FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO TOBIAS

Implementação do Padrão Capturador de Dados em Linguagens Orientadas a Aspectos

MARÍLIA
2007

THIAGO TOBIAS

Implementação do Padrão Capturador de Dados em Linguagens Orientadas a Aspectos

Monografia apresentada como Trabalho de Conclusão de Curso ao Curso de Bacharelado em Ciência da Computação do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino "Eurípides Soares da Rocha", para obtenção do Título de Bacharel em Ciência da Computação. (Área de Concentração: Engenharia de Software).

Orientador:
Prof. Dr. Valter Vieira Camargo

MARÍLIA
2007

TOBIAS, Thiago.
Implementação do Padrão Capturador de Dados em Linguagens Orientadas a Aspectos / Thiago Tobias; orientador: Prof. DR. Valter Vieira Camargo.
Marilia, SP: [s.n.],2007.
69f.

Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino Eurípedes Soares da Rocha

1. Aspectos 2. Adendos 3. Pontos de Junção 4. Conjunto de Junções 5. Capturador de Dados

CDD:005.757

THIAGO TOBIAS

IMPLEMENTAÇÃO DO PADRÃO CAPTURADOR DE DADOS EM LINGUAGENS ORIENTADAS A ASPECTOS

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação, do UNIVEM,/F.E.E.S.R., como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Resultado: Aprovado

ORIENTADOR: Prof. DR. Valter Vieira Camargo.

1º EXAMINADOR: Prof. Dr. Edmundo Ségio Spoto

2º EXAMINADOR: Profa. Dra. Maria Istela Cagnin Machado

Marília, 19 de Novembro de 2007.

AGRADECIMENTOS

Ao meu pai, à minha mãe e minha irmã que sempre me apoiaram nos momentos de dificuldade e alegria da minha vida.

Ao meu orientador, Professor Dr. Valter Vieira Camargo, por poder trabalhar ao seu lado, pela ajuda concebida, pela paciência que teve ao trabalhar comigo.

A todos os professores do curso de Ciência da Computação com quem tive a alegria de compartilhar uma das fases mais importantes da minha vida.

Aos meus amigos e colegas da Faculdade, em especial, ao Guilherme Oelsen Franchi Junior e Rafael Euclides da Silva que me incentivaram e ajudaram com as dificuldades encontradas no decorrer do curso.

A minha amiga Larissa Buffulin Ribeiro que sempre me motivou e ajudou nos momentos que encontrei maiores dificuldades, e me deu força para continuar em frente.

Aos meus amigos: Fernando, Kleber, André, Leandro, Ana Paula.

TOBIAS, Thiago. **Implementação do Padrão Capturador de Dados em Linguagens Orientada a Aspectos.** 2007, 69f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino Eurípedes Soares da Rocha, Marília, 2007.

RESUMO

O paradigma de programação orientada a objeto (POO) surgiu com o conceito de bons níveis de reúso e facilitação da manutenção, porém com o passar do tempo notou-se alguns problemas de interesses transversais, sendo os que não podem ser modularizados em classes como persistência de objetos, segurança de acesso, acesso concorrente, gerenciamento de transações. Um novo paradigma foi estabelecido, o qual procura modularizar os interesses transversais, nomeado de aspectos, que possui em seu contexto adendos (*advices*), pontos de junção (*join point*) e conjunto de junções (*pointcuts*). Deste modo, um notável número de programação orientada a aspectos (POA) surgiu a fim de suprir as necessidades existentes na orientação a aspectos, como AspectJ, AspectWerkz, JAsCO entre outras. Estas três linguagens serão analisadas com o objetivo de informar se os seus padrões podem ser implementados no conceito de um framework orientado a aspectos. Entretanto, o mais importante deste estudo é verificar se as linguagens suprem o conceito de um padrão de projeto existente, denominado Capturador de Dados, onde o entrecorte do aspecto tenha que capturar dados do código-base para poder manipulá-lo da maneira que achar conveniente.

Palavras-chave: Aspectos, Adendos, Pontos de Junção, Conjunto de Junções, Capturador de Dados, Frameworks Orientado a Aspectos.

TOBIAS, Thiago. **Implementação do Padrão Capturador de Dados em Linguagens Orientada a Aspectos.** 2007, 69f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino Eurípedes Soares da Rocha, Marília, 2007.

ABSTRACT

The object-oriented paradigm emerged with the concept of improve the levels of reuse and maintenance, but over time it is noticed some problems of crosscutting concerns, which can not be modularized in classes. Examples of such concerns are persistence, security, concurrency and transaction management. In this direction, a new paradigm has been established, which seeks modularize the crosscutting concerns with new concepts, such as advices, join points and pointcuts. This new paradigm is called Aspect Oriented Programming. Over the last years, a remarkable number of the aspect-oriented languages has emerged, such as AspectJ, AspectWerkz, JAsCO among others. In this work, these three languages will be analyzed in order to know if a pattern implemented in a previous work can be implemented.

Key words: Aspects, Advices, Join Points, Pointcuts, Data Catcher, Frameworks Aspects Oriented.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVEATURAS

JVM – Java Virtual Machine

OA – Orientação a Aspectos

OO – Orientação a Objeto

POA – Programação Orientada a Aspectos

POJO - Plain Old Java Objects

POO – Programação Orientada a Objetos

SUMÁRIO

INTRODUÇÃO

Por muito tempo o paradigma da orientação a objetos foi considerado adequado para a implementação de sistemas de software por propiciar bons níveis de reúso e facilitar a manutenção. Mais recentemente, por volta de 1997, começou-se a perceber que esse paradigma possui algumas deficiências para modularizar determinadas tipos de interesses, chamados interesses transversais (*crosscutting concerns*), fazendo com que o código desses requisitos fique entrelaçado e espalhado pelos módulos funcionais do sistema. Sendo assim, Gregor Kiczales propôs um novo paradigma de programação chamado "programação orientada a aspectos", que visa modularizar os interesses transversais de um sistema com um tipo especial de módulo, chamado "aspecto" (Kiczales, 1997).

Os exemplos mais clássicos de interesses transversais são os requisitos não-funcionais como persistência, distribuição, controle de acesso, autenticação, registro de execução de operações (*log*), rastreamento (*tracing*) e concorrência. Contudo, há também interesses transversais funcionais, como por exemplo, regras de negócio que sejam dependentes de um período de tempo e que não são diretamente relacionadas com a funcionalidade do sistema (Camargo, 2006).

Com a evolução da programação orientada a aspectos (POA) muitas tecnologias que já estavam bem consolidadas tiveram que ser reavaliadas sob a ótica dos seus conceitos introduzidos pela POA. Assim, Frameworks Orientados a Aspectos surgiram, e um grande número desses frameworks tem sido propostos. Entretanto, padrões para o projeto e a implementação desses frameworks ainda é uma área pouco explorada. Camargo (2008) propôs um padrão que permite aumentar os níveis de reusabilidade de um FOA. Entretanto, para que esse padrão cumpra de forma adequada seus objetivos, o mesmo deve poder ser implementado em diversas linguagens orientadas a aspectos. Várias linguagens orientadas a

aspectos já se encontram disponíveis, entre elas: AspectJ (ASPECTOS, 2007), JAsCo (JAsCo, 2007) e AspectWerkz (AspectWerkz,2007).

1.1 Motivações e Objetivos

A motivação principal para a realização do trabalho proposto é a carência de estudos sobre a implementação de padrões para o projeto e a implementação de FOAs. Além disso, outra motivação também é aumentar os níveis de reusabilidade em frameworks implementados com programação orientada a aspectos.

O objetivo deste estudo é averiguar se um padrão proposto por Camargo (2008), chamado Capturador de Dados, pode ser adequadamente implementado em três linguagens orientadas a aspectos: AspectJ, JAsCo e AspectWerkz.

1.2 Estrutura do Trabalho

O Trabalho esta dividido em cinco Capítulos, incluindo o Capítulo introdutório.

O Capítulo 2 apresenta a definição da Programação Orientada a Aspectos, contendo os seus conceitos, funcionalidade e os mecanismos utilizados, além de conter uma parte descrevendo o Padrão Capturador de Dados e sua importância na criação de um projeto e alternativas que lhe concede ao utilizá-lo.

O Capítulo 3 apresenta as linguagens AspectJ, AspectWerkz e JAsCO, estudadas para serem utilizadas na implementação do projeto PEDIDOS junto com o conceito do Padrão Capturados de Dados, mostrando o tipo de arquitetura de cada linguagem e seus recursos.

O Capítulo 4 descreve o estudo de caso do projeto Pedido e o processo de desenvolvimento e a implementação, utilizando as linguagens AspectJ, AspectWerkz e JAsCO.

O trabalho é concluído no Capítulo 5, avaliando e comparando cada linguagem, verificando se seus conceitos satisfazem o Padrão Capturador de Dados, a viabilidade de trabalhos futuros e as dificuldades obtidas ao decorrer do trabalho.

2. REVISÃO BIBLIOGRÁFICA

2.1. Programação Orientada a Aspectos

Programação Orientada a Aspectos mais conhecida como POA é um paradigma da programação de computadores que possibilita aos programadores de *software* separar e organizar o código de acordo com a necessidade do *software.* A POA foi proposta em 1997 por Kiczales (1997), como uma técnica objetiva para melhorar o suporte à modularização dos interesses transversais por meio de abstrações que possibilitem a separação e composição destes interesses na construção dos sistemas de *software*.

Os paradigmas de programação mais antigos, como a Programação Procedural e Programação Orientada a Objeto, implementam a separação do código, através de entidades únicas, fazendo com que as classes chamadas em POO ou funções em procedural sejam espalhadas por toda a aplicação.

Tipicamente uma implementação da POA busca encapsular essas chamadas através de uma nova construção chamada de aspecto. Um aspecto pode alterar o comportamento de um código (parte do programa não orientada a aspecto) pela aplicação de comportamento adicional, adendo (*advice)*, sobre um ponto de execução, ou pontos de junção(*join point*). A descrição lógica de um conjunto de Ponto de junção (Join Points) é chamada de conjunto de junções (*pointcut*) (Kiczales, 1997).

Do ponto de vista matemático, os aspectos formam uma extensão de lógica de segunda ordem para qualquer paradigma de programação, enquanto paradigmas usuais levam a um raciocínio baseado em funções, mensagens e assim por diante, através de uma assinatura função/mensagem. A POA possibilita um raciocínio baseado em conjuntos destas entidades utilizando os conjuntos de junções (*pointcuts*) com um caracter de substituição (*wildcard*) na

sua assinatura. Portanto, pode-se enxergar a POA mais como uma extensão lógica poderosa, do que como um paradigma de programação. Esta visão foi proposta por Friedrich Stelmann.

Os defensores da POA (Camargo, 2006), a promovem como um pacote externo que pode ser entregue junto com a aplicação. Por exemplo, se um programa por si não tem suporte à segurança, um pacote POA pode servir como uma extensão modular para a aplicação, disponibilizando a segurança.

A depuração é um dos maiores problemas. No nível sintático, o código POA aparece separado, porém junto do restante do código em tempo de execução. A inserção de *advices* pode se tornar imprevisível se não estiver definido aonde o aspecto deve entrecortar ou seja interferir no código-base. Os projetistas devem considerar meios alternativos para conseguir a separação do código, porém estas abordagens não têm um mecanismo de quantificação que permite que o programador chegue a diversos pontos de junção com apenas uma declaração. Outro problema com a POA é a captura não intencional de ponto de junção (Kiczales, 1997).

