UNIVEM
FUNDAÇÃO DE ENSINO “EURÍPEDES SOARES DA ROCHA”

José Uetanabara Júnior

Uma Comparação entre Abordagens de Modelagem Orientada a Aspectos

Marília - SP
Dezembro/2006

UNIVEM
FUNDAÇÃO DE ENSINO “EURÍPEDES SOARES DA ROCHA”

José Uetanabara Júnior

Uma Comparação entre Abordagens de Modelagem Orientada a Aspectos

“Monografia apresentada como requisito do Trabalho de Conclusão de Curso, orientada pelo professor Dr. Valter Vieira de Camargo.”

Marília - SP
Dezembro/2006

AGRADECIMENTOS

Aos meus pais e a minha família, pois seria impossível a realização deste trabalho sem vocês! Agradeço por terem acreditado em mim durante esses longos anos de faculdade!
Ao meu orientador professor Dr. Valter Vieira de Camargo, por ter me suportado durante um ano inteiro fazendo com que a concretização desse trabalho fosse possível!
Aos meus amigos de faculdade dos anos de 2000 até 2006, aos meus amigos de infância, da Dynamis, do futebol são tantos... Ana, Ana Flávia, Ba, Bruna, Eliana, Fernandinha, Flávia, Grazi, Lílian, Luana, Ma, Merley, Miiii, Nat, Nat Rondina, Pat, Adriano, Anderson, Carlim, Danilo, Dedé, Flávio, Guilherme, Gustavo, Gustavo Santana, Gustavo Rondina, Gzus, Henrique, João Mirolha, Naza Nazi, Lâmpada, Porks, Matheus, Rafael, Reinaldo, Roberto, Rodrigo, Thiago, Vitor, Wendel... Agradeço a todos pelas risadas e pelas coisas estúpidas que foram possíveis de serem realizadas graças a vocês!
A DEUS por ter me dado paciência e força para agüentar os inúmeros obstáculos que surgiram durante esses anos...

Obrigado a TODOS!

UETANABARA JÚNIOR, José. **Uma Comparação entre Abordagens de Modelagem Orientada a Aspectos**. 2006. 48f. Monografia (Bacharelado em Ciências da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino "Eurípedes Soares da Rocha", Marília, 2006.

RESUMO

Várias propostas de abordagens para modelar sistemas orientados a aspectos são encontradas na literatura. Entretanto, o estado da arte ainda carece de uma abordagem padrão e que represente adequadamente os mecanismos básicos desse novo tipo de programação. Nesse sentido, esta monografia apresenta uma comparação entre quatro abordagens de modelagem para sistemas orientados a aspectos. O objetivo é fornecer diretrizes e cenários que possam guiar o desenvolvedor na escolha por uma determinada abordagem de acordo com as características de um determinado projeto. A análise se concentrou em abordagens de modelagem para a fase de projeto, embora algumas também ofereçam mecanismos para a fase de análise. Para realizar a comparação foram utilizados seis critérios relevantes. Um estudo de caso foi feito em que as abordagens foram utilizadas na modelagem de um mesmo sistema hipotético de compra de produtos. O interesse transversal modelado foi o de Rastreamento de Operações que constitui um exemplo clássico de interesse transversal bastante apropriado para a comparação que foi feita.

Palavras-chave: Programação Orientada a Aspectos, UML, AspectJ.

UETANABARA JÚNIOR, José. **Uma Comparação entre Abordagens de Modelagem Orientada a Aspectos**. 2006. 48f. Monografia (Bacharelado em Ciências da Computação) – Centro Universitário Eurípedes de Marília, Fundação de Ensino “Eurípedes Soares da Rocha”, Marília, 2006.

ABSTRACT

A lot of proposals of notations for aspect oriented systems can be found in literature. However, the state of the art still lacks of a standard notation that represents adequately the basic mechanisms of this new type of programming. In this sense, this monograph presents a comparison between four notations for modeling systems with aspects. The objective is to supply lines of direction and scenes that can guide the developer in the choice for one determined notation that better represents the characteristics of the project. The analysis was concentrated in notations for the project phase, even so some also offers mechanisms for the analysis phase. To carry through the comparison six criterias had been used. A case study was made where the notations had been used in the modeling of one same hypothetical system of purchase of products. The crosscuting concern shaped was Tracking of Operations that constitutes a classic example of crosscuting concern sufficiently appropriate for the comparison that was made.

Keywords: Aspect Oriented Programming, UML, AspectJ

SUMÁRIO

Introdução **9**
Contexto 9
Motivação 10
Objetivos 10
Organização do Texto 10
CAPÍTULO 2 ..... **11**
Conceitos Fundamentais **11**
2.1 Considerações Iniciais 11
2.2 Conceitos Fundamentais 11
2.2.1 Programação Orientada a Aspectos 11
2.2.2 A Linguagem AspectJ 13
2.2.3 Implementando um interesse transversal utilizando aspectos 14
2.2.4 A Linguagem de Modelagem UML 16
2.3 Considerações Finais 17
CAPÍTULO 3 ..... **18**
Trabalhos Relacionados **18**
3.1 Considerações Iniciais 18
3.2 Abordagens Estudadas 18
3.3 Considerações Finais 29
CAPÍTULO 4 ..... **30**
Critérios de comparação e Comparação **30**
4.1 Considerações Iniciais 30
4.2 Critérios de comparação 30
4.3 Comparação 31
4.4 Considerações Finais 40
Conclusão **41**
Referências Bibliográficas **42**
Anexo **44**

LISTA DE FIGURAS

Figura 2.1 – Entrelaçamento de código (KICZALES, 2001). 12
Figura 2.2 – Interesses encapsulados em um Aspecto (KICZALES, 2001)................... 12
Figura 2.3 – Estrutura de programa no AspectJ (MONTEIRO,2004)............................ 14
Figura 2.4 – Trecho de código OO. 15
Figura 2.5 – Exemplo de um aspecto. 16
Figura 3.1 – Exemplo de uma modelagem utilizando <<*pointcut*>> e <<*aspect*>>...... 19
Figura 3.2 – Exemplo de um Tema-base (parte superior), Tema-transversal (parte inferior) e relacionamento de ligação (parte central).. 22
Figura 3.3 – Modelo resultante.. 23
Figura 3.4 - Exemplo de modelagem com a abordagem de GIMENES *et al*. (GIMENES *et al*., 2004)... 24
Figura 3.5 – Exemplo de um <<*pointcut*>> (GIMENES *et al*., 2004)........................... 25
Figura 3.6 – Exemplo que utiliza a abordagem de GIMENES *et al.* (2004)................. 26
Figura 3.7 – Pacotes *Aspect*, *Connector* e Base (GROHER, BAUMGARTH, 2004). .. 28
Figura 3.8 – Exemplo utilizando os três pacotes (GROHER, BAUMGARTH, 2004).. 28
Figura 3.9 – Exemplo que utiliza a abordagem de GROHER e BAUMGARTH, 2004. 28

LISTA DE TABELAS

Tabela 4.1 – Comparação pelo critério de nível de abstração. 31
Tabela 4.2 – Comparação pelo critério de separação de interesses. 33
Tabela 4.3 – Comparação pelo critério de rastreabilidade. 33
Tabela 4.4 – Comparação pelo critério de composição. 35
Tabela 4.5 - Comparação pelo critério de evolução. 36
Tabela 4.6 - Comparação pelo critério de escalabilidade 37
Tabela 4.7 – Tabela final. 39

Introdução

Contexto

A programação orientada a aspectos (POA) foi criada por Gregor Kiczales em 1997 com a finalidade de permitir que desenvolvedores de software possam modularizar trechos de código que ficam espalhados e/ou entrelaçados na aplicação em uma nova construção denominada Aspecto (KICZALES *et al*., 1997).

Depois de seu surgimento vários conceitos que já estavam bem estabelecidos tiveram que ser revistos à luz desse novo modo de programação como, por exemplo, *frameworks* (CAMARGO, 2006), linhas de produtos (Kiczales e Mezini, 2005) entre outros.

O AspectJ (KICZALES *et al*., 2001) é a mais conhecida para a POA. Existem também outras linguagens como o AspectC (http://www.aspectc.org/) e o HyperJ (http://www.alphaworks.ibm.com/tech/hyperj), mas neste trabalho o AspectJ é utilizado por ser mais difundida.

Embora existam várias linguagens que implementam aspectos como, por exemplo, o AspectC++, o AspectC# e o AspectS, o mesmo não acontece com abordagens de modelagem para sistemas orientados a aspectos. Como a maior parte dos sistemas utiliza a linguagem AspectJ como técnica de implementação, neste trabalho é mostrada uma comparação entre algumas abordagens de modelagem que sejam adequadas a sistemas implementados em AspectJ.

O trabalho tem enfoque na fase de projeto porque essa é a fase mais carente de uma abordagem de modelagem.

Além disso, um bom modelo de projeto auxilia grandemente o desenvolvedor e a implementação bem como ajuda na compreensão do sistema todo.

Motivação

A motivação para a realização desse trabalho é a inexistência de uma abordagem de modelagem padrão para programação orientada a aspectos e a carência de softwares que implementem as abordagens existentes de modelagem orientada a aspectos. Outra motivação é a oportunidade que este trabalho oferece para o aprendizado do funcionamento das abordagens de modelagem orientada a aspecto.

Objetivos

Este trabalho tem como objetivo principal, fornecer subsídios para que o desenvolvedor possa escolher uma determinada abordagem de modelagem dependendo do contexto do projeto que está sendo conduzido. Outro objetivo está no referencial teórico que o trabalho irá deixar que pode ser utilizado para o desenvolvimento de outros trabalhos.

Organização do Texto

Esta monografia está estruturada da seguinte forma: No Capítulo 2 são apresentados alguns conceitos fundamentais sobre POA, AspectJ e UML; no Capítulo 3 são apresentadas as abordagens de modelagem estudadas; no Capítulo 4 são apresentados os critérios de comparação utilizados e é feita a comparação entre essas abordagens e em seguida é feita a conclusão.

CAPÍTULO 2

Conceitos Fundamentais

2.1 Considerações Iniciais

Neste capítulo o objetivo é fornecer um panorama dos conceitos básicos da POA, Aspectj, UML (*Unified Modelling Language*) [BOOCH *et al*., 2000] e a diferença entre uma notação e um perfil UML (BOOCH *et al*., 2000) para que depois seja possível entender o funcionamento das abordagens de modelagem proposta pelos autores estudados [GROHER e BAUMGARTH (2004), CLARKE *et al*. (2005), Gimenes *et al*. (2004), Pawlak *et al*. (2002)].

