



# Desenvolvimento de Software Orientado a Aspectos

**João Roberto Silva de Almeida**

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação
Orientador: Prof. Tarcísio de Souza Lima


Juiz de Fora, MG
Dezembro de 2007

# Agradecimentos

Em primeiro lugar, agradeço a meus pais e minha irmã, por todo o carinho e força nesta jornada. Agradeço também a todos outros familiares pela amizade e apoio. Aos grandes amigos que conquistei nestes últimos anos e especialmente àqueles que me acompanham de longa data. Danielle por toda a atenção, compreensão e amor.

# Sumário

# Lista de Reduções

| | |
|---|---|
| AODM | *Aspect Oriented Design Model* |
| DSOO | Desenvolvimento de Software Orientado a Objetos |
| DSOA | Desenvolvimento de Software Orientado a Aspectos |
| ER | Engenharia de Requisitos |
| EROA | Engenharia de Requisitos Orientada a Aspectos |
| MOA | Modelagem Orientada a Aspectos |
| MPOS | Modelo de Projeto Orientado a Sujeitos |
| POA | Programação Orientada a Aspectos |
| POO | Programação Orientada a Objetos |
| POS | Programação Orientada a Sujeitos |
| OO | Orientação a Objetos |
| UML | *Unified Modeling Language* |

# Lista de Figuras

# Lista de Tabelas

# Resumo

A consolidação da Orientação a Aspectos como paradigma capaz de suprir as deficiências da Orientação a Objetos gera a necessidade de adaptações em diversas etapas do desenvolvimento de um software. Este trabalho apresenta algumas das propostas já realizadas para a utilização da Orientação a Aspectos nas etapas de levantamento de requisitos, modelagem e testes. Estas propostas, mesmo que recentes e sem grandes aplicações práticas apresentadas, visam facilitar a utilização correta dos conceitos do novo paradigma, aumentando a qualidade do software.

**Palavras-chave:** Orientação a Aspectos, Desenvolvimento de Software Orientado a Aspectos, Separação de Interesses, Testes Orientados a Aspectos, Modelagem Orientada a Aspectos, Engenharia de Requisitos Orientada a Aspectos.

# Capítulo 1
# Introdução

Durante os últimos anos a Orientação a Objetos (OO) se tornou o paradigma em destaque para a solução de problemas no desenvolvimento de software. A divisão dos requisitos de um sistema em classes e a facilidade de analisar o problema de diferentes ângulos foram, entre outras, as grandes razões da grande aceitação da OO.

Porém, com o passar do tempo, os problemas deste paradigma começaram a surgir, e a demanda por qualidade e funcionalidades crescente do mercado de software, criou a necessidade de novas soluções. Segundo FILMAN *et al.* (2004), a OO está cada vez mais próxima do seu limite. A necessidade de tratar cada vez mais interesses em uma aplicação e a falta de ferramentas que tornem esta tarefa mais clara e objetiva cria um cenário favorável para o surgimento de novas tecnologias de desenvolvimento. Dentre elas, a Orientação a Aspectos (OA) é, até o momento, a que mais se destaca.

A Programação Orientada a Aspectos (POA), segundo CLARKE e BANIASSAD (2005), proporciona a codificação de interesses que entrecortam (*crosscutting concerns*) outras partes de uma aplicação. Estes interesses podem ser facilmente propagados para quaisquer pontos da aplicação de acordo com a necessidade imposta pelos requisitos. Um aspecto é o resultado da separação no código destes interesses.

Alguns anos após o surgimento do termo OA e com o fortalecimento de linguagens que tratam deste paradigma, outras etapas do ciclo de desenvolvimento de uma aplicação começaram a receber propostas de alterações. Segundo CLEMENTE *et al.* (2004), a resistência à utilização da POA em grandes projetos não se deve à falta de ferramentas ou pela descrença nos benefícios providos por este paradigma. Mas sim pela ausência de métodos de aplicação da OA em outros estágios do desenvolvimento.

Segundo FILMAN *et al.* (2004), o desenvolvimento de software orientado a aspectos (DSOA) é proposto como uma técnica para melhorar a separação de interesses na construção de software e apoiar ao aumento da reusabilidade e facilidade de evolução. O DSOA é, portanto, uma área de pesquisa emergente cujo objetivo é promover a separação avançada de interesses em todas as etapas do desenvolvimento de software.

Este trabalho apresenta uma abordagem sobre algumas técnicas já existentes para alcançar os objetivos do DSOA e tem como público alvo desenvolvedores de software e pesquisadores interessados neste paradigma e nas suas aplicações persistentes. Uma iniciação em OA é um pré-requisito básico para o bom entendimento deste trabalho e o trabalho de CAMPOS (2006) é uma boa referência inicial para o assunto.

## 1.1 Organização da Monografia

A monografia divide-se em cinco capítulos, além de um apêndice, desta introdução e das referências bibliográficas.

No segundo capítulo é apresentado um breve resumo das tecnologias de desenvolvimento de software até a o surgimento da OA. Recebem destaque a teoria de separação de interesses e os pontos falhos da OO.

O terceiro capítulo trata da POA e da linguagem AspectJ. Por questões cronológicas, a POA é apresentada antes das propostas baseadas em OA para outras etapas do desenvolvimento de software. Os conceitos da POA e os termos da linguagem AspectJ são amplamente utilizados em pesquisas para utilização de OA em outras etapas.

No quarto capítulo são apresentadas as propostas de diversos pesquisadores para a expansão das etapas do desenvolvimento de um software a fim de abranger as mudanças impostas pela OA. Levantamento de requisitos, modelagem e testes são as etapas abordadas neste trabalho.

O quinto capítulo traz o desenvolvimento do estudo de caso. É criada uma pequena aplicação, com funcionalidades de auditoria e autenticação, utilizando propostas apresentadas para diversas etapas do ciclo de desenvolvimento.

O sexto capítulo conclui o trabalho e apresenta propostas para trabalhos futuros. No apêndice I é feita uma breve revisão sobre conceitos importantes da *Unified Modelling Language* (OMG, 2005) que são utilizados em várias partes do trabalho.

# Capítulo 2

# Desenvolvimento de Software Orientado a Aspectos

## 2.1 Origens

Nos primórdios da Ciência da Computação, a programação era feita diretamente no nível da máquina. Gastava-se mais tempo pensando sobre questões ligadas aos conjuntos de instruções de uma determinada máquina do que com o problema em si. Aos poucos ocorreu a migração para linguagens de mais alto nível e com maior poder de abstração. O paradigma estruturado surgiu com as estruturas de procedimentos e funções como facilitadoras para a divisão de problemas. Evoluindo o modelo estruturado, surgiu a programação modular que interliga módulos contendo diferentes funções através de interfaces.

A programação estruturada e, em seguida, a programação modular, apesar de todas as suas limitações, satisfizeram as necessidades da comunidade responsável pela criação de sistemas durante alguns anos. Porém, na década de 1970, os conhecimentos e técnicas para desenvolvimento de software não eram suficientes para contornar alguns problemas, necessidade que ficou ainda mais evidente com a crescente demanda do público consumidor de software. A Orientação a Objetos (OO) surgiu da necessidade de simular a realidade, criando uma abstração do cotidiano, na tentativa de representar as características relevantes dos objetos envolvidos no sistema que se tenta simular. Ao se definir um programa em termos de objetos, passa-se a compreender o software de um modo mais profundo, o que nos obriga a abordar as funções dos objetos em um nível conceitual (WINCK e JÚNIOR, 2006).

Nos últimos anos, a OO tornou-se o paradigma mais utilizado no desenvolvimento de software. Porém, diversas questões ligadas, principalmente, à separação de interesses em uma aplicação, começaram a se tornar evidentes com o crescimento constante da complexidade dos softwares desenvolvidos. Estas limitações foram o fator estimulante para a busca por um paradigma complementar à OO. Para entender melhor estas limitações, precisa-se aprofundar os conceitos de interesses em um sistema e sua separação, seguindo critérios desejáveis.

## 2.2 Separação de Interesses

A estratégia de dividir um problema complexo em partes menores para solucioná-lo de forma mais eficiente é comum em várias naturezas. Segundo DIJKSTRA (1976), a principal característica do pensamento inteligente é avaliar um aspecto de um determinado problema de forma isolada, mas sem se esquecer que outros aspectos serão influenciados pelas soluções alcançadas.

Segundo WINCK e JÚNIOR (2006), o surgimento de linguagens de programação capazes de representar o conceito de procedimentos possibilitou trazer para as linguagens esta estratégia. A divisão de um problema o torna mais compreensível e possibilita que mais pessoas possam trabalhar simultaneamente em uma solução.

Sistemas de software reúnem diferentes interesses de diferentes aspectos. Estes interesses são modularizados por meio de diferentes abstrações providas pelas linguagens e paradigmas de programação (FIGUEIREDO, 2006). Na programação estruturada, os interesses são divididos de acordo com as funcionalidades do software. Funções são implementadas em módulos ou procedimentos únicos. Desta forma, interesses relativos a dados ficam distribuídos em diversos módulos, como exemplificado na figura 2.1.

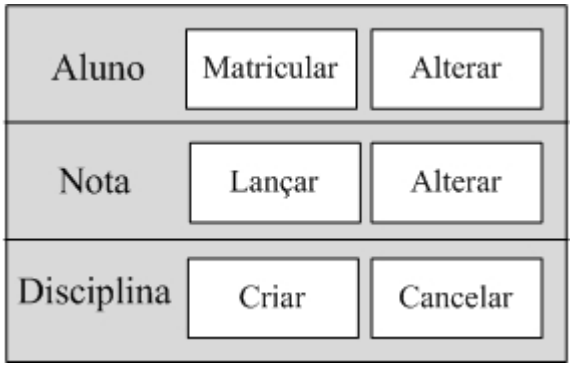


**Figura 2.1 Separação em dados e funções**

Na orientação a objetos, a utilização de classes, métodos e atributos como estruturas de abstração torna possível uma separação em duas dimensões. Passa-se a lidar com os dados e as funções que utilizam o tipo de dados como visto na figura 2.2.

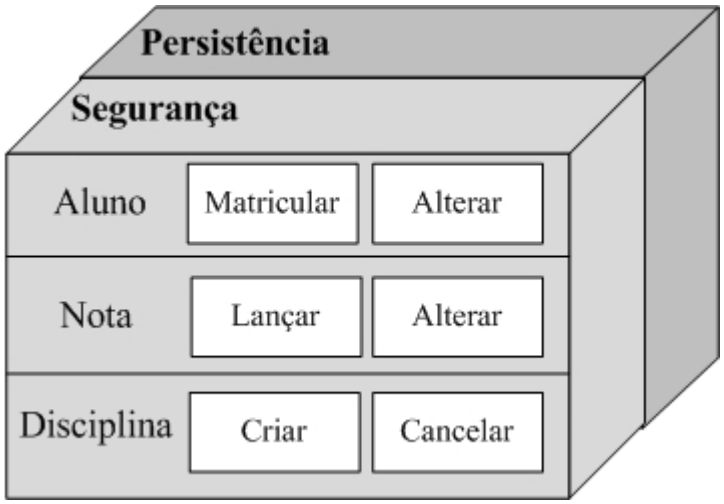


**Figura 2.2 Separação em dados, funções e aspectos**

Dada a complexidade de sistemas de software atuais, o número de interesses envolvidos em um único projeto se torna cada vez maior. Pode-se visualizar um projeto de software atual como uma grande combinação de interesses implementados em diversos módulos (LADDAD, 2002). A figura 2.3, adaptada de LADDAD (2002), ilustra esta situação.

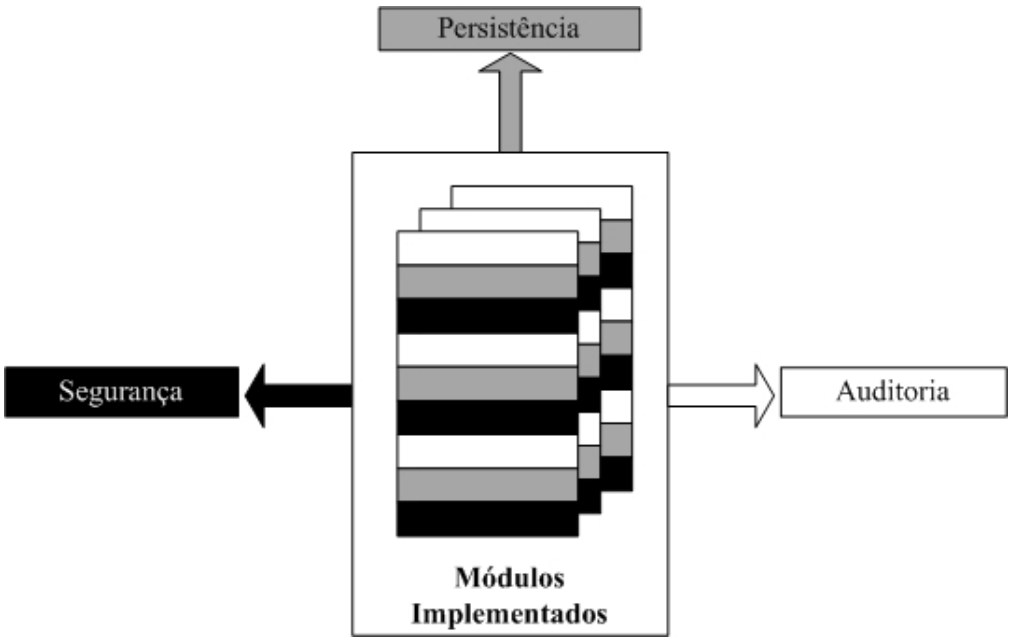


**Figura 2.3 Implementação de um software como um conjunto de interesses**

Desta forma, segundo FIGUEIREDO (2006), seria interessante concentrar em uma única localidade física toda a parte dedicada ao tratamento de certo interesse, facilitando seu estudo e entendimento.

Para esta situação a Orientação a Objetos oferece, em suas estruturas, meios para se representar os elementos e os conceitos que determinam certo problema. Daí, para cada problema específico, pode-se retirar o conceito de domínio da aplicação. Domínio é o agrupamento de informações sobre o mundo real que podem ser associadas de forma a tornarem identificáveis as relações entre os itens que compõem o domínio (WINCK e JÚNIOR, 2006).

As limitações da OO surgem quando precisamos demonstrar, em uma solução, não somente os elementos necessários de acordo com o domínio da aplicação, mas, também, elementos que afetam, geralmente, assuntos relacionados com a programação em si. Como exemplo, temos o tratamento de exceções, gerenciamento de uma camada de persistência etc. A presença destes interesses em várias partes do código determina a sua transversalidade.

## 2.3 Orientação a Aspectos

No inicio dos anos 1990, pesquisadores[1] voltaram suas atenções para a dificuldade da OO e seus padrões de projetos, até então conhecidos, em lidar com estes interesses. Estes estudos resultaram nas primeiras definições de **interesses entrecortantes**[2] e, em seguida, da programação orientada a aspectos.

"Encontramos diversos problemas de programação onde técnicas OO não são suficientes para capturar todas as importantes decisões do modelo que o programa deve implementar. Analisamos o porquê destas decisões serem tão difíceis de se capturar. Chamamos essas questões de aspectos, e a razão que os tornavam difíceis de se capturar é que estas entrecortam as funcionalidades básicas do sistema. Apresentamos a base para uma nova técnica de programação chamada Programação Orientada a Aspectos que torna possível expressar claramente programas que envolvem esses aspectos, incluindo o isolamento, composição e reuso apropriados do código do aspecto" (KICZALES *et al.*, 1997).

[1] Em especial o grupo liderado por Gregor Kickzales, no XEROX Palo Alto Research Center (PARC).
[2] Alguns termos de destaque deste trabalho seguem a tradução proposta em WASP (2004).

Utilizando apenas a OO como ferramental, ao programar-se interesses entrecortantes, pode surgir dois problemas de acoplamento e coesão no código: o espalhamento e o entrelaçamento do código. Um interesse é dito **espalhado** quando afeta vários componentes do sistema e **entrelaçado** quando se mistura com outros interesses dentro de um módulo (FIGUEIREDO**,** 2006).

A seguir é apresentado um exemplo de implementação de um interesse transversal na ferramenta servidora de aplicações TomCat[3]. Na figura 2.4 a barra destacada representa a inexistência de código espalhado na implementação do analisador XML da aplicação exemplo.

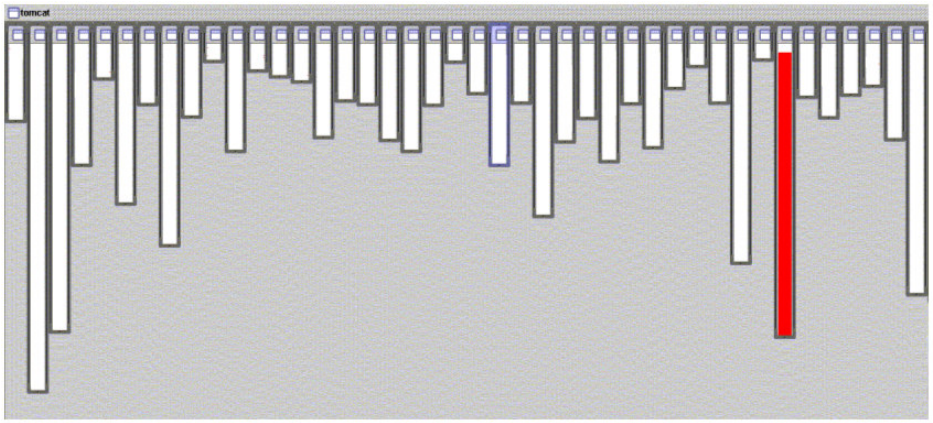


**Figura 2.4 Código do analisador XML no TomCat**

Já na figura 2.5, a implementação da funcionalidade de auditoria na aplicação é destacada. Percebe-se que o código encontra-se espalhado por diversas classes no sistema.

## 2.4 Desenvolvimento Orientado a Aspectos

Desde a sua primeira definição concreta, a orientação a aspectos cresceu e apresenta-se, segundo LADDAD (2002), como o próximo passo na evolução das metodologias de programação. Neste contexto, o Desenvolvimento de Software Orientado a Aspectos (DSOA) surge como uma nova subárea da Engenharia de Software preocupada em desenvolver métodos, técnicas e ferramentas que dêem apoio a todas as fases do desenvolvimento (SILVA, 2006).

[3] Disponível em: <http://tomcat.apache.org/>

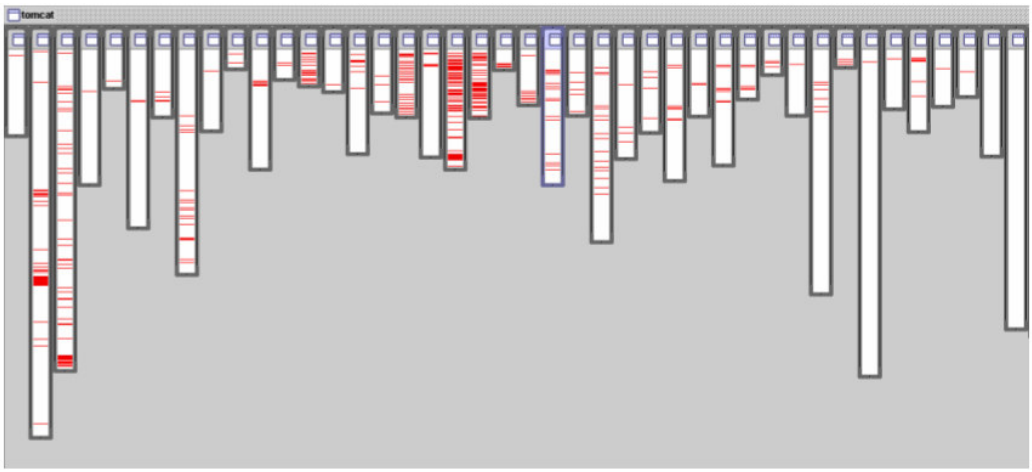

**Figura 2.5 Funcionalidade de auditoria**

Dentre estas fases podemos destacar, além da programação, outras etapas em que a OA já possui trabalhos específicos como levantamento de requisitos, modelagem, refatoração e testes. Assim como na OO, somente após as ferramentas de programação OA terem conseguido atingir certa estabilidade é que as outras etapas do ciclo de vida do desenvolvimento OA começaram a receber maiores atenções. A tarefa de um engenheiro de software sobre um sistema OA apresenta-se como um desafio, dadas as diferentes possibilidades de técnicas disponíveis em cada estágio do ciclo e o fato de poucas delas possuírem casos reais de uso divulgados. A escolha de certa ferramenta ou abordagem pode ser determinada por diversos fatores como requisitos do sistema, práticas organizacionais, restrições determinadas pelo ambiente de desenvolvimento ou pelas ferramentas utilizadas etc (FILMAN *et al.*, 2004).

Em geral, todas as pesquisas já apresentadas sobre estas etapas em OA utilizam experiências e ferramentas da OO como base. No caso do levantamento de requisitos, os interesses transversais previamente identificados são analisados e, de acordo com a técnica utilizada, são marcados como futuros aspectos na aplicação. Já para a modelagem, é clara a influência da OO e de sua principal linguagem de modelagem em uso, a UML. Os principais trabalhos de Modelagem Orientada a Aspectos derivam dos modelos UML utilizados para os conceitos de OO. Para a fase de testes algumas propostas teóricas demonstram grande eficiência para tratar dos possíveis problemas causados pela interação entre classes e aspectos. Mas, infelizmente, não existem ferramentas consolidadas para a aplicação destas teorias.