Fundamentalmente, o modo como o aspecto interage com o programa é definido como modelo de pontos de junção (*join point*) no qual o aspecto é escrito (Kiczales, 1997).

Pontos de junção (*Join points*) são os locais no código do componente onde os aspectos podem interferir, portanto um ponto de junção é um ponto bem definido no fluxo de execução de um programa. Um ponto de junção indica um lugar no código do componente em que deseja-se inserir um novo comportamento. Exemplos de pontos de junção são: uma chamada a uma função, uma declaração de variável, o construtor de uma classe, etc. Os aspectos podem adicionar comportamento antes ou depois do ponto de junção, bem como obter o controle completo sobre o ponto de junção, executando o código do aspecto no lugar do código original do ponto de junção. Deve-se observar que para a utilização efetiva da POA uma linguagem de programação, é necessário que se tenha um mecanismo capaz de definir um bom conjunto de pontos de junção, logo, uma ferramenta que possibilite o uso da POA

deve ser bastante flexível para permitir a descrição de um vasto conjunto de pontos de junção (QUEIROZ, 2006).

Em POA é introduzido um novo mecanismo para abstração e composição, que facilita a modularização dos interesses transversais, o aspecto (*aspect*). Desta forma, os sistemas de software são decompostos em componentes e aspectos. Assim, os requisitos funcionais normalmente são organizados em componentes através de uma linguagem de programação orientada a objetos (POO), como Java, e os requisitos não funcionais como aspectos relacionados as propriedades que afetam o comportamento do sistema (Kiczales, 2001).

Conjuntos de Junções (*Pointcut*) é uma construção de linguagem que junta um conjunto de pontos de junções baseando-se em um critério pré-definido(Kiczales, 1997).

Depois de definirem os pontos a serem criados, é usado o adendo (*advice*) para completar implementação. Adendo é o trecho de código que é executado antes (*before* ()), depois (*after* ()) e simultaneamente (*around* ()) a um ponto de junção é definindo em três partes (Kiczales, 1997):

- onde o aspecto pode ser aplicado.
- um modo para especificar, ou quantificar múltiplos ponto de junções, os chamados *pointcuts* ou conjuntos de junções, são na verdade uma consulta sobre todos os pontos de junção de um programa para selecionar um conjunto menor deles.
- meios para alterar o comportamento dos pontos de junção. São considerados como adendos (*advice*).

A POA envolve basicamente três etapas distintas de desenvolvimento (Kiczales, 1997):

- decompor os interesses (*aspectual decomposition*) – identificar e separar os interesses transversais dos interesses do negócio;
- implementar os interesses (*concern implementation*) – implementar cada um dos interesses identificados separadamente;
- recompor o aspectual (*aspectual recomposition*) – nesta etapa, tem-se integrador de aspectos que especifica regras de recomposição para criação de unidades de modularização – aspectos. A esse processo de junção da codificação dos componentes e dos aspectos é denominada combinação (*weaving*).

Uma implementação de POA consiste dos seguintes elementos (Kiczales, 2001):

- linguagem de componentes – responsável por implementar interesses do negócio do sistema de software, por exemplo, Java;
- linguagem de aspecto – deve suportar a implementação de interesses transversais de forma clara e concisa, fornecendo meios para construção de estruturas que descrevam o comportamento do aspecto e definam em que situações estes ocorrem, por exemplo, AspectJ;
- combinador de aspectos – sua tarefa é combinar aspectos (*aspect weaver*) com programas escritos na linguagem de componentes com os programas escritos na linguagem de aspectos.

As linguagens de aspectos são classificadas em linguagens de propósitos específicos e de propósitos gerais. Como o próprio nome diz, as linguagens de propósitos específicos tratam somente de determinados aspectos, impondo geralmente algumas restrições quanto ao uso das linguagens de componentes. Por outro lado, as linguagens de propósitos gerais permitem a implementação de qualquer tipo de aspecto, sendo de uso mais familiar e de fácil adoção pelos desenvolvedores, uma vez que geralmente a linguagem de aspecto compartilha o

mesmo ambiente de desenvolvimento utilizado pela linguagem de componente (Kiczales, 2001).

2.2. O Padrão Capturador de Dados

No padrão Capturador de Dados a parte responsável pelo código base ser implementada junto com o padrão de projeto, dando sentido ao padrão Capturador de Dados se for utilizado com FTs dependente de contexto, onde o entrecorte tenha que capturar dados do código-base para que possa funcionar.

Para ficar clara a importância do padrão capturador de dados, a Figura 1 mostra a implementação convencional de um aspecto onde entrecorta o código base durante a execução do método `comprarProduto()` onde ".." mostra que o método pode receber qualquer tipo de parâmetro O aspecto tem por objetivo interromper a criação de conexões do código-base.

```
public aspect Aspecto
{
    public pointcut Point():
        execution (void comprarProduto(..));

    void around() : Point()
    {
        thisJoinPoint.getArgs();
        System.out.println(getArgs + " terminando");
        proceed();
        Trace.meioMethod("Around");
    }

}
```

Figura 1 - Implementação de um aspecto capturador de dados

Por meio desta implementação nota-se que o código do adendo sempre realiza a captura do objeto por meio da declaração `getArgs()`, mas não se pode ter certeza de que em todo código-base ao qual o aspecto está atuando haverá um ponto de junção cujo objeto alvo (*target*) seja o objeto requerido pelo aspecto.

Assim, algumas alternativas de composição dão mais flexibilidade ao código deixando que o responsável pelo desenvolvimento escolha um modo adequado. No Quadro 1 algumas alternativas de composição do padrão Capturador de Dados são mostradas, tais como *This, Target, Args*.

Quadro 1 - Alternativas do Padrão de Composição

Alternativa de Composição	Descrição
This	Esta alternativa consiste em estender o aspecto This e fornecer um ponto de junção cujo objeto *this* possa ser utilizado para a obtenção do valor requerido pelo FT. Esse ponto de junção pode ser uma chamada ou execução de um método, ou o acesso a um atributo.
Target	Semelhante à alternativa anterior, porém o aspecto a ser estendido é Target e o objeto *target* do ponto de junção fornecido é que será utilizado para obtenção do dado requerido pelo FT.
Args	Semelhante à alternativa anterior, porém o aspecto a ser estendido é o Args e o parâmetro do ponto de junção fornecido será determinado pelo índice fornecido pelo engenheiro de aplicação por meio da concretização de um método abstrato.

A escolha da alternativa que deve ser utilizada depende dos dados que o aspecto requer do ponto de junção, isto é, de sua interface, e também da estrutura do código-base.

A Figura 2 mostra uma hierarquia de classes que representa os papéis do padrão. Os aspectos abstratos representam algumas alternativas de composição que podem ser especializadas. A classe/aspecto que deve ser criada durante o processo de reúso é apresentada MeusDesconto.

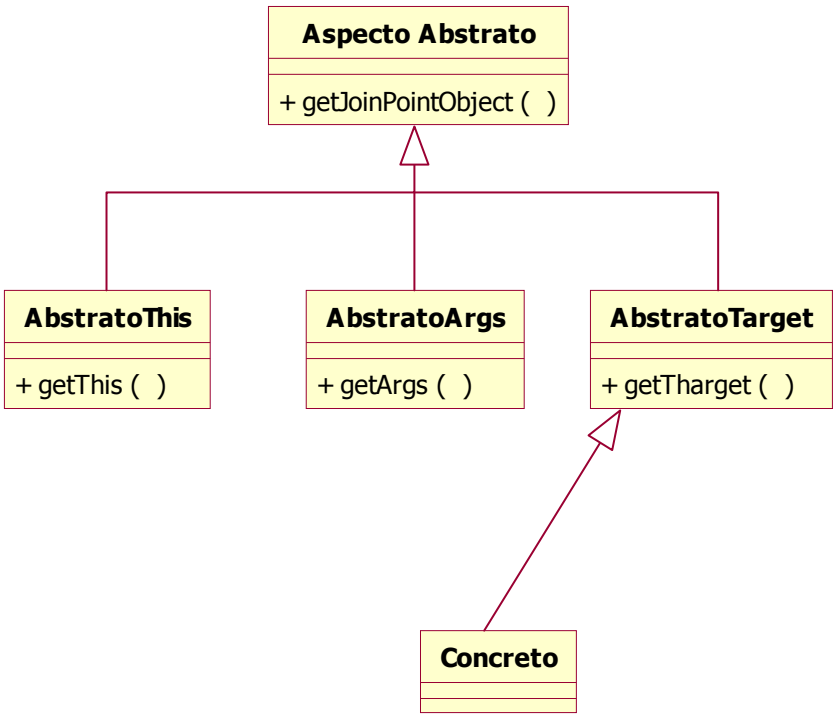

Figura 2 - Projeto Genérico Aspecto

Um detalhe interessante ocorre na hierarquia do aspecto Desconto utilizando o método abstrato `getJoinPointObject()` para obter um objeto do contexto de execução que possa ser utilizado para capturar o dado desejado. Esse método é redefinido em cada um dos aspectos especializados (*This*, *Target* e *Args*) com o objetivo de retornar diferentes objetos do contexto de execução, parte do código `Desconto` demonstrada na Figura 3.

```
1  package pedido;
2
3  import org.aspectj.lang.*;
4
5  public abstract aspect Desconto {
6
7      public abstract pointcut desconto();
8
9      before(): desconto() {
10
11             ...
12             Object[] argumento = thisJoinPoint.getArgs();
13             ...
14         }
15     }
16     public abstract Object getJoinPointObject(JoinPoint jp);
17 }
```

Figura 3 - Parte do código Desconto

A implementação concreta do método abstrato `getJoinPointObject()` pode ser observada na Figura 4 onde captura o objeto desejado com o `getTarget()` retornando-o.

```
1  package pedido;
2
3  import org.aspectj.lang.*;
4
5  public abstract aspect DescontoTarget extends Desconto {
6
7
8          public Object getJoinPointObject(JoinPoint jp){
9
10             return jp.getTarget();
11
12         }
13
14 }
```

Figura 4 - Código abstrato DescontoTarget

O adendo do aspecto Desconto utiliza o método abstrato `getJoinPointObject()` para obter um objeto do contexto de execução a partir do ponto de junção passado por parâmetro. Esse objeto pode ser o T*arget, This* ou um argumento dependendo do aspecto que foi especializado. O método é implementado nos aspectos *Target*, *This* e *Args,* retornando o

objeto apropriado. Caso o adendo do aspecto `Desconto` não utilizasse o método abstrato, ele deveria fixar o objeto capturando do contexto ao qual diminuiria bastante as possibilidades de acoplamento.

No Padrão Capturador de Dados, além dos elementos básicos (aspectos, conjuntos de junção e métodos abstratos), outros elementos podem ser adicionados nessa hierarquia de acordo com o problema e o interesse transversal em que está encapsulado.

As alternativas do padrão formam um conjunto de opções para o acoplamento quando se torna necessário capturar dados do código base. Tais alternativas podem ser vistas como interfaces de entrecortes que requerem determinadas propriedades para o acoplamento.