2.2 Conceitos Fundamentais

2.2.1 Programação Orientada a Aspectos

Proposta por Gregor Kiczales em 1997 (KICZALES *et al*., 1997), a programação orientada a aspectos (POA) é um paradigma de programação que tem como objetivo básico amenizar dois problemas: o espalhamento e o entrelaçamento de código. Esses problemas são causados quando os mecanismos tradicionais da POO são utilizados na implementação de "interesses transversais". Exemplos de interesses transversais são: Persistência, Distribuição, *Logging*, *Tracing* e Criptografia. O espalhamento de código ocorre quando trechos de um determinado interesse está espalhado por vários módulos de um mesmo sistema. O entrelaçamento de código ocorre quando vários trechos de códigos de diferentes interesses estão entrelaçados em um mesmo módulo. Um interesse transversal consiste em trechos de código da aplicação que serão encapsulados em um aspecto. A parte base é a parte da aplicação que é afetada pelo interesse transversal. Para que isso ocorra é necessário que haja uma composição (*weaving*) para gerar a aplicação. A composição consiste em compor o interesse transversal com a parte base para que seja gerada uma aplicação final que possuirá

um ou mais aspectos. A POA permite que os desenvolvedores de software possam separar esses trechos de código na aplicação para que os mesmos sejam encapsulados e modularizados em uma nova construção denominada Aspecto.

Na Figura 2.1 é mostrado o código que implementa o interesse transversal de `Logging` dentro do servidor Tomcat. Cada coluna representa uma classe desse servidor e as linhas alaranjadas representam linhas de código do interesse de *Logging*. É possível observar que as linhas desse interesse encontram-se espalhadas pelas várias classes do sistema.

Na Figura 2.2 tem-se em alaranjado no canto superior esquerdo, a representação dos interesses transversais separados e encapsulados em um Aspecto.

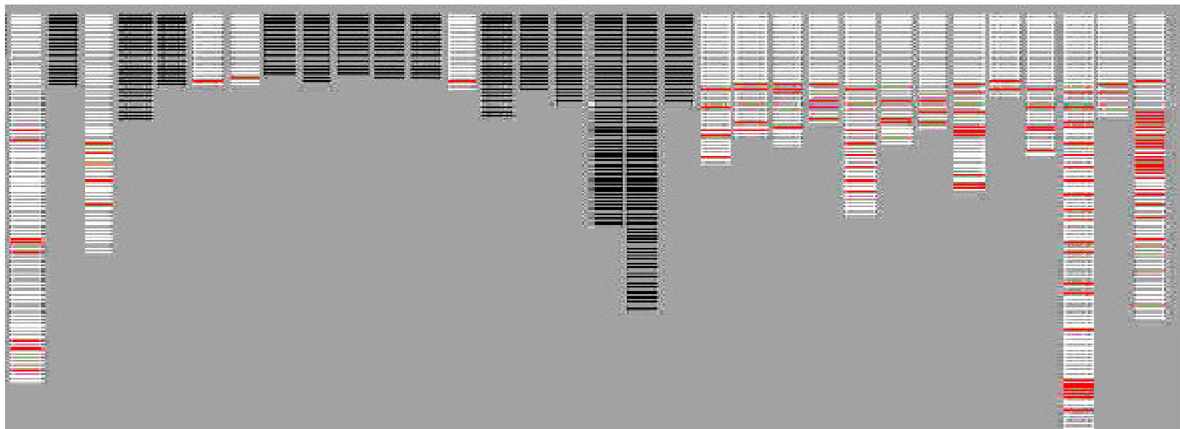

Figura 2.1 – Entrelaçamento de código (KICZALES, 2001).

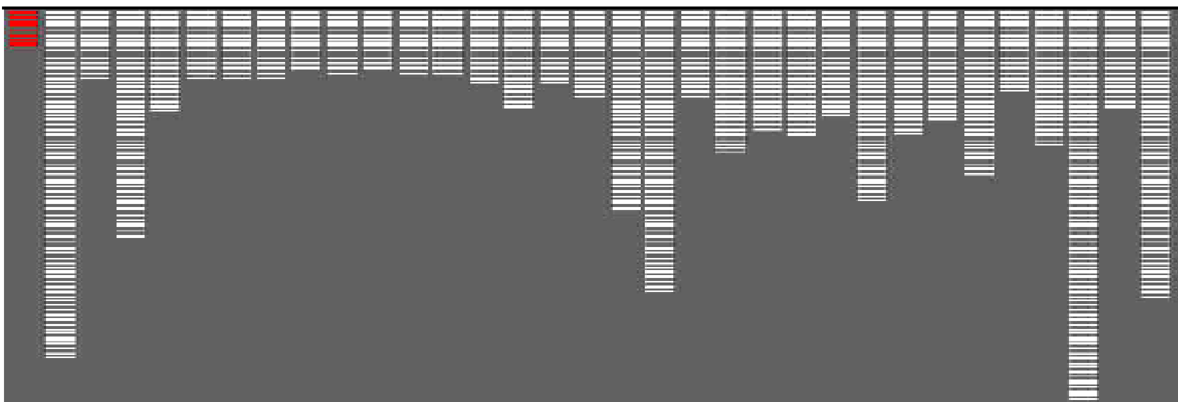

Figura 2.2 – Interesses encapsulados em um Aspecto (KICZALES, 2001).

A POA possui alguns conceitos novos como, por exemplo, pontos de junção (*join points*), conjuntos de junção (*pointcut*), adendo (*advice*), declarações inter-tipos (*inter-type declaration*) e aspectos (*aspects*). Todos esses conceitos serão explicados mais adiante.

2.2.2 A Linguagem AspectJ

A Linguagem AspectJ, que é uma extensão da Linguagem Java, é a Linguagem mais difundida atualmente para implementar programas orientados a aspectos. Com esta inguagem é possível implementar separadamente os interesses transversais e a parte base do sistema. A Linguagem AspectJ possui alguns elementos básicos que são os pontos de junção, os conjuntos de junção, os adendos, as declarações inter-tipos e os aspectos.

Pontos de Junção são pontos existentes dentro de um programa que podem ser afetados pelo aspecto. Vários pontos de junção podem ser agrupados para formar um conjunto de junção.

Um conjunto de junção é formado por vários pontos de junção. Quando a execução de um programa chegar em um desses pontos de junção declarados no conjunto de junção um adendo é executado. Isso permite ao programador descrever aonde e quando um trecho de código deve ser executado durante a execução do programa.

Um adendo é executado em um conjunto de junção. Um adendo possui três tipos básicos: *Before*, *After* e *Around*. No *before*, o trecho de código é executado antes do ponto de junção; no *after* o trecho de código é executado depois do ponto de junção e no *around* o trecho de código é executado simultaneamente ao ponto de junção.

As declarações Inter-Tipo permitem ao programador adicionar métodos, campos ou interfaces nas classes existentes utilizando o aspecto.

A unidade modular “aspecto” encapsula conjuntos de junção, adendos e declarações inter-tipos em uma unidade modular de implementação, que pode alterar a estrutura estática ou dinâmica de um programa. A estrutura estática pode ser alterada usando as declarações

inter-tipos citadas acima. A estrutura dinâmica ocorre em tempo de execução por meio dos pontos de junção que são selecionados pelos conjuntos de junção e por meio da edição dos adendos.

Com essa linguagem é possível definir vários pontos de junção dentro de um programa, que por sua vez, serão agrupados em um conjunto de junção e assim, os pontos de junção poderão ser afetados pelo aspecto. Exemplos de pontos de junção:

- execução de métodos e construtores;
- recebimento de chamadas a construtores;
- acesso a campos;
- e execução de manipuladores de exceção.

Na Figura 2.3 é mostrada a estrutura de um programa no AspectJ com a composição.

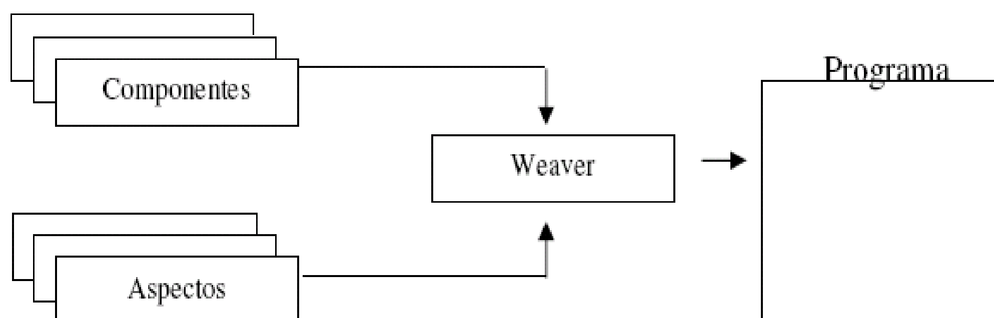

Figura 2.3 – Estrutura de programa no AspectJ (MONTEIRO,2004).

2.2.3 Implementando um interesse transversal utilizando aspectos

Nas Figuras 2.4 e 2.5 é mostrado como implementar um interesse transversal `RastreiaOperacao` utilizando um aspecto. Na Figura 2.4 é possível observar a existência de duas classes: `Cliente` e `RastreiaOperacao`. Dentro da classe `Cliente`, todos os métodos possuem uma chamada ao método `beginningMethod()` da classe `RastreiaOperacao` no início e no fim de cada método. Sendo assim, toda vez que algum método da classe `Cliente` é chamado, o método `beginningMethod()` é executado. Esse

método emite uma mensagem que o método chamado da classe `Cliente` está iniciando sua execução. Quando este método está para terminar sua execução, ele chama o `finishingMethod()` que também está na classe `RastreiaOperacao`. Esse método emite uma mensagem que o método está terminando. Note que o `beginningMethod()` e o `finishingMethod()` se repetem muitas vezes ao longo do código. Criar um aspecto para eliminar essa repetição de código tornaria o código mais legível e modularizaria mais adequadamente o interesse transversal que está sendo tratado.

```
     public Cliente(String n, String cpf)
     {
        RastreiaOperacao.beginningMethod("Cliente(" +n+", "+cpf+")");
        …
        RastreiaOperacao.finishingMethod("Cliente(" +n+", "+cpf+")");
     }
     public Produto removerProduto(int cod,int aux,int i)
     {
        RastreiaOperacao.beginningMethod("removerProduto()");
        …
        RastreiaOperacao.finishingMethod("removerProduto()");
        …
     }
     public Produto comprarProduto(String n,int c)
     {
          RastreiaOperacao.beginningMethod("comprarProduto()");
            …
          RastreiaOperacao.finishingMethod("comprarProduto()");
          …
     }
     public class RastreiaOperacao
        {
         public static void beginningMethod(String methodName)
          {
               System.out.println("O método" + methodName + "está
iniciando sua execução");
          }

         public static void finishingMethod(String methodName)
          {
               System.out.println("O método" + methodName + "está
terminando sua execução ");
          }
        }
```

Figura 2.4 – Trecho de código OO.