Analisar o suporte da Engenharia de Software para o estado-da-arte da OA nos leva à conclusão de que muitas técnicas já surgiram para as etapas citadas, porém é necessária experiência para validar as capacidades destas técnicas e reduzir assim o leque de abordagens. No capítulo quatro, alguns trabalhos já realizados sobre outras etapas que não a POA são apresentados.

# Capítulo 3
# Programação Orientada a Aspectos

Como visto no capítulo anterior, os problemas existentes na Programação Orientada a Objetos começaram a ser analisados nos anos 1990 por KICZALES *et al.* (1997) e sua equipe. Segundo CAMPOS (2006), nesta mesma época estudantes da Northeastern University nos Estados Unidos desenvolviam pesquisas similares. O termo POA nasceu, então, dessa união, entre novembro de 1995 e maio de 1996, com o intuito de apresentar técnicas de programação capazes de cobrir as limitações das linguagens OO (CAMPOS, 2006).

A POA busca a separação do código relacionado aos interesses transversais de um sistema de forma bem definida e centralizada. Para isto, em POA é introduzido um novo mecanismo para abstração e composição, que facilita a modularização destes interesses, **o aspecto**. Segundo WINCK e JÚNIOR (2006), a POA possibilita um nível maior de abstração no desenvolvimento de software, o que deixa os interesses sistêmicos de uma aplicação bem separados e em locais definidos, aumentando sua reusabilidade, facilitando, assim, sua manutenção e sua legibilidade.

Para alcançar todos esses resultados, sistemas que utilizam POA são compostos dos seguintes componentes:

- **Linguagem de Componentes** – através dela o programador cria as funcionalidades básicas do sistema sem se preocupar com a implementação futura dos aspectos. Como exemplos temos Java, PHP, C#, C++ etc.
- **Linguagem de Aspectos** – são responsáveis por prover o ferramental necessário para a codificação dos aspectos e as estruturas necessárias para determinar o funcionamento dos mesmos.
- **Combinador de Aspectos** – sua execução é que torna capaz a junção dos códigos escritos na linguagem de componentes e na linguagem de aspectos. Na figura 3.1 é ilustrado o processo de criação de um sistema orientado a aspectos a partir destes três componentes.

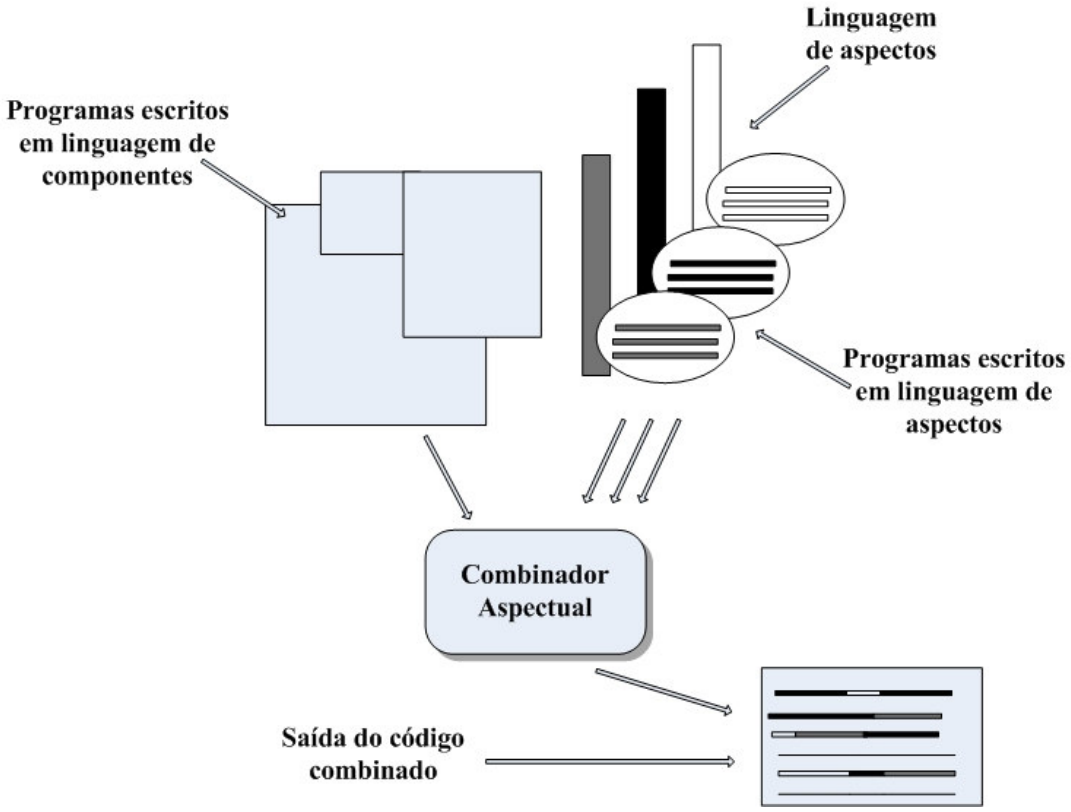


**Figura 3.1 Composição de um sistema orientado a aspectos (WINCK e JÚNIOR, 2006)**

## 3.1 AspectJ

Dentre as diversas opções de linguagens para POA existentes, AspectJ (KICZALES *et al.*, 2001) ganhou grande destaque pois, além de ser baseada em uma linguagem muito popular, foi também estruturada pela mesma equipe responsável pelos primeiros passos deste novo paradigma.

AspectJ é uma extensão de Java com mecanismos que provêem suporte para a implementação de duas estruturas entrecortantes. Segundo KICZALES *et al*. (2001) estes mecanismos são classificados como dinâmicos e estáticos. O primeiro tipo torna capaz a criação de código que será executado somente quando condições pré-determinadas sejam atingidas, já o segundo possibilita a definição de novas estruturas em classes existentes.

A escolha de Java como base para a criação da linguagem AspectJ cria algumas preocupações de compatibilidade. Apesar de muitos dos problemas imaginados se demonstrarem praticamente nulos, tais preocupações ainda existem e as mesmas são citadas abaixo segundo WINCK e JÚNIOR (2006):

- **Compatibilidade Total:** um programa Java válido deve ser também um programa AspectJ válido.
- **Compatibilidade de Plataforma:** todo programa AspectJ deve ser executado em uma máquina virtual Java.
- **Compatibilidade de Ferramentas:** ferramentas existentes, como ambientes de desenvolvimento, ferramentas de documentação, ferramentas de projeto etc., devem ser possivelmente estendidas para AspectJ de forma natural.
- **Compatibilidade para o programador:** para o programador, AspectJ deve ser como uma extensão Java.

Segundo CAMPOS (2006), os principais pontos fortes da linguagem AspectJ são a declaração concisa de aspectos, a checagem estática de pontos de atuação, a integração madura com importantes ambientes de programação e a ampla documentação.

## 3.2 Conceitos da orientação a aspectos

A seguir serão apresentados os principais elementos da POA e a sua implementação correspondente na linguagem AspectJ.

### 3.2.1 Pontos de Junção

Pontos de Junção são locais muito bem definidos em uma aplicação onde um ou vários aspectos poderão atuar. Os pontos de junção sempre estão associados a um contexto de execução e é através deles que se originam os pontos de atuação.

Dentre os contextos de execução que caracterizam pontos de junção pode-se citar a chamada e a execução de métodos e construtores, a inicialização e utilização de objetos e seus atributos etc. Desta forma, em AspectJ a identificação de um ponto de junção torna-se uma tarefa simples. A figura 3.2 destaca a ocorrência de um ponto de junção ligado à chamada do método `imprimeData`. O contexto associado ao ponto de junção ilustrado na figura 3.2 pode englobar o objeto chamador do método, o objeto alvo e os argumentos do método.

```
public class Impressao {

    private Calendar calendario;

    public Impressao()
    {
        calendario = Calendar.getInstance();
        this.imprimeData();
    }

    public void imprimeData()
    {
        System.out.println("Olá, a data atual é: "+calendario.getTime());
    }
}
```

**Figura 3.2 Ponto de junção na classe** `Impressao.java`

### 3.2.2 Pontos de Atuação

Pontos de Atuação são estruturas capazes de agrupar um conjunto de pontos de junção. Através deles é possível especificar regras genéricas para definir os pontos em todo o código da aplicação que serão considerados pontos de junção.

É responsabilidade dos pontos de junção também apresentar os dados do contexto da execução de cada ponto de junção (WINCK e JÚNIOR, 2006). Na figura 3.3 é apresentado um ponto de atuação que utiliza o ponto de junção da figura 3.2.

```
pointcut ptcImpressao() : call
                          (public void Impressao.imprimeData());
```

**Figura 3.3 Ponto de atuação para o método** `imprimeData`

### 3.2.3 Adendos

Um adendo é a implementação do código que deverá ser executado quando certo ponto de junção é alcançado. Este código geralmente é executado antes, durante ou depois de se alcançar o ponto de junção. Sua estrutura básica é dividida em duas partes: definições para o

ponto de atuação que determina a execução do código e a implementação em si do código do adendo.

Em AspectJ, o adendo possui mecanismo similar a um método. Ele engloba, como elementos chave, a definição do ponto de atuação que determinará a sua execução e o código que deve ser executado em tal situação. Ele faz parte da implementação de um aspecto e possui três possíveis posições de execução pré-definidas pelas seguintes palavras reservadas: ***before*** (antes), ***around*** (durante) e ***after*** (após).

Na figura 3.4 é apresentada a implementação de um adendo para o ponto de atuação da figura 3.3. Este adendo determina que antes da execução do método deverá ser impressa uma mensagem na tela.

```
before() : ptcImpressao()
{
    System.out.println("Aguarde ...");
}
```

**Figura 3.4 Exemplo de adendo em AspectJ**

### 3.2.4 Declarações Inter-tipos (Inserções ou Introduções)

É a forma proposta pela POA para alteração de estruturas estáticas da codificação. Declarações inter-tipos são capazes de incluir atributos e alterar a herança de uma classe por exemplo. A figura 3.5 mostra a declaração de uma inserção que define a herança da classe `Impressao` em relação à classe `ImpressaoBasica` em tempo de execução.

```
declare parents : Impressao extends ImpressaoBasica;
```

**Figura 3.5 Alteração da estrutura da classe `Impressao` em tempo de execução**

### 3.2.5 Aspectos

Em POA, aspecto é a estrutura disponibilizada para agrupar os interesses transversais do sistema, evitando os problemas já citados decorrentes da POO, como o código espalhado e o código emaranhado.

Aspectos encapsulam pontos de atuação, declarações inter-tipos e adendos numa unidade modular capazes de expressar de forma clara e objetiva certo interesse transversal de uma aplicação. Além disso, um aspecto em AspectJ pode conter atributos e métodos como uma classe Java qualquer.

Na figura 3.6 é apresentado o aspecto `AspectoImpressao` que reúne todos os elementos das figuras anteriores e, desta forma, define como este interesse transversal, mesmo que sem objetivos práticos, está construído nesta aplicação exemplo.

O resultado da execução desta aplicação, com a mensagem definida no adendo `ptcImpressao`, sendo exibida exatamente antes da execução do método `imprimeData` da classe `Impressão` pode ser visualizado na figura 3.7.

```
public aspect AspectoImpressao {

    declare parents : Impressao extends ImpressaoBasica;


    pointcut ptcImpressao() : call
                              (public void Impressao.imprimeData());


    before() : ptcImpressao()
    {
        System.out.println("Aguarde ...");
    }
}
```

**Figura 3.6** `AspectoImpressao` **englobando todos os elementos previamente abordados**

```
Aguarde ...
Olá, a data atual é: Tue Nov 13 02:43:45 BRST 2007
```

**Figura 3.7 Resultado da execução da aplicação exemplo**

## 3.3 Onde aplicar POA

Como visto, a POA é uma excelente ferramenta para se alcançar qualidades desejáveis em um software. Segundo WINCK e JÚNIOR (2006) o uso de POA é aconselhado na codificação dos seguintes interesses sistêmicos:

- **Sincronização de Objetos Concorrentes:** Com o uso da POA é possível modularizar de forma eficaz políticas de sincronização.
- **Distribuição:** A POA pode ser utilizada em vez de técnicas de refatoração para adaptar certa versão de um software para uma versão considerada apta para distribuição.
- **Tratamento de Exceções:** Através da POA é possível centralizar políticas para tratamento de exceções para unidades elementares mais legíveis e de manutenção facilitada.
- **Persistência:** Utilizando aspectos a tarefa de abstrair e criar esta camada de negócios torna-se mais simples.
- **Auditoria:** Com o uso da POA, o interesse de auditoria pode ser implementado independentemente do contexto da aplicação.

No próximo capítulo serão demonstrados trabalhos que propõem a aplicação dos conceitos da OA em outras etapas do desenvolvimento de um software. A linguagem AspectJ ganha destaque nestes trabalhos por ser, em muitos deles, utilizada como padrão na nomenclatura de elementos.

# Capítulo 4

# Orientação a Aspectos no Ciclo de Desenvolvimento

O objetivo das técnicas de Desenvolvimento de Software Orientado a Aspectos é prover meios para a identificação, modularização e composição de interesses entrecortantes de forma sistemática durante o ciclo de vida de um software (BLAIR *et al.*, 2004). Com várias abordagens de programação concretas para este paradigma, os conceitos da OA passam então a ser aplicados em outros estágios.

Este capítulo visa apresentar algumas das principais propostas de aplicação de OA em outras etapas do ciclo de vida de desenvolvimento de software além da programação. Serão abordadas as etapas de levantamento de requisitos, modelagem e testes do software.

## 4.1.Levantamento de Requisitos Orientado a Aspectos

O papel da Engenharia de Requisitos (**ER**) no desenvolvimento do software é conciliar as características desejáveis de um software com a necessidade de se atender amplamente a todos os objetivos e restrições do escopo em questão. É necessário nesta etapa, elicitar, analisar conflitos, validar, priorizar, modificar e reusar requisitos, rastreá-los considerando sua origem, os componentes arquiteturais e o código que os implementa, dentre outras tarefas (SILVA, 2006).

Como visto, a OA propõe uma melhor separação dos interesses transversais de uma aplicação utilizando uma nova estrutura denominada aspecto. Esta modificação cria a necessidade de adaptação dos modelos de levantamento de requisitos para o novo paradigma. Segundo RASHID *et al.* (2002), a separação antecipada das propriedades entrecortantes facilita a localização de seus alcances nas etapas futuras assim como a sua influência em outros artefatos do sistema.

Enquanto algumas abordagens de ER se preocupam somente em identificar as necessidades do negócio da aplicação, não deve-se ignorar também as necessidades ligadas a questões técnicas. Durante as etapas iniciais do ciclo, alguns interesses não-funcionais podem ser enumerados facilmente de acordo com o escopo do projeto. Em casos, como, por exemplo,

de aplicações bancárias e médicas, estes interesses podem ser citados pelos próprios interessados no projeto.

Os estudos sobre modelos de Engenharia de Requisitos Orientados a Aspectos (EROA) ainda não estão consolidados. Uma das primeiras propostas foi realizada por GRUNDY (1999), criando um modelo voltado para o desenvolvimento baseado em componentes. A proposta de RASHID *et al.* (2002, 2003) tem sido adotada em outras pesquisas e, apesar de incompleta, apresenta novos conceitos específicos para a OA. Destaque também para as pesquisas de integração de interesses transversais na modelagem de requisitos de SILVA (2006).

### 4.1.1. Aspectos em Requisitos Baseados em Componentes

GRUNDY (1999) é referenciado em diversas pesquisas sobre EROA por ser o primeiro a utilizar conceitos de aspectos nesta etapa de desenvolvimento. Porém, sua abordagem é focada no desenvolvimento baseado em componentes e, segundo RASHID *et al.* (2002), não mostra evidências de que seja utilizável no desenvolvimento de software em geral. Apesar de não detalhar as técnicas utilizadas em suas pesquisas, GRUNDY determina que cada componente do software pode caracterizar diversos aspectos do sistema como um todo.

### 4.1.2. Aspectos Candidatos

Esta proposta apresenta recursos para a EROA de forma mais ampla, possibilitando a sua utilização não só no desenvolvimento baseado em componentes. Existem duas versões para esta proposta: a primeira é voltada para o conceito de PREView[4] e a segunda voltada para o uso com a *Unified Modelling Language*[5]. O autor destaca a importância de se avaliar todos os interesses de um sistema desde a fase de requisitos até a implementação, visando a identificação antecipada dos interesses que entrecortam outras funcionalidades.

[4] PREView é uma metodologia baseada nas etapas iniciais da Engenharia de Requisitos. Sua abordagem possibilita a conversão de metas de alto-nível em requisitos e restrições no sistema (SOMMERVILLE, 1997).

[5] A *Unified Modelling Language*(UML) é a linguagem utilizada tanto na Engenharia de Requisitos como no Projeto de um software por diversos autores. No presente trabalho, termos e diagramas da UML serão utilizados em diversas situações. No anexo I é realizado um breve apanhado dos principais elementos desta linguagem.

#### 4.1.2.1. PREView

Esta versão do modelo de aspectos candidatos é composta de seis etapas como vistas na figura 4.1. Inicialmente, duas etapas se entrelaçam de acordo com os métodos usados para recolhimento dos dados relativos ao projeto e análise dos requisitos. Estas etapas têm como propósito justamente descobrir requisitos, identificar interesses e relacionar estes itens. Diferentes ângulos são utilizados para se analisar todos os requisitos e relacioná-los aos interesses identificados. Estas diferentes formas de se análise são chamadas de **Pontos de Visão**.

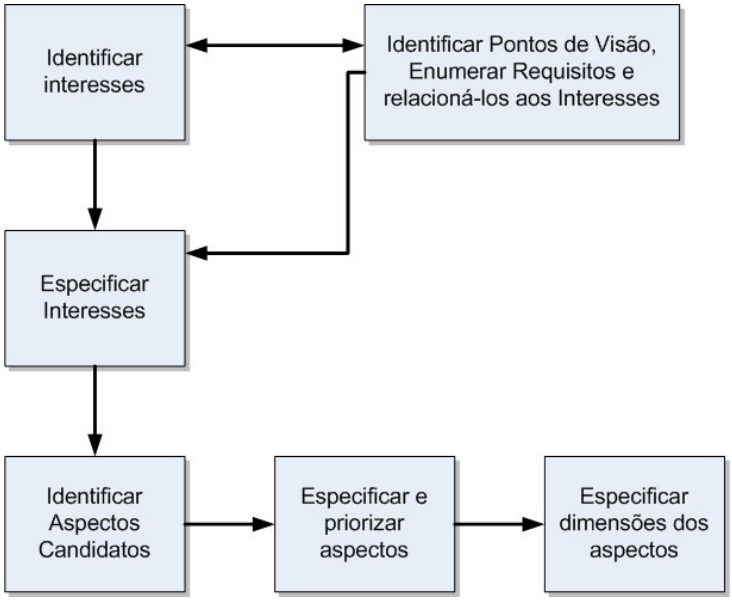


**Figura 4.1 Modelo de aspectos candidatos com PREView**

Em seguida é preciso especificar com mais detalhes cada interesse. Se um requisito cruza vários pontos de visão então ele é um **aspecto candidato** (SILVA, 2006). Então, deve-se detalhar estes candidatos a aspectos para, assim, aprimorar suas características e, possivelmente, detectar interações e/ou conflitos entre eles. Estes conflitos são solucionados na etapa seguinte, onde é aplicado o conceito de **prioridade sobre aspectos**.

A última atividade consiste em identificar as **dimensões dos aspectos**. Segundo RASHID *et al.* (2002), nesta fase do DSOA, os aspectos já definidos podem afetar o restante das etapas nas duas dimensões descritas abaixo:

- **Mapeamento**: um aspecto pode convergir para uma característica/função do sistema (por exemplo, um método), uma decisão (por exemplo, uma decisão de arquitetura) ou para um aspecto. Por isso, nesta fase é utilizado o termo **aspecto candidato**, já que certo interesse transversal não necessariamente será convertido em um aspecto nas etapas futuras do desenvolvimento.
- **Influência**: representa a influência do aspecto sobre outros componentes definidos durante o levantamento de requisitos. Daí a necessidade de se solucionar todos os conflitos na etapa anterior deste modelo. Estes conflitos podem alterar completamente a influência de um aspecto sobre o sistema.

#### 4.1.2.2. UML

Neste modelo, o pesquisador seguiu os mesmos princípios adotados na estratégia baseada em PREView, porém, utilizando os recursos da UML. Nesta segunda proposta o processo é dividido em três partes principais. São elas: interesses entrecortantes, interesses funcionais e requisitos compostos, como visto na figura 4.2.

- **Interesses entrecortantes**: nestas etapas são definidos os interesses funcionais entrecortantes, e segundo o conceito de RASHID *et al.* (2002), os aspectos candidatos.
- **Interesses funcionais**: apresenta uma abordagem clássica em UML para levantamento de requisitos. São utilizados diagramas de caso-de-uso e de seqüência nesta etapa.
- **Requisitos compostos**: iniciada pela composição dos interesses funcionais com os aspectos; daí são identificados e solucionados conflitos resultantes desta composição.