Após o processo de reúso ser completado e o sistema final obtido deve-se decidir por continuar ou descartar as alternativas de composição que não foram utilizadas. Esta decisão depende da natureza da aplicação que foi desenvolvida, pois se ela possuir uma alta probabilidade de mudança de requisitos torna-se interessante que essas alternativas permaneçam na arquitetura do código, que facilitará instanciar uma nova variabilidade caso a alternativa de composição já exista.

3. LINGUAGENS ORIENTADAS A ASPECTOS

3.1. A Linguagem AspectJ

A linguagem AspectJ (Kiczales et al., 2001) é uma extensão orientada a aspectos de propósito geral da linguagem Java. Foi criada pela *Xerox Palo Alto Research Center* em 1997 e, posteriormente, agregada ao projeto *Eclipse* da IBM EM 2002. Além dos elementos oferecidos pela POO (Programação Orientada a Objetos) como classes, métodos, atributos e etc, são acrescentados novos conceitos e construções ao AspectJ, tais como: aspectos (*aspects*), conjuntos de junção (*pointcuts*), pontos de junção (*join points*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*).

A seguir, são apresentados cada um dos conceitos e construções que compõem o AspectJ.

3.1.1. Pontos de Junção (Join Points)

Para o entendimento do AspectJ é de fundamental importância o conceito de ponto de junção. Como comentados no capítulo anterior, pontos de junção são pontos na execução de um programa, onde os aspectos serão aplicados. O AspectJ pode detectar e operar sobre os seguintes tipos de pontos de junção (Gradecki e Lesiecki, 2003):

- chamada e execução de métodos;
- chamada e execução de construtores;
- execução de inicialização;
- execução de contrutores;
- execução de inicialização estática;
- pré-inicialização de objetos;

- inicialização de objetos;
- referência a campos;
- execução de tratamento de exceções.

Na Figura 5 é apresentado um exemplo apresentado, de um fluxo de execução entre dois objetos, identificando alguns pontos de junção retirado de Soares e Borba, (2002).

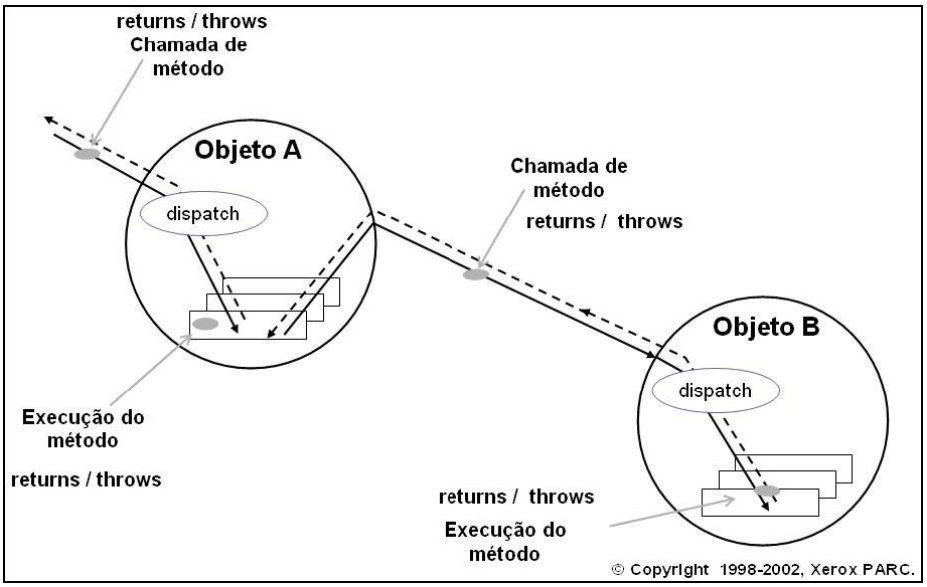

Figura 5 - Pontos de junção de um fluxo de execução

O primeiro ponto de junção é a invocação de um método de um objeto A, ao qual pode retornar sucesso ou lançar uma exceção. O próximo ponto de junção é a execução deste método, que por sua vez também pode retornar sucesso ou lançar uma exceção. Durante a execução do método do objeto A é invocado um método de um objeto B, podendo retornar sucesso ou lançar uma exceção. A invocação e execução destes métodos são pontos de junção.

3.1.2. Conjuntos de Junção (Pointcuts)

Os conjuntos de junção (*pointcuts*) são responsáveis por selecionar ponto de junção, ou seja, eles detectam em que ponto do programa os aspectos deverão interceptar.

Conjunto de Junção (*pointcuts*) comportam-se semelhante aos membros de classes Java. E eles podem ser declarados públicos, privados ou finais, mas não podem ser sobrecarregados. Também podem ser declarados abstratos, mas somente dentro de aspectos abstratos, e ainda podem ser nomeados ou anônimos. A declaração de um pointcut nomeado deve seguir a seguinte sintaxe (Monteiro e Piveta, 2003):

pointcut <Nome> (Argumentos): <corpo>;

onde:

- <Nome>, nome do adendo;
- Argumentos, argumentos passado pelo adendo;
- <corpo>, conjunto de junção a ser escolhido.

Para definir um conjunto de junção utiliza-se construtores de AspectJ nomeados de designadores. Os principais estão listados no Quadro 2:

Quadro 2 - Listagem dos designadores em AspectJ

Designador	Características
Call (Signature)	Invocação do método / construtor identificado por assinatura
Executiom (Signature)	Execução do método / construtor identificado por assinatura
Get (Signature)	Acesso a atributo identificado por assinatura
Set (Signature)	Atribuição do atributo identificado por assinatura
This (Type pattern)	Objeto em execução é instância do padrão tipo
Target (Type pattern)	Objeto de destino é instância do padrão tipo
Args (Type pattern)	Os argumentos são instância do padrão tipo
Within (Type pattern)	O código em execução está definido em padrão tipo

Um pointcut pode ser alinhado em outro pointcut com os operadores e, ou e não representados pelos caracteres: &&, ||, e !. Por exemplo:

```
call(void CadastrarReservas())||call(void AlterarSenha(int));
```

Em AspectJ elementos *wildcards* são utilizados. Estes permitem que em especificações de assinatura (*signature*) sejam definidos o número de caracteres (*) e o número dos argumentos (..). Por exemplo: `public void set*(.., String)`. Isto irá refletir sobre todos os métodos que iniciam com a palavra set e que tenham zero ou mais argumentos como parâmetro. No padrão tipo (*type pattern*) utilizam-se os seguintes *wildcards* listados no Quadro 3.

Quadro 3 - Tipos de Wildcards

Wildcards	Descrição
*	Qualquer seqüência de caracteres não contendo pontos
+	Qualquer subclasse de uma classe
..	Qualquer seqüência de caracteres, inclusive pontos

3.1.3. Adendos (Advices)

Adendos é o código para ser executado em um ponto de junção que é referenciado pelo conjunto de junção. Existem três tipos básicos de adendos: antes, durante e depois (*before*, *around* e *after*). Portanto, de acordo com seus nomes, *before* executa antes, *around* durante e *after* executa depois do ponto de junção onde serão demonstrados nas subseções 3.1.3.1, 3.1.3.2, 3.1.3.3.

O adendo pode modificar a execução do código no ponto de junção, substituir ou passar por ele. Usando o adendo pode-se registrar as mensagens antes de executar o código de determinados pontos de junção, que estão espalhados em diferentes módulos. O corpo de um adendo é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um ponto de junção é alcançado (Monteiro e Piveta, 2003).

3.1.3.1. Adendo Before

O adendo *before* tem a seguinte forma:

before(Parametros) : Nome {corpo}

onde:

- <Nome>, nome do adendo;
- Parametros, parâmetros passado pelo adendo;
- <corpo>, onde será adicionado a programação do entrecorte.

Esta é a forma mais direta entre os tipos, sempre executando antes de qualquer ponto de junção (*join point*, exemplo demonstrado na Figura 6 (Monteiro e Piveta, 2003).

```
public aspect exemploBefore{

pointcut    altSenha(String    Senha):    within(Usuario)    &&    execution(void
AlterarSenha(String,*,*)) && args(Senha,*,*);

before (String Senha):altSenha(Senha){
  System.out.println("Senha atual: "+Senha);
}
}
```

Figura 6 - Exemplo do Advice Before

3.1.3.2. Adendo After

O adendo *after* tem a sintaxe similar à do adendo *before* e está descrita a seguir:

after(Parametros) : pointcut {corpo}

```
public aspect exemploAfter{

pointcut altSenha(String Senha,String Nova): within(Usuario) && execution(void
AlterarSenha(String,String,*)) && args(Senha,Nova,*);

after (String Senha,String Nova):altSenha(Senha,Nova){
  System.out.println("Nova Senha: "+Nova);
}
}
```

Figura 7 -Exemplo do Advice After

Na Figura 7 tem-se um exemplo de um adendo do tipo *after*. No adendo *after* possui duas variações. Isto ocorre porque executará sempre após o retorno de um pontos de junção. Entretanto, neste caso, podem haver exceções, o que impossibilita o código de ser executado por inteiro e, retornar o que lhe foi especificado. Desta maneira, têm-se: o adendo *after returning*, a forma que executa somente depois do retorno normal e o adendo *after trowing* que somente reflete sobre as exceções (Monteiro e Piveta, 2003).

A sintaxe da versão returning é a seguinte:

after(Parametros) returning [(UmParametro)] : pointcut {corpo}

A sintaxe do after throwing é a descrita a seguir:

after(Parametros) throwing [(Um Parametro)] : Pointcut {corpo}

3.1.3.3 Adendo Around

O adendo *around* é uma das características mais importantes da linguagem AspectJ. A sintaxe é a que segue a seguir (Monteiro e Piveta, 2003):

ReturnType around(Parametros) [throws ListOfExceptionTypes] : pointcut{corpo}

Dentro do corpo do *around* existe uma sintaxe que prevê executar o código do ponto de junção original, o *proceed*, que introduz os argumentos expostos pelo conjunto de junção, e deve retornar o tipo do adendo declarado. Sua sintaxe é descrita da seguinte forma:

proceed(Argumentos);

3.1.4. Declarações Inter-tipos (Inter-type Declarations)

O AspectJ prevê uma maneira de alterar a estrutura estática de uma aplicação. Isto ocorre por meio das declarações inter-tipos que são descritas como interesses estáticos (*static crosscutting*). Estas declarações provêm uma construção chamada *Introduction*.

Introduction é um interesse estático, que introduz alterações nas classes, interfaces e aspectos do sistema. Alterações estáticas em módulos não têm efeito direto no comportamento. Por exemplo, pode ser adicionado um método ou um atributo na classe (Gradecki e Lesiecki, 2003).

3.1.5 Aspectos (*Aspects*)

Do mesmo modo que classe é a unidade central em Java, aspecto é a unidade central do *AspectJ*. Aspectos encapsulam conjuntos de junção (*point cuts*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*) em uma unidade modular de implementação. Assim como as classes em Java, os aspectos podem conter atributos, métodos e classes internas.

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*). 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 de junção, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução (Soares e Borba, 2002).

3.1.6 Exemplo de Programa em AspectJ

Nesta subseção é demonstrado um programa básico da Linguagem AspectJ, sendo implementado o HelloWorld.