Na Figura 2.5 o aspecto chamado `Aspecto1` foi criado. Dentro do `Aspecto1` existe um conjunto de junção denominado `methodToBeTraced()`. Dentro desse conjunto de junção existem os métodos `Cliente.new(..)`, `removerProduto(int,int,int)`, `comprarProduto(String,int)` e `comprarProduto(Produto)` que representam pontos de junção da aplicação que devem ser afetados pelo interesse transversal. Toda vez que alguns

desses métodos forem executados, o aspecto irá entrecortar essa execução e chamará o método `beginningMethod()` que está na classe `RastreiaOperacao` e a mensagem "Método começando" é impressa. Esse comportamento do aspecto está definido no adendo `before()` que atua sobre o conjunto de junção `methodtoBeTraced()`. Quando o método chamado pela classe `Cliente` estiver terminando a sua execução, novamente o aspecto irá afetá-lo, porém desta vez irá chamar o método `finishingMethod()` que está na classe `RastreiaOperacao`, e a mensagem "Método Terminando" é exibida. Isto acontece em conseqüência da existência do adendo `after()` que atua sobre o conjunto de junção `methodToBeTraced()`. Desta forma, todas as chamadas aos métodos da classe `RastreiaOperacao` que estão espalhados pelo sistema podem ser eliminadas.

```
public aspect Aspecto1
{
	public pointcut methodToBeTraced():
		execution (Cliente.new(..)) ||
		execution (void removerProduto(int,int,int)) ||
		execution (void comprarProduto(String,int)) ||
		execution (void comprarProduto(Produto));
	before() : methodToBeTraced()
	{
		RastreiaOperacao.beginningMethod("String
methodName");
	}
	after() : methodToBeTraced()
	{
		RastreiaOperacao.finishingMethod("String
methodName");
	}
}
```

Figura 2.5 – Exemplo de um aspecto.

No final deste trabalho, no anexo, há um exemplo de um simples programa que possui a implementação um aspecto.

2.2.4 A Linguagem de Modelagem UML

A UML (*Unified Modeling Language*) surgiu em junho de 1996 e trouxe recursos eficientes para modelar cada fase do ciclo de vida do software de maneira estruturada e padronizada. A UML é baseada em técnicas orientadas a objetos e destina-se à visualização,

especificação, construção e documentação dos artefatos de um sistema de software (BOOCH *et al.*, 2000).

A UML, em seu meta-modelo, possui blocos de construções básicos, como por exemplo: abstrações, relacionamentos e diagramas. Abstrações representam aspectos estruturais e comportamentais de um sistema, relacionamentos declaram como as abstrações se relacionam e diagramas apresentam um resumo do conjunto de abstrações e seus relacionamentos

A UML é composta por diversos diagramas que permitem descrever as particularidades mais relevantes dos sistemas que são implementados utilizando a abordagem orientada a objetos. Cada um dos diagramas foca uma dada visão do sistema e propositadamente enfatiza algumas particularidades e negligencia outras. A notação UML é independente da linguagem de programação e do processo de desenvolvimento adotados (BOOCH *et al.*, 2000).

Um perfil (*profile*) UML consiste em definir e agrupar alguns mecanismos de extensão para um determinado projeto. Esses mecanismos são os estereótipos (s*tereotypes*), as etiquetas valoradas (*tagged values*) e as restrições (*constraints*). Os estereótipos estendem o vocabulário UML com novos tipos de elementos (exemplo: <<*aspect*>>). Estereótipos podem possuir etiquetas valoradas que é basicamente um par que consiste de um nome e um valor (exemplo: {ponto de junção = método( )}) e restrições, que são condições que devem ser mantidas verdadeiras para que a semântica do estereótipo fique correta (BOOCH *et al.*, 2000).

2.3 Considerações Finais

Foram mostrados neste capítulo os conceitos fundamentais sobre POA, UML e a Linguagem AspectJ. Agora é possível obter entendimento das abordagens de modelagem que são apresentadas no próximo capítulo.

CAPÍTULO 3

Trabalhos Relacionados

3.1 Considerações Iniciais

Neste capítulo são apresentadas algumas das abordagens de modelagem encontradas na literatura juntamente com um estudo de caso que foi realizado com cada uma delas.

3.2 Abordagens Estudadas

Pawlak *et al.* (2002), propuseram um perfil UML para a modelagem de sistemas orientados a aspectos utilizando os estereótipos <<*aspect*>> (para representar os aspectos), <<*pointcut*>> (para representar onde eles afetam), <<*before*>> e <<*after*>> (que representam os adendos) e as etiquetas valoradas `?metodo(..)` ou `!metodo(..)`. O caractere "?" representa que o método é afetado pelo aspecto no contexto de execução, já o caractere "!"representa que o método é afetado pelo aspecto no contexto de chamada.

Na Figura 3.1, pode-se ver um exemplo que utiliza a notação desses autores. É um simples sistema de venda de produtos que modulariza o interesse transversal `RastreiaOperacao`. As classes `Cliente` e `Produto` representam a parte base da aplicação e as unidades modulares `RastreiaOperacao` e `Rastrear` formam o comportamento transversal. O aspecto `Rastrear` afeta as classes `Cliente` e `Produto`. Isso é observado pelo relacionamento de ligação que existe entre o aspecto e as classes `Cliente` e `Produto`. Note que em ambos os relacionamentos é possível ver em anexo o nome do conjunto de junção que irá afetar a classe (no caso `methodToBeTraced`), o estereótipo *`<<pointcut>>`* (que representa que o conjunto de junção `methodToBeTraced` irá afetar a classe que está conectada) e as etiquetas valoradas `?Cliente.new(..)`, `?adicionarProduto(..)`, `?comprarProduto(..)`, `?removerProduto(..)` na classe `Cliente` e

`?Produto.new(..)` na classe `Produto`. Essas etiquetas valoradas definem quais métodos serão afetados pelo aspecto nas respectivas classes. Toda vez que algum desses métodos for chamado pela classe `Cliente` ou `Produto`, antes do método começar a sua execução (observe o *`<<before>>methodToBetraced(`* `)`), o aspecto chama o método *`beginningMethod`*`()` que está na classe `RastreiaOperacao` para ser executado. Após a execução do *`beginningMethod`*`()`, o método afetado continua executando normalmente. Quando o método afetado estiver para terminar a sua execução, o aspecto novamente chama a classe `RastreiaOperacao` e o método *`finishingMethod`*`()` é executado. Após isso, o método afetado termina a sua execução.

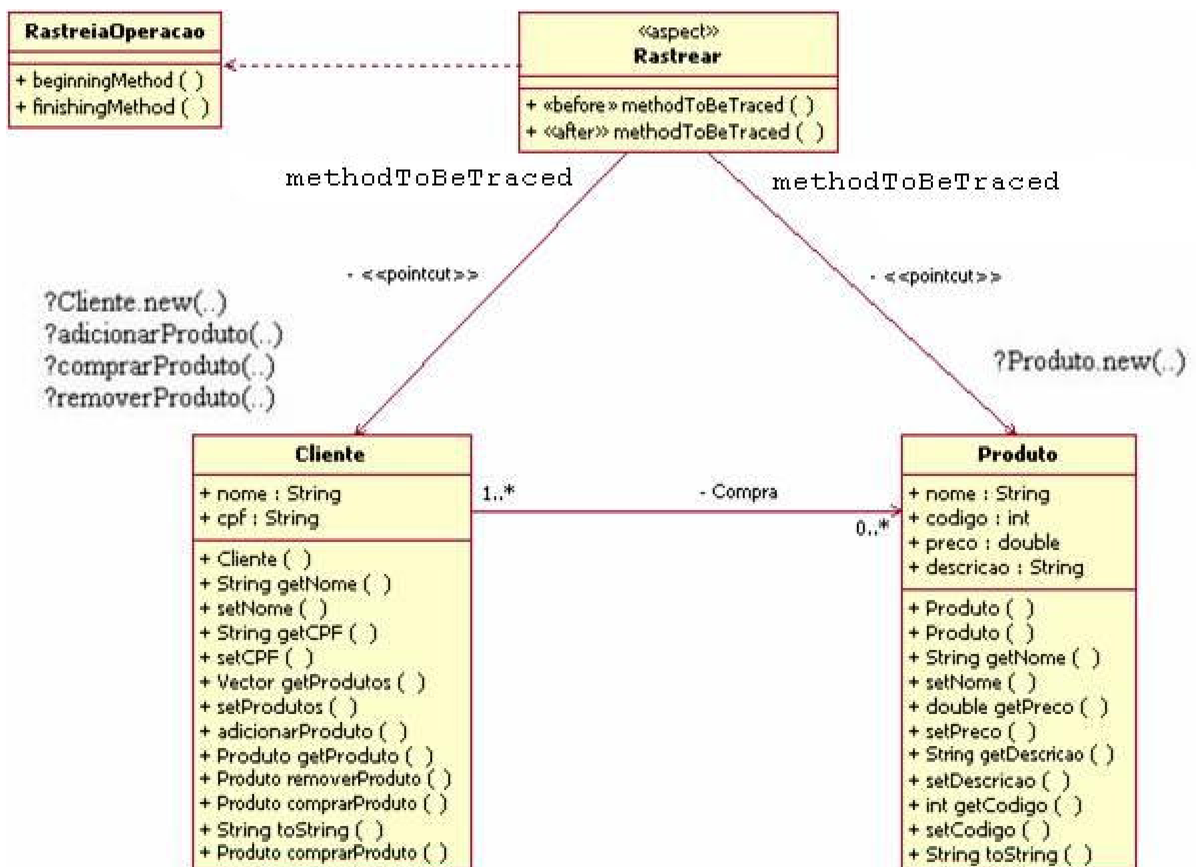

Figura 3.1 – Exemplo de uma modelagem utilizando <<*pointcut*>> e <<*aspect*>>.

Siobhán Clarke (CLARKE *et al*., 2005) propôs uma abordagem de desenvolvimento de software orientado a aspectos denominada Tema. A abordagem Tema faz uso de um perfil

UML e trabalha com dois tipos de temas: o base e o transversal. O Tema-base possui as funcionalidades do domínio do problema, já o Tema-transversal encapsula os interesses transversais que afetam o Tema-base. Um tema é o encapsulamento de um interesse que é representado graficamente por meio das operações-gabaritos (*templates*).

Definidos os temas base e transversal, é necessário que seja feita a composição entre eles por meio de um relacionamento de ligação denominado *bind*. É por meio desse relacionamento que é possível saber quais métodos do Tema-base serão afetados pelo Tema-transversal. Após o relacionamento de ligação ser feito, é possível fazer a composição para que seja gerado o modelo resultante da aplicação.

Na Figura 3.2 pode-se observar um exemplo de um Tema-base, um relacionamento de ligação e um Tema-transversal. O Tema-base possui duas classes: `Cliente` e `Produto`. Note que "Aplicação" é o nome do Tema-base. Esse tema é afetado pelo Tema-transversal quando for feita a composição.

O nome do Tema-transversal é *`TraceTheme`* e ele possui duas classes: *`Trace`* e *`TracedClass`*. *`Trace`* possui dois métodos que são o *`beginingMethod()`* e o *`finishingMethod()`*. O primeiro método imprime uma mensagem "método começando" quando um método que está em uma classe do Tema-base é afetado, e o segundo, idem, porém a mensagem é "método terminando". A classe *`TracedClass`* também possui dois métodos: *`tracedMethod()`* e *`do_tracedMethod()`*. O método *`tracedMethod()`* possui o comportamento transversal e uma chamada para o comportamento original (que no caso seria o *`do_tracedMethod()`*). Em outras palavras, *`tracedMethod()`* encapsula o *`do_tracedMethod()`*. O *`do_tracedMethod()`* possui o comportamento original do método *`tracedMethod()`*. Esse "*do_*" é a principal característica da abordagem. Observe o diagrama de seqüência dentro do Tema-transversal: Toda vez que *`Main`* chamar o método *`tracedMethod()`*, *`TracedClass`* chama o método *`beginningMethod()`* que está na

classe `Trace`. Feito isso, `tracedClass` chama o método `do_tracedClass()` (que é uma cópia original de `tracedClass`). Essa foi a maneira que Clarke encontrou para representar aspectos utilizando modelagem OO. Então, `TracedClass` chama o método `finishingMethod()` que está na classe `Trace` e finaliza a sua execução. `TracedClass` e `tracedMethod()` que estão no retângulo tracejado, são a classe e o método gabarito respectivamente.