Para a identificação e especificação dos interesses entrecortantes é proposta uma tabela que contém informações como a prioridade do interesse, os requisitos e os modelos UML afetados pelo mesmo. Esta tabela é representada na tabela 4.1.

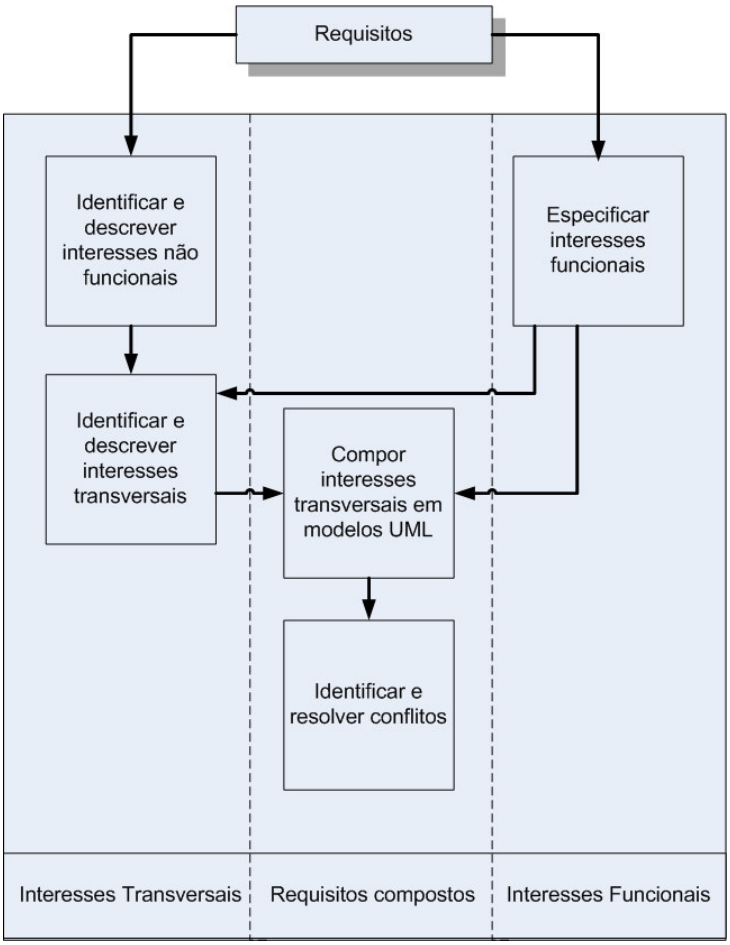


**Figura 4.2 Modelo de aspectos candidatos com UML**

**Tabela 4.1 Tabela para especificação de interesses entrecortantes**

| **Interesse Entrecortante** | <Nome> |
|---|---|
| **Descrição** | <Descrição Executiva> |
| **Prioridade** | <Máxima, média ou mínima> |
| **Lista de Requisitos** | <Requisitos que descrevem o interesse> |
| **Lista de Modelos** | <Modelos UML influenciados pelo interesse> |

Na fase de composição dos interesses entrecortantes identificados e especificados com os demais interesses avaliados, o pesquisador lança mão de um novo elemento para os diagramas de caso de uso já gerados. São incluídas no modelo estruturas estereotipadas com o

nome do interesse entrecortante que representam. Estas estruturas são relacionadas com as já existentes no modelo através de relacionamentos estereotipados de acordo com as suas características:

- **Modificação:** o interesse entrecortante modifica o interesse funcional;
- **Sobreposição:** o interesse entrecortante sobrepõe o interesse funcional;
- **Encapsulamento:** o interesse entrecortante encapsula o interesse funcional.

Além de utilizar a UML como linguagem para organização dos requisitos avaliados, RASHID *et al.* (2003) traz uma melhoria na etapa de resolução dos conflitos entre aspectos candidatos. Esta melhoria é dada através da criação de uma matriz de contribuição entre estes aspectos. A contribuição negativa de um candidato a aspecto para a realização de outro gera valores baseados no peso desta contribuição.

### 4.1.3. Integração de Características Transversais

SILVA (2006) destaca que, ao contrário de RASHID *et al.* (2002, 2003), não está interessada em tratar a EROA especificamente, mas sim, auxiliar a ER em detectar os interesses de um sistema de forma mais clara, mas que, possivelmente, esses interesses podem se tornar aspectos. Esta proposta também é baseada no modelo de metas e tem como ponto forte a divisão em três etapas com funções bem definidas. Cada uma destas etapas possui parâmetros de entrada e saída pré-definidos e ao final de uma iteração são obtidas diferentes visões do modelo já integrado.

A primeira etapa é responsável por prover apoio à modelagem de requisitos através do modelo de metas. Cada modelo de metas (*goals*) é utilizado para retratar um grupo de características do sistema tais como as funcionalidades específicas da aplicação, requisitos de segurança, tratamento de exceções, dentre outros (SILVA, 2006). Requisitos, funcionais ou não, são modelados como modelos de metas e podem ser considerados como interesses entrecortantes de acordo com os seus relacionamentos. A característica mais interessante desta etapa é que, cada requisito é analisado e modelado de forma independente. Assim, certa característica pode ser considerada funcional em um contexto e não-funcional em outro.

Na fase de composição, os modelos gerados na etapa anterior são utilizados como entrada, além de regras de composição que irão controlar o processo. Com estes atributos em

mãos é realizada a geração automatizada de um modelo integrado. Deste modelo integrado, diversas visões podem ser derivadas.

Na atividade final, o modelo integrado e informações sobre visões comuns servem de entrada. Esta etapa tem como objetivo exibir o sistema em subdivisões facilitando seu entendimento. Cada visão retrata o sistema por um certo ângulo, por exemplo: uma visão de quais partes do sistema utilizam um mecanismo de autenticação ou sofrem impacto dele, uma visão que mostre quais usuários tem autorização para utilizar quais partes do sistema, o impacto que o sistema sofre se uma outra característica transversal for adicionada, uma visão das matrizes de rastreabilidade, dentre outras (SILVA, 2006). A figura 4.3 ilustra este processo.

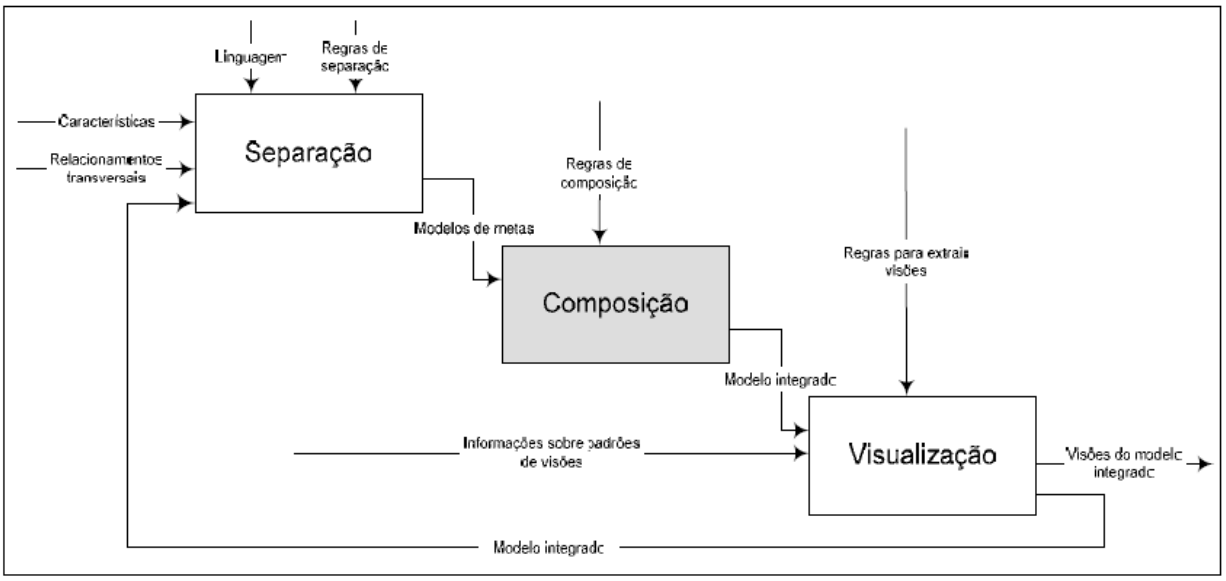


**Figure 4.3 Modelo de integração de requisitos (SILVA, 2006)**

### 4.1.4 Comparação entre as propostas

A proposta de GRUNDY, apesar da importância histórica para os estudos de EROA, não apresenta grandes acréscimos para este campo. Por outro lado, as propostas de RASHID *et al.* (2002, 2003) têm grande destaque por apresentarem formas simples e objetivas de se propagar o paradigma OA para esta etapa do desenvolvimento. Já SILVA (2006), apesar de não abordar diretamente a EROA, tem grande destaque em seu trabalho por separar os interesses de uma aplicação e apresentá-los de uma forma de fácil entendimento.

## 4.2 Modelagem Orientada a Aspectos

A modelagem de software contemporânea adota uma perspectiva orientada a objetos, na qual os principais blocos básicos dos sistemas de software são objetos e classes (CHAVEZ, 2004). Porém, desde o surgimento da POA, percebeu-se a necessidade de uma linguagem padrão para a criação de modelos voltados para esta nova realidade de programação. Dentre as principais mudanças já apresentadas que a OA trouxe pode-se destacar a definição de Aspecto em si e a forma que estes aspectos entrecortam outros elementos e blocos do restante da aplicação.

Expressar de forma clara estes novos elementos e características é tarefa vital da **Modelagem Orientada a Aspectos** (MOA) para alcançar a melhoria na qualidade do software OA e facilitar o entendimento dos novos conceitos pela comunidade. A fase de modelagem possui fator crítico também por ser a principal ligação entre os resultados obtidos do levantamento de requisitos e a implementação da solução esperada.

Levando-se em conta as principais regras e normas que ditam as técnicas atuais de modelagem de software é possível enumerar as qualidades desejáveis de uma proposta de linguagem para MOA:

- Facilidade para se capturar qualquer aspecto de um domínio de problemas e suas dependências;
- Fornecer notação independente de linguagem de programação;
- Possuir uma notação expressiva e bem definida;
- Tratar aspectos como cidadãos de primeira classe;
- Oferecer suporte à representação explícita de interações de aspectos;
- Oferecer suporte à expressão de interfaces de aspectos;
- Oferecer suporte à decomposição de interfaces de aspectos.

### 4.2.1 Proposta de Suzuki e Yamamoto (1999)

Esta proposta é baseada na introdução de uma nova meta-classe "*Aspect*" ao meta-modelo da UML. Esta nova classe assume definições muito próximas a uma classe ou interface na UML. Seguindo esta proposta, um aspecto pode contar com diversos atributos e operações, além de poder participar de diversas associações, generalizações e relações de dependências com

outros elementos. Por outro lado, a proposta tem como ponto fraco a alta ligação com a linguagem AspectJ.

Para tratar dos conceitos como o impacto de um aspecto sobre outras classes (seu alcance) e as modificações provocadas por situações características, como o processo de **combinação** (*weaving*), o modelo utiliza a classificação de operações com estereótipos da UML. Especificamente para o processo de combinação, podem ser adicionadas ao modelo informações sobre a etapa onde o processo ocorre. Como exemplo, na linguagem AspectJ podemos definir um aspecto que afeta certo componente, através de **inserções**. O estereótipo `<<weave>>` indica a declaração de um adendo ou de uma inserção. Na figura 4.4 temos um aspecto com duas inserções declaradas segundo SUZUKI e YAMAMOTO (1999).

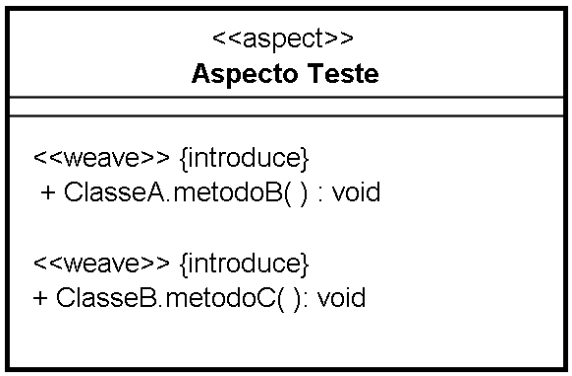


**Figura 4.4 Modelo de aspecto**

Ainda tratando do impacto de alterações causadas pelos Aspectos na modelagem, esta proposta utiliza um campo chamado "designador" que indica os atributos, métodos, classes e/ou pacotes afetados. Na figura 3.4 as classes `ClasseA` e `ClasseB` representam estas classes.

Para representar a relação entre um Aspecto e suas classes base, os autores utilizam dependências do tipo estereótipo `<<realize>>` da UML. Isto se deve ao pressuposto de que uma realização é a relação entre um elemento de especificação do modelo e o modelo que o implementa. O modelo das classes geradas após o processo de combinação utiliza classes com o estereótipo `<<wovenclass>>`. A figura 4.5 representa o processo de combinação segundo esta proposta.

Esta foi uma das primeiras propostas de MOA. Os pesquisadores tratam o processo de desenvolvimento de um software OA de forma muito estática e pouco detalhada e a ausência de modelos para características cruciais, como pontos de junção, adendos e interesses transversais, reforçam a fraqueza do modelo.

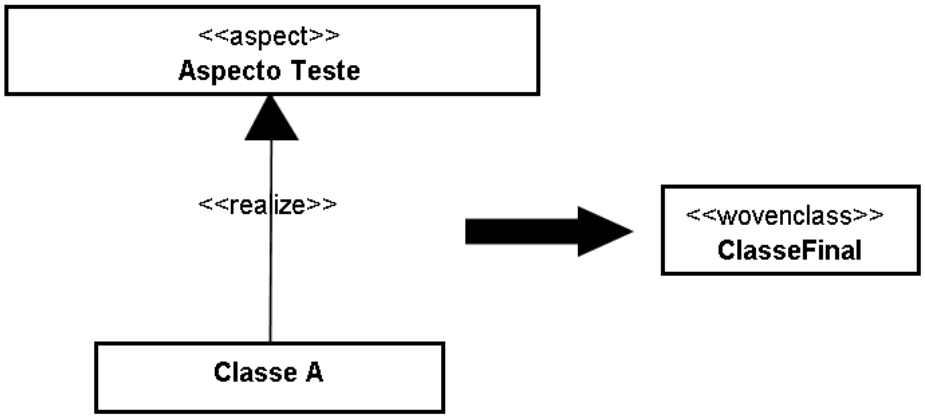


**Figura 4.5 Modelo do resultado do processo de combinação**

### 4.2.2 Padrões de Composição

Originado da **Programação Orientada a Sujeitos** (POS) (MELO e BET, 2003) e fundamentado por CLARKE (2001), o **Modelo de Projeto Orientado a Sujeitos** (MPOS) apresenta um novo conceito, os **Padrões de Composição** (PC). Planejados inicialmente para lidar com interesses transversais na POS, logo os Padrões de Composição foram adaptados para o ambiente Orientado a Aspectos, especificamente para a linguagem AspectJ.

Padrões de Composição são modelos em UML para projetos baseados em sujeitos que esperam classes (ditas classes de padrão) e operações (referidas como operações modelo) como parâmetros. Estes padrões ocorrem quando um sujeito de *design* com interesses transversais é mesclado com outro(s) sujeito(s) da mesma forma em ocasiões diferentes. O MPOS usa pacotes UML para encapsular todos os elementos relacionados com um requisito em particular.

#### 4.2.2.1 Modelo de Projeto Orientado a Sujeitos (MPOS)

A seguir são apresentados alguns conceitos relacionados à Orientação a Sujeitos. Estes conceitos são fundamentais para a definição de Padrões de Composição e, conseqüentemente, a sua adaptação para Orientação a Aspectos.

- **Sujeitos de Design (*Design Subjects*):** São os objetos da modelagem responsáveis por reunir todos os elementos de certo requisito. São separados como visões diferentes e caracterizados pelo estereótipo `<<subject>>` na UML.
- **Relacionamentos de Composição:** No MOPS diversos sujeitos podem ser relacionados entre si gerando um novo sujeito. Para isto são utilizadas as regras definidas nos relacionamentos de composição.
- **Estratégias de Integração:** Estas estratégias definem como os relacionamentos de composição serão utilizados na geração dos sujeitos de saída do processo de composição. A estratégia utilizada nos Padrões de Composição é chamada ***merge*.** Esta estratégia une efetivamente os sujeitos de entrada, organizando diferenças nas especificações de cada elemento (exceto para operações) baseado em estratégias de reconciliação especificas. Operações de ***merge*** combinam os comportamentos assumidos por cada operação correspondente. Isto é alcançado com a geração de um modelo de interação que realizará a operação composta através de cada operação original correspondente.

Na figura 4.6 tem-se um exemplo da estratégia ***merge*,** na qual ocorrem dois sujeitos (S1 e S2), representados por duas classes de mesmo nome "A". O arco entre as classes, com setas em suas extremidades, determina que os sujeitos sejam mesclados. Além disso, a representação "match[name]", indica que os elementos de mesmo nome correspondam entre si. Como resultados têm-se o sujeito "S1S2" que reúne os atributos dos dois sujeitos de entrada e suas operações. Seguindo a estratégia ***merge***, a operação "op1" é composta pela delegação das operações originais presentes renomeadas de acordo com o sujeito de origem. É apresentado no modelo um diagrama de interação que representa o comportamento da delegação.

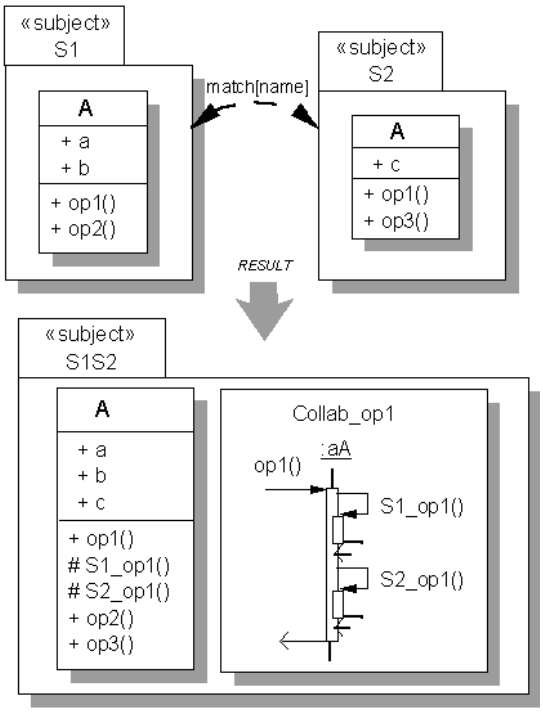


**Figura 4.6 Estratégia de integração *merge* (CLARKE, 2001)**

### 4.2.2.2 Padrões de Composição e a UML

A seguir são apresentados os principais elementos estendidos da UML para satisfazer as regras e definições da MPOS. Nos diagramas de UML, um padrão de composição é representado como um tipo especial de pacote, estereotipado como `<<subject>>` e com os parâmetros não vinculados identificados como em uma classe parametrizada da UML. Os sujeitos de design (não transversais) comuns são apresentados como uma classe normal. Um padrão de composição pode conter um diagrama de interação para cada operação composta, que descreve seu comportamento transversal como uma delegação a cada uma das operações de entrada (renomeadas) correspondentes.

A figura 4.7 mostra a representação de um comportamento transversal no modelo de padrões de composição. A classe de padrão *Subject* representa sujeitos cujas modificações de estado estão sendo observadas, já a classe de padrão *Observer* representa qualquer classe que esteja observando o estado do sujeito. As operações `_aStateChange`, `update`, `start` e `stop` representam parâmetros não vinculados no contexto do sujeito `Observer`.

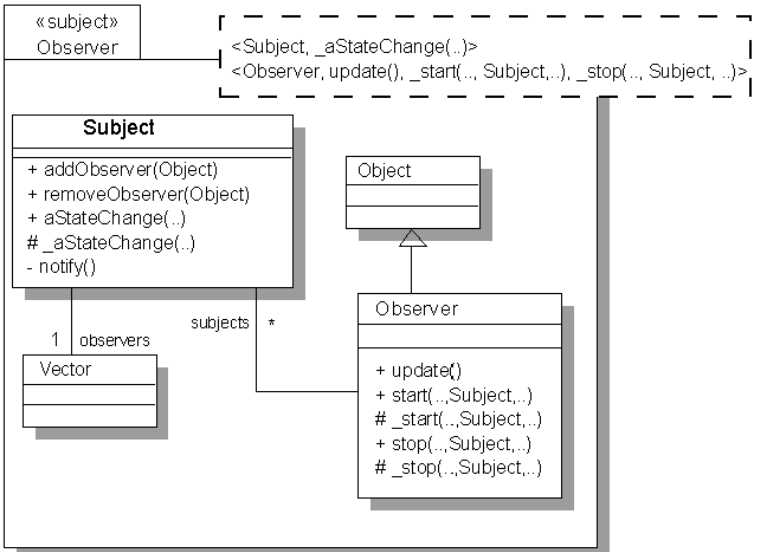


**Figura 4.7 Interesse transversal em Padrões de Composição (CLARKE, 2001)**

#### 4.2.2.2.1 Relacionamentos de Composição de Vinculação

Estes relacionamentos são a representação dos Relacionamentos de Composição do MPOS. São utilizados entre dois ou mais sujeitos que devem ser combinados especificando também uma estratégia de integração para ser utilizada. Esse relacionamento é representado por uma linha tracejada com o complemento "*bind*[ ]", que define os elementos que substituem os parâmetros não vinculados dentro do padrão de composição. Na figura 4.8, por exemplo, o padrão de composição *Observer* é vinculado ao sujeito *Library* através de um relacionamento de composição.