Na Figura 8 implementa-se um programa normal em Java com a classe `main()` e `dizHello()` onde imprime na tela “Hello World!”

```
3  public class Hello {
4
5      public static void main(String args[]) {
6          Hello world = new Hello();
7          world.dizHello();
8      }
9
10     public void dizHello() {
11         System.out.println("Hello World!");
12     }
13 }
```

Figura 8 – Código base de Java

O programa montado em AspectJ tem por sua finalidade entrecortar a classe `dizHell()` durante a sua execução e imprimir antes do *Hello World* a palavra *after*. Na Figura 9 o aspecto se chama `AspHello` e o adendo recebe o nome de `EstAdendo` onde o conjunto de junção tem que entrecortar o código-base durante a execução e na classe `dizHello()` através do comando `execution(* helloasp.Hello.dizHello(..));`

```
3  public aspect AspHello {
4      pointcut EstAdendo(): execution(* helloasp.Hello.dizHello(..));
5      after() : EstAdendo() {
6      System.out.println("after");
7  }
8  }
```

Figura 9 - Aspecto HelloWorld do AspectJ

Esta é uma pequena demonstração do que um programa em aspecto possa fazer.

3.2. A Linguagem JAsCo

JAsCo (JAsCO, 2007) é uma linguagem de programação orientada a aspecto, cujas contribuições principais são seus aspectos reusáveis e seu mecanismo de composição para combinações dos aspectos. A linguagem JAsCo fornece a integração e a remoção dinâmica dos aspectos. Essa linguagem é mantida tão perto possível à sintaxe e aos conceitos regulares de Java, introduzindo dois novos conceitos: *aspect beans* (simples aspectos) e conectores (*connectors*), ligando o aspecto ao código-base.

Algo mais sobre os novos conceitos: *aspect beans* e conectores:

- Um *aspect bean* é altamente reusável, porque busca descrever *crosscutting,* que são interesses transversais, independentes dos tipos componentes concretos e de APIs.
- Um conector desdobra-se em um ou mais *aspect beans,* dentro de um contexto de componente e especifica uma combinação explícita de seu comportamento aspectual.

A tecnologia JAsCo é baseada em um tempo de execução (*run-time*), que são reservados para adicionar, alterar e remover aspectos quando a aplicação for executada.

JAsCo é uma aproximação avançada de POA que suporta uma escala larga de conceitos de POA, como conjuntos de junção, aspectos independentes, distribuição explícita nos conectores, combinações expressivas do aspecto, o polimorfismo aspectual, aspectos stateful, mixins virtuais e a programação adaptável, descritos na seção sobre POA.

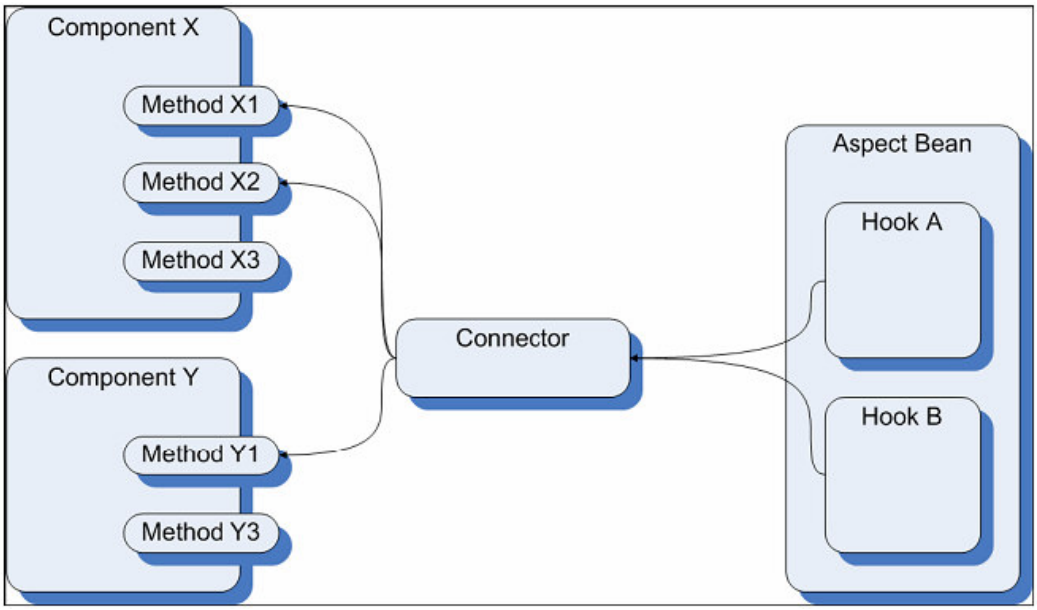

Figura 10- Contexto Esquemático de JAsCO

Para tornar a linguagem de JAsCo operacional, introduziram um novo modelo de componente, o *"aspect-enable"*, fazendo com que o aspecto seja executado durante o tempo de execução do código-base. O modelo de componente de JAsCo permite a adição e a remoção de tempo de execução do aspecto.

Na Figura 10 é mostrado de forma esquemática o contexto da linguagem JAsCo. No lado esquerdo, dois componentes normais de Java são descritos, declarando um número de métodos e alguns eventos. No lado direito, um aspecto é mostrado contendo dois ganchos (*hooks*). Um conector é usado instantaneamente unindo os métodos e os eventos do componente. Observa-se que o aspecto pode também declarar métodos e eventos normais.

Principais benefícios de JAsCo (JAsCO, 2007):

- Independência e reúso de *aspect beans*. não é um código difícil de aplicar-se em um contexto concreto, dando sustentação a polimorfismo.
- Suporte à distribuição de aspectos avançados: meta data e aspectos de aspectos.
- Possibilita a composição avançada do aspecto:

  - Estratégias baseadas em exemplos explícitos dá precedência em um conector. Os aspectos não têm que se relacionar com a precedência do código-base e a precedência de diversos aspectos *cooperating*. Podem mudar aplicações diferentes excedentes destes aspectos.
  - Estratégias explícitas da combinação em um conector, que podem controlar o comportamento combinado de aspectos cooperativos.
- Programação adaptável a conectores e os Aspectos são baseados em conjuntos de junções.

3.2.1. Pontos de Junção (*Join Point*)

Um *aspect beans* é um esquema regular do JAsCO, que pode declarar um ou mais ganchos (*hooks*), logicamente relacionado com uma classe interna. Os ganchos são entidades genéricas e reusáveis que podem ser considerados como uma combinação abstrata de conjunto de junções (*poincut*) e adendos (*advice*). *Aspect beans* são descritos independentemente de um contexto específico e podem ser aplicados sob uma variedade de componentes (NICOARÃ, 2004).

O conector tem o propósito de instanciar o *aspect beans* abstrato em um contexto concreto, e desse modo, ligar os conjuntos de junções abstratos com conjuntos de junções concretos.

Um conector reserva especificar a precedência e a combinação estratégicas entre os aspectos e os componentes.

3.2.2. Conjuntos de Junção (*Pointcuts*)

Os conjuntos de junção em JAsCO são declarados como ganchos (hook), sendo responsável pela especificação de um modo abstrato definindo quando o comportamento do gancho deve ser acionado. Nesse sentido, o gancho construtor é semelhante ao conjunto de junção explicado nos tópicos anteriores. Um gancho construtor contém um ou mais métodos abstratos que denotam o contexto em que o gancho pode ser instanciado. O método abstrato está vinculado aos métodos concretos ou eventos. Quando o gancho é instanciado em um conector, podem-se declarar três tipos de parâmetros no método abstrato (JAsCO, 2007):

- Método pode estar vinculado a dois tipos de parâmetros um do tipo "*String"* e outro do tipo "*int*".
- Também é possível especificar um método abstrato vinculado a qualquer método usando o construtor "..", podendo ser combinado com tipos específicos, mas deverá ser sempre o último argumento.
- Pode ser qualquer outro tipo de argumento criado pelo Java, especificando que o método pode ser sujeito a qualquer argumento do tipo criado.

Um código JAsCo, a primeira vista, tende a entender que seria um Java normal, sendo capaz de declarar variáveis, métodos e até eventos. O próprio comportamento transversal é especificado utilizando o construtor gancho (*hook*), que é um tipo especial de classe interna. Um gancho especifica pelo menos um construtor e um adendo (*advice*), e é capaz de conter qualquer membro de classes Java. Um construtor gancho especifica as condições em que o gancho é acionado. Os parâmetros de um construtor gancho são métodos de parâmetros abstratos, que estão vinculados a métodos concretos ou eventos a um conector. O corpo do construtor especifica a condição para que o gancho entre em ação. Os adendos de um gancho são usados para especificar as várias ações de um gancho quando é acionado.

Destacar um aspecto dentro de um código-base é feito através de conectores. Na Figura 9 temos um exemplo de conector. Sua sintaxe se caracteriza como:

```
static connector NomeConector {
        Aspects.LoggingAspect.LoggingHook hook0 =
            new
aspects.LoggingAspect.LoggingHook(corpo);
        hook0.Tipo();
}
```

Onde:

- ***NomeConector:*** nome do método do conector criado;
- ***Aspects:*** nome do pacote onde se encontra o aspect;
- ***LoggingAspect:*** nome do aspect criado;
- ***LoggingHook:*** nome da classe gancho (Hook)que se encontra dentro do aspecto;
- ***Corpo:*** expressão onde deseja que o aspecto entrecorte;
- ***Tipo:*** tipo desejadodo adendo.

```
static connector LoggingConnector {
        aspects.LoggingAspect.LoggingHook hook0 =
            new aspects.LoggingAspect.LoggingHook(
                    void application.Application.test());

        hook0.before();
}
```

Figura 11 - Conector JAsCO

Um conector contém três tipos de construção: um ou mais ganchos, zero ou mais adendos e qualquer construtor Java.

Na Figura 11 são apresentados alguns parâmetros que podem ser passados no método para a execução.

Os parâmetros do método abstrato estão vinculados aos métodos concretos ou eventos, quando o gancho é instanciado em um conector.

Os parâmetros do método abstrato têm um alcance que se estende por todo o gancho, que pode ser acessado diretamente nos adendos.

Caracterizando a sintaxe do aspecto como e demonstrado na Figura 12:

```
class LoggingAspect {hook LoggingHook {
            LoggingHook(method(Parâmetros)) {
                  Tipo(method) && Tipo(method);}
            around() {Corpo}}}
```

Onde

- ***LoggingAspect:*** nome do aspect;
- ***LoggingHook:*** nome do gancho (Hook);
- ***Parâmetros:*** parâmetros passado pelo method;
- ***Tipo:*** tipo do conjunto de junção;
- ***Corpo:*** corpo do adendo.

```
package aspects;
class LoggingAspect {
    hook LoggingHook {
        LoggingHook(method(..args)) {
            execution(method);
        }
        around() {
            System.out.println("Method executed");
        }
    }
}
```

Figura 12 - Modo de como passar os Parâmetros

Os tipos de parâmetros concretos especificados no método abstrato permitem condicionar o gancho para que apenas os métodos corretos sejam aplicados.

O corpo do construtor especifica a condição que aciona o parâmetro do método abstrato. A palavra chave é desencadeada no gancho cada vez que o método tiver que ser executado (JAsCO, 2007).