As operações-gabaritos podem ser classes ou métodos. Uma classe gabarito é uma classe que representa uma classe qualquer do código base, já um método gabarito é o conjunto de junção. A idéia do tema é desenvolver interesses transversais sem o conhecimento dos pontos em que esse interesse é acoplado (caracterizando o conceito de inconsciência da POA).

De posse do Tema-base e o Tema-transversal, agora é necessário ligá-los por meio de um relacionamento de ligação (parte central da figura). É nessa linha bidirecional tracejada que estão anexados os métodos (ou construtores) que serão afetados pelo Tema-transversal (no caso seria o `Cliente.comprarProduto()`, `Cliente.removerProduto()`, `Cliente.new()` e `Produto.new()`). Nela também, é definido qual é o nome do modelo resultante que é o `ThemeName` (no caso é App). Na Figura 3.3, foi gerado o modelo resultante da combinação do Tema-base com o modelo resultante com o relacionamento de ligação. `App` no canto superior esquerdo é o nome do modelo resultante (que foi definido no relacionamento de ligação). É importante notar que cada método afetado pelo Tema-transversal, possui uma cópia que começa com "*do_*" dentro do diagrama de classes. Todos aqueles métodos que são renomeados possuem o comportamento original dos métodos que são afetados (o "*do_*" diferencia o método original do método transversal). Note-se também que para cada método e construtor afetado, foi criado um diagrama de seqüência. O funcionamento destes diagramas é parecido com o que foi explicado na Figura 3.2

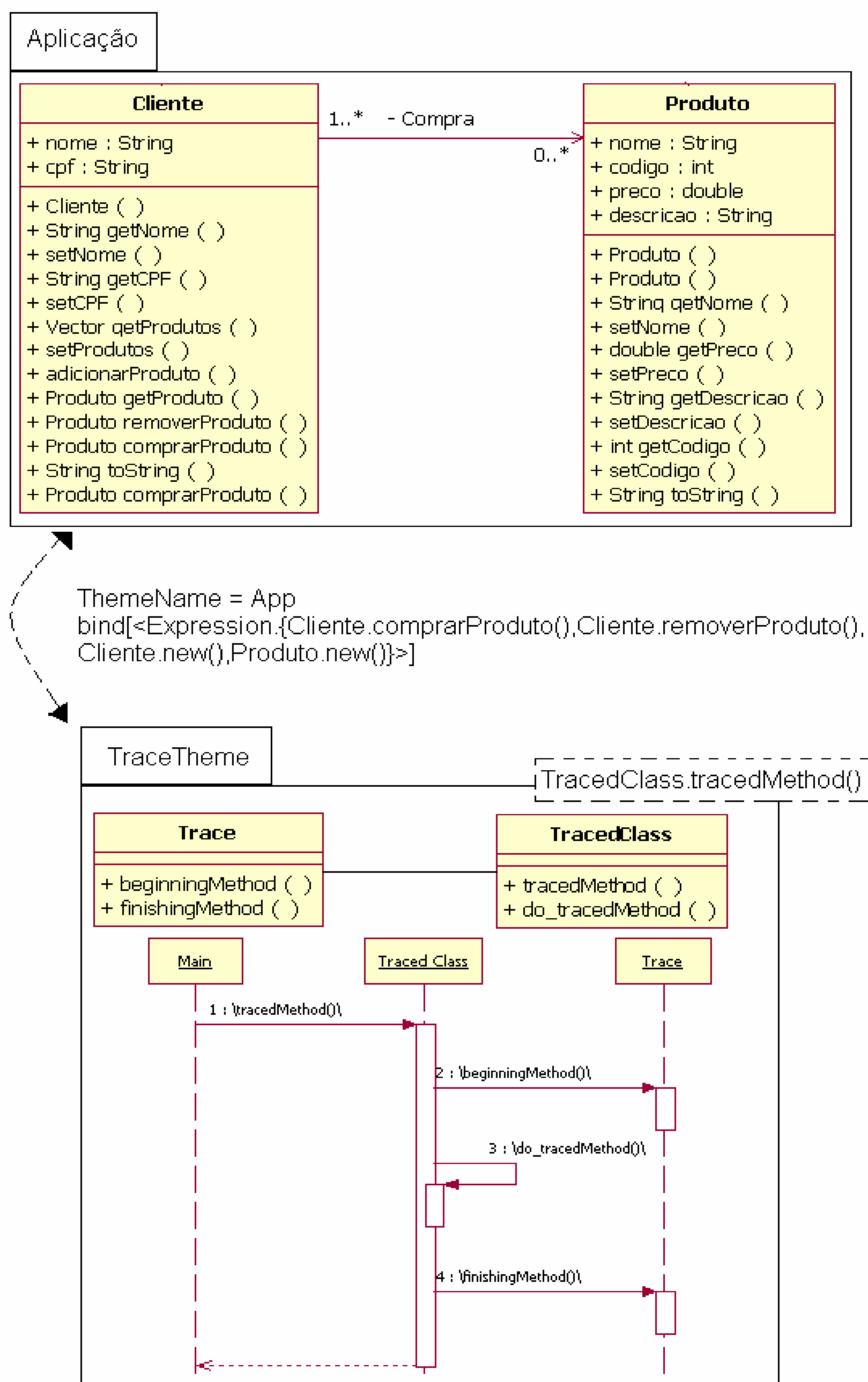

Figura 3.2 – Exemplo de um Tema-base (parte superior), Tema-transversal (parte inferior) e relacionamento de ligação (parte central).

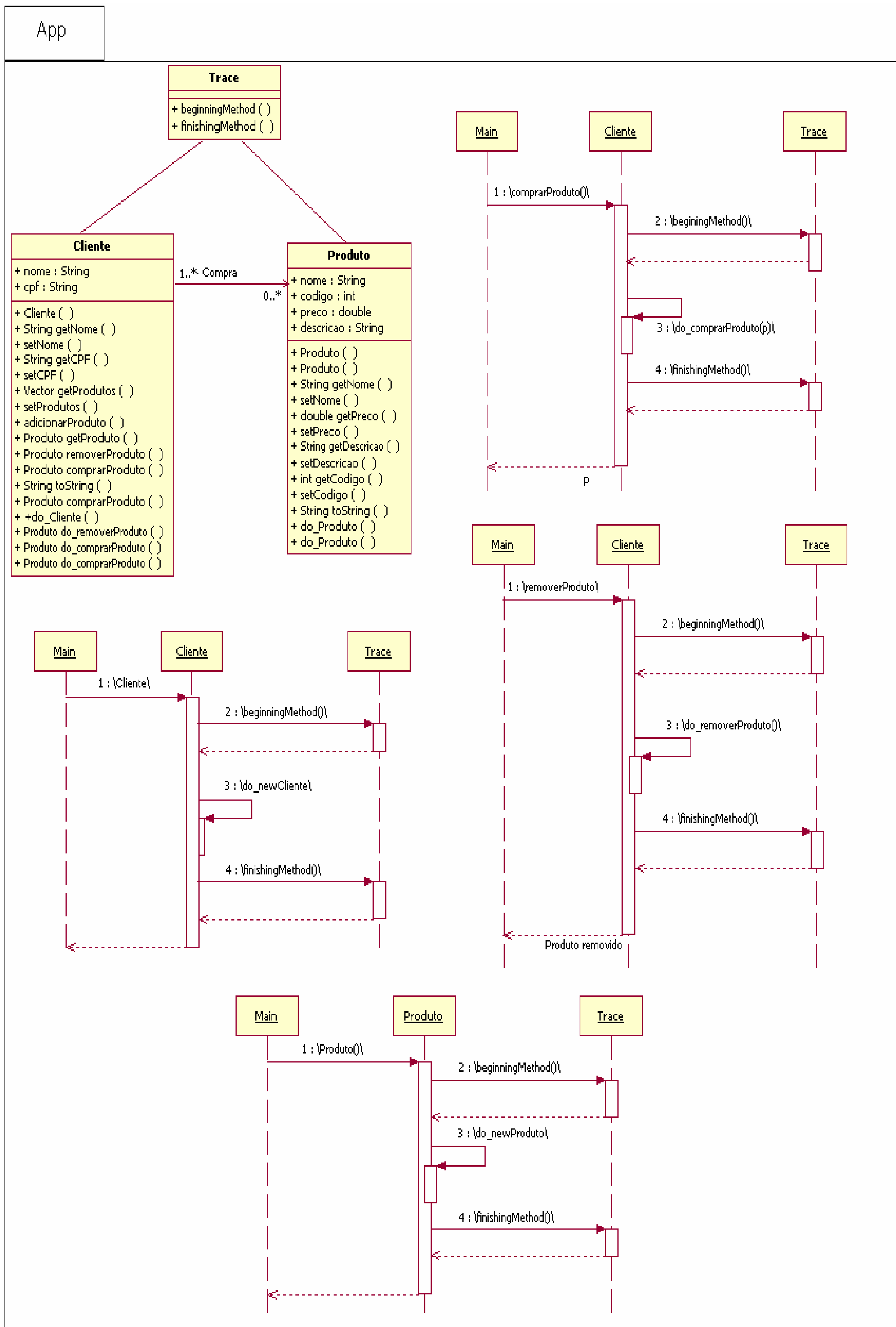

Figura 3.3 – Modelo resultante.

Gimenes *et al*. (GIMENES *et al*., 2004) propuseram um perfil UML para aspectos utilizando os estereótipos <<aspecto>>, <<*pointcut*>>, <<*joinpoint*>> e <<*advices*>>. Na Figura 3.4 é mostrado um exemplo que utiliza esse perfil. É possível observar uma classe `Círculo` e um aspecto abstrato chamado *`Trace`*. A classe Círculo possui apenas uma diferença em relação às classes normais da UML: há um novo compartimento que possui o estereótipo `<<aspecto>>` *`Trace`*. Isso significa que essa classe é afetada pelo aspecto *`Trace`*.