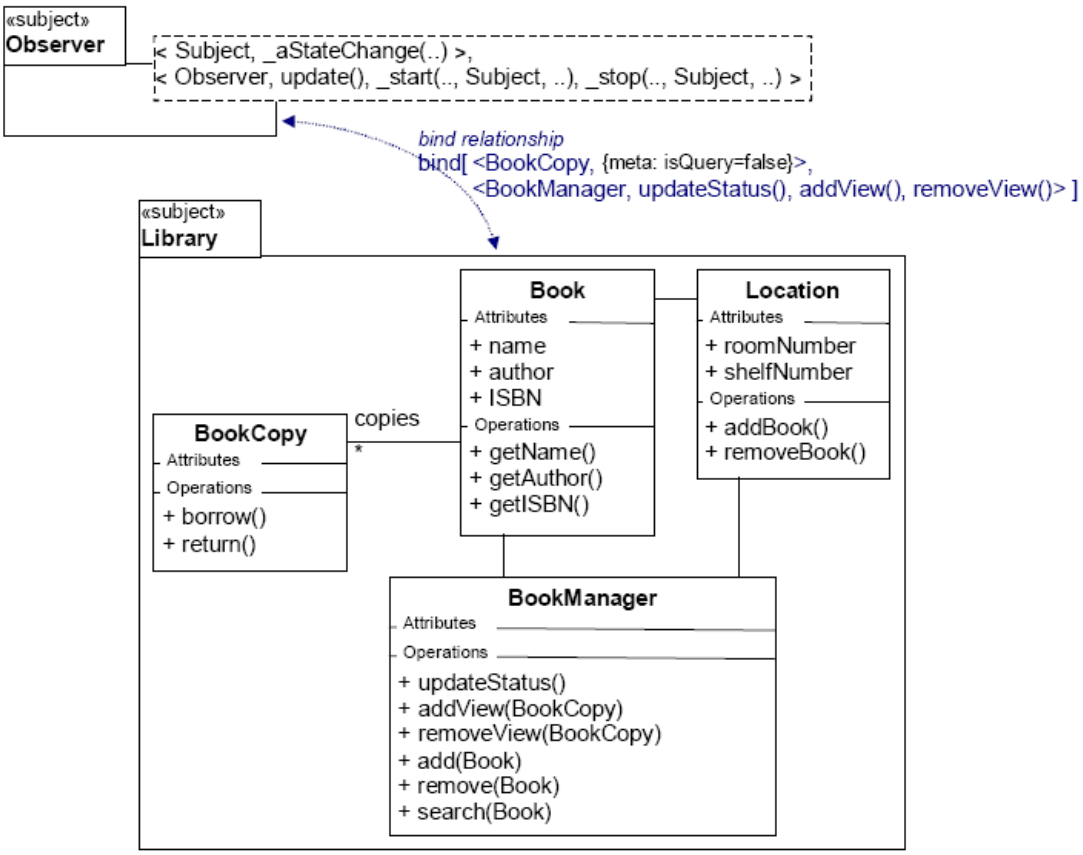


**Figura 4.8 Vinculação em Padrões de Composição (CLARKE, 2001)**

#### 4.2.2.2.2 Integração de Sujeitos

Após a junção do padrão de composição *Observer* com o sujeito *Library*, temos a saída representada na figura 4.9. Percebe-se que cada entrada transversal e saída integrada (por exemplo, `_aStateChange`) é substituída tanto na classe *BookCopy* como no diagrama de colaboração. Seguindo a estratégia de integração, as operações de entrada são renomeadas utilizando o nome do sujeito em questão para evitar duplicidade nos nomes. Todas estas modificações são atualizadas nos locais devidos a fim de se evitar referências falhas.

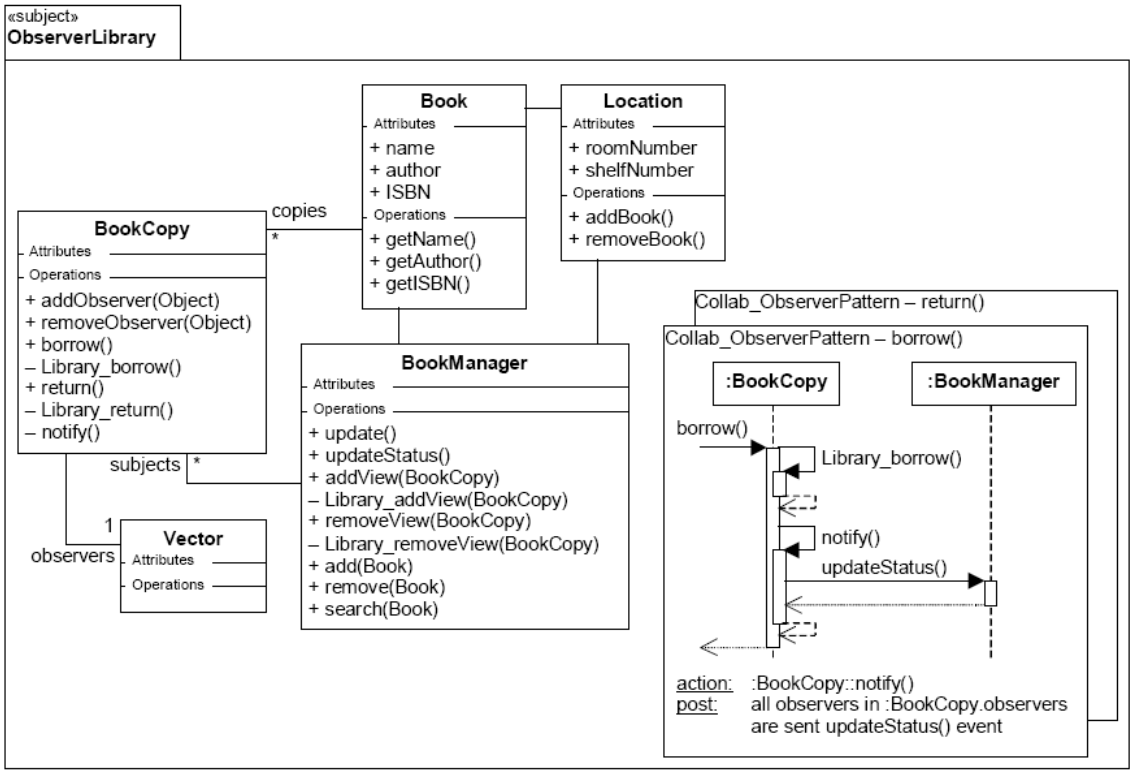


**Figura 4.9 Sujeitos *Observer* e *Library* integrados (CLARKE, 2001)**

### 4.2.2.3 Utilizando Padrões de Composição para DSOA

O modelo de Padrões de Composição é independente de qualquer modelo de programação. Padrões de composição propiciam a separação e a modelagem de comportamentos entrecortantes no MPOS assim como AspectJ propicia separação e programação de comportamentos entrecortantes no DSOA. Na tabela 4.2 é representado o mapeamento apresentado por CLARKE para realizar essa interação entre os modelos.

**Tabela 4.2 Relação entre elementos da POS e elementos da linguagem AspectJ**

| AspectJ | Padrão de Composição |
|---|---|
| Aspecto | Sujeito de Design |
| Conjunto de Pontos de Junção | Operações parametrizadas não vinculadas devem ser definidas e referenciadas com especificações de interação, denotando que se tratam de conjuntos de pontos de junção. |
| Comportamento Transversal | Com um diagrama de interação, comportamentos transversais devem ser executados quando uma operação parametrizada não vinculada é acionada. |

#### 4.2.2.4 Considerações sobre Padrões de Composição

A proposta de CLARKE (2001) apresenta como a UML deve ser usada para se estender requisitos entrecortantes em um sistema. Porém, segundo STEIN (2003), a proposta é ineficiente quando se deseja trabalhar com sistemas utilizando o paradigma OA e AspectJ em diversas formas. Abaixo são destacados os principais pontos de falha da tentativa de adaptação dos Padrões de Composição de CLARKE (2001) para o DSOA, segundo STEIN (2003):

- Padrões de Composição não representam devidamente os conjuntos de pontos de junção como uma característica de aspectos. Eles deveriam ser extraídos dos diagramas de interação para descrever o tipo de conjunto e qual a ação deve ser realizada pelo mesmo de forma clara;
- Com Padrões de Composição somente interesses transversais estáticos são tratados;
- Aspectos em AspectJ devem conter componentes comuns de classes Java como atributos e operações, porém os modelos de PC utilizam pacotes da UML, impedindo o uso de tais elementos.

### 4.2.3 *Aspect Oriented Design Model* (AODM)

O modelo proposto por STEIN (2003) é baseado em uma extensão da UML para conceitos da orientação a aspectos, utilizando os termos da linguagem AspectJ. O trabalho compara os elementos do paradigma OA e as suas representações na linguagem com elementos da UML e, baseado em algumas considerações, utiliza os mecanismos de extensão da UML para obter a representação desejada. O pesquisador aborda também o processo de combinação realizado pelos compiladores de linguagens OA.

Este trabalho utiliza um exemplo removido do guia oficial da linguagem AspectJ, que apresenta o padrão de projeto `Observer`. Este padrão realiza um rótulo colorido (*color label*) que exerce papel de observador e muda sua cor sempre que um botão (*button*), que exerce papel de sujeito, é clicado.

#### 4.2.3.1 Elementos de AspectJ x UML

A seguir os principais elementos da linguagem AspectJ são relacionados com elementos da UML de acordo com o AODM.

#### 4.2.3.1.1 Pontos de Junção

Em AspectJ, pontos de junção são determinados como os pontos do código afetados em tempo de execução por um interesse transversal. O modelo base dos pontos de junção define quatro ações determinantes para a criação destes pontos: acesso a atributos, manipulação de erros, criação de objetos e invocação de métodos. Para representar pontos de junção no AODM são utilizados componentes de ligação dos Diagramas de Seqüência. A diferenciação das ligações que representam pontos de junção é feita através da espessura dos traços referentes a pontos de junção.

Estereótipos são utilizados para determinar o tipo de ponto de junção explícito no diagrama, especialmente para pontos de junção de criação de objetos e invocação de métodos. Na tabela 4.3 temos a relação entre estes tipos e os estereótipos correspondentes.

**Tabela 4.3 Relação entre ações específicas no código e os estereótipos da AODM**

| Tipo de Ação | Estereótipo |
|---|---|
| Invocação de Métodos de Atribuição (set) de valores de atributos | <<*set*>> |
| Invocação de Métodos de Leitura (get) de valores de atributos | <<*get*>> |
| Execução de determinado ponto de junção | <<*execute*>> |
| Inicialização de código necessário para execução de ponto de junção | <<*initialize*>> |

Na figura 4.10 é apresentado um exemplo de Diagrama de Seqüência utilizando as notações do AODM. As regiões afetadas pela existência de pontos de junção destacam-se e estereótipos facilitam o entendimento das modificações geradas pelos interesses transversais no código.

#### 4.2.3.1.2 Pontos de Atuação

Durante a análise dos pontos de atuação, ou conjuntos de pontos de junção, STEIN observou duas características importantes para a criação de uma extensão na UML para este elemento:

- Conjuntos de pontos de junção mostram ao combinador como os adendos devem ser inseridos no código base das classes;
- Por outro lado, a estrutura destes conjuntos na programação é muito similar a estrutura de métodos.

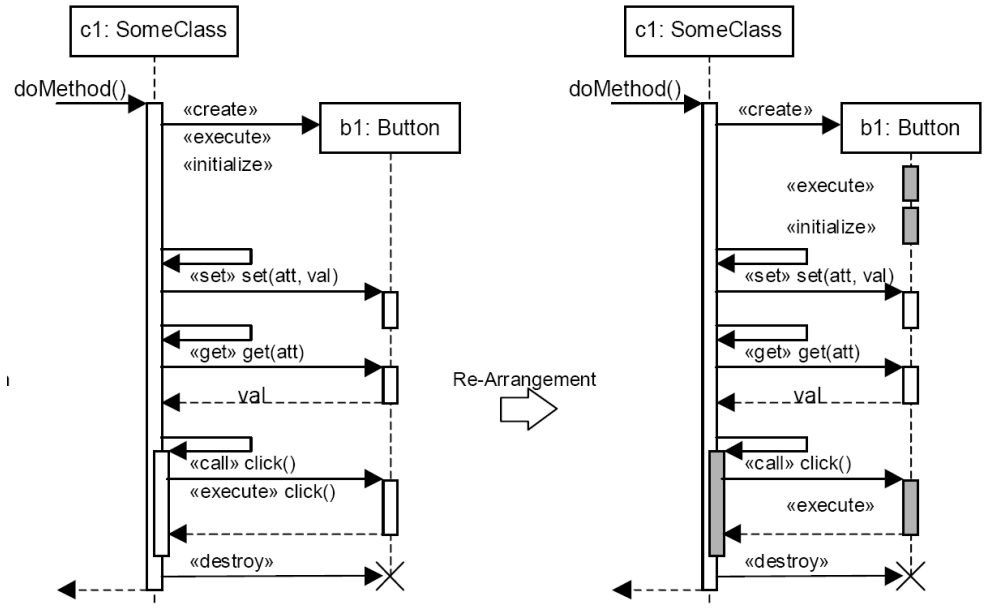


**Figura 4.10 Impacto da inserção de um interesse transversal em um diagrama de seqüência segundo o AODM (STEIN, 2003)**

Considerando-se que um conjunto de pontos de junção pode ser declarado dividido em duas partes, assim como um método que possui sua assinatura e sua implementação, a proximidade entre estas estruturas torna-se ainda mais evidente, como visto na figura 4.11. Desta forma, o AODM trata os conjuntos de pontos de junção como métodos sob um estereótipo específico: `<<pointcut>>`.

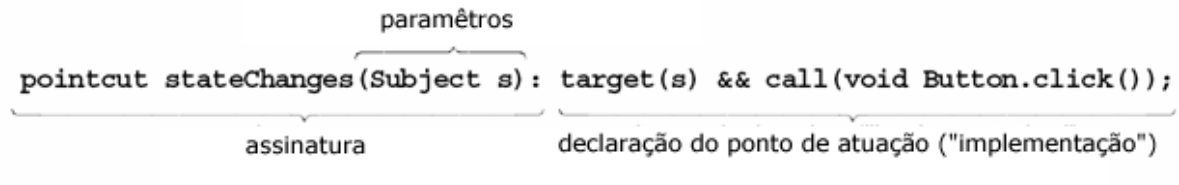


**Figura 4.11 Semelhanças entre a declaração de um ponto de atuação e de um método**

Os conjuntos de pontos de junção devem possuir, além do estereótipo característico, o meta-atributo "`base`" junto a sua assinatura. Este meta-atributo deve conter a declaração do conjunto de pontos de junção na notação de AspectJ. Na figura 4.12 é apresentada a notação completa de um conjunto de pontos de junção no AODM.

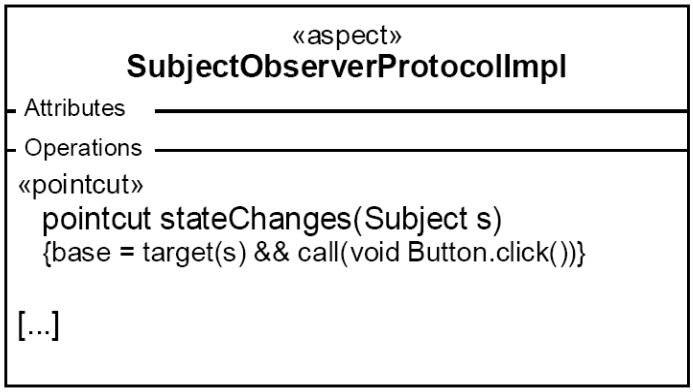


**Figura 4.12 Exemplo de conjunto de pontos de junção em AODM (STEIN, 2003)**

#### 4.2.3.1.3 Adendos

Um adendo contém o código que deve ser executado quando certo ponto de junção é alcançado. Cabe ao adendo indicar o ponto de junção responsável pela sua ativação. Segundo Stein, a declaração de um adendo, assim como os conjuntos de pontos de junção, podem ser comparados à declaração de um método quando dividida em duas partes, como na figura 4.13.

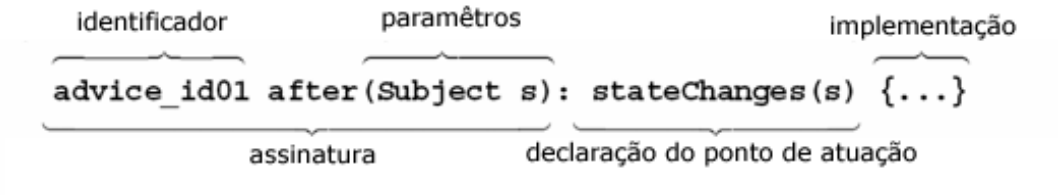


**Figura 4.13 Semelhanças entre a declaração de um adendo e a de um método**

Da mesma forma que os conjuntos de ponto de junção, os adendos são representados no AODM utilizando-se a mesma estrutura definida na UML para uma operação acrescida do estereótipo `<<advice>>`. Porém, a utilização de métodos estereotipados na representação de adendos causa um conflito com as especificações da UML. Afinal, como adendos não possuem um nome em AspectJ, é possível que dois ou mais adendos de um mesmo aspecto possuam a mesma assinatura. Para resolver este conflito, durante a modelagem orientada a aspectos utilizando AODM, adendos devem receber um "pseudo" identificador em sua assinatura.

A implementação do adendo é representada de forma separada da sua assinatura, sendo inserida em um lugar diferenciado, como uma anotação. O AODM define ainda que os adendos, assim como os pontos de junção, apresentem o meta-atributo "`base`". No caso dos adendos o valor do meta-atributo representa a expressão que contém o adendo. Um exemplo de notação de adendo seguindo AODM é apresentado na figura 4.14 e, na figura 4.15, tem-se a representação detalhada da implementação do adendo "`AfterAdvice_id`01".

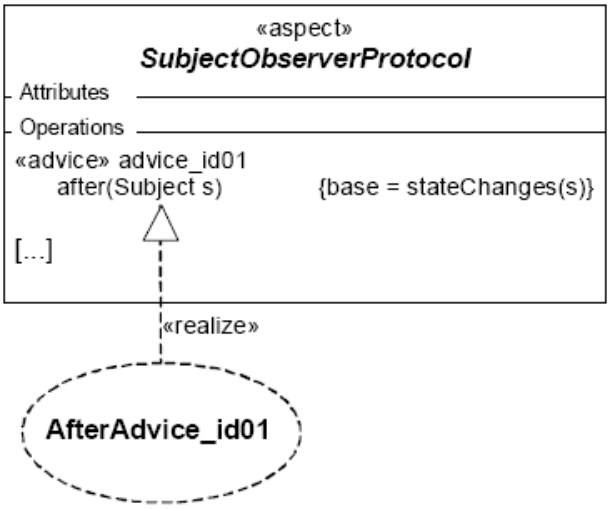


**Figura 4.14 Exemplo de um adendo em AODM (STEIN, 2003)**

#### 4.2.3.1.4 Introduções

Introduções entrecortam a estrutura estática de classes, podendo inserir novos membros a classe, como construtores, métodos e atributos. Além disso, introduções podem alterar a estrutura de herança e também as interfaces implementadas pela classe.

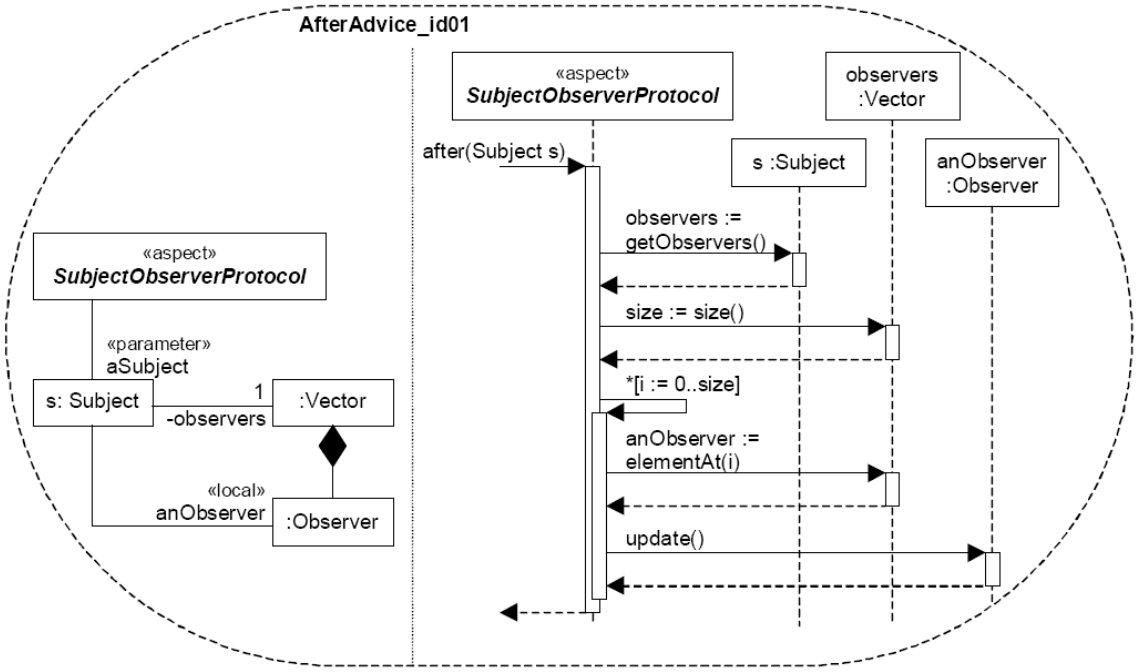


**Figura 4.15 Implementação detalhada de um adendo em AODM (STEIN, 2003)**

Para criar a notação adequada para introduções no AODM, Stein lançou mão de duas estruturas da UML: Parametrizações e Colaborações. A primeira estrutura, segundo a especificação da UML, não deve ser utilizada diretamente nos modelos. Esta estrutura parametrizada deve servir somente como forma de vinculação dos seus parâmetros com os argumentos de determinada estrutura, uma classe, por exemplo. Já as Colaborações descrevem a forma como uma classe ou pacote e suas ligações ou relacionamentos são utilizadas para alcançar o comportamento desejado de um classificador da UML (um caso de uso ou uma operação, por exemplo). O uso de parametrizações e colaborações com o estereótipo `<<introduction>>` é a forma proposta pelo AODM para situações de Introduções no projeto OA.