As características e seus respectivos comandos estão descritos no Quadro 4.

Quadro 4 - Composição dos Conjuntos de Junção em JAsCO

Palavras Chaves	Características
execution(x)	Ativada durante a execução do método /construtor identificado.
call(x)	Ativada durante a invocação do método / construtor identificado.
cflow (x)	Expressão que avalia se é verdadeiro quando o método está atualmente em execução no controle do fluxo vinculado ao parâmetro do método abstrato x.
withincode (x)	Mais restritiva do que cflow e obriga o método vinculado a x, ser a chamada direta do método atualmente em execução.
target (x)	Avalia quando um tipo objeto dinâmico onde um método foi invocado quando é x ou uma subclasse de x. (Delimitador)

Os conjuntos de junções podem ser alinhados com outros utilizando alguns operadores como: e (&&), ou (||) ou o operador de negação (!), na Figura 13 tem-se um pequeno exemplo de como podem ser empregados os operadores.

```
package aspects;
class LoggingAspect {
    hook LoggingHook {
        LoggingHook(method(..args)) {
            execution(method) && cflow(method);
        }
        around() {
            System.out.println("Method executed");
        }
    }
}
```

Figura 13 - Operadores de comparação

3.2.3. Adendos (*Advices*)

Os adendos são utilizados para especificar o que um gancho deve fazer. Pelo menos um adendo tem que ser especificado, senão o gancho não faz nada. Nota-se que isto não é forçado em uma implementação JAsCO.

Seis tipos de adendos são válidos*: before, around, after, throwing, after returning, around throwing e around returning*(JONCKERS, 2004).

Onde as sintaxes dos adendos é semelhante a sintaxe descrita na subseção 3.2.2..

3.2.3.1. Adendo Before

Um adendo *before* é capaz de especificar um comportamento que deveria ser executado antes do método original que causa o acionamento dos ganchos. Tem-se um exemplo na Figura 14.

```
package aspects;
class LoggingAspect {
    hook LoggingHook {
        LoggingHook(method(..args)) {
            execution(method);
        }
        before() {
            System.out.println("Method executed");
        }
    }
}
```

Figura 14 - Adendo Before do JAsCO

3.2.3.2. Adendo After

A declaração do adendo é constituída de três tipos de adendos, sendo o mais comum o *after,* que especifica o comportamento que é executado, após o ponto de junção onde o gancho é acionado. Tem-se um exemplo na Figura 15.

```
package aspects;
class LoggingAspect {
    hook LoggingHook {
        LoggingHook(method(..args)) {
            execution(method);
        }
        after() {
            System.out.println("Method executed");
        }
    }
}
```

Figura 15 - Adendo after do JAsCO

Os outros dois tipos do adendo *after* têm como finalidade os pressupostos descritos no Quadro 5, tais como:

Quadro 5 - Adendos after JAsCO

Tipo	Descrição
afterReturning	Depois de executar o método se: terminou normalmente (não lançou exceção) e retornou o parâmetro definido no tipo.
afterThrowing	Depois de executar o método se: ele levantou a exceção definida no tipo.

3.2.3.3. Adendo Around

O adendo *around* especifica o comportamento que é executado, durante o ponto de junção onde o gancho é acionado, constituído na Figura 16.

```
package aspects;
class LoggingAspect {
    hook LoggingHook {
        LoggingHook(method(..args)) {
            execution(method);
        }
        around() {
            System.out.println("Method executed");
        }
    }
}
```

Figura 16- Adendo around do JAsCO

Os outros dois tipos do adendo *around* têm como finalidade os pressupostos descritos no Quadro 6.

Quadro 6 - Adendos around JAsCO

Tipo	Descrição
aroundReturning	Durante a execução do método se: terminou normalmente (não levantou exceção) e retorna o parâmetro definido no tipo.
aroundThrowing	Durante a execução do método se: ele levantou a exceção definida no tipo.

Around returning e *around throwing* são semelhantes ao *after returning* e *after throwing* respectivamente, exceto que eles permitem alterações no retorno válido ou decidam não retroagir a exceção lançada.

3.2.4. Aspectos

Um aspecto em JAsCo é capaz de possuir um ou mais ganchos que contêm o *crosscutting*. Um gancho é um participante de um aspecto, e é utilizado para especificar:

- Quando a execução normal de um método ou de um componente deve ser "entrecortada".
- Comportamentos extras que devem ser executados naquele exato momento de tempo.

Para especificar quando isto deve acontecer, ganchos e os métodos são utilizados. Esta parte é implementada na parte do adendo (Advice).

3.2.5. Exemplo de Programa em JAsCO

O programa HelloWorld em JAsCO é exibido nesta seção onde temos o programa Hello em Java com as mesmas funcionalidades descrito na subseção 3.1.6., demonstrado na Figura 17.

```
3  public class Hello {
4
5      public static void main(String args[]) {
6          Hello world = new Hello();
7          world.dizHello();
8      }
9
10     public void dizHello() {
11         System.out.println("Hello World!");
12     }
13 }
```

Figura 17 - Programa HelloWorld em JAsCO

Encontra-se algumas diferença logo de começo em um aspecto desenvolvido em JAsCO que são seu conector e o adendo denominado gancho (hook), sendo no conector que colocamos os conjuntos de junções, Figura 18, e o tipo do adendo e as características do

entrecorte, Figura 19, então citados no aspecto `LoggingAspect` onde temos o `Hook01` e o *after*.

```
static connector LoggingConnector {
        aspects.LoggingAspect.LoggingHook hook0 =
            new aspects.LoggingAspect.LoggingHook(
                    void helloasp.Hello.dizHello());

        hook0.after();
}
```

Figura 18 - Conector do aspecto HelloWorld

```
class LoggingAspect {

    hook Hook01 {

        Hook01(method(..args)) {
            execution(method);
        }

        after() {
            System.out.println("After");

        }
    }
}
```

Figura 19 - Aspecto do programa HelloWorld do JAsCO

3.3. A Linguagem AspectWerkz

AspectWerkz (AspectWerkz,2007) é uma estrutura dinâmica para a Programação Orientada a Aspectos (POA) em Java. Consiste de uma biblioteca do núcleo para o gerenciamento de aspectos dinâmicos. Aproxima-se às chamadas de inserção à biblioteca em tempo de execução a fim de permitir a execução do adendo (Mezini,2005).

Sendo uma Linguagem simples, de elevada performance, dinâmica, leve, e de grande poder de programação orientada a aspectos para Java, oferece simplicidade para a integração em projetos novos ou existentes de POA. (AspectWerkz, 2007)

AspectWerkz utiliza a modificação do *bytecode* para criar suas classes no projeto, no tempo de carga da classe ou no tempo de execução. Os ganchos (*hooks)* usam o padrão das APIs de JVM (*Java Virtual Machine*). Os aspectos, os adendos e as introduções são escritos em Java puro e suas classes alvos podem ser escritas em POJOs. Tem-se a possibilidade para adicionar, remover e reestruturar a execução de suas tarefas nas introduções do tempo de execução. Seus aspectos podem ser definidos usando XML (AspectWerkz, 2007).

È fornecida uma API para usar os mesmos aspectos para *proxies*, tendo uma experiência transparente, permitindo uma transição para os usuários familiares com os *proxies* (AspectWerkz, 2007).

A arquitetura de AspectWerkz contem quatro novos objetivos diferentes(AspectWerkz, 2007):

- Performance: o *weaver* de AspectWerkz é baseada completamente na compilação estática, significando que o código escrito é rápido. Os criadores do AspectWerkz introduziram um *microbenchmark*, que simplesmente o comparasse com todas as outras execuções principais de POA.

- Extensões: recipiente novo. É muito aberto e extensível. Reserva *plugings* em extensões diferentes do modelo do aspecto que faz funcionar o aspecto da outra estrutura de AOP dentro do recipiente extensível do aspecto de AspectWerkz.

- Dinâmico: o AspectWerkz permite que o usuário modifique seus aspectos em tempo de execução. Esta execução foi baseada em *HotSwap,* e garante que o desempenho da aplicação seja o mesmo quando um aspecto for *undeployed,* como era antes de ser iniciado.

3.3.1. Pontos de Junção (Join Point)

Pontos de Junção (*Join Points*) são pontos bem definidos no fluxo do programa. Em AspectWerkz a construção de pontos de junção é representada pela classe *JoinPoint*. Um exemplo de *JoinPoint* está disponível nos *advices,* e pode ser usado ao *introspect* e recuperar

o tempo de execução sobre a classe. O exemplo criado é registrado assim que o *join point* for chamado podendo ser recuperado posteriormente (Boner J., 2003).

Os seguintes tipos de *join point* que podem ser definidos no AspectWerkz são: *execution, call, set*, *get* e *cflow*. Para definir *pointcuts*, uma linguagem de teste padrão é usada desde que seja muito similar àquela de AspectJ. A linguagem de teste padrão também pode ser usada pôr p*ointcuts* complexos (Mezini.2005).

3.3.2. Conjuntos de Junção (Pointcuts)

Uma das características mais importantes de uma estrutura de POA é a possibilidade de escolher o ponto de junção. Onde conduziu à execução de uma linguagem padrão que utiliza a sintaxe de AspectJ (Mezin,2005).

A declaração de um conjunto de junções deve ser nomeado da seguinte maneira

```
<aspectwerkz>
    <system id="NomeTrabalho">
        <package name="Pacote">
            <aspect class="ClassAspecto">
                <pointcut name="Nomepointcut" expression="Corpo"/>
                <advice name="NomeAdendo" type="Tipo" bind-
to="Link"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Onde:

- NomeTrabalho: seria o nome do projeto criado em Java;
- Pacote: pacote utilizado do projeto Java;
- ClassAspecto: nome da classe em aspecto criada;
- Nomepointcut: nome do conjunto de junção;
- Corpo: contem as expressões para o entrecorte do aspecto;
- NomeAdendo: nome do método criado na classe aspecto;

- Tipo: coloca-se o tipo do adendo
- Link: nome do conjunto de junção que será utilizado geralmente é o mesmo que o “Nomeponitcut”.

Na Figura 20 é mostrado como um ponto de junção pode ser definido, sendo desenvolvido na extensão XML do AspectWerkz.

```
<aspectwerkz>
    <system id="AspectWerkzTcc">
        <package name="pacpedido">
            <aspect class="MeuDesconto">
                <pointcut name="greetMethod"
      expression="execution(* pacpedido.Pedido.adicionarItem(..))"/>
                <advice name="beforeDesconto" type="around"
      bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Figura 20 - Join Points em AspectWerkz

A composição dos conjuntos de junção q possibilita encontrar os pontos de junção nos métodos ou aplicativos, seguindo a regra definida no padrão passado como argumento, onde são definidos por (Mezini, 2005) e explicados no Quadro 7.

Quadro 7 - Composição dos Conjuntos de Junção

Composição	Descrição
execution(* *. *(..))	(padrão de método ou construtor) **-** Intercepta a execução de um ponto de junção definido para um método ou um construtor.
set(* *. *(..))	(padrão de atributos) **-** Intercepta a execução de um ponto de junção definido para um atributo quando este é modificado.
get(* *. *(..))	(padrão de atributos) **-** Intercepta a execução de um ponto de junção definido para um atributo quando este é lido.
call(* *. *(..))	(padrão de método ou construtor) **-** Intercepta a chamada de um ponto de junção definido para um método ou um construtor.
within(* *. *(..))	(tipo de parâmetro) **-** Usado para delimitar o *scopo* de quais tipos podem ser utilizados em conjuntos de junção do tipo *call*() ou *execution*(). Também pode ser utilizado para delimitar as classes que entram em uma operação de *mixin*.