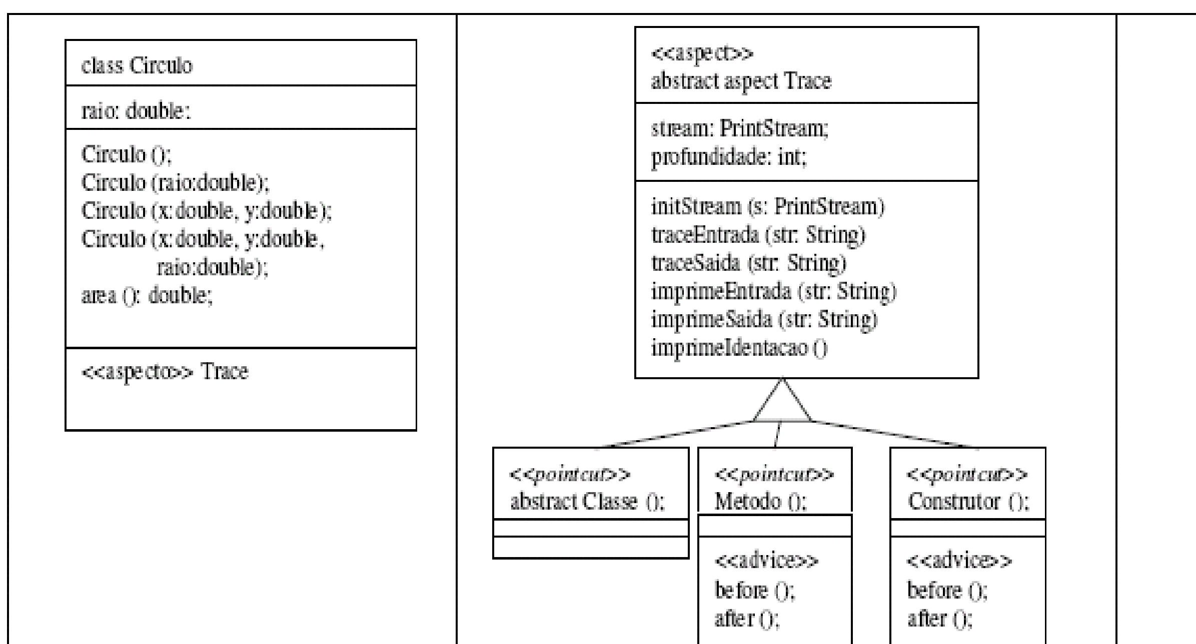

Figura 3.4 - Exemplo de modelagem com a abordagem de GIMENES *et al*. (GIMENES *et al*., 2004).

Observe agora o aspecto *`Trace`*. Note que ele possui três conjuntos de junção que são uma generalização do aspecto. A generalização relaciona os *pointcuts* aos seus respectivos aspectos. Isto significa que cada um dos `<<pointcut>>` possui um relacionamento com o aspecto *`Trace`*. Cada <<*pointcut*>> possui seus próprios métodos entrecortados, seus pontos de junção e adendos. Na Figura 3.5 é apresentado um exemplo de um `<<pointcut>>`.

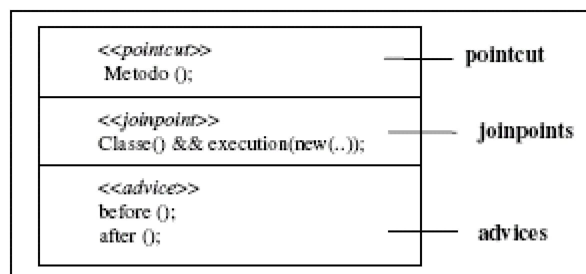

Figura 3.5 – Exemplo de um <<*pointcut*>> (GIMENES *et al.*, 2004).

O nome do <<*pointcut*>> é Método. Dentro de <<*joinpoint*>> estão os método que serão afetados pelo aspecto (nesse caso, `Classe()` e *execution*(*new*(..)). Por fim visualiza-se os adendos *before*() e *after*() (neste exemplo eles não realizarão nenhuma operação). Na Figura 3.6 é apresentado o mesmo exemplo utilizado anteriormente, porém utilizando a abordagem de GIMENES *et al.* (2004). `Cliente` e `Produto` são as classes que serão afetadas pelo aspecto abstrato `ExemploAspecto` pelo o que indica estereótipo `<<aspect>>`. O aspecto `ExemploAspecto` possui dois métodos: `beginningMethod()` e `finishingMethod()`. Há um relacionamento de generalização entre o aspecto e os dois conjuntos de junção. Este relacionamento mostra que os conjuntos de junção `Construtor()` e `Método()` utilizam os métodos do aspecto `ExemploAspecto`. No conjunto de junção `Construtor()` os métodos que serão afetados pelo aspecto são `Cliente.new(..)` e `Produto(..)`. O adendo `<<advice>>` mostra que o método `beginningMethod()` será chamado antes de algum dos métodos iniciar e mostra também que o método `finishingMethod()` será chamado após algum dos métodos finalizar. O mesmo raciocínio vale para o conjunto de junção `Método()`.

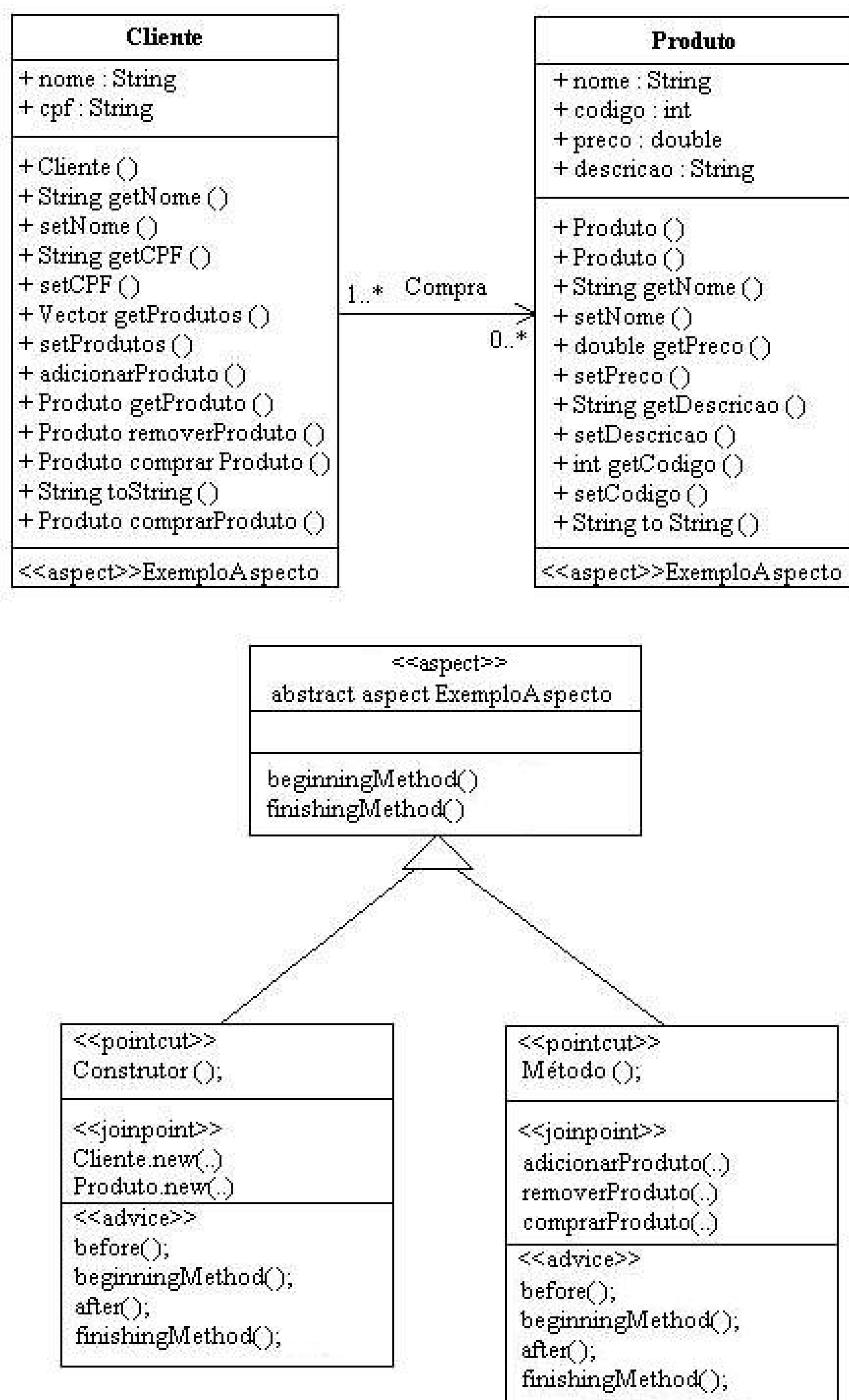

Figura 3.6 – Exemplo que utiliza a abordagem de GIMENES *et al.* (2004).

Groher (GROHER, BAUMGARTH, 2004) também propuseram uma notação para sistemas desenvolvidos com programação orientada a aspectos. A abordagem dos autores utiliza três pacotes o `base`, o *`aspect`* e o *`connector`*. O pacote *`Aspect`* possui o interesse

transversal da aplicação, o pacote `Base` possui a aplicação que é afetada pelo o aspecto e o pacote *`Connector`* é o responsável de fazer a ligação entre o aspecto e os elementos base.

Na Figura 3.7 há um exemplo que ilustra a abordagem em um nível de abstração alto porque é possível ver apenas os pacotes com poucos detalhes de projeto. Já na Figura 3.8 é possível identificar mais detalhes, como por exemplo, os conjuntos de junção, os adendos e o método afetado, o que a caracteriza como nível de abstração baixo. Por exemplo, toda a vez que o método *`getValues()`* que está na classe *`Server`* no pacote base for chamado, ele é afetado durante a chamada pelos métodos *`traceEntry()`* e *`traceExit()`*, que estão na classe *`Trace`* no pacote transversal *`Tracing`*. É possível observar isso por meio do conjunto de junção *`in Server$$getValues : CALL`* e do adendo *`Advice`*: `+Tracing$$Trace$$traceEntry(in tracePoincut : BEFORE)` e `+Tracing$$Trace$$traceEntry(in tracePoincut : AFTER)`. Na Figura 3.9 é apresentado o mesmo exemplo utilizado anteriormente, porém utilizando a abordagem de GROHER e BAUMGARTH (2004). No pacote base é possível visualizar as classes `Cliente` e `Produto`. Os métodos que serão afetados pelo pacote `Tracing` estão dentro do pacote `connector` no conjunto de junção Pointcut. O `conjuntojuncao(in Cliente$$Cliente.new(..) : EXECUTION)` e os adendos `Tracing$$Trace$$beginningMethod(in conjuntojuncao : BEFORE)` e `Tracing$$Trace$$finishingMethod(in conjuntojuncao : AFTER)`, representam que o construtor `Cliente.new(..)` que está na classe `Cliente` no pacote `base`, será afetado pelo aspecto no contexto de execução antes e depois da execução. Em relação ao símbolo “$$” ele é usado no lugar de um ponto porque o ponto não pode ser usado para separar pacotes, classes e métodos. Os autores então decidiram usar o “$$” por ser raramente usado em nomes de classes (GROHER, BAUMGARTH, 2004).

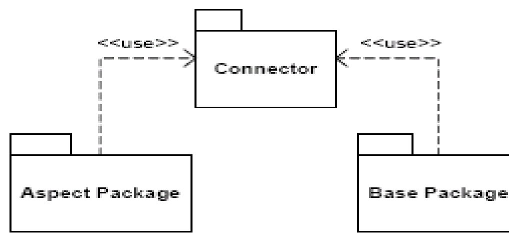

Figura 3.7 – Pacotes *Aspect*, *Connector* e Base (GROHER, BAUMGARTH, 2004).