Para complementar esta representação, é utilizado o estereótipo `<<containsWeavingInstructions>>` que determina que certo parâmetro apresenta informações vitais para o futuro processo de combinação a ser realizado. Os parâmetros apresentam também o meta-atributo "`base`" que, neste caso, indica a classe referente ao

processo de Introdução que será realizado. Para demonstrar a utilização de introduções no AODM é apresentada a figura 4.16:

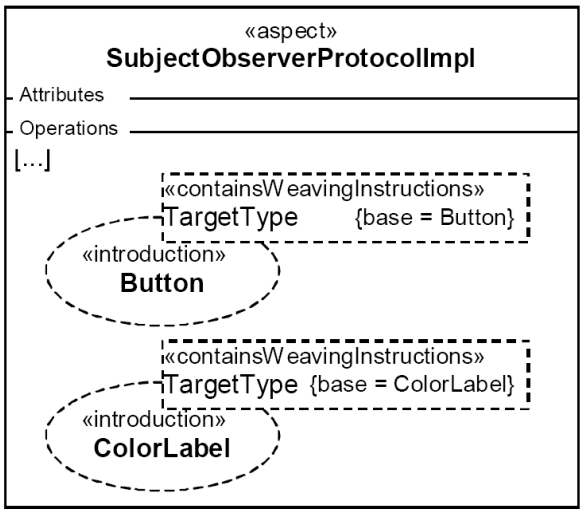


**Figura 4.16 Aspecto contendo inserções em AODM (STEIN, 2003)**

#### 4.2.3.1.5 Aspectos

São os responsáveis pela definição de como interesses e estruturas transversais irão atuar através de conjuntos de pontos de junção, adendos e introduções em um sistema orientado a aspectos (STEIN, 2003). A notação do AODM para aspectos é baseada no uso de classes com o estereótipo `<<aspect>>`.

Classes com este estereótipo são acrescidas de meta-atributos, capazes de englobar as meta-propriedades de um aspecto, como a cláusula de instanciação, a ocorrência de acesso privilegiado às classes base etc, que serão avaliadas durante o processo de combinação. Essas meta-propriedades são utilizadas na geração de código por parte do compilador e os meta-atributos não causam conflitos com a especificação da UML. Desta forma, um aspecto pode ser representado segundo a AODM de acordo com o exemplo da figura 4.17:

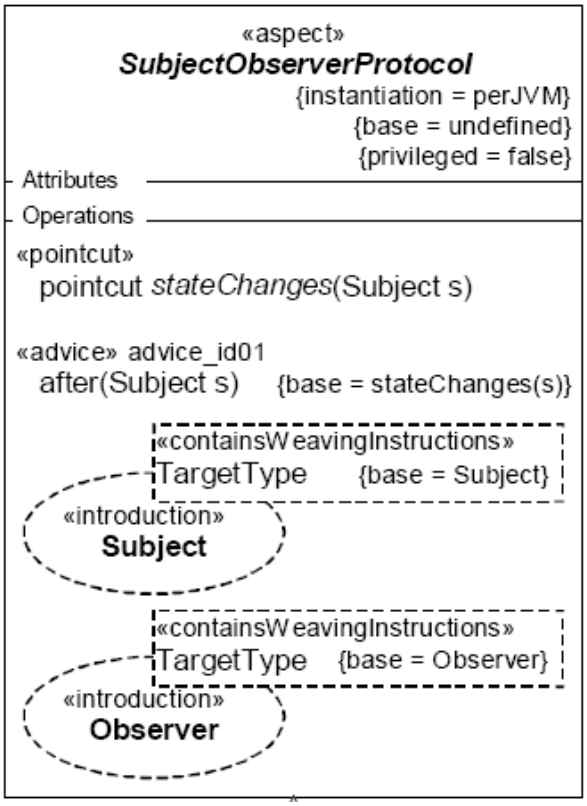


**Figura 4.17 Exemplo completo de um aspecto em AODM (STEIN, 2003)**

#### 4.2.3.2 O Processo de Combinação

O AODM apresenta também modelos e definições para a representação do processo de combinação realizado pelo compilador da linguagem AspectJ no momento da geração do código. Segundo Stein, o processo de combinação em AODM é subdividido em duas etapas: a combinação de adendos e a combinação de introduções. Nesta fase todas as informações já descritas são utilizadas para a criação de um novo modelo integrado e atualizado.

Este novo modelo é capaz de representar facilmente não só as estruturas previamente definidas pelo projetista como aspectos, classes, conjuntos de pontos de junção etc, mas também as relações geradas entre estes elementos pelos interesses transversais. A figura 4.18 demonstra esta saída do processo de combinação. Os relacionamentos entre os aspectos e todas os elementos afetados por eles são identificados pelo estereótipo `<<crosscut>>`.

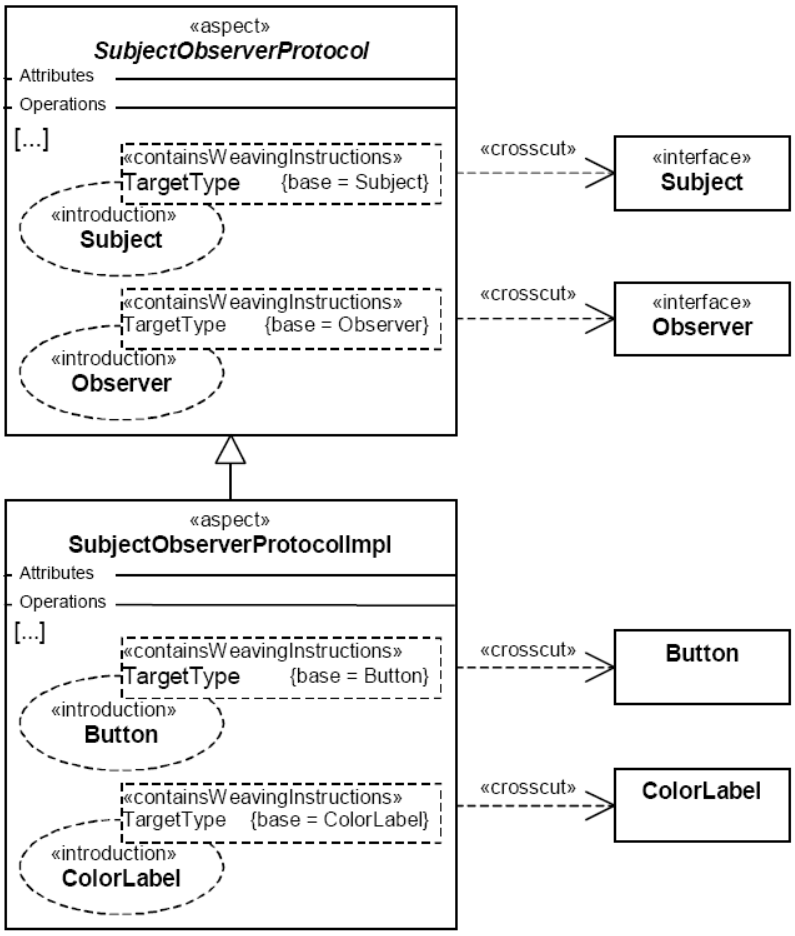


**Figura 4.18 Modelo atualizado após o processo de combinação aspectual (STEIN, 2003)**

### 4.2.4 Comparação entre as propostas

O modelo de SUZUKI e YAMAMOTO (1999), apesar de simples e pouco detalhado, foi um dos primeiros propostos para MOA. Em diversos trabalhos, especialmente sobre POA, são encontradas representações muito semelhantes a esta proposta. Já CLARKE (2001), apesar de iniciar seus estudos focando a POS, cria um modelo que, depois de adaptado para a linguagem AspectJ, apresenta-se muito eficaz. Outra grande vantagem dos Padrões de Composição está na sua independência de linguagem de programação, ao contrário de STEIN (2003) que deixa clara a dependência de sua proposta, o AODM, à linguagem AspectJ. Apesar desta dependência, o AODM é completo, tratando situações específicas de um projeto OA. Um ponto positivo comum dos três trabalhos apresentados é a utilização da UML como base, visto que esta linguagem já esta consolidada em projetos OO.

## 4.3 Testes Orientados a Aspectos

A etapa de testes de um software é o momento em que a equipe de desenvolvimento e os demais interessados colocam o software sob avaliação seguindo técnicas e metodologias de testes a fim de encontrar erros no código. Para esta tarefa é desejado que se gaste o mínimo de tempo e esforço possível.

Sob vários aspectos, o teste é um processo independente e o número de tipos diferentes de teste varia tanto quanto as diferentes abordagens de desenvolvimento (PRESSMAN, 2006). Atualmente, os testes podem ser diferenciados por diferentes técnicas e, em cada técnica, por suas fases. As principais técnicas de testes são os testes de caixa-preta e os testes de caixa-branca. Já as fases podem ser divididas em testes de unidade, testes de integração, testes de aceitação, testes de sistema, entre outras fases.

Testes de caixa-preta são executados sobre o software já existente, preocupando-se com funcionalidades e resultados esperados pela execução de determinada operação sem se preocupar com os passos realizados para se alcançar este resultado. Já os testes de caixa-branca, segundo PRESSMAN (2006), analisam a estrutura procedimental do software, validando caminhos lógicos e os possíveis relacionamentos entre diferentes componentes e estruturas.

No DSOO duas fases das técnicas de testes podem ser destacadas: testes de unidade e testes de integração. Na primeira, a menor unidade de um projeto de software é submetida a testes de forma isolada para verificar sua consistência e eficiência. Já na segunda, as unidades já testadas pela primeira fase são reunidas em estruturas definidas pelo projeto e então testadas. No caso de um software OO pode-se determinar como uma boa estratégia de testes a verificação do comportamento de cada classe do projeto, englobando seus métodos e seu comportamento global, e em seguida, o teste dos conjuntos de classes interligados desde as classes independentes até se alcançar todo o sistema.

A área de Testes Orientados a Aspectos ainda não possui muitas propostas. Os poucos trabalhos que abordam o tema preocupam-se com os testes estruturais, principalmente com as fases de destaque dos testes OO: unidade e integração. Dentre os trabalhos existentes pode-se destacar a proposta baseada em estados de XU e XU (2004) e o modelo de testes abordado por ZHAO (2003) e LEMOS (2005). Todas estas pesquisas concentram-se nos testes estruturais

do software e para isto, grafos de fluxo de controle são utilizados como representação do programa que será submetido aos testes.

Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos; a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando (BARBOSA *et al.*, 2000). No grafo de fluxo de um programa P os nós representam os pontos de decisão do programa e os arcos o fluxo de controle. Ao se percorrer um nó do grafo, todos os comandos representados por este nó devem ser executados, portanto, cada nó deve caracterizar um bloco indivisível de comandos. Na figura 4.19 são apresentados exemplos de grafos de fluxo para conjuntos básicos de instruções.

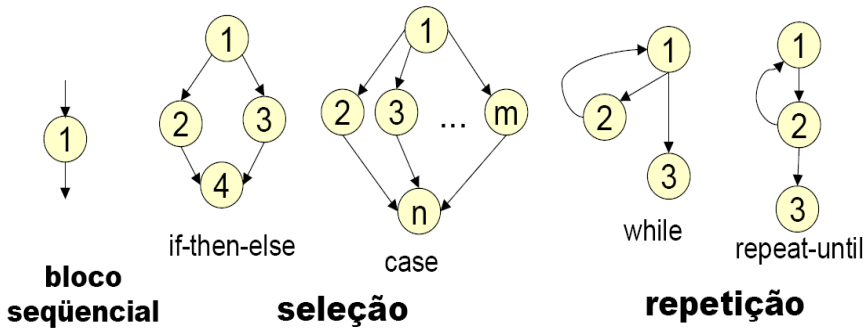


**Figura 4.19 Grafos de fluxo para estruturas básicas**

Segundo BARBOSA *et al.* (2000), os critérios analisados em um teste estrutural podem ser classificados como:

- **Critérios Baseados em Fluxo de Controle:** somente características de controle, como instruções de desvio, servem como parâmetro para determinar as estruturas necessárias;
- **Critérios Baseados em Fluxo de Dados:** utilizam os blocos que representam a definição ou o uso de uma ou mais variáveis como critério para realizar a movimentação sobre o grafo;
- **Critérios Baseados na Complexidade:** a complexidade do programa define os critérios de teste.

### 4.3.1 Testes Estruturais Orientados a Aspectos

Segundo LEMOS (2005), as fases de uma estratégia de testes OA podem ser divididas em:

- **Testes de Unidade: T**este de cada método e adendo isoladamente, também chamado de teste intramétodo ou intra-adendo.
- **Testes de Integração:** Testes de coleções de unidades dependentes – unidades que interagem por meio de chamadas ou interações com adendos. LEMOS subdivide essa fase nos seguintes tópicos:
    - **Inter-Método:** Teste de métodos públicos juntamente com outros métodos da mesma classe ativados direta ou indiretamente;
    - **Adendo-Método:** Teste de cada adendo com os métodos ativados por ele direta ou indiretamente;
    - **Método-Adendo:** Teste de cada método público e os adendos que o afetam direta ou indiretamente sem levar em conta a integração destes métodos com outros métodos ativados por eles;
    - **Adendo-Adendo:** Teste de cada adendo com outros adendos que o afetam direta ou indiretamente;
    - **Inter-Método-Adendo:** Teste que reúne as possibilidades geradas pelos quatro tipos descritos acima;
    - **Intraclasse:** Teste das interações entre métodos públicos ativados em diferentes seqüências considerando ou não a interação com aspectos;
    - **Inter-Classe:** Teste das interações entre classes considerando ou não a interação com aspectos.
- **Testes de Sistema:** Teste de módulos integrados gerando o sistema completo. Nesta etapa aplica-se o teste funcional.

As pesquisas de LEMOS (2005) e ZHAO (2003) preocupam-se basicamente com duas fases dos testes estruturais: os testes de unidade e os testes de integração.

#### 4.3.1.1 Testes de Unidade

Para os testes de unidade, LEMOS (2005) considera como a menor unidade de um sistema OA os métodos e adendos ao contrário de outras linhas de pesquisa, como ZHAO (2003) que considera o aspecto como a menor unidade. Porém, vem de ZHAO (2003) uma das principais definições utilizadas em testes estruturais de unidades OA. Segundo ele, é impossível a realização de testes em classes ou aspectos de forma isolada em sistemas OA. É necessária a aplicação de testes sobre os aspectos e todos os métodos cujos comportamentos são afetados por estes aspectos e também sobre as classes e os adendos que possivelmente irão alterar seu código através de introduções.

A principal modificação gerada pela inserção do conceito de Aspectos nos testes de um sistema está ligada ao fluxo de controle desta aplicação. A passagem do controle da aplicação para certo adendo quando um ponto de junção é alcançado possui características muito próximas com a chamada de um método. Porém, segundo MASIERO *et al.* (2006), a principal diferença está na inversão de dependência destes casos. A chamada de um método é prevista no código e realizada de forma explicita pelo programador, já no caso da execução de um adendo, os aspectos presentes na aplicação são os responsáveis por determinar a sua execução de acordo com as definições dos pontos de junção.

Para adequar os grafos de fluxo a esta nova realidade, LEMOS (2005) propõe a diferenciação dos nós que identificam ativações de adendos. Estes nós devem ser representados como elipses tracejadas e ainda conter as informações sobre o tipo de adendo e o aspecto que definem aquele ponto. Na figura 4.20 é apresentado um exemplo deste novo formato de nó introduzido por LEMOS (2005). Desta forma, pode-se facilmente identificar em um grafo de fluxo os nós que compõem um determinado interesse transversal da aplicação, uma vez que todos os pontos em que adendos são executados estão representados e trazem consigo a identificação do aspecto que causa sua ocorrência.


**Figura 4.20 Nó para representação de um aspecto em um grafo de fluxo**

A proposta de XU e XU (2004) é a criação de um modelo para criação de testes capazes de exercitar corretamente a interação entre classes e aspectos de um sistema baseando-se em um modelo UML. Este modelo UML não segue nenhum dos casos de modelagem orientada a aspectos citados em 3.2, porém é dada grande ênfase aos diagramas de seqüência gerados após o processo de combinação dos aspectos com as demais classes. A partir desses diagramas e da escolha de metas para a realização dos testes são criados grafos de fluxo capazes de analisar todos os caminhos possíveis do código.

Além da diferente definição de unidade mínima utilizada para testes e da nova representação gráfica do grafo de fluxo, o trabalho de LEMOS (2005) diferencia-se das pesquisas de XU e XU (2004) principalmente pela fonte utilizada para a criação do grafo. Enquanto LEMOS (2005) utiliza o código combinado intermediário gerado pela execução do combinador de aspectos seguida da execução do compilador da linguagem, XU e XU (2004) utilizam diagramas de seqüência adaptados para OA.

Para demonstrar as diferenças causadas de acordo com as fontes utilizadas por LEMOS (2005) e por XU e XU (2004) serão criados dois grafos de fluxo seguindo o método *Impressao* da classe *Impressao.java* apresentada no capítulo 3. Na figura 4.21 é apresentado o código intermediário para uma máquina virtual Java gerado após a execução do compilador AspectJ seguida da execução de um compilador Java.

```
public Impressao();
     0  aload_0 [this]
     1  invokespecial classes.ImpressaoBasica() [14]
     4  aload_0 [this]
     5  invokestatic java.util.Calendar.getInstance() : java.util.Calendar [20]
     8  putfield classes.Impressao.calendario : java.util.Calendar [22]
    11  aload_0 [this]
    12  invokestatic classes.AspectoImpressao.aspectOf() : classes.AspectoImpressao [67]
    15  invokevirtual classes.AspectoImpressao.ajc$before$classes_AspectoImpressao() : void [70]
    18  invokevirtual classes.Impressao.imprimeData() : void [25]
```

**Figura 4.21 Código intermediário Java após processo de combinação aspectual**

Segundo LEMOS (2005), é a partir deste código que se deve gerar o grafo de fluxo para os testes OA. Utilizando esta proposta, foi gerado o grafo de fluxo demonstrado na figura 4.22. Neste grafo, percebe-se que os nós 14, 20 e 22 são relacionados com a invocação de classes (*invokespecial*) e atribuição de valores (*putfield*) dentro do método. Ao se alcançar um ponto de junção determinado pelo aspecto *AspectoImpressao* o grafo indica o salto para a

execução do conteúdo do adendo especificado no aspecto. Após a execução o método retorna à sua execução original e é finalizado.

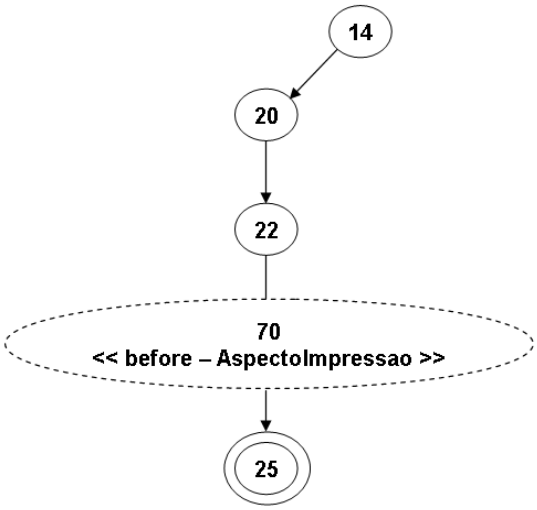


**Figura 4.22 Grafo de fluxo para o método `ImprimeData`**

Já na figura 4.23 temos o diagrama de seqüência modificado para representar a atuação do aspecto *AspectoImpressao* seguindo a proposta de XU e XU (2004). É a partir destes diagramas que esta proposta cria os grafos de fluxo utilizados nos testes. Na figura 4.24 temos o grafo gerado levando em consideração as seguintes restrições impostas por este trabalho:

- O grafo deve considerar somente as mensagens enviadas e não as mensagens de retorno.
- O grafo deve ser construído a partir da última mensagem enviada (modelo *bottom-up*).

