao nosso exemplo, o efeito produzido será similar ao abaixo (figura 6):

```
public static void main(String[] args) {
    debug_before("InicializaEstoque");
    if (InicializaEstoque()){
        debug_before("iterator");
        Iterator inter = Estoque.iterator();
        debug_before("hasNext");
        while (inter.hasNext()){
            debug_before("println");
            debug_before("next");
            System.out.println(inter.next());
            debug_after("next");
            debug_after("println");
        }
        debug_after("hasNext");
        debug_after("iterator");
    }
    debug_after("InicializaEstoque");
}
```

```
public static void debug_after(String nomeMetodo){
    System.out.println("========= Debug AspectJ ==========");
    System.out.println(" Saindo do metodo -> '" + nomeMetodo + "'" );
    System.out.println("==================================");
}

public static void debug_before(String nomeMetodo){
    System.out.println("========= Debug AspectJ ==========");
    System.out.println(" Entrando no metodo -> '" + nomeMetodo + "'" );
    System.out.println("==================================");
}
```

Figura 6 – Código visto após a pré-compilação executada pelo *Aspect Weaver*

Percebam que a classe que será compilada tem a inclusão dos *Advices* definidos no aspecto, aplicados em seus respectivos *Join Points* (Chamadas de métodos em vermelho).

4. Conclusão

A POA é um paradigma novo e por isto tem ainda muito a crescer, mas desde já proporciona um ambiente de desenvolvimento capáz de produzir softwares concisos, de fácil manutenção e com uma separação clara entre as camadas/regras/rotinas de negócio e de sistema, tornando a aplicação mais confiável, com menos possibilidades de erros, além de melhor a reutilização destas rotinas (debug por exemplo) em outras aplicações sem a necessidade de mudanças no fonte.

Outra aplicação interessante para a POA é na implementação de novos recursos em sistemas antigos, que se tornam simples e fácil de se fazer.

Bibliografia:

LADDAD, Ramnivas. AspectJ in Action: Practical Aspect-Oriented Programming. Ed. Mannig, Canadá, 2003.

WAZLAZWICK, Raul Sidnei. Análise de Projeto de Sistemas de Informação Orientados a Objetos. Rio de Janeiro: Elsevier, 2004.

GOETTEN Junior, WINCK Diogo, 2006. AspectJ - Programação Orientada a Aspectos com Java. São Paulo - SP: Novatec Editora, 2006.

ASPECTJ Team. Disponível em: http://www.eclipse.org/aspectj Acessado em 17/03/2008.

GRADECK, Joe, LESIECKI, Nicolas. Mastering AspectJ: aspect-oriented programming in Java. Indianopolis, Indiana. Wiley, 2003, p. 453.

IRWIN, John, et al. Aspect – Oriented Programming. Proceeding of ECOOP'97, Finland: Springer – Verlag,1997.