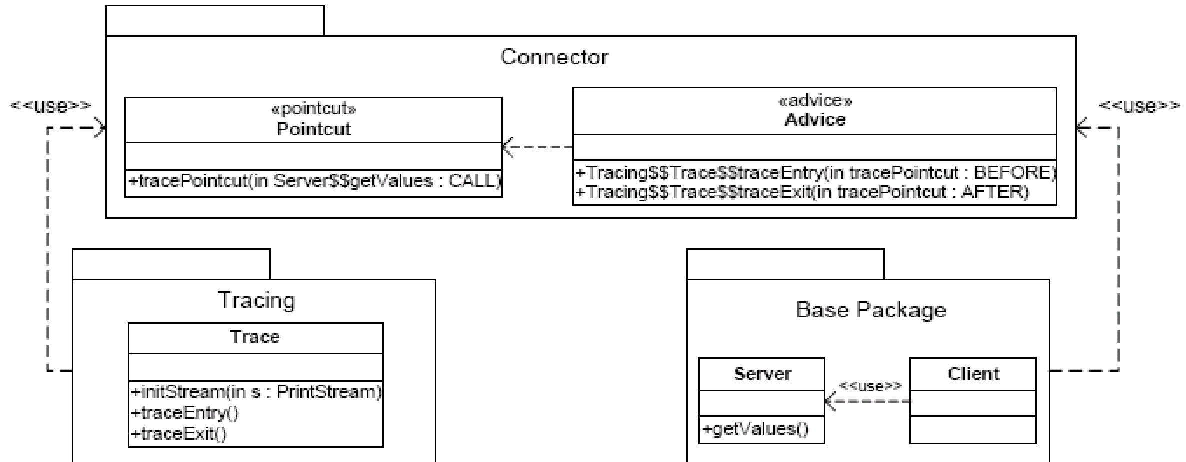

Figura 3.8 – Exemplo utilizando os três pacotes (GROHER, BAUMGARTH, 2004).

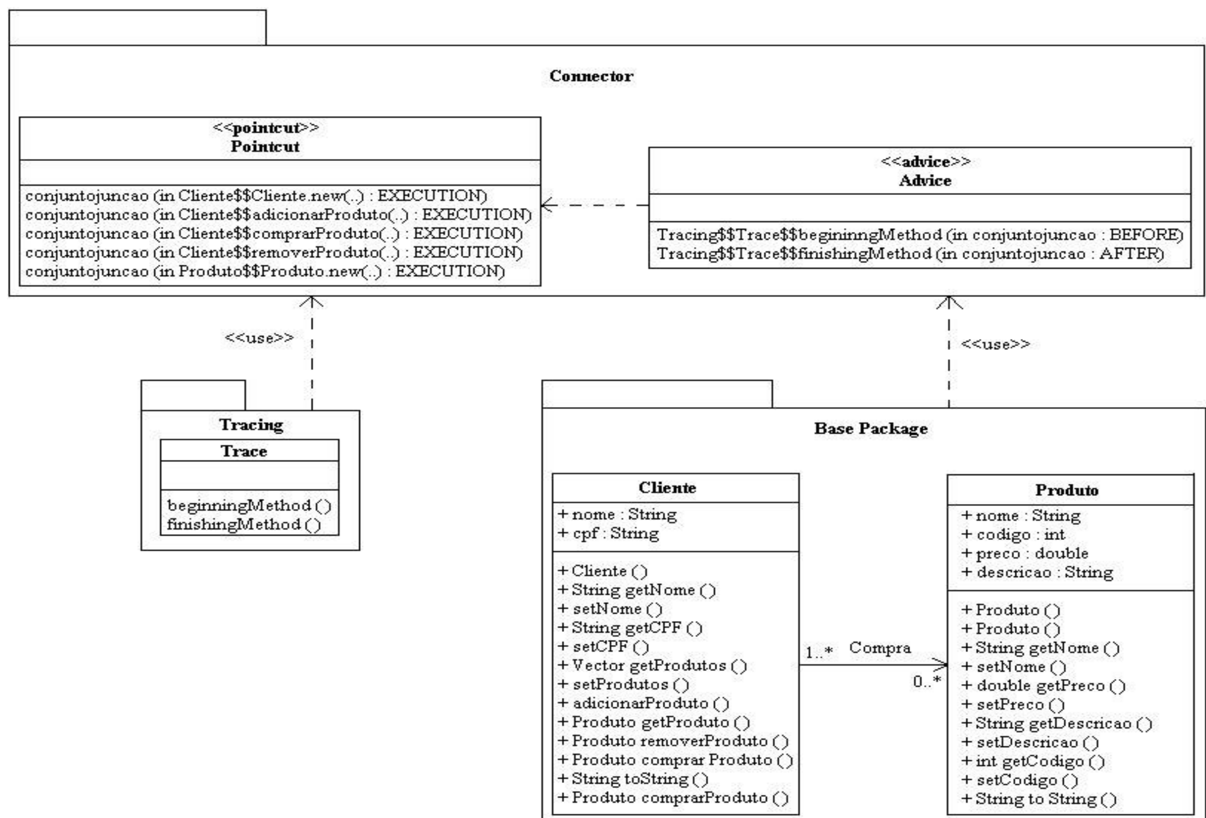

Figura 3.9 – Exemplo que utiliza a abordagem de GROHER e BAUMGARTH, 2004.

3.3 Considerações Finais

Cada abordagem apresentada usa diferentes maneiras para representar as características da POA na modelagem. A abordagem de Pawlak *et al.* (2002) utiliza o estereótipo <<*aspect*>> para a representação dos aspectos nas classes, <<*pointcut*>> para representar os conjuntos de junção, <<*before*>> e <<*after*>> para os adendos e as etiquetas valoradas que mostram quais métodos serão afetados, e aonde serão afetados. GIMENES *et al.* (2004) também utiliza os estereótipos <<*aspect*>>, <<*joinpoint*>> e <<*pointcut*>> mais o estereótipo <<*advice*>> para representar os adendos. Entretanto ela faz uso de uma generalização para relacionar os conjuntos de junção aos seus respectivos aspectos, eliminando o uso de relacionamentos e tornando o modelo mais legível. CLARKE *et al.* (2005) não utiliza estereótipos. Em seu lugar ela utiliza o "`do_`" para representar os aspectos utilizando OO. Os conjuntos de junção são representados por meio de etiquetas valoradas no relacionamento de ligação *bind* e os adendos não possuem visualização no modelo. Também não é possível saber se os métodos serão afetados durante a chamada ou a execução. GROHER e BAUMGARTH (2004) utilizam os estereótipos <<*pointcut*>>, <<*advice*>> e <<*use*>> junto com os pacotes *base*, *connector* e *aspect* para representar o modelo. Por meio desses pacotes é possível identificar quais métodos entrecortarão quais classes e se o entrecorte ocorrerá durante a chamada ou a execução.

CAPÍTULO 4

Critérios de comparação e Comparação

4.1 Considerações Iniciais

Neste capítulo são mostrados os critérios de comparação e as tabelas comparativas em relação às abordagens apresentados no Capítulo2. Após a comparação, é feita uma conclusão em cada critério sobre as abordagens.

4.2 Critérios de comparação

Para evitar que os modelos fossem comparados de uma maneira subjetiva, foram utilizados alguns critérios de comparação. Todos os critérios de comparação utilizados são baseados nos critérios de comparação proposto no trabalho AOSD-Europe-ULANC-9, 2005 e foram utilizados pelo fato de não serem subjetivos. Existem mais alguns critérios como legibilidade e semântica, entretanto eles não foram utilizados neste trabalho por serem critérios subjetivos.

O critério nível de abstração irá definir se uma modelagem possui muitos detalhes (nível de abstração baixo), se possui uma quantidade média de detalhes (nível de abstração médio) ou se possui poucos detalhes (nível de abstração alto). Uma abordagem pode possuir mecanismos que representam um alto nível de abstração, mas também fornecer subsídios para analisar detalhes de mais baixo nível. Já o critério de nível de separação de interesses determina se uma abordagem é simétrica ou assimétrica. Uma abordagem simétrica separa todos os interesses de um sistema, sem fazer distinção entre transversais e não transversais. Já uma abordagem assimétrica, só separa os interesses transversais do resto do sistema. O critério de rastreabilidade determina se o nível de rastreabilidade da abordagem é baixo, médio ou alto. Este critério determina se a abordagem permite que o modelo seja rastreado

durante as suas três fases de desenvolvimento que são a análise o projeto e a implementação. Quanto maior a capacidade de manter um mapeamento entre os artefatos de desenvolvimento que são criados ao longo do ciclo de vida, maior será sua rastreabilidade (o que o caracterizaria como alto). O critério de composição determina a habilidade de compor artefatos e conseqüentemente, ver e entender um conjunto de artefatos e as suas inter-relações. A composição é subdividida em especificação da composição e a composição final. A especificação da composição define como é feita a composição dos interesses base e transversal para se chegar a um modelo composto. A composição final é o modelo resultante da junção de todos os interesses da fase anterior em um único modelo. A evolução determina o nível de evolução fornecido pela abordagem. Consiste na possibilidade de se poder alterar, adicionar ou remover os artefatos existentes em um projeto. Artefatos é tudo o que é criado para se desenvolver o software (atores, estereótipos, classes, temas etc.). O critério de escalabilidade determina a adequabilidade da abordagem a medida com que o sistema adquire um grande número de componentes/classes. A abordagem deve ser apropriada tanto para grandes quanto para pequenos sistemas. A classificação de uma abordagem em relação a sua escalabilidade poderá ser baixa (fácil modelar um sistema pequeno e difícil para modelar sistemas grandes), média ou alta (fácil modelar sistemas pequenos e grandes).

4.3 Comparação

Nesta seção, são apresentadas as comparações entre as abordagens de modelagem mostradas no Capítulo 3 utilizando critérios de comparação citados acima.

Na Tabela 1 é mostrado a comparação entre as abordagens utilizando o critério "nível de abstração".

Tabela 4.1 – Comparação pelo critério de nível de abstração.

Abordagem	Nível de abstração
Temas – CLARKE *et al.*	Médio
Pawlak *et al.*	Baixo
Gimenes *et al.*	Baixo
Pacotes – GROHER e BAUMGARTH	Baixo - Alto

A abordagem de CLARKE *et al.* (2005) está sendo considerada de nível médio por apresentar uma quantidade média de detalhes na modelagem. Comparada com a abordagem de GROHER e BAUMGARTH (2004), a abordagem de CLARKE *et al.* (2005) não apresenta visualização dos adendos e também, não é possível saber se os métodos serão afetados durante a chamada ou a execução. Entretanto, é possível visualizar os conjuntos de junção, os pontos de junção e as etiquetas valoradas.

A abordagem de Pawlak *et al.* (2002) é considerada de nível baixo pelo fato de que é possível extrair uma grande quantidade de informações por meio do modelo de projeto. Entre algumas dessas informações podemos citar os conjuntos de junção, os pontos de junção, os adendos, os métodos que podem ser afetados na chamada ou execução etc.