A partir deste grafo, XU e XU sugerem a criação de uma árvore de fluxo que deve representar todas as alternativas possíveis da execução do código, levando em conta inclusive a existência de diversas classes concretas a partir de uma possível chamada a uma interface no código.

Para avaliação destes grafos, LEMOS (2005) propõe os seguintes critérios relativos ao fluxo de controle:

- **Todos-nós-transversais:** cada nó transversal, ou seja, cada adendo relativo a uma unidade afetada, deve ser executado ao menos uma vez por algum caso de teste T.

- **Todas-arestas-transversais:** todas arestas, que contenham um nó relativo à execução de um adendo em seu início ou fim, devem ser percorridas ao menos uma vez por algum caso de teste T.

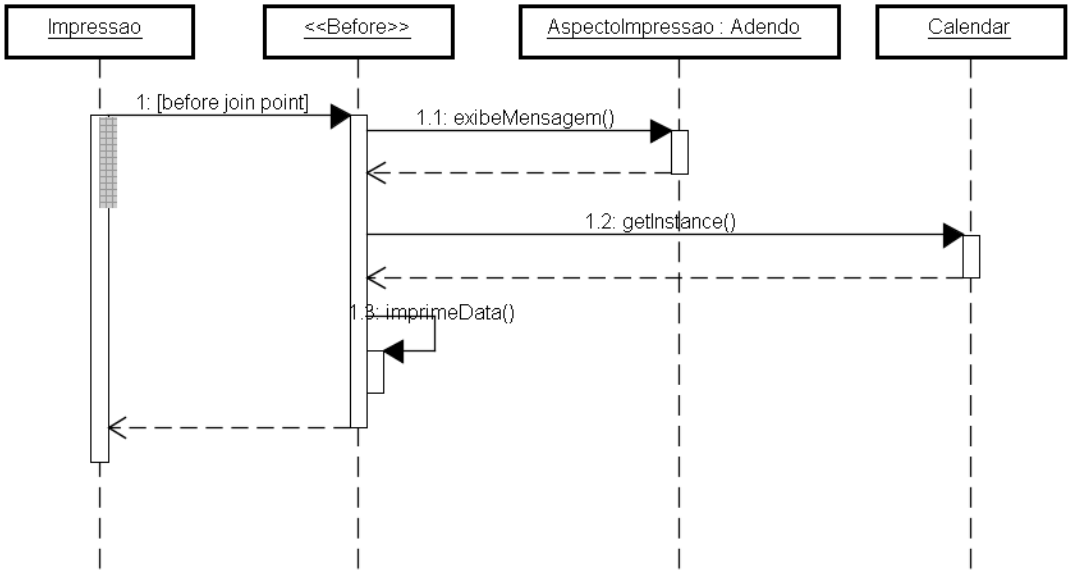


**Figura 4.23 Diagrama de seqüência do método `ImprimeData` alterado pelo adendo de `AspectoImpressao`**

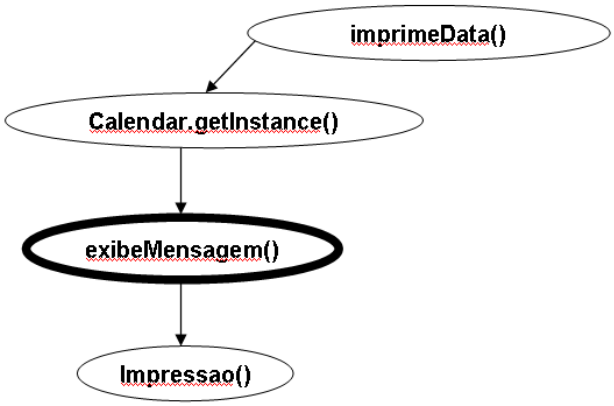


**Figura 4.24 Grafo de fluxo para o método `ImprimeData`**

#### 4.3.1.2 Testes de Integração

Para os testes de integração, LEMOS (2005) concentra esforços nas interações entre métodos e adendos. Segundo ele, a realização de todos os testes de unidade para validação dos critérios criados para OA pode ser uma tarefa custosa. Dessa forma, pode-se restringir a profundidade das chamadas e interações com adendos, para apoiar uma atividade de teste praticável.

A importância deste tipo de testes é evidente, já que a possível alteração de dados por parte da atuação de um aspecto através de um adendo pode, por exemplo, modificar os parâmetros da chamada de um método, comprometendo os resultados desejados.

Nestes testes são utilizados os mesmos conceitos apresentados durante o teste de unidades. Porém, são criados dois novos nós característicos, o nó de aprimoramento (**enh)** e o nó de retorno (**ret).** Estes nós apresentam-se como nós de desvio do controle do fluxo no grafo. O nó de aprimoramento é ligado ao primeiro nó relacionado à execução do adendo desejado em certo ponto do grafo, e o nó de retorno liga o nó final da execução do mesmo adendo com o nó que deveria ser o próximo na execução original do grafo.

Na figura 4.25 é apresentado o grafo de fluxo para aplicação do teste Método-Adendo a partir do exemplo da figura 4.22. Para garantir a consistência dos dados após a execução do adendo, LEMOS (2005) sugere a inclusão para cada nó de informações sobre a definição e a atribuição de valores para variáveis.

### 4.3.2 Ferramentas de Teste OA

Devido ao recente interesse nas pesquisas em Testes Orientados a Aspectos, não existem ferramentas em versões estáveis disponíveis. Nos trabalhos de LEMOS (2005) e MASIERO *et al.* (2006) é apresentada uma versão adaptada da ferramenta Jabuti, resultante do trabalho de VICENZI *et al.* (2004), chamada Jabuti/AJ.

O diferencial da ferramenta Jabuti é o fato de dispensar o código fonte da aplicação para a realização de testes. Estes testes são realizados a partir do código intermediário gerado para máquinas virtuais Java. Além disso, a ferramenta é capaz de construir grafos de fluxo e realizar a análise de diversos critérios desejados em uma aplicação OO. A extensão para OA implementada por LEMOS (2005) e MASIERO *et al.* (2006) tornam a ferramenta capaz de criar grafos incluindo instruções da linguagem AspectJ e, aumentam também o número de

critérios para analise, incluindo agora os critérios citados para testes de softwares orientados a aspectos.

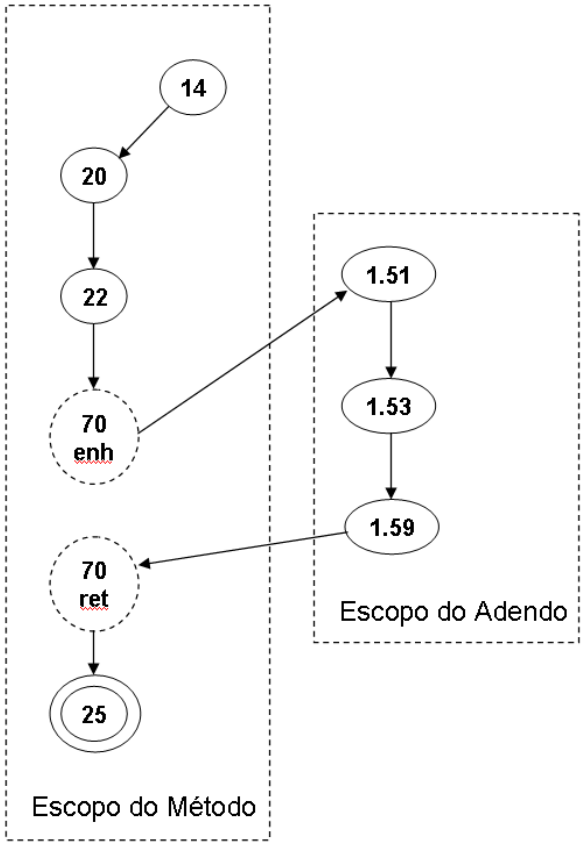


**Figura 4.25 Grafo de fluxo de dados do método `ImprimeData`**

Apesar de não possuir uma versão disponível para uso até o presente momento, a figura 4.26 apresenta a interface da tela de apresentação do grafo de fluxo da ferramenta Jabuti/AJ, de acordo com MASIERO *et al.* (2006).

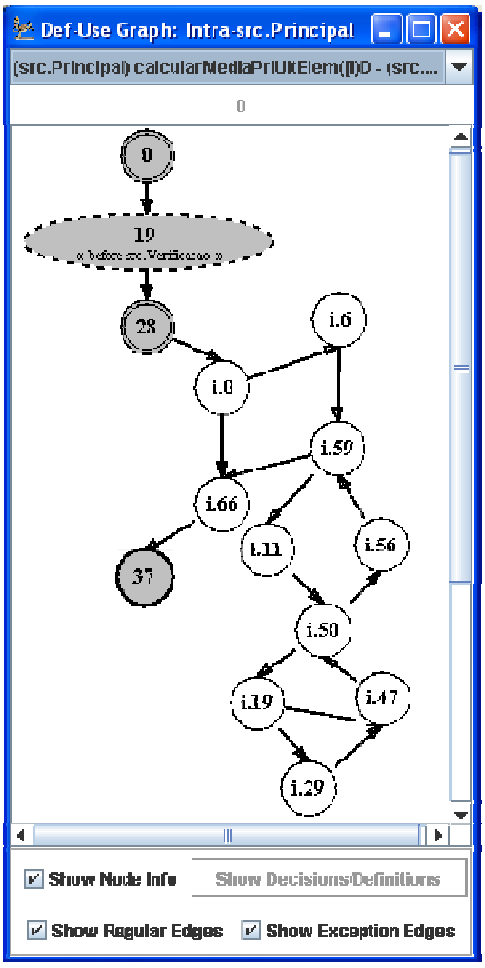


**Figura 4.26 Grafo de fluxo na ferramenta Jabuti/AJ**

Outra ferramenta de testes de software OA é aUnit[6]. Baseada na ferramenta já consolidada nos testes de unidade JUnit[7], aUnit já possui uma versão disponível para uso mesmo que com pouquíssima documentação.

Um ponto positivo da ferramenta aUnit é o fato de seu código estar disponível na página Web do projeto. Desta forma os usuários têm a liberdade de alterar a ferramenta ou propor modificações baseadas neste código. Na figura 4.27 tem-se a representação gráfica da ferramenta. Percebe-se que a interface utilizada é proveniente da ferramenta JUnit, porém os testes envolvem componentes da linguagem AspectJ.

[6] Site oficial: <http://www.aunit.org>
[7] Site oficial: <http://www.junit.org>

Por outro lado, os desenvolvedores da ferramenta não deixam claro em qualquer momento as técnicas utilizadas para a realização dos testes.

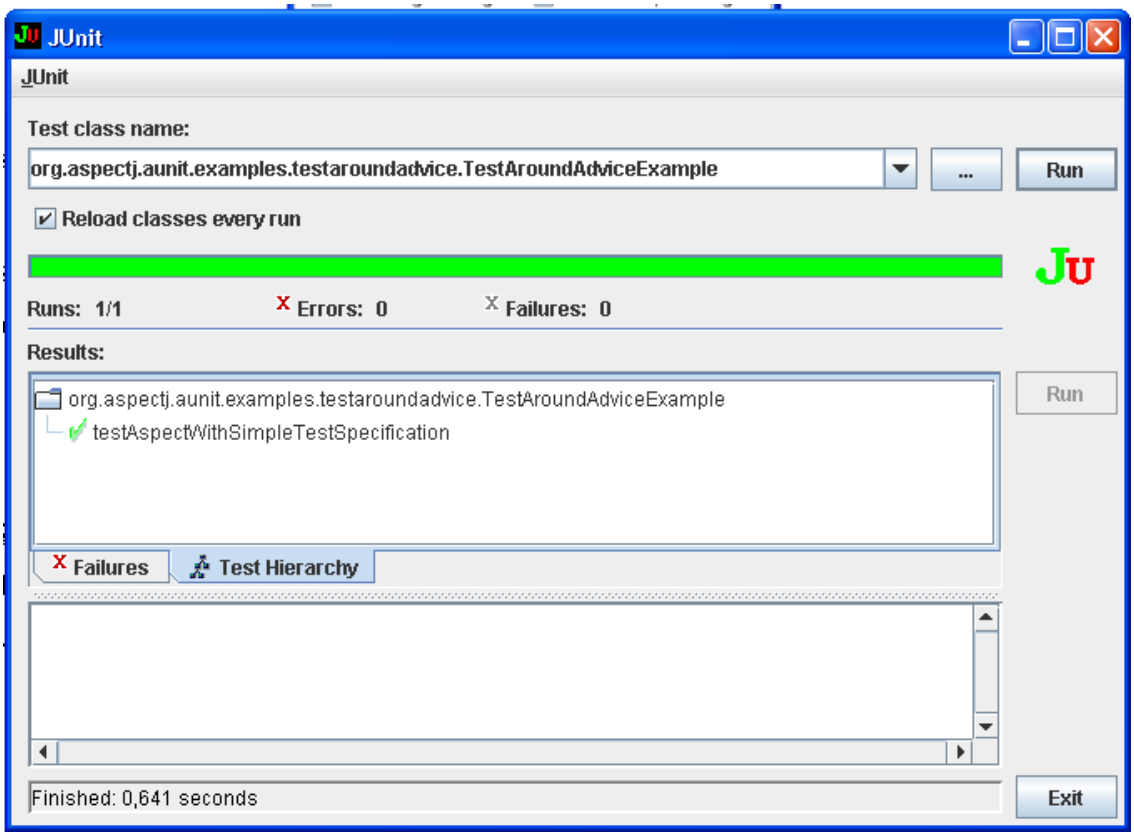


**Figura 4.27 Execução de testes de unidade na ferramenta aUnit**

A fase de testes é, das três abordadas neste capítulo, a que apresenta menor evolução até o momento em sua adaptação para o novo paradigma. Apesar de já apresentar modificações bem fundamentadas para os testes OO, a falta de ferramentas estáveis interfere diretamente nesta avaliação.

No próximo capítulo será apresentado um estudo de caso utilizando pesquisas apresentadas neste capítulo para cada etapa do desenvolvimento de uma aplicação OA.

# Capítulo 5
# Estudo de caso: Autenticação e Auditoria em um Módulo de Vendas

Neste capítulo será apresentado um estudo de caso baseado na criação de um módulo de vendas, planejado para ser adaptável para diversas implementações de interfaces gráficas, como interfaces web ou para dispositivos móveis, por exemplo. Não estão especificadas também restrições para a forma de armazenamento dos dados.

Este módulo de vendas é uma adaptação do sistema de vendas proposto em WINCK e JÚNIOR (2006). Em sua obra, os autores propõem diferentes aplicações da POA em um sistema de vendas de itens de uma loja. Neste estudo de caso dois destes interesses – autenticação e auditoria – serão demonstrados nas etapas de levantamento de requisitos, modelagem e testes, além da etapa de programação.

## 5.1 Levantamento de Requisitos

A seguir é apresentado um texto fictício que caracteriza os principais requisitos do software:

É necessário desenvolver um módulo de vendas que, futuramente, será integrado a outros módulos de um sistema consideravelmente maior. Este módulo, portanto deve estar apto para receber diversos formatos de interface com o usuário final e não deve restringir a forma de armazenamento dos dados.

As principais funções que o módulo de vendas deve apresentar são: cadastro de produtos, cadastro de categorias de produtos e registro das vendas dos produtos. Cada produto deve possuir uma categoria e deve existir também o controle da quantidade em estoque de um determinado produto.

O sistema deve reconhecer dois tipos de usuários: vendedores e gerentes. Somente gerentes podem manipular dados relativos a produtos, categorias e outros usuários. Os vendedores só estão autorizados a registrar vendas de produtos.

Para esta primeira versão todas as ações realizadas no sistema devem ser registradas. Desta forma será possível analisar possíveis falhas ocorridas durante a execução do módulo.

A partir destas informações, abaixo são seguidos os passos propostos por RASHID *et al.* (2003) para o levantamento de requisitos OA.

### 5.1.1 Identificação e Especificação dos interesses não funcionais

Os principais requisitos não-funcionais relacionados com a aplicação estão relacionados com a sua manutenibilidade, sua portabilidade e sua segurança. Para os dois primeiros, as utilizações do paradigma orientado a aspectos e da linguagem Java, são suficientes para se alcançar os resultados esperados.

Já o interesse não-funcional relativo à segurança pode ser enumerado através dos seguintes requisitos:

**R1.** Autenticação e validação das operações realizadas de acordo com o usuário atual do sistema.
**R2.** Todas as atividades realizadas devem ser registradas para futura auditoria.

### 5.1.2 Especificação dos Requisitos Funcionais

Analisando os requisitos funcionais, pode-se identificar os seguintes atores que, em seguida, serão utilizados no diagrama de caso de uso.

**Vendedor:** representa o tipo de usuário responsável somente pela venda de produtos.

**Gerente:** representa o tipo de usuário responsável pela administração de todos os dados da aplicação.

Estes atores são utilizados nos seguintes casos-de-uso:

**Cadastrar Produto:** Um gerente pode cadastrar novos produtos informando seu valor, a sua quantidade em estoque, sua descrição e a categoria deste produto.

**Alterar Produto:** Um gerente pode alterar todos os dados referentes a um produto.

**Cadastrar Categoria:** Um gerente pode criar novas categorias informando seu nome e, opcionalmente, os produtos desta categoria.

**Alterar Categoria:** Um gerente pode alterar tanto o nome da categoria como os produtos pertencentes a ela.

**Cadastrar Usuário:** Um gerente pode criar novos usuários, informando seu nome, endereço, nome para acesso ao sistema, senha, código e o tipo do usuário (Gerente ou Vendedor).

**Alterar Usuário:** Um gerente ou um usuário pode alterar seus próprios dados no sistema.

**Efetuar Venda:** Um gerente ou um vendedor pode registrar uma venda no sistema, informando o produto e a quantidade vendida.

Na figura 5.1 é apresentado o diagrama de caso de uso gerado a partir dos dados levantados na etapa 5.1.2.

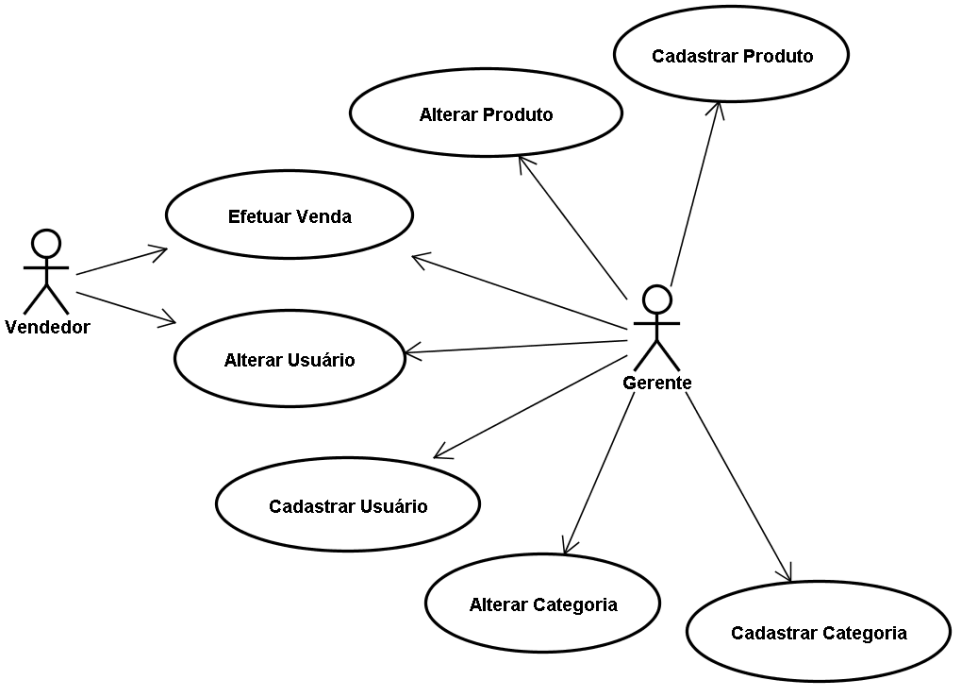


**Figura 5.1 Diagrama de caso de uso dos requisitos do sistema**

### 5.1.3 Identificação e Especificação dos Interesses Entrecortantes

O interesse não-funcional ligado à segurança das ações realizadas no software é convertido em dois interesses que entrecortam outros interesses da aplicação. A partir deste interesse não-funcional são derivados dois interesses entrecortantes: Auditoria e Autenticação. Seguindo o modelo proposto por RASHID *et al.* (2003), são construídas as tabelas 5.1 e 5.2 que representam informações sobre estes dois candidatos a aspectos.

**Tabela 5.4 Especificação do Interesse Entrecortante de Auditoria**

| **Interesse Entrecortante** | Auditoria |
|---|---|
| **Descrição** | Interesse que certifica o registro de todas as operações realizadas na aplicação. |
| **Prioridade** | Máxima |
| **Lista de Requisitos** | R2 |
| **Lista de Modelagens** | Cadastrar Produto, Alterar Produto, Cadastrar Categoria, Alterar Categoria, Cadastrar Usuário, Alterar Usuário, Efetuar Venda |

**Tabela 5.5 Especificação do Interesse Entrecortante de Autenticação**

| **Interesse Entrecortante** | Autenticação |
|---|---|
| **Descrição** | Interesse que certifica a validade das operações realizadas por cada tipo de usuário. |
| **Prioridade** | Máxima |
| **Lista de Requisitos** | R1 |
| **Lista de Modelagens** | Cadastrar Produto, Alterar Produto, Cadastrar Categoria, Alterar Categoria, Cadastrar Usuário, Alterar Usuário |

### 5.1.4 Integração dos Interesses Entrecortantes à modelagem UML

A integração é feita através da inclusão de casos de uso especiais para os interesses entrecortantes. Estes casos de uso recebem os estereótipos <<*Auditoria*>> e <<*Autenticacao*>> e suas ligações com os casos de uso listados na etapa 5.1.3 recebem estereótipos <<*wrappedBy*>> de acordo com o modelo de RASHID *et al.* (2003). Este estereótipo demonstra que o interesses transversal encapsula os casos de uso originais. O novo diagrama é representado na figura 5.2.