A ajuda de curingas nos métodos dos conjuntos de junções permitem anexar um pequeno desvio aos comportamentos das assinaturas dos elementos dos conjuntos de junção, localizados no Quadro 8.

Quadro 8 - Coringas do AspectWerkz

Coringas	**Descrição**
*	Permite aceitar qualquer assinatura para um elemento do pacote. É aplicável a somente um nível do mesmo.
..	Para declaração de classes, significa que quaisquer elementos depois dos dois pontos são válidos (neste caso, só pode ser usado como último elemento da declaração do pacote). Para parâmetros de métodos, significa que a quantidade de parâmetros passados pode ser variável.
+	Usado para selecionar apenas objetos que são subclasses do valor selecionado ou instâncias do mesmo.

Em AspectWerkz encontra-se alguns operadores que nos permitem unir um ou mais padrões de conjuntos de junção, criando uma nova regra composta entre os mesmo descritos no Quadro 9.

Quadro 9 - Operadores do AspectWerkz

Operadores	Desscrição
!	Nega um elemento de um Pointcut, realizando a característica contraria a que foi declarada.
|| ou OR	Une dois ou mais Pointcuts, permitindo que a regra seja aplicada se qualquer uma das alternativas for verdadeira. Geralmente também é usado para criar Pointcuts Mixados, ou seja, unindo dois Pointcuts de tipos diferentes (por exemplo, unindo um Pointcut de método e um de atributo).
&& ou AND	Une dois ou mais Pointcuts, permitindo que a regra seja aplicada somente se todas as alternativas forem verdadeiras.

3.3.3. Adendos (Advices)

O requisito funcional será implementado dentro do adendo esperando para ser anexado a um ponto de junção. Trazendo para POO, seria um método que executaria o ponto

de junção (lugar no Objeto) definido pelo conjunto de junção (regra definida), quando fosse interceptado ou entrecortado. Até mesmo por isso, é comum que um adendo venha anexado a um conjunto de função. Na Figura 21 contém adendo Desconto que por sua vez escreve na tela conforme as regras definidas na parte XML do código onde será mostrado mais abaixo.

```
package desconto;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspect {

    public void Desconto(JoinPoint joinPoint) {
        System.out.println("Lugar onde entrecortado conforme a regra");
    }

}
```

Figura 21 - Aspecto simples em AspectWerkz

3.3.3.1. Adendo Before

Este adendo sempre executará antes de qualquer ponto de junção, tendo o seu formato mostrado na Figura 22. Tendo sua sintaxe semelhante a apresentada no subseção 3.3.2..

```
<aspectwerkz>
    <system id="AspectWerkzTcc">
        <package name="pacpedido">
            <aspect class="MeuDesconto">
                <pointcut name="greetMethod"
      expression="execution(* pacpedido.Pedido.adicionarItem(..))"/>
                <advice name="beforeDesconto" type="before"
      bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Figura 22 - Adendo Before em AspectWerkz

3.3.3.2. Adendo After

Este adendo sempre executará depois de qualquer ponto de junção, tendo o seu formato mais simples implementado na Figura 23. Contendo uma sintaxe igual a apresentada no subseção 3.3.2..

```
<aspectwerkz>
    <system id="AspectWerkzTcc">
        <package name="pacpedido">
            <aspect class="MeuDesconto">
                <pointcut name="greetMethod"
      expression="execution(* pacpedido.Pedido.adicionarItem(..))"/>
                <advice name="beforeDesconto" type="after"
      bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Figura 23 - Adendo After em AspectWerkz

AspectWerkz tem mais dois tipos de adendos, descritos no Quadro 10.

Quadro 10 - Adendos after aspectWerkz

Tipo	Descrição
afterReturning	Depois de executar o método se: terminou normalmente (não levantou exceção) e retornou o parâmetro definido em *type*.
afterThrowing	Depois de executar o método se: ele levantou a exceção definida em *type*.

3.3.3.3. Adendo Around

Este adendo sempre será executado durante a execução de qualquer ponto de junção, representado na Figura 24. Adquirindo a mesma sintaxe demonstrada na subseção 3.3.2..

```
<aspectwerkz>
    <system id="AspectWerkzTcc">
        <package name="pacpedido">
            <aspect class="MeuDesconto">
                <pointcut name="greetMethod"
      expression="execution(* pacpedido.Pedido.adicionarItem(..))"/>
                <advice name="beforeDesconto" type="around"
      bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Figura 24 - Adendo Around em AspectWerkz

3.3.4. Declaração Inter-Tipos

A definição do modelo AspectWerkz tem sua funcionalidade bem próxima a da Linguagem de AspectJ, passando por algumas mudanças importantes, até que atingiu o seu estado atual tendo a mesma essência do modelo AspectJ (Bonér J., 2004).

3.3.5. Aspectos

Em AspectWerkz, uma classe de aspecto é uma classe originada do Java que é definida para ser um aspecto. Isto é feito adicionando um comentário de JavaDoc que contenha o *doclet* do *@Aspect* na frente da classe. Porém, um ou os mais métodos no corpo da classe é definido para ser um método do adendo (Mezini.2005).

Um adendo é um método especial, que executa a funcionalidade do adendo. O método do adendo é forçado para ser um membro de sua classe do aspecto. O método do adendo também espera um exemplo do ponto de junção como um parâmetro, e pode retornar um *Java lang Objeto*. O exemplo do ponto de junção está passado sobre o adendo e pelo ponto de junção quando o adendo é executado. Com o parâmetro do ponto de junção, o adendo pode inspecionar o contexto do ponto de junção (Mezini.2005).

Atualmente, AspectWerkz suporta cinco tipos diferentes de adendos: *around, before, after, after returning* e *after throws*.

AspectWerkz fornece um modelo para sua definição de XML que define aspectos no formato de *aop.xml* e separa a definição dos aspectos de sua execução. Uma vantagem desta separação é que a definição do aspecto pode também, por exemplo, na altura da distribuição de componentes de Java/J2EE em usuários da aplicação (Mezini, 2005).

3.3.6 Exemplo de Programa em AspectWerkz

A implementação de um HelloWorld na Linguagem AspectWerkz é demonstrado nesta subseção.

Na Figura 25 implementa-se um programa normal em Java com a classe main() e dizHello() onde imprime na tela “Hello World!”

```
3  public class Hello {
4
5      public static void main(String args[]) {
6          Hello world = new Hello();
7          world.dizHello();
8      }
9
10     public void dizHello() {
11         System.out.println("Hello World!");
12     }
13 }
```

Figura 25 - HelloWorld em AspectWerkz

O programa em AspectWerkz tem por sua finalidade entrecortar a classe dizHell() durante a sua execução e imprimir antes do *Hello World* a palavra *after greeting...*. Diferente do AspectJ no AspectWerkz tem-se um arquivo em XML, Figura 26, onde é posto todas as

informações necessárias para ocorrer o entrecorte, a classe em Java MyAspect tem-se o corpo do adendo como demonstrado na Figura 27.

```
<aspectwerkz>
    <system id="AspectWerkzExample">
        <package name="heloo">
            <aspect class="MyAspect">
                <pointcut name="greetMethod" expression="execution(* helloasp.Hello.dizHello(..))"/>
                <advice name="afterGreeting" type="after" bind-to="greetMethod"/>
```

Figura 26 - Arquivo xml do AspectWerkz