A abordagem de Gimenes *et al.* (2004) também é considerada de nível baixo, porque é possível visualizar várias informações como o aspecto, os conjuntos de junção, os pontos de junção e os adendos.

A abordagem de GROHER e BAUMGARTH (2004) é considerada de nível baixo - alto. Na Figura 3.7 é possível ter uma visão geral do sistema, o que a caracteriza como sendo de nível alto. Entretanto na Figura 3.8, nota-se a mesma modelagem com um número bem mais expressivo de detalhes, caracterizando-a também como nível baixo.

A comparação mostrada na Tabela 4.1 pode auxiliar o engenheiro de software na escolha por uma determinada abordagem. Quando o objetivo é detalhar características especificas de uma linguagem de programação, uma abordagem de mais baixo nível como a de Pawlak *et al.* (2002), Gimenes *et al.* (2004) e GROHER e BAUMGARTH (2004) devem ser utilizadas. Já quando o objetivo é abstrair os detalhes da implementação, uma abordagem de mais alto nível como a de GROHER e BAUMGARTH (2004) deve ser empregada pelo fato dessa abordagem possuir uma representação em alto nível. As vantagens de uma ou outra depende das características do projeto e dos objetivos da organização. Geralmente há um

misto entre ambas, em que tanto pode-se concentrar nos detalhes quanto na funcionalidade de mais alto nível.

Na Tabela 4.2 é mostrada uma comparação entre as abordagens utilizando o critério "nível de separação de interesses".

Tabela 4.2 – Comparação pelo critério de separação de interesses.

Abordagem	Nível de separação de interesses
Temas - CLARKE *et al.*	Simétrica
Pawlak *et al.*	Assimétrica
Gimenes *et al.*	Assimétrica
Pacotes – GROHER e BAUMGARTH	Assimétrica

A abordagem Tema é considerada simétrica, pois separa todos os interesses do resto do sistema utilizando os temas base e transversal.

As abordagens de Pawlak *et al.* (2002) e Gimenes *et al.* (2004) são consideradas assimétricas por separarem somente o interesse transversal do resto do sistema (observa-se isso por meio da classe com o estereótipo `<<aspect>>`).

A abordagem de GROHER e BAUMGARTH (2004) também é considerada assimétrica por separar somente o interesse transversal do resto do sistema usando o pacote *Aspect*.

Esse critério também pode ser utilizado para guiar a escolha de uma determinada abordagem. Abordagens simétricas devem ser escolhidas quando as linguagens de programação utilizadas são simétricas, como, por exemplo, HyperJ. Já quando linguagens assimétricas serão empregadas na implementação, as abordagens assimétricas é que devem ser escolhidas. Um exemplo de um linguagem assimétrica é a AspectJ.

Na Tabela 4.3 é mostrada a comparação entre as abordagens utilizando o critério "Rastreabilidade".

Tabela 4.3 – Comparação pelo critério de rastreabilidade.

Abordagem	Rastreabilidade
Temas – CLARKE *et al.*	Alto
Pawlak *et al.*	Baixo
Gimenes *et al.*	Baixo
Pacotes – GROHER e BAUMGARTH	Médio

A abordagem de CLARKE *et al*. (2005) é considerada como sendo de nível alto pelo fato de que é possível fazer o rastreamento do sistema, começando pelo modelo resultante e voltando até a parte da análise de requisitos.

As abordagens de Pawlak *et al*. (2002) e Gimenes *et al*. (2004) são consideradas como sendo de nível baixo por não possuírem uma modelagem específica para a implementação nem para a análise, focando apenas para a parte de projeto.

A abordagem de GROHER e BAUMGARTH (2004), é considerada como sendo de nível médio por permitir que a aplicação possa ser rastreada do projeto para a implementação por meio do *Code Generator*. O *Code Generator* proposto pelos autores gera o esqueleto do código de acordo com o modelo projetado.

Em relação ao critério de rastreabilidade, a abordagem de CLARKE *et al*. (2005) é a única que permite o rastreamento durante as três fases de desenvolvimento do projeto, portanto essa abordagem é a mais apropriada para projetos que têm como objetivo manter um bom mapeamento entre os artefatos de desenvolvimento que são criados ao longo do ciclo de vida. Já a abordagem de GROHER e BAUMGARTH (2004), não apresentam uma representação para o modelo na fase de análise, sendo assim essa abordagem é mais recomendada para projetos que focam mais a parte de projeto e implementação. Quanto às abordagens de Pawlak *et al*. (2002) e Gimenes *et al*. (2004), ambas não possuem representação na fase de análise nem na fase de implementação, o que as tornam menos convenientes de serem utilizadas caso o projeto foque neste critério.

Na Tabela 4.4 é mostrada a comparação entre as abordagens utilizando como critério a Composição.

Tabela 4.4 – Comparação pelo critério de composição.

Abordagem	**Especificação da Composição**	**Composição**
Temas - CLARKE *et al.*	*Merge*, *Override* ou *Bind*.	Temas são compostos em um tema resultante.
Pawlak *et al.*	Pontos de junção e conjuntos de junção.	Não possui.
Gimenes *et al.*	Pontos de junção e conjuntos de junção.	Não possui.
Pacotes – GROHER e BAUMGARTH	Pacote *connector*.	Não possui.

A especificação da composição na abordagem Tema é classificada como *Merge* ou *Override*. Quando um Tema-base é combinado (*merged*), os elementos base são ligados com os elementos gabarito. No caso de *Override*, ele permite que os elementos do projeto de um tema sobreponham os elementos de projeto de outro tema. Já o *Bind* é o relacionamento de ligação que especifica quais métodos do Tema-base serão afetados pelo Tema-transversal. Em relação à composição da abordagem Tema, temas são compostos formando o modelo resultante (Figura 3.3).

As abordagens de Pawlak *et al.* (2002) e Gimenes *et al.* (2004), realizam a especificação da composição por meio dos conjuntos de junção, dos pontos de junção (observe Figuras 3.1 e 3.6). Essas abordagens não possuem visualização de uma composição final.

A abordagem de GROHER e BAUMGARTH (2004) realiza a especificação da composição por meio do pacote *connector*, pois é ele que indica quais métodos do pacote base vão ser afetados pelo pacote transversal.

Em relação ao critério de composição, a abordagem de CLARKE *et al.* (2005) é a única que possui a visualização de um modelo resultante que utiliza diagramas de seqüência, sendo assim, ao final da fase de projeto é possível ver o modelo completo antes de partir para a fase de implementação. Assim, se um dos grandes objetivos do projeto for visualizar o comportamento final do sistema, essa abordagem pode ser usada. As demais abordagens não

apresentam um modelo de composição final, apenas à especificação da composição. Dessa forma, aumenta-se a dificuldade de se entender o comportamento do sistema todo.

Na Tabela 4.5 é mostrada a comparação entre as abordagens utilizando como critério a Evolução.

Tabela 4.5 - Comparação pelo critério de evolução.

Abordagem	Evolução	
	Mudança	**Adição - Remoção**
Temas – CLARKE *et al.*	Alterar um interesse de um tema pode exigir a mudança de outros temas.	Temas podem ser adicionados ou removidos.
Pawlak *et al.*	Uma mudança no núcleo pode exigir mudanças nas classes dependentes.	Aspectos podem ser adicionados e removidos sem necessidade de alteração no núcleo.
Gimenes *et al.*	Uma mudança no núcleo pode exigir mudanças nas classes dependentes.	Aspectos podem ser adicionados e removidos sem necessidade de alteração no núcleo.
Pacotes – GROHER e BAUMGARTH	Alterar o pacote *connector* muda todo o projeto final.	Pacote *connector* não é removido (caso haja apenas um). Pacotes base e aspecto podem ser adicionados ou removidos.

Um tema encapsula um projeto relacionado a um interesse. O tema composto é descrito como a composição das relações entre os temas. A mudança em um dos temas pode ou não afetar outros temas que serão compostos.

Temas não são dependentes uns dos outros, então, eles podem ser adicionados ou removidos sem afetar outros temas.

Em relação às abordagens de Pawlak *et al.* (2002) e Gimenes *et al.* (2004), os aspectos podem ser inseridos ou removidos, pois não haverá alterações no núcleo (apenas nas classes afetadas). Já em relação à parte base, uma mudança nela pode exigir mudança nas partes (classes) dependentes.

Quanto à abordagem de GROHER e BAUMGARTH (2004), o pacote *connector* é responsável pela união do pacote aspecto com o pacote base, portanto mudá-lo ou retirá-lo irá

gerar mudanças no sistema. Já os pacotes base e aspecto podem ser alterados sem maiores problemas.

Em relação ao critério de evolução, pode-se chegar à conclusão de que, de uma maneira geral, todas as abordagens permitem mudanças no modelo de projeto do sistema. O que diferencia uma abordagem da outra é o lugar em que será feita a mudança. No caso da abordagem Tema, alterar o interesse de um tema pode exigir a alteração de outros temas dependentes. Em relação às abordagens de Pawlak *et al*. (2002) e Gimenes *et al*. (2004), uma alteração no núcleo do projeto, pode exigir a alteração das classes dependentes do mesmo. Já na abordagem de GROHER e BAUMGARTH (2004), alterar o pacote *connector* resultará em um modelo de projeto final diferente, visto que após união entre os pacotes base e aspecto, outros métodos serão afetados pelo aspecto na classe base.

Na Tabela 4.6 é mostrada a comparação entre as abordagens utilizando como critério a Escalabilidade.

Tabela 4.6 - Comparação pelo critério de escalabilidade

Abordagem	Escalabilidade
Temas - CLARKE *et al.*	Médio
Pawlak *et al.*	Baixo
Gimenes *et al.*	Médio
Pacotes – GROHER e BAUMGARTH	Alto

Temas podem ser vistos de um alto nível por meio dos pacotes. Esta visão permite que o projetista veja como cada tema opera no sistema (uma visão geral). Sendo assim, podemos classificar essa abordagem como sendo de escalabilidade média.

Na abordagem de Pawlak *et al*. (2002), o modelo poderá ficar muito poluído em sistemas de grande porte que tenham uma grande quantidade de métodos afetados pelos aspectos. Por esse motivo, podemos classificar essa abordagem como sendo de escalabilidade baixa.

Algo interessante em relação à abordagem de Gimenes *et al*. (2004) é o fato de não existirem relacionamentos entre classes e aspectos, o que torna o modelo menos poluído e

consequentemente de mais fácil entendimento. Devido a isso, podemos classificar essa abordagem como sendo de escalabilidade média.

Sobre a abordagem de GROHER e BAUMGARTH (2004), é possível extrair um conjunto maior de informações através do modelo projetado, entretanto, a abordagem apresenta uma maneira de mostrar o sistema em uma visão geral, o que a torna possível de ser classificada como sendo de escalabilidade alta.

Em relação ao critério de escalabilidade, pode-se chegar à conclusão de que para o projeto de sistemas de grande porte a abordagem de GROHER e BAUMGARTH (2004) é a mais cabível para ser utilizada, pelo fato de apresentar uma representação em alto nível por meio dos pacotes. As abordagens de Gimenes *et al*. (2004) e CLARKE *et al*. (2005) devem ser empregadas preferencialmente em projetos para sistemas de médio porte, visto que, se empregadas em sistemas de grande porte poderão apresentar problemas de legibilidade. Já a abordagem de Pawlak *et al*. (2002), deve ser utilizada somente se o objetivo principal for o de obter uma visão detalhada do sistema na fase de projeto.