Após a conclusão deste passo o levantamento dos requisitos do sistema é dado como concluído. Na próxima etapa será apresentada a modelagem seguindo o AODM, visto no capítulo 4.

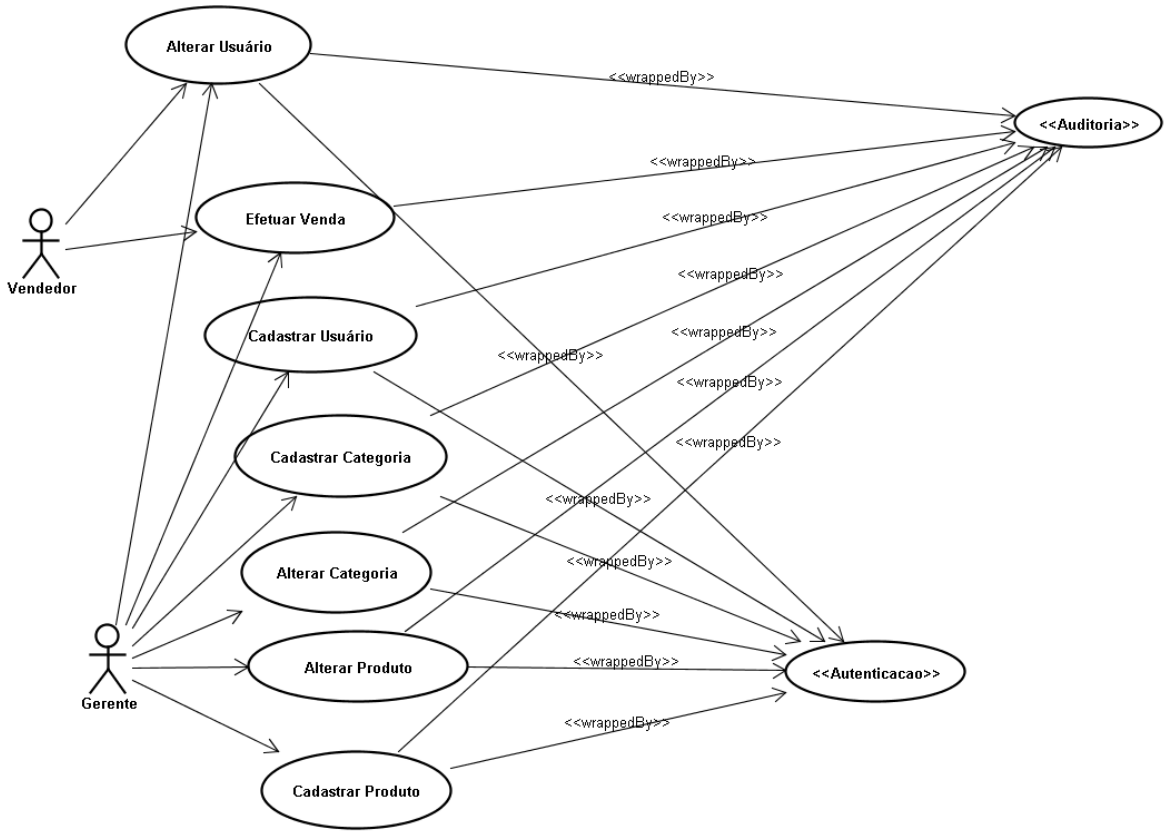


**Figura 5.2 Diagrama de caso de uso integrado aos interesses entrecortantes**

## 5.2 Modelagem

Para a modelagem serão utilizados alguns dos modelos propostos por STEIN (2003) através do AODM. Como este estudo de caso é voltado para a visão dos analistas e desenvolvedores do sistema, serão desconsideradas questões relativas à processos como a combinação aspectual.

Inicialmente, é realizada a diagramação do modelo de classes simplificado do sistema a partir dos dados gerados pela etapa de levantamento de requisitos. A figura 5.3 traz este diagrama. Nele são representados dois pacotes: *modelo* que contém as classes básicas do modelo e *gui* que contém uma interface que, futuramente, pode ser utilizada por qualquer implementação de interface gráfica.

Para adequar o diagrama de classes à realidade OA é necessário inserir, segundo o AODM, os elementos referentes aos aspectos `Auditoria` e `Autenticacao` e relacionar estes elementos com as classes afetadas por eles. Antes de integrá-los ao modelo de classes

original, a modelagem destes aspectos é apresentada nas figuras 5.4 e 5.5 respectivamente. A maior dificuldade para a criação destes novos elementos é a necessidade de se conhecer bem a linguagem que será utilizada na etapa de programação, afinal, a modelagem dos aspectos já apresenta a assinatura de todos os pontos de atuação e adendos que serão utilizados.

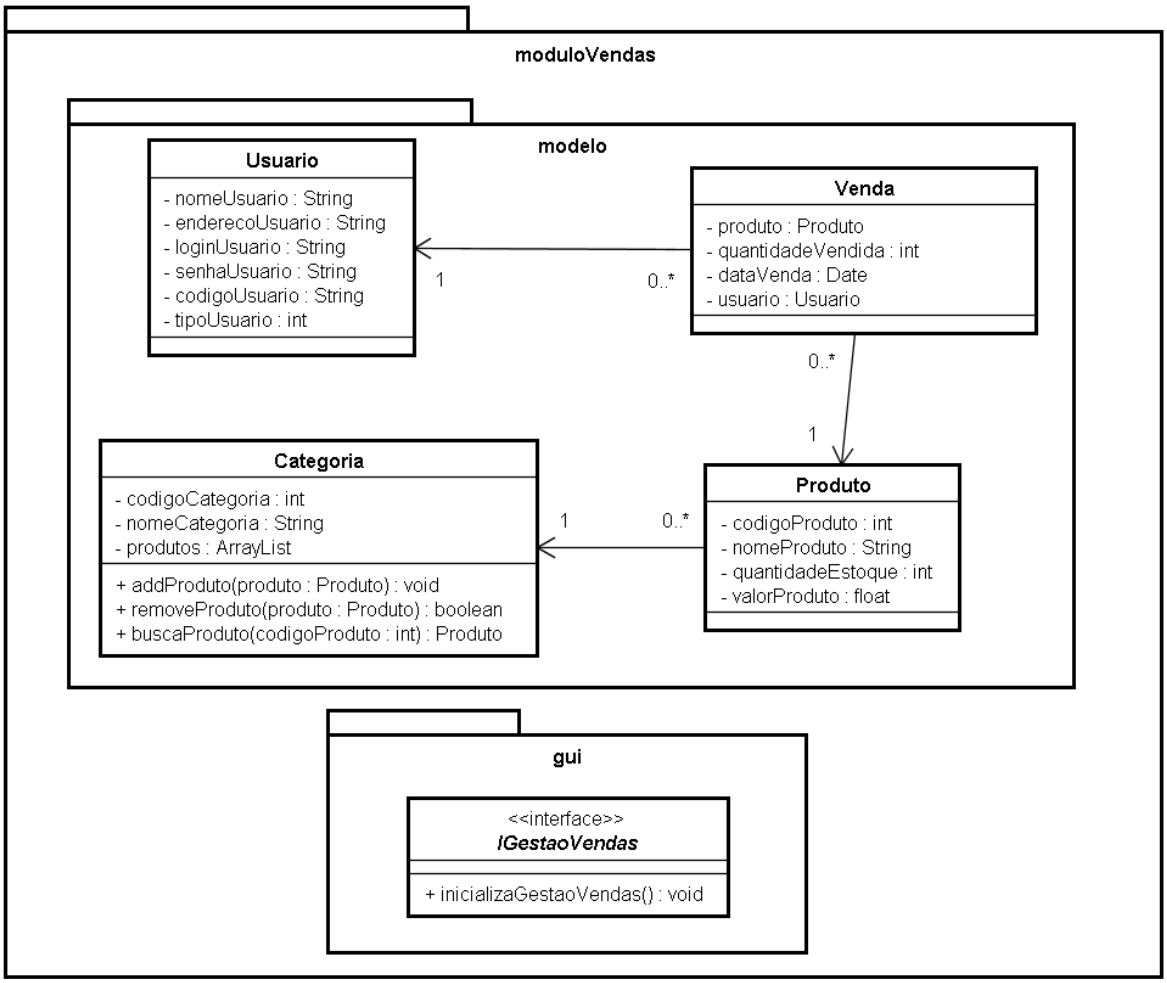


**Figura 5.3 Diagrama de classes do módulo de vendas**

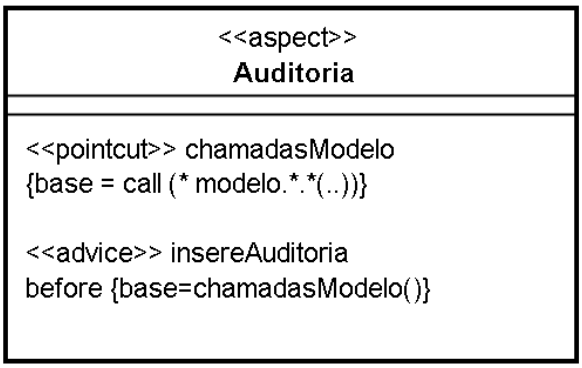


**Figura 5.4 Representação em UML do aspecto de auditoria**

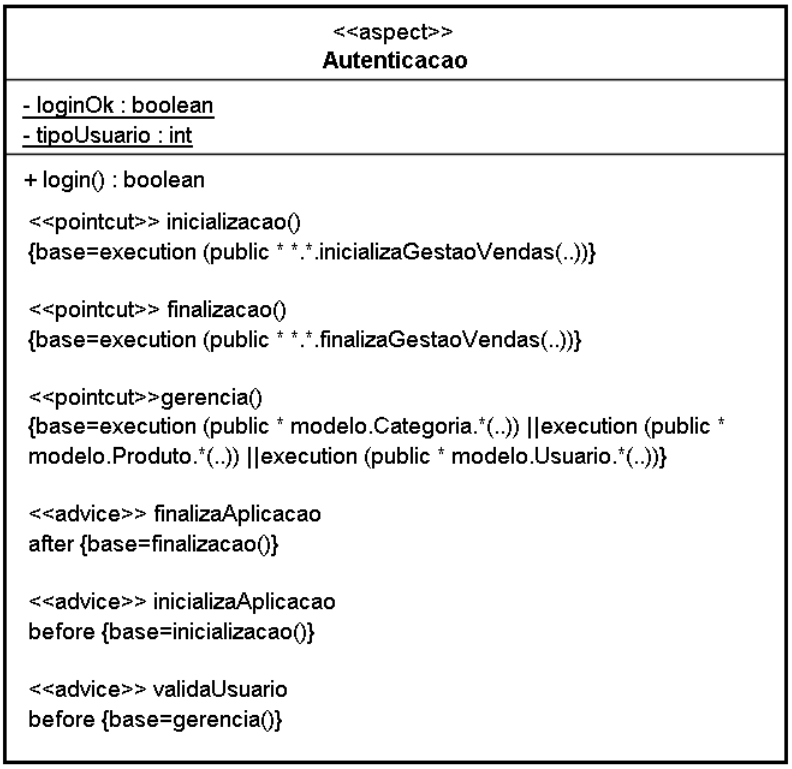


**Figura 5.5 Representação em UML do aspecto de autenticação**

A figura 5.6 representa o modelo após a execução do combinador aspectual sobre as classes e aspectos. As classes são relacionadas com os aspectos que as entrecortam através de ligações com o estereótipo <<*crosscut*>>. Este modelo finaliza a etapa de modelagem do sistema e serve de entrada para o início da codificação, que será demonstrada na etapa 5.3.

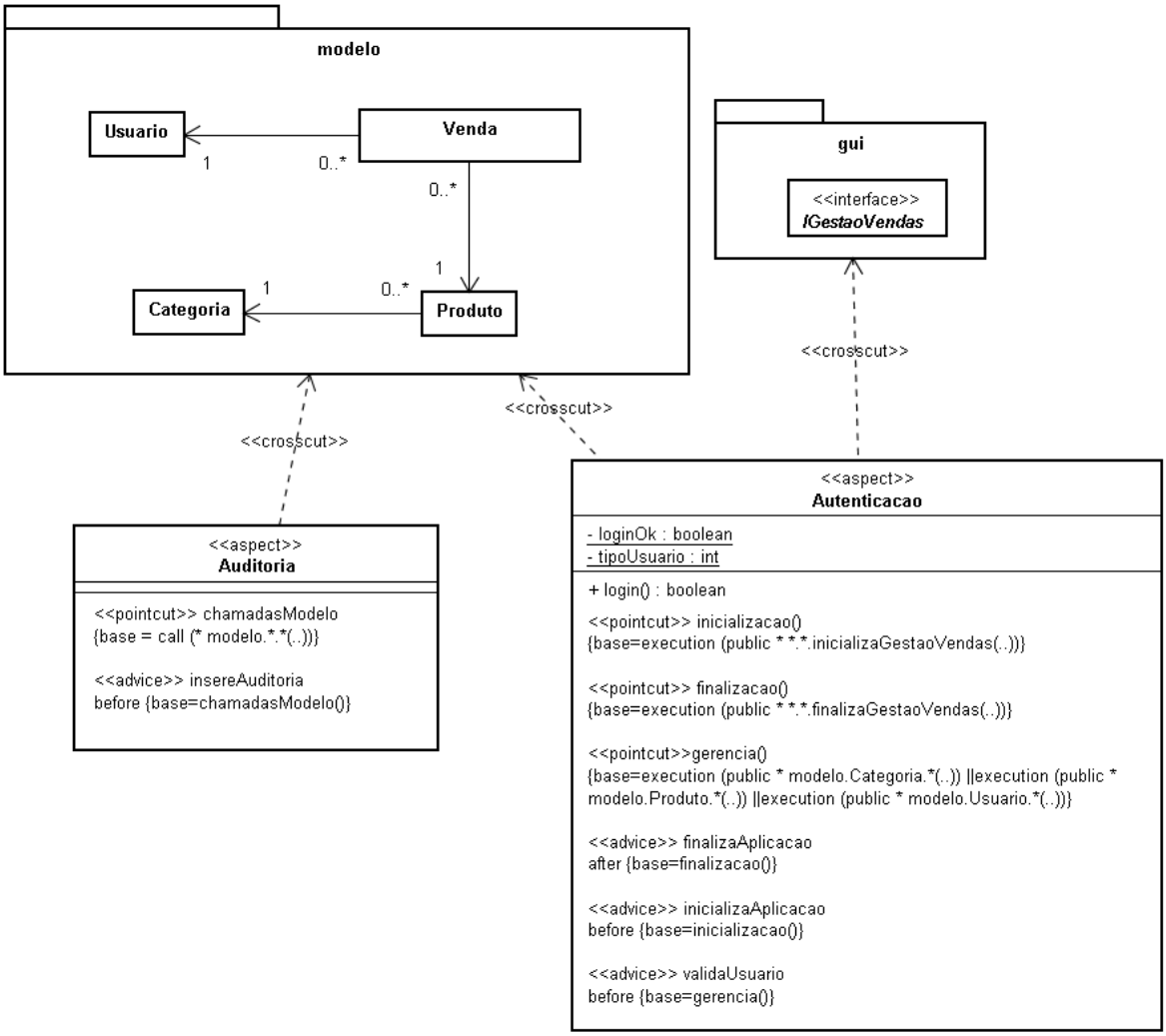


**Figura 5.6 Diagrama de classes após o processo de combinação aspectual**

## 5.3 Programação

O desenvolvimento do módulo de vendas foi realizado utilizando a linguagem Java em sua versão 1.5, foi utilizada também o ambiente de programação Eclipse 3.3[8] e o *plugin* para AspectJ chamado AJDT[9] na versão 1.5. Foram seguidas práticas recomendadas por ASPECTJ (2007).

[8] Site oficial: <http://www.eclipse.org/>
[9] Site oficial: <http://www.eclipse.org/ajdt/>

Com os requisitos bem estruturados e a modelagem já definida, a atividade de codificação se torna muito simples. A primeira tarefa realizada nesta fase foi a criação das classes do pacote modelo, que, basicamente, possuem alguns atributos, métodos construtores e operações para definição e utilização dos valores dos atributos. A figura 5.7 apresenta um trecho do código da classe `Produto.java`.

Para satisfazer a portabilidade do software para qualquer definição de representação gráfica, foi criada no pacote `gui` uma interface Java chamada `IGestaoVendas`. Esta interface define a existência de dois métodos: `inicializaGestaoVendas` e `finalizaGestaoVendas`. O aspecto de autenticação, como visto na etapa de modelagem, entrecorta todas a chamadas a métodos com estas duas assinaturas. Portanto a restrição gerada pela criação desta interface se torna a melhor solução para garantir a execução correta deste requisito.

A criação dos aspectos também se torna uma tarefa trivial uma vez que, todas as definições destes já foram realizadas na etapa de modelagem. O aspecto de Auditoria foi representado pela classe `AspectoAuditoria.aj` e possui somente um ponto de atuação e um adendo. Este ponto de atuação entrecorta todas as chamadas a métodos das classes do modelo e o adendo gera um texto que contém informações sobre o método invocado, a classe que declara este método e os parâmetros passados. Na figura 5.8 é apresentado o código de `AspectoAuditoria.aj`.

O aspecto responsável pelo controle da execução das funcionalidades do sistema de acordo com o usuário atual é representado pela classe `AspectoAutenticacao.aj`. Nele são declarados três conjuntos de pontos de atuação: `inicializacao()`, `finalização()` e `gerencia()`. Os dois primeiros estão relacionados com chamadas aos métodos definidos na interface `IGestaoVendas` citada anteriormente enquanto o terceiro intercepta as chamadas a métodos restritos aos usuários com cargo de gerência. O código referente a estes três pontos de atuação é demonstrado na figura 5.9.

```
public class Produto {


    private int         codigoProduto;
    private String      nomeProduto;
    private int         quantidadeEstoque;
    private float       valorProduto;


    public Produto() {
        // TODO Auto-generated constructor stub
    }

    public Produto(int codigoProduto, String nomeProduto,
            int quantidadeEstoque, float valorProduto) {
        super();
        this.setCodigoProduto(codigoProduto);
        this.setNomeProduto(nomeProduto);
        this.setQuantidadeEstoque(quantidadeEstoque);
        this.setValorProduto(valorProduto);
    }
```

**Figura 5.7 Classe `produto.java`**

```
public aspect AspectoAuditoria {
    pointcut chamadasModelo() : call (* modelo.*.*(..));
    // Adendo insereAuditoria
    before() : chamadasModelo()
    {
        Signature assinatura = thisJoinPointStaticPart.getSignature();
        String classe = assinatura.getDeclaringType().getName();
        String metodo = assinatura.getName();
        String argumentos = "";

        for (int i = 0; i < thisJoinPoint.getArgs().length; i++) {
            argumentos += thisJoinPoint.getArgs()[i].toString();
            argumentos += " | ";
        }
        String log = "LOG - "+classe+" - "+metodo+" paramêtros ("+argumentos+")";
        // Inserção da auditoria na forma de armazenamento desejada.
    }
}
```

**Figura 5.8 Aspecto de auditoria**

```
pointcut inicializacao() : call (public * gui.*.inicializaGestaoVendas(..));
pointcut finalizacao() : call (public * gui.*.finalizaGestaoVendas(..));

pointcut gerencia() :   execution (public * modelo.Categoria.*(..)) ||
                        execution (public * modelo.Produto.*(..)) ||
                        execution (public * modelo.Usuario.*(..));
```

**Figura 5.9 Conjuntos de pontos de atuação do aspecto de autenticação**

Para cada um destes pontos de atuação é declarado um adendo. O adendo `inicializaAplicacao` executa o método `login()` criado no próprio aspecto. Este método autoriza ou não a utilização do sistema de acordo com o nome de usuário e a senha inserida e salva informações sobre o tipo de usuário atual. Já o adendo `finalizaAplicacao` anula as informações armazenadas pelo adendo `inicializaAplicacao`. Por último o adendo `validaUsuario` verifica o tipo de usuário atual e aplica as restrições especificadas pelos requisitos. A figura 5.10 exibe o código referente a estes adendos.

Como planejado na etapa de levantamento de requisitos, todas as ações do sistema são rastreadas pelo aspecto de auditoria e todas as ações restritas a gerentes são validadas pelo aspecto de autenticação. As chamadas realizadas por estes aspectos podem ser comprovadas pela figura 5.11. Nela as linhas pretas representam a atuação do aspecto de auditoria e as linhas cinzas a atuação do aspecto de autenticação. Como não existem classes concretas para as representações gráficas do sistema, o aspecto de autenticação não atua diretamente em classes do pacote `gui`.

```
// Adendo finalizaAplicacao
after() : finalizacao()
{
    loginOk = false;
    tipoUsuario = TipoUsuario.Invalido.ordinal();
}
// Adendo inicializaAplicacao
before(): inicializacao()
{
        loginOk = false;

        loginOk = this.login();

        if (!loginOk)
        {
            // Retorna mensagem de erro segundo a interface gráfica adotada.
        }
}

// Adendo validaUsuario
before() : gerencia()
{
    if(tipoUsuario != TipoUsuario.Invalido.ordinal())
    {
        if(tipoUsuario != TipoUsuario.Gerente.ordinal())
        {
            // Retorna mensagem de erro segundo a interface gráfica adotada.
        }
    }
    else
    {
        // É necessário realizar o login novamente.
    }
}
```

**Figura 5.10 Adendos do aspecto de autenticação**

Um relacionamento que não foi previsto nas etapas anteriores acabou sendo revelado na etapa de programação: o relacionamento entre os aspectos de auditoria e autenticação. Este relacionamento é dado pelo registro das ações realizadas pelos adendos do aspecto de autenticação. A figura 5.12 comprova as chamadas realizadas por `AspectoAuditoria` em `AspectoAutenticacao`.

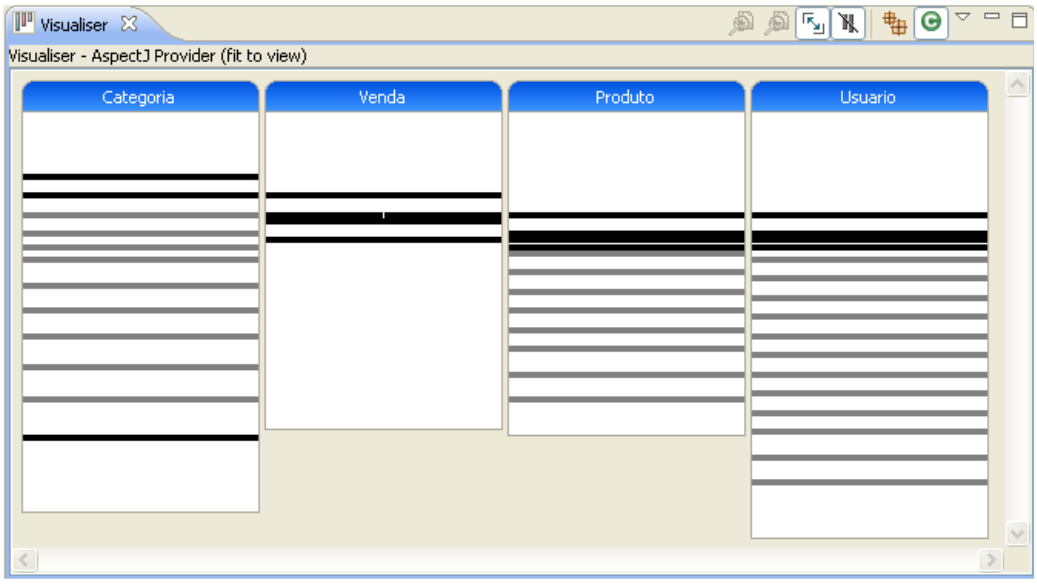


**Figura 5.11 Demonstração dos pontos interceptados pelos aspectos de auditoria e autenticação**

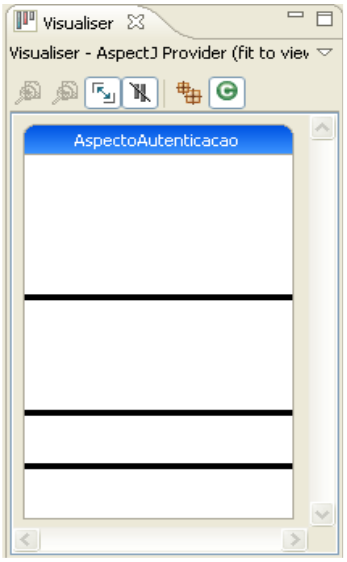


**Figura 5.12 Pontos interceptados pelo aspecto de auditoria no aspecto de autenticação**

## 5.4 Testes

Como visto em 4.3.2, uma das poucas opções disponíveis para uso nesta etapa é o arcabouço aUnit. Para a realização dos testes devem ser criadas classes de acordo com os padrões definidos pelo arcabouço. A primeira tarefa é adaptar o aspecto que será submetido aos testes

com marcações sobre a declaração do aspecto e dos adendos. Neste estudo de caso serão realizados testes com o aspecto `AspectoAuditoria` e seu adendo `insereAuditoria`.

As adaptações realizadas neste aspecto são demonstradas na figura 5.13. Basicamente devem-se adicionar os atributos `@TestableAspect` e `@TestableAdvice` aos aspectos e adendos que serão testados respectivamente. Os adendos devem também ser identificados com um atributo `id`.

```
@TestableAspect
public aspect AspectoAuditoria {
    pointcut chamadasModelo() : call (* modelo.*.*(..));
    // Adendo insereAuditoria
    @TestableAdvice(id="insereAuditoria")
    before() : chamadasModelo()
    {
        Signature assinatura = thisJoinPointStaticPart.getSignature();
        String classe = assinatura.getDeclaringType().getName();
        String metodo = assinatura.getName();
        String argumentos = "";

        for (int i = 0; i < thisJoinPoint.getArgs().length; i++) {
            argumentos += thisJoinPoint.getArgs()[i].toString();
            argumentos += " | ";
        }
        String log = "LOG - "+classe+" - "+metodo+" paramêtros {"+argumentos+"}";
        // Inserção da auditoria na forma de armazenamento desejada.
    }
}
```

**Figura 5.13 Adaptações no aspecto de auditoria para realização de testes**

Feitas estas adaptações, devem ser criadas classes que simulem a execução de métodos de uma ou mais classes afetadas pelo aspecto. Esta classe pode ser dividida em duas partes com funções distintas: a criação de parâmetros para a execução do teste e a chamada das interfaces gráficas do arcabouço.

Para a criação dos parâmetros devem ser especificados os métodos que serão executados e os adendos cuja execução é esperada. Na figura 5.14 o trecho de código que cria estes elementos é apresentado. Já na figura 5.15 é apresentado o código que realiza os testes.

```
public TestStep[] criacaoParametros() throws Exception
{

   TestStep[] steps = new TestStep[1];

   // Cria a classe alvo dos testes
   Categoria classeTeste = new Categoria();

   // Especifica os tipos dos parametros do método que será testado
   Class[] parametrosMetodo = { int.class };

   // Seleciona o método que será testado
   Method metodoTeste = classeTeste.getClass().getMethod("buscaProduto", parametrosMetodo);

   // Cria os valores estáticos do ponto de junção
   JoinPoint.StaticPart staticPart = Factory.makeEncSJP(metodoTeste);

   // Especifica os valores dos parametros do método.
   Object[] args = { 15 };

   // Especifica o ponto de junção passando seus atributos estáticos e dinâmicos
   JoinPoint joinPoint = Factory.makeJP(staticPart, classeTeste,
           classeTeste, args);

   // Cria o contexto de execução do ponto de junção.
   JoinPointContext contexto = new JoinPointContext(joinPoint);

   // Especifica o adendo que deve ser executado
   String adendoSelecionado = "insereAuditoria";

   // Cria o passo de teste que será executado
   TestStep testStep = TestStepFactory.createControlledTestStep(
         (AspectObject) AspectoAuditoria.aspectOf(), contexto,
         adendoSelecionado);
   steps[0] = testStep;


   return steps;
}
```

**Figura 5.14 Definição dos passos do teste para o aspecto de auditoria**

A execução da classe de teste ativa a interface gráfica do arcabouço aUnit. Esta interface é a mesma utilizada pelo arcabouço de testes OO JUnit. Na figura 5.16 tem-se a tela resultante da execução dos testes descritos nesta fase do estudo de caso. Os testes são realizados com sucesso sem a presença de qualquer falha. Estes resultados determinam o fim do estudo de caso.

```
public void executaTestes()
{
   // Cria a especificação do teste seguindo o padrão aUnit.
   TestSpecification testSpecification = TestSpecificationFactory
         .createBaseTestSpecification();

   try
   {
      // Cria os paramêtros segundo o método criacaoParametros
      TestStep[] steps = this.criacaoParametros();

      // Adiciona os paramêtros à especificação do teste
      for (TestStep step : steps)
      {
         testSpecification.addStep(step);
      }

      // Executa a especificação
      testSpecification.execute();
   }
   catch (Exception e)
   {
      // Imprime erros encontrados
      e.printStackTrace();
   }
}
```

**Figura 5.15 Método responsável pela execução do teste**

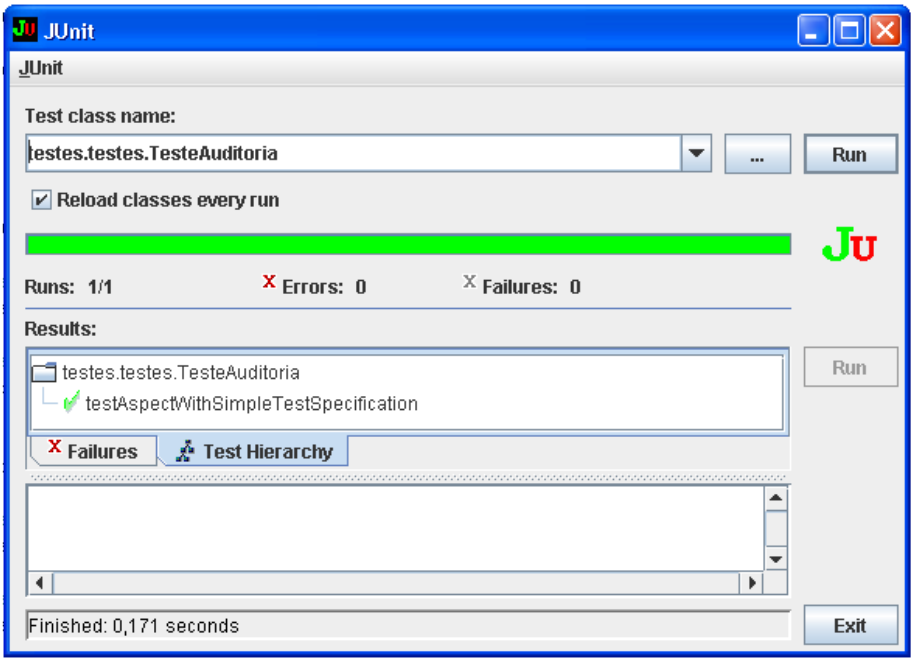


**Figura 5.16 Interface da ferramenta de testes aUnit após a execução do teste**

Este estudo de caso demonstrou que a aplicação da OA desde as primeiras etapas do desenvolvimento facilita a percepção de interesses entrecortantes no software e a conseqüente criação de aspectos para absorver estes interesses. A maior fragilidade de todas estas etapas ainda está na fase de realização de testes OA visto a falta de ferramentas bem documentadas. O próximo capítulo traz as considerações finais sobre este trabalho e sugestões para trabalhos futuros.

# Capítulo 6
# Considerações Finais

A Orientação a Aspectos cada vez mais consolida suas forças como o paradigma capaz de cobrir as falhas existentes na OO. Este novo paradigma destaca-se pela capacidade de separar interesses que entrecortam uma aplicação de forma simples e objetiva. Porém, o uso incorreto e mal planejado destas capacidades pode trazer conseqüências negativas.

Como visto, após a consolidação de algumas linguagens e ferramentas facilitadoras da POA, surge a necessidade de se expandir estes conceitos para outras etapas do ciclo de desenvolvimento. O DSOA reúne então técnicas e metodologias para aplicar de forma correta a OA em fases como engenharia de requisitos, modelagem, testes, refatoração etc.

Dentre estas, as etapas anteriores à fase de implementação apresentam maior nível de maturidade. A facilidade de adaptação destas etapas no DSOO para o DSOA é notável, principalmente pelo uso da UML. O uso dos mecanismos de extensão desta linguagem de modelagem dispensa a criação de ferramentas especificas para OA nas fases de levantamento de requisitos e de modelagem.

Já a fase de testes, apesar de também aproveitar conceitos do DSOO em suas propostas, depende de ferramentas específicas para comprovar sua maturidade. E é justamente esse o principal ponto de falha dos testes OA: a carência de ferramentas estáveis. Contudo, Jabuti/AJ e aUnit, mesmo que não consolidadas, demonstram que futuros esforços podem suprir essa carência.

## 6.1 Trabalhos Futuros

Este trabalho teve o foco voltado para as etapas de levantamento de requisitos, modelagem e testes além da programação orientada a aspectos. Alguns assuntos relativos ainda ao DSOA não foram tratados e podem ser abordados em trabalhos futuros como refatoração, arquitetura de software e evolução do software orientado a aspectos.

Dentre os assuntos abordados neste trabalho, trabalhos futuros podem se aproveitar de novas propostas e ferramentas, assim como explorar com maior detalhes uma das etapas especificamente.

Outra sugestão para trabalhos futuros seria a avaliação da qualidade, através de métricas específicas, de softwares OA. O fato de um aspecto afetar, de forma dinâmica, diversas classes torna esta tarefa mais complexa e a necessidade de ferramentas específicas é clara. Por último, a análise de casos reais de uso dos conceitos apresentados também é uma perspectiva para trabalhos futuros.

# Glossário

**Adendos:** trechos de código executados quando um ponto de atuação é alcançado.

**AspectJ:** linguagem com suporte aos conceitos da Orientação a Aspectos, estendido da linguagem Java.

**Aspecto:** unidade de um sistema capaz de reunir todos os componentes que determinam um interesse transversal.

**Grafo de Fluxo:** grafo auxiliar para a elaboração de testes de software. Demonstram todos os possíveis passos do código da aplicação.

**Interesses Transversais:** Interesses não funcionais que afetam a organização e o entendimento do código. Também chamados de interesses entrecortantes.

**Pontos de Junção:** pontos do código onde um aspecto pode ser aplicado.

**Pontos de Atuação:** regras que definem a atuação de um aspecto sobre pontos de junção.

# Referências Bibliográficas


ASPECTJ, 2007 *The AspectJ Programming Guide*. 2007. Disponível em <http://www.eclipse.org/aspectj/doc/released/progguide/index.html> Último acesso em novembro de 2007.

BARBOSA, E. F.; MALDONADO, J. C.; VICENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. Introdução ao teste de software. 2000. Disponível em: <http://www.inf.ufpr.br/silvia/topicos/apostilaUSP.pdf.gz> Último acesso em novembro de 2007.

BLAIR, G. S.; RASHID, A.; MOREIRA, A.; ARAÚJO, J.; CHITCHYAN, R. Engineering Aspect-Oriented Systems. In: FILMAN, R. *et al.* Aspect-Oriented Software Development. 1.ed. Boston: Addison Wesley Professional, 2004. 800p.

CAMPOS, F. Programação Orientada a Aspectos. Juiz de Fora, 2006. Monografia (Bacharelado) – Departamento de Ciência da Computação, UFJF – MG.

CHAVEZ, C. Um Enfoque Baseado em Modelos para Design Orientado a Aspectos. Rio de Janeiro, 2004. Tese (Doutorado) – Departamento de Informática, PUC – RJ.

CLARKE, S. *Composition of Object-Oriented Software Design Model.* Dublin, 2001. Tese (Doutorado) - School of Computer Applications, Dublin City University

CLARKE, S. ; BANIASSAD E. Aspect-Oriented Analysis and Design: The Theme Approach. 1.ed. Boston: Addison Wesley Professional, 2005. 400p.

CLEMENTE, P.; HERNEÁNDEZ, J. HERRERO, J. L.; MURILLO, J.M.; SEÁNCHEZ, F. Aspect-Orientation in the Software Lifecycle: Fact and Fiction In: FILMAN, R. *et al.* Aspect-Oriented Software Development. 1.ed. Boston: Addison Wesley Professional, 2004.

DIJKSTRA, E. A Discipline of Programming. 1.ed. New Jersey: Prentice Hall, 1976. 240p.

FIGUEIREDO, E. Uma Abordagem Quantitativa para Desenvolvimento de Software Orientado a Aspectos. Rio de Janeiro, 2006. Dissertação (Mestrado) – Departamento de Informática, PUC – RJ.

FILMAN, R.; CLARKE, S.; AKSIT, M. Aspect-Oriented Software Development. 1.ed. Boston: Addison Wesley Professional, 2004. 800p.

GRUNDY, J. *Aspect-oriented Requirements Engineering for Component-based Software Systems*. 1999. In: IEEE International Symposium on Requirements Engineering. Limerick, Ireland, 1999.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.V.; LOINGTIER, J.; IRWIN, J. *Aspect-Oriented Programming* In: European Conference on Object Oriented Programming (ECOOP). Finland: Springer-Verlag, 1997.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W.G. *An Overview of AspectJ*. European Conference on Object-Oriented Programming (ECOOP). Budapest, Hungary, 2001.

LADDAD, R. *I want my AOP*. 2002. Disponível em: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html> Último acesso em novembro de 2007.

LEMOS, O. A. L. Testes de Programas Orientados a Aspectos: uma abordagem estrutural para AspectJ. São Carlos, 2005. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, USP – São Carlos.

MELO, A.C.; BET, S. *SOP - Subject-Oriented Programming*. 2003. Disponível em: <http://www.lisha.ufsc.br/~guto/teaching/sce/ine65100-2003-2/work/sop/> Último acesso em dezembro de 2007.

MASIERO, P. C.; LEMOS, O. A. L.; FERRARI, F. C.; MALDONADO, J. C. Teste de Software Orientado a Objetos e a Aspectos:Teoria e Prática. 2006 Disponível em: <http://www.sbc.org.br/bibliotecadigital/download.php?paper=637> Último acesso em: setembro de 2007.

OMG (Object Management Group). *Unified Modeling Language Specification* - Versão 1.4.2. Janeiro, 2005

PRESSMAN, R.S. Engenharia de Software. 6.ed. São Paulo: McGraw-Hill, 2006.

RASHID, A.; MOREIRA, A.; ARAÚJO, J; SAWYER, P. *Early Aspects: a Model for Aspect-Oriented Requirements Engineering*. 2002. In: IEEE Join Conference on Requirements Engineering. Essen, Germany, 2002.

RASHID, A.; MOREIRA, A.; ARAÚJO, J . *Aspect-Oriented Requirements with UML*. 2003 Disponível em: <http://www.comp.lancs.ac.uk/computing/aod/papers/AORE_UMLWS2002.pdf> Último acesso em outubro de 2007.

SILVA, L. *Uma Estratégia Orientada a Aspectos para Modelagem de Requisitos*. Rio de Janeiro, 2006. Tese (Doutorado) – Departamento de Informática, PUC – RJ.

STEIN, D. *An Aspect-Oriented Design Model Based on AspectJ and UML*. Essen, 2002. Tese (Mestrado) - Department of Business Arts, Economics, and Management Information Systems, University of Essen

SUZUKI, J.; YAMAMOTO, Y. *Extending UML with Aspects:Aspect Support in the Design Phase*. 3rd Aspect-Oriented Programming Workshop - ECOOP. Lisbon, Portugal, 1999.

VICENZI, A. M.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. Jabuti – *Java Bytecode Understanding and Testing – user´s guide – version 1.0*. 2004. Disponível em: <http://jabuti.incubadora.fapesp.br/portal/documentos/jabuti.pdf > Último acesso em: outubro de 2007.

WASP. 1o. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos – Relatório Final. 2004.

WINCK, D.; JÚNIOR, G. AspectJ: Programação Orientada a Aspectos com Java. 1.ed. São Paulo: Novatec, 2006. 228p.

XU, W; XU, D. *A Model-Based Approach to Test Generation for Aspect-Oriented Programs*. Disponível em: <http://www.cs.ndsu.nodak.edu/~dxu/publications/xu-xu-wtaop05.pdf> Último acesso em setembro de 2007.

ZHAO, J. *Data-Flow-Based Unit Testing of Aspect-Oriented Program.* 2003. Disponível em: < http://cse.sjtu.edu.cn/~zhao/pub/pdf/compsac03.pdf> Último acesso em outubro de 2007.

# Apêndice I
# Visão Geral sobre a UML

A UML é uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software, assim como para modelagem de outros tipos de sistemas que não softwares. Ela representa uma coleção das melhores praticas de engenharia que se comprovaram eficientes para o projeto de sistemas amplos e complexos (OMG, 2005). Adotada pela comunidade de desenvolvimento OO como padrão para a criação de modelos e diagramas conceituais, a UML proporciona também a criação de variações através de um mecanismo de extensão próprio.

Os artefatos principais da linguagem são representações gráficas de componentes e formas de relacionamento de um sistema. Estes elementos são agrupados nos seguintes diagramas:

- Diagrama de Caso de Uso
- Diagrama de Classes
- Diagramas de Comportamento
    - Diagrama de Estado
    - Diagrama de Atividades
    - Diagramas de Iteração
        - Diagrama de Seqüência
        - Diagrama de Colaboração
    - Diagramas de Implementação

As etapas de concepção, especificação, construção, testes e entrega de uma solução podem ser descritas através deste conjunto de diagramas, que suportam conceitos de alto nível (estrutura, padrões e componentes). É importante ressaltar também que a UML não substitui nem está atrelada a qualquer linguagem de programação especifica. Para a modelagem de sistemas e soluções OO a UML possui grande destaque por apresentar-se como uma alternativa completa