```
public class MyAspect {

    public void afterGreeting(JoinPoint joinPoint) {
        System.out.println("after greeting...");
    }
}
```

Figura 27 - Classe que contém o adendo do AspectWerkz

3.4. Comparando as Linguagens

Com o desenvolvimento dos programas básicos nas Linguagens citadas nas seções anteriores nota-se que em AspectJ tem-se somente um arquivo em aspecto onde ele contem todas as informações necessárias para a implementação de uma POA (Programaçao Orientada a Aspectos), em AspectWerkz temos um arquivo em XML onde contem todos os dados necessários e o corpo do adendo se localiza em uma classe Java normal, agora em JAsCO encontra-se um conector onde nele é inserido o pondo do entrecorte e no aspecto temos o gancho (hook) onde seria o Adendo (advice) do AspectJ, contendo o tipo do entrecorte e o tipo do adendo no casso gancho. Porém as três linguagem satisfazem o conceito de POA.

4. IMPLEMENTAÇÃO DO "PADRÃO CAPTURADOR DE DADOS" EM LINGUAGENS ORIENTADAS A ASPECTOS

4.1 Considerações Iniciais

Nesta seção é apresentado alguns conceitos do Padrão Capturador de Dados em cima da hierarquia do aspecto Desconto, que utiliza o método abstrato `getJoinPointObject()` para obter um objeto do contexto de execução que possa ser utilizado para capturar o dado desejado.

4.1. Estudo de Caso

Com a finalidade de mostrar e exemplificar a utilização do padrão Capturador de Dados, será mostrado um estudo de caso de um sistema de Pedidos. O objetivo principal desse sistema é o cadastramento de Pedidos, Produtos e Clientes que após serem inseridos todos os dados, retornará o valor unitário de cada Pedido e também o valor total dos pedidos. Na Figura 28 é mostrado o diagrama de classes do sistema de Pedidos.

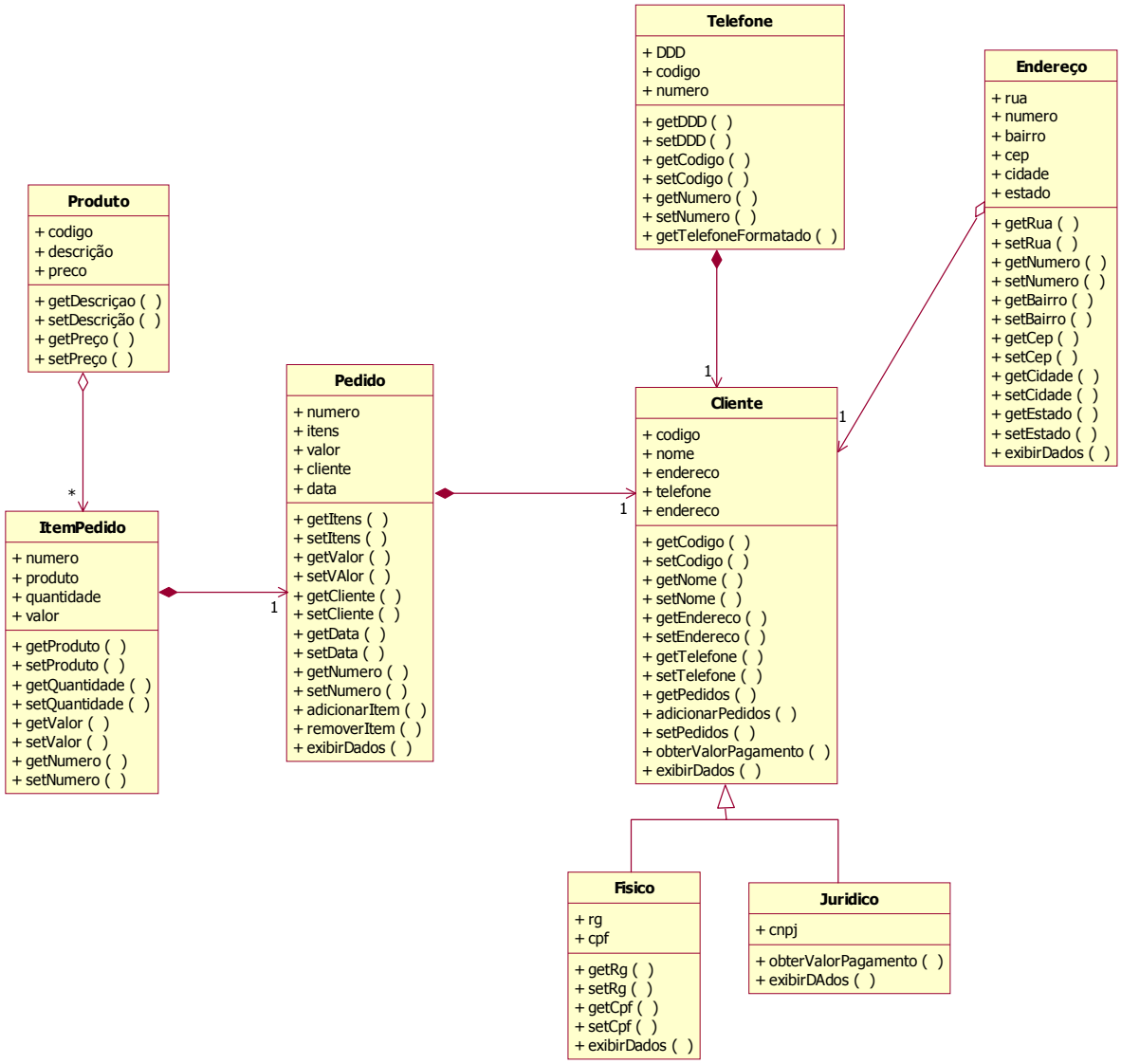

Figura 28 - Diagrama de classes do sistema Pedidos

Como já comentado anteriormente, o padrão Capturador de Dados deve ser utilizado para o projeto e a implementação de FTs. Entretanto, o mesmo padrão também pode ser aplicado em funcionalidades genéricas. No contexto deste trabalho, o padrão é utilizado para se projetar de forma genérica um interesse chamado “Cálculo de Descontos”. O objetivo desse interesse é fornecer descontos para clientes no momento em que um item é adicionado ao pedido de um cliente jurídico. Na Figura 29, é mostrado o projeto desse interesse genérico de acordo com o padrão Capturador de Dados.

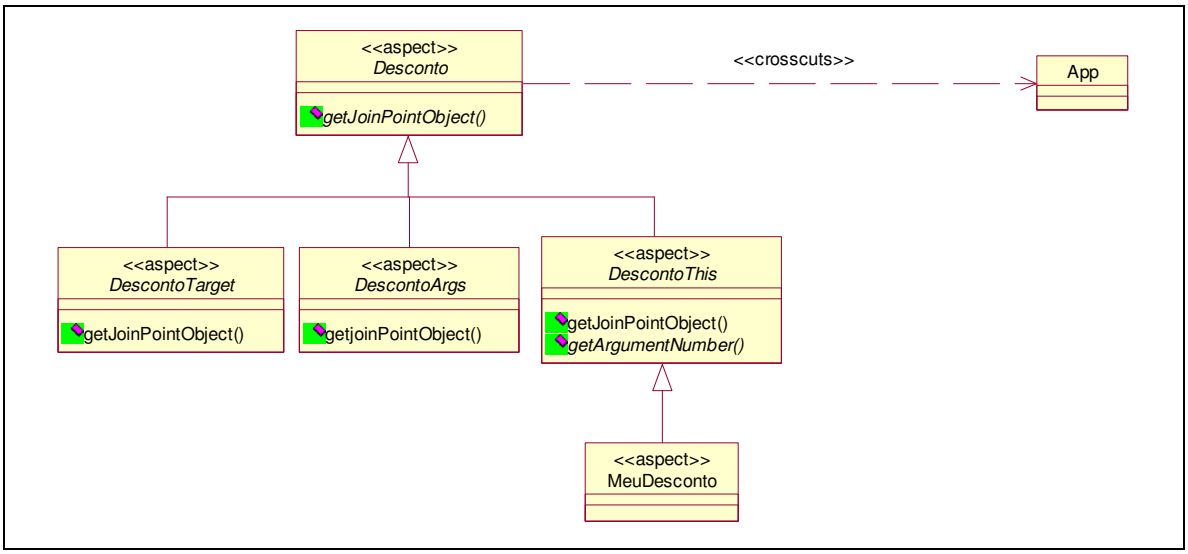

Figura 29- Diagrama Calculo Desconto

O FT consiste em três alternativas de composição localizados no Quadro 11:

Quadro 11 - Alternativas de composição do Capturador de Dados

Nome	Descrição
This	Esta alternativa consiste em estender o aspecto `DescontoThis` e fornecer um ponto de junção que a partir do seu objeto `this` pode-se obter o valor a ser modificado. Também deve ser possível atribuir o valor modificado a esse mesmo objeto. Esse ponto de junção pode ser uma chamada ou execução de um método, ou o acesso a um atributo.
Target	Semelhante à alternativa anterior, porém o aspecto a ser estendido é o `DescontoTarget` e objeto `target` do ponto de junção fornecido é que será utilizado para obtenção e atribuição do valor modificado.
Args	Semelhante às duas alternativas anteriores, porém o aspecto a ser estendido é o `DescontoArgument` e o primeiro argumento do ponto de junção fornecido é que será utilizado para obtenção e atribuição do valor modificado.

É possível no processo de escolha de alternativas de composição incluir uma atividade de exclusão das alternativas que não foram escolhidas, deixando a aplicação apenas com o código necessário.

4.3. Implementação em AspectJ

Nesta seção é mostrada a implementação do padrão Capturador de Dados na linguagem AspectJ. Onde o `getJoinPointObject()` é redefinido em cada um dos aspectos especializados (`DescontoThis`, `DescontoTargt` e `DescontoArgs`) tendo como objetivo de retornar diferentes objetos do contexto de execução. Na Figura 30 é mostrado o código fonte do aspecto Desconto, onde o adendo do aspecto utiliza o método abstrato `getJoinPointObject()` para obter um objeto do contexto de execução a partir do ponto de junção passado por parâmetro.

```
1  package pedido;
2  import org.aspectj.lang.*;
3  public abstract aspect Desconto {
4      public abstract pointcut desconto();
5      before(): desconto() {
6          Pedido pedido = (Pedido)getJoinPointObject(thisJoinPoint);
7          if (pedido.getCliente().getClass().toString().indexOf("Juridico") != -1){
8              Object[] argumento = thisJoinPoint.getArgs();
9              ItemPedido item = (ItemPedido)argumento[0];
10             item.setValor(new Float(item.getValor() * 0.80).floatValue());
11         }
12     }
13     public abstract Object getJoinPointObject(JoinPoint jp);
14 }
```

Figura 30 - Aspecto Desconto AspectJ

O objeto pode ser *target, this* ou um argumento, que irá decidir qual será o aspecto que foi especializado sendo implementado nos aspectos `DescontoTarget`, `DescontoThis`, `DescontoArgs` que retornam o objeto apropriado como podem ser visto na Figura 31 o método `DescontoArgument` onde o objeto é capturo utilizando o método `getArgs()`.

```
1 package pedido;
2 import org.aspectj.lang.*;
3 public abstract aspect DescontoArgument extends Desconto {
4     public Object getJoinPointObject(JoinPoint jp){
5         Object[] objects = jp.getArgs();
6         return objects[getArgumentNumber()];
7     }
8     public abstract int getArgumentNumber();
9 }
```

Figura 31 - Aspecto DescontoArgument AspectJ

No aspecto `DescontoThis` é utilizado no método `getThis()` para capturar o objeto desejado. Na Figura 32 tem-se o aspecto implementado.

```
1 package pedido;
2 import org.aspectj.lang.*;
3 public abstract aspect DescontoThis extends Desconto {
4     public Object getJoinPointObject(JoinPoint jp){
5         return jp.getThis();
6     }
7 }
```

Figura 32 - Aspecto DescontoThis AspectJ

No aspecto `DescontoTarget` é utilizado método `getTarget()` para capturar o objeto desejado. Na Figura 33 tem-se o aspecto implementado.

```
1 package pedido;
2 import org.aspectj.lang.*;
3 public abstract aspect DescontoTarget extends Desconto {
4         public Object getJoinPointObject(JoinPoint jp){
5             return jp.getTarget();
6         }
7 }
```

Figura 33 - Aspecto DescontoTarget AspectJ

Todos estes métodos são abstratos para adquirir o objeto desejado tem-se que implementar um método concreto que chama `MeuDesconto` que se encontra na Figura 34, onde nele é escolhido um dos três tipos de capturador de dados.

O aspecto irá entrecortar onde método de entrecorte, que no caso foi escolhido o *call*. O motivo da escolha é o entrecorte, pois acontece na chamada do método. Assim impede que ele execute antes do aspecto.

```
1 package pedido;
2 public aspect MeuDesconto extends DescontoTarget {
3       public  pointcut desconto():
4           call(* Pedido.adicionarItem(ItemPedido));
5 }
```

Figura 34 - Aspecto MeuDesconto AspectJ

No Quadro 12 mostra-se o que cada método retorna contendo a sua descrição(Aspectos, 2007).

Quadro 12 - Descrição dos metodos utilizados

Método	Descrição
Object[] getArgs()	retorna os argumentos da função, semelhante ao que se obtém com o ponto de corte args().
Object getTarget()	retorna o objeto que foi alvo do ponto de corte, semelhante a ponto de corte target() - por exemplo, no caso de uma chamada a método, o alvo é o objeto que recebe a chamada do método.
Object getThis()	retorna o objeto onde está o ponto de corte, semelhante a ponto de corte this() - por exemplo, no caso de uma chamada a método, o objeto é aquele que está fazendo a chamada.

4.4. Implementação em JasCo

A implantação do método Capturador de Dados na linguagem JAsCo é descrita nesta seção pelos aspectos `DescontoThis`, `DescontoTargt` e `DescontoArgs` onde o método `getJoinPointObject()` é definido, para que possa capturar diferentes tipos de dados do código-base. Na Figura 35 temos o aspecto abstrato `Desconto` que em seu adendo utiliza o método `getJoinPointObject()`,que recebe um objeto de contexto passado por parâmetro definido a partir do ponto de junção para assim conseguir os dados desejados.

```
package aspectos;
import pedido.*;
abstract class Desconto {
    hook HookDesconto {
        HookDesconto(method(..args)) {
            execution(method);
        }
        before() {
            Pedido pedido = (Pedido)getJoinPointObject(thisJoinPointObject);
                if (pedido.getCliente().getClass().toString().indexOf("Juridico") != -1){
                        Object[] argumento = thisJoinPoint.getArgumentsArray();
                        ItemPedido item = (ItemPedido)argumento[0];
                                        item.setValor(new Float(item.getValor() * 0.80).floatValue());
                    }
                }
        }
    public abstract Object getJoinPointObject(Object obj);
}
```

Figura 35 - Desconto em JAsCO

O objeto de retorno podem ser de três tipos como descrito na sessão anterior, retornando o objeto apropriado, na Figura 36 o aspect `DescontoArgument` esta capturando o dado do código-base utilizando o método `getArgumentsArray()`, semelhante ao método `getArgs()` do AspectJ.

```
package aspectos;
import jasco.runtime.MethodJoinpoint;
abstract class DescontoArgument extends Desconto {
    public Object getJoinPointObject(MethodJoinpoint jp){
        Object[] objects = jp.getArgumentsArray();
        return objects[getArgumentNumber()];
    }
    public abstract int getArgumentNumber();
}
```

Figura 36 - DescontoArgument em JAsCO

No aspecto `DescontoThis` não foi possível a captura do elemento desejado onde por vez o adendo do JAsCo não da suporte para a captura do mesmo (getThisnot, 2004).

Para a captura utilizando o aspecto `DescontoTarget` é utilizado o método `getCalledObject()` capturando o dados no código-base descrito na Figura 37.

```
package aspectos;
import jasco.runtime.MethodJoinpoint;
class DescontoTarget extends Desconto{
    public Object getJoinPointObject(Object jp){
            MethodJoinpoint aux = (MethodJoinpoint)jp;
            return aux.getCalledObject();
    }
}
```

Figura 37 - DescontoTarget em JAsCO

Para concretizarmos os aspectos abstratos tem-se o MeuDesconto contendo o gancho(hook) vazio, descrito na Figura 38, somente para a concretização dos demais aspectos relacionados.

```
package aspectos;
class MeuDesconto extends DescontoTarget{
        hook HookMeu extends HookTarget {
    }
}
```

Figura 38 - Aspecto Concredo MeuDesconto em JAsCO

Na linguagem JAsCO são definidos os pontos de junção e os conjuntos de junção para que possa haver entrecorte no código-base. É definido em uma classe chamada de conector como descrito na Figura 39.

```
static connector aop {
    aspectos.MeuDesconto.HookMeu hook0 =
    new aspectos.MeuDesconto.HookMeu(* pedido.Pedido.adicionarItem(*));
}
```

Figura 39- Conector JAsco

4.5. Implementação em AspectWerkz

A linguagem AspectWerkz, utilizando o método Capturador de Dados, é descrita nesta seção para a capturar diferentes tipos de dados no código-base utilizando dos aspectos `Desconto`, `DescontoThis`, `DescontoTargt` e `DescontoArgs` contendo o método `getJoinPointObject()` definido para manipular os dados capturados pelos aspectos. Na Figura 40 demonstramos o aspecto abstrato `Desconto` onde utiliza o método `getJoinPointObject()` para receber os dados capturados e manipulá-los.

```
package pacpedido;
import org.codehaus.aspectwerkz.joinpoint.*;
public abstract class Desconto {
    public void beforeDesconto(JoinPoint joinPoint){
        Pedido pedido = (Pedido)getJoinPointObject(joinPoint);
            if (pedido.getCliente().getClass().toString().indexOf("Juridico") != -1){
                MethodRtti rttiTemp = (MethodRtti) joinPoint.getRtti();
                Object[] argumento = rttiTemp.getParameterValues();
                ItemPedido item = (ItemPedido)argumento[0];
                item.setValor(new Float(item.getValor() * 0.80).floatValue());
            }
        }
    public abstract Object getJoinPointObject(JoinPoint jp);
}
```

Figura 40 - Desconto em AspectWerkz

Tendo como objeto de retorno do tipo `agumento`, `this ou target`, onde são implementados nos aspectos `DescontoThis`, `DescontoTargt` e `DescontoArgs`, a Figura 41 apresenta o aspecto `DescontoArgument` onde utiliza o método `getRtti()` e `getParameterValues()` para poder capturar o mesmo dado com o método `getArgs()` na linguagem AspectJ .

```
package pacpedido;
import org.codehaus.aspectwerkz.joinpoint.*;
public abstract class DescontoArgument extends Desconto{
    public Object getJoinPointObject(JoinPoint jp){
        MethodRtti rttiTemp = (MethodRtti) jp.getRtti();
        Object[] objects = rttiTemp.getParameterValues();
        return objects[getArgumentNumber()];
    }
    public abstract int getArgumentNumber();
}
```

Figura 41 - DescontoArgument em AspectWerkz

No aspecto `DescontoTarget` é utilizado método `getTarget()` para capturar o objeto desejado, na Figura 42 tem-se o aspecto implementado.

```
package pacpedido;
import org.codehaus.aspectwerkz.joinpoint.*;
public class DescontoTarget extends Desconto {
    public Object getJoinPointObject(JoinPoint jp){
        return jp.getTarget();
    }
}
```

Figura 42 - DescontoTarget em AspectWerkz

No aspecto `DescontoThis` é utilizado método `getThis()` para capturar o objeto desejado, na Figura 43 tem-se o aspecto implementado.

```
package pacpedido;
import org.codehaus.aspectwerkz.joinpoint.*;
public class DescontoThis extends Desconto {
    public Object getJoinPointObject(JoinPoint jp){
        return jp.getThis();
    }
}
```

Figura 43 - DescontoThis em AspectWerkz

Estes métodos são abstratos tendo o `MeuDesconto` como método concreto, para escolher qual objeto deseja, onde nele é escolhido um dos três tipos de capturador de dados, demonstrado na Figura 44.

```
package pacpedido;

public class MeuDesconto extends DescontoTarget {
}
```

Figura 44 - Aspecto concreto em AspectWerkz

O aspecto irá entrecortar onde o método de entrecorte, que no caso foi escolhido o *call*. É descrito no arquivo XML como mostra a Figura 45.

```
<aspectwerkz>
    <system id="AspectWerkzTcc">
        <package name="pacpedido">
            <aspect class="MeuDesconto">
                <pointcut name="greetMethod"
      expression="execution(* pacpedido.Pedido.adicionarItem(..))"/>
                <advice name="beforeDesconto" type="before"
      bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Figura 45 - Arquivo XML do AspectWerkz

CONCLUSÃO

Notamos que a deficiência voltada para o paradigma da programação orientada a objetos está sendo sanada pelo modelo de programação orientada a aspecto, onde seus conceitos são aplicados em requisitos não-funcionais como persistência, distribuição, controle de acesso, autenticação, registro de execução de operações (*log*), rastreamento (*tracing*) e concorrência. Contudo, há também interesses transversais funcionais, como por exemplo, regras de negócio que sejam dependentes de um período de tempo.

A composição do código-base é executada junto com a implementação do padrão de projeto denominado Capturador de Dados que será usado junto com as linguagens de programação orientada a aspectos onde o entrecorte precisa capturar dados do código-base.

Neste trabalho alguns testes foram feitos nas linguagens, tais como AspectJ, JAsCo e AspectWerkz, de POA com a designação de examinar se o padrão Capturador de Dados proposto por Camargo (2008) pode ser implementado nas linguagens citadas acima.

É possível a implementação do Padrão Capturador de Dados nas linguagens apresentadas no contexto desta monografia, porém algumas não contêm todos os requisitos necessários para o capturamento de dados, como falta de métodos na captura de dados como descrito na Tabela 13 abaixo.

Tabela 1 - Comparação das Linguagens AspectJ, JAsCO e AspectWerkz

	AspectJ	AspectWerkz	Jasco
Declarações inter-tipo	Sim	Sim	Não
getTarget()	getTarget()	getTarget()	getCalledObject()
getThis()	getThis()	getThis()	Não possui
getArgs()	getArgs()	getRtti() + getParameterValues()	getArgumentsArray()
thisJoinPoint	thisJoinPoint	JoinPoint	MethodJoinPoint

Sendo assim conclui-se que nem todas as linguagens satisfazem a implementação do padrão Capturador de Dados devido ao contexto inserido na programação das mesmas.

No presente trabalho as dificuldades encontradas foram: o acervo científico é pequeno sobre algumas linguagens apresentadas e quando encontrado é em língua inglesa e de pouco esclarecimento tendo uma grande carga de palavras específicas. A dificuldade de obtenção de informação se situa na falta de estudo sobre determinadas linguagens. A falta de manuais para a instalações das linguagens dizendo quais versões são necessárias para a sua instalação. Mal esclarecimento nas linhas de código da criação dos programas.

Como trabalhos futuros, propomos estender a comparação com outras linguagens orientadas a aspectos,como AspectC++ e AspectS.

Propomos também submeter os programas a testes de desempenho e realizar experimentos para delimitar o contexto de aplicabilidade do padrão.

REFERÊNCIAS

ASPECTOS. **Programação Orientada a Aspectos Com AspectJ.** Disponível em: < http://www.aspectos.org/courses/aulasaop/curso_poa.html>. Acesso em 19 de maio de 2007.

ASPECTWERKZ. **AspectWerkz – Plain Java AOP**. Disponível em: < http://aspectwerkz.codehaus.org/>. Acesso em 10 de junho 2007.

Bonér J.. **AspectWerkz – dynamic AOP for Java**. BEA Systems. 2003**.**

CAMARGO, V.V. **Frameworks Transversais: Definições, Classificações, Arquitetura e Utilização em um Processo de Desenvolvimento de Software**. Tese de Doutorado, ICMC-USP, São Carlos, 2006.

CAMARGO, V.V., MASIERO, P.C. **A Pattern to Design Crosscutting Frameworks. In: Proceedings of the Annual ACM Symposium on Applied Computing**, Fortaleza, Brasil, 2008. (*Aceito para publicação*)

CAMARGO, V.V., MASIERO, **P.C. Frameworks Orientados a Aspectos.** In: Anais do Simpósio Brasileiro de Engenharia de Software (SBES´05), Uberlândia, Brasil, 2005.

Dinkelaker T., Mezini M.. **Advanced Method Bytecode Management For Steamloom**. Technische Universitat Darmstadt. 2005

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

JAsCO**. JAsCo Quick Reference for 0.8.6**. Disponível em: < https://ssel.vub.ac.be/jasco/documentation:quick>. Acesso em 8 de agosto 2007.

JONCKER, Viviane, Apprenticeship Report: **Integrating JAsCo Artifacts within the Concern Manipulating Environment**. 2004. Laboratório de Engenharia de software e sistemas, 2004.

MONTEIRO, Elaine da Silva; PIVETA, Eduardo Kessler, **Programação Orientada a Aspectos em AspectJ**. 2003. Curso de Sistemas de Informação - Centro Universitário Luterano de Palmas (CEULP/ULBRA), Tocantins, 2003.
NICOARÃ, Angela; GYGER, Johann; ALONSO, Gustavo, **Improving the efficiency of adaptive middleware based on dynamic AOP**. 2004. Department of Computer Science - Swiss Federal Institut of Technology Zürich, 2004.

PROGRAMAÇÃO Orientada a Aspectos; Departamento de Sistemas e Computação - Universidade Federal de Campina Grande, 2007.

KICZALES, G; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.; IRWIN, J. **Aspect-Oriented Programming**, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag LNCS 1241, June 1997.

Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, Viviane Jonckers. **Adaptive Programming** in JAsCo, Vrije Universiteit Brussel
Pleinlaan 2 1050 Brussel, Belgium