Na Tabela 4.7 são mostradas todas as abordagens com todos os critérios de comparação que foram utilizados neste trabalho. Para diminuir o tamanho da figura, para cada critério foi atribuído uma sigla, como pode ser observado na legenda posicionada na parte inferior da tabela.

Tabela 4.7 – Tabela final.

Abordagem	**Abst.**	**Sep. Int.**	**Rast.**	**Composição**		**Evolução**		**Escala.**	**Cenários**
				Esp. C.	**Comp.**	**Mudança**	**Adição-Remoção**		
Temas – CLARKE *et al*.	Médio	Simétrica	Alto	*Merge*, *Override* ou *Bind*.	Temas são compostos em um tema resultante.	Alterar um interesse de um tema pode exigir a mudança de outros temas.	Temas podem ser adicionados ou removidos.	Médio	Interesse moderado nos detalhes da linguagem.
Pawlak *et al*.	Baixo	Assimétrica	Baixo	Pontos de junção e conjuntos de junção.	Não possui.	Uma mudança no núcleo pode exigir mudanças nas classes dependentes.	Aspectos podem ser adicionados e removidos sem necessidade de alteração no núcleo.	Baixo	Interesse alto em representar os detalhes da linguagem.
Gimenes *et al*.	Baixo	Assimétrica	Baixo	Pontos de junção e conjuntos de junção.	Não possui.	Uma mudança no núcleo pode exigir mudanças nas classes dependentes.	Aspectos podem ser adicionados e removidos sem necessidade de alteração no núcleo.	Médio	Alto interesse na legibilidade da linguagem.
Pacotes – GROHER e BAUMGARTH	Baixo - Alto	Assimétrica	Médio	Pacote *connector*.	Não possui.	Alterar o pacote *connector* muda todo o projeto final.	Pacote *connector* não é removido (caso haja apenas 1). Pacotes base e aspecto podem ser adicionados ou removidos.	Alto	Muito interesse nos detalhes da linguagem.

Legenda: Abst. = Nível de abstração, Sep. Int. = Separação de interesses, Rast. = Rastreabilidade, Esp. C. = Especificação da composição, Comp. = Composição, Escala. = Escalabilidade.

4.4 Considerações Finais

Foi mostrada uma comparação entre cada abordagem utilizando os critérios de comparação previamente definidos e feita uma breve conclusão em cada um dos critérios de comparação sobre as abordagens estudadas. Foi mostrada também uma tabela completa contendo todas as abordagens com todos os critérios analisados que representa o produto deste trabalho.

Conclusão

Este trabalho foi elaborado com o intuito de ajudar o projetista de software a escolher uma determinada abordagem de modelagem para um sistema que utiliza Aspectos. Para isto, foram apresentados alguns cenários utilizando a abordagem de modelagem de cada autor para que assim fosse possível obter um breve entendimento sobre o funcionamento da mesma.

Uma das limitações deste trabalho é a quantidade de abordagens analisadas. Existem muitas outras abordagens de modelagem para POA, entretanto, apenas quatro foram apresentadas pelo fato de que algumas delas são de difícil entendimento e o tempo também é escasso. Sendo assim, seria necessário um maior estudo das mesmas para que fosse possível apresentá-las e compará-las neste trabalho. A carência de cenários mais elaborados também é uma limitação deste trabalho visto que os cenários apresentados para o estudo são triviais.

Outra deficiência deste trabalho está no fato das abordagens terem sido apenas estudadas e comparadas na fase de projeto. Um maior estudo e comparação entre as abordagens durante as três fases de desenvolvimento (análise, projeto e implementação) com cenários mais elaborados deverá ser feito em trabalhos futuros.

A contribuição deste trabalho está no referencial deixado, que pode auxiliar os desenvolvedores na escolha por uma determinada abordagem de modelagem. Outra contribuição é o levantamento bibliográfico que pode auxiliar o desenvolvimento de outros projetos semelhantes.

Referências Bibliográficas

BOOCH, Grady. RUMBAUGH, James. JACOBSON, Ivat. **UML – Guia do usuário.** Rio de Janeiro. Editora Campus,2000.

CAMARGO, Valter. **Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software.** Tese de Doutorado – ICMC/USP, 2006.

CLARKE, Siobhán, BANIASSAD, Elisa. **Aspect Oriented Analysis and Design – The Theme Approach.** Editora Addison-Wesley, Primeira Edição, 2005.

GIMENES, Itana, STEINMACHER, Fábio, LIMA, José. Artigo: **Uma Extensão da UML para Modelagem Orientada a Aspectos Baseada em AspectJ.** UFRGS – Universidade Federal do Rio Grande do Sul, Porto Alegre – RS, UEM – Universidade Estadual de Maringá, Maringá – PR.

GROHER, Iris, BAUMGARTH, Thomas. Artigo: **Aspect-Orientation from Design to Code.** Munich, Alemanha, 2004.

KICZALES, Gregor *et al*. **Aspect-oriented programming.** In Proc. of ECOOP (European Conference on Object-Oriented Programming), Springer-Verlag, 1997.

KICZALES, Gregor *et al*. **Aspect–oriented programming with AspectJ**. In *OOPSLA* (Object-Oriented Programming, Systems, Languages and Applications)*'01, Tutorial*, Tampa FL.

KICZALES, G., MEZINI, M. **Aspect-Oriented Programming and Modular Reasoning. In: Proceedings of International Conference on Software Engineering (ICSE'05)**, St. Louis, Missouri, USA, pp. 49-58, 2005.

MONTEIRO, Elaine. Monografia: **Um Estudo Sobre Modelagem Orientada a Aspectos Baseada em AspectJ e UML.** 2004. Centro Universitário Luterano de Palmas, 2004.

PAWLAK, Renaud, DUCHIEN, Laurence, FLORIN, Gerard, LEGOND-AUBRY, Fabrice, SENTURIER, Lionel, MARTELLI, Laurent. Artigo: **A UML Notation for Aspect Oriented Software Design.** 2002. França, 2002.

STEIN, Dominik. **An Aspect-Oriented Design Model Based on AspectJ and UML**. Dissertação de Mestrado (Mestrado em Gerenciamento de Sistemas de Informação)– Universidade de Essen, Germany, 2002.

Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke, Andrew Jackson. **AOSD-Europe-ULANC-9**. 18/05/2005. Europa, 2005.

Anexo

Segue em anexo o código fonte de um simples programa feito utilizando a linguagem AspectJ.

Aspecto

```
public aspect Aspecto
{
    public pointcut Point():
         execution (void adicionarProduto(Produto)) ||
         execution (Cliente.new(..)) ||
         execution (void removerProduto(int,int,int)) ||
         execution (void comprarProduto(String,int)) ||
         execution (void comprarProduto(Produto)) ||
         execution (Produto.new(..));
    before() : Point()
    {
         thisJoinPoint.getArgs();
         thisJoinPoint.getTarget();
         Trace.beginningMethod("String methodName");
    }
    after() : Point()
    {
         thisJoinPoint.getArgs();
         thisJoinPoint.getTarget();
         Trace.finishingMethod("String methodName");
    }
}
```

Classe Cliente

```
import java.util.*;
public class Cliente
{
    public Vector produtos;
    public String nome;
    public String cpf;

    public Cliente(String n, String cpf)
    {
    produtos = new Vector(0);
    setNome(n);
    setCPF(cpf);
    }

    public String getNome()
    {
        return nome;
    }

    public void setNome(String n)
    {
        nome = n;
```

```
}

public String getCPF()
{
    return cpf;
}

public void setCPF(String c)
{
    cpf = c;
}

public Vector getProdutos()
{
return produtos;
}

public void setProdutos(Vector v)
{
produtos = v;
}

private void adicionarProduto(Produto p)
{
produtos.add((Produto) p);
}

public Produto getProduto(int cod)
{
Produto p = (Produto) produtos.get(cod);
return p;
}

public Produto removerProduto(int cod,int aux,int i)
{
Produto p = getProduto(cod);
return p;
}

public Produto comprarProduto(String n,int c)
{
    Produto p = new Produto(n,c);
    adicionarProduto(p);
    return p;
}

public Produto comprarProduto(Produto p)
{
if (p != null)
     adicionarProduto(p);
return p;
}

public void exibirProdutos()
{
```

```
                Produto p;
                if (produtos.size() > 0)
                {
                    for (int i = 0; i < produtos.size(); i++)
                    {
                        p = (Produto) produtos.get(i);
                        System.out.println(p);
                    }
                }
                else
                {
                    System.out.println("Nao ha produtos");
                }
            }

            public String toString()
            {
            String s = getNome() + ", " + getCPF() + ", " +
getProdutos().size() + " produtos.";
            return s;
            }
        }
```

Classe Produto

```
        public class Produto
        {
            public String nome;
            public int cod;
            public double preco;
            public String desc;

            public Produto(String n,int c, String d, double p)
            {
            setNome(nome);
            setCod(c);
            setDesc(d);
            setPreco(p);
            }

            public Produto(String n, int c)
            {
                setNome(nome);
                setCod(c);
                setDesc("");
                setPreco(0.0);
            }

            public String getNome()
            {
                return nome;
            }

            public void setNome(String n )
            {
                nome = n;
```

```
        }

        public double getPreco()
        {
            return preco;
        }

        public void setPreco(double p)
        {
            preco = p;
        }

        public String getDesc()
        {
            return desc;
        }

        public void setDesc(String d)
        {
            desc = d;
        }

        public int getCod()
        {
            return cod;
        }

        public void setCod(int c)
        {
            cod = c;
        }

        public String toString()
        {
        return (getNome() + ", " + getCod() + ", " + getPreco() +
", " + getDesc());
        }
    }
```

Classe Principal

```
    public class Principal
    {
        public static void main(String args[])
        {
         //cria dois clientes
         Cliente c1 = new Cliente("Cli1", "001");
         Cliente c2 = new Cliente("Cli2", "002");

         //cria três produtos
         Produto p1 = new Produto("calça",1,"roupa",25.50);
         Produto p2 = new Produto("tenis",2,"calçado",50.50);
         Produto p3 = new Produto("oculos", 3,"acessórios",
         42.20);

         //adiciona 2 produtos no cliente 1
```

```
        c1.comprarProduto(p1);
        c1.comprarProduto(p2);

        //adiciona 4 produtos no cliente 2
        c2.comprarProduto(p2);
        c2.comprarProduto(p3);
        c2.comprarProduto(p3);
        c2.comprarProduto(p3);

        //imprime os clientes
        c1.exibirProdutos();
        c2.exibirProdutos();

        //finaliza a aplicação
        System.exit(0);
    }
}
```

Classe Trace

```
public class Trace
{
    public static void beginningMethod(String methodName)
    {
        System.out.println(methodName + " iniciando");
    }

    public static void finishingMethod(String methodName)
    {
        System.out.println(methodName + " terminando");
    }
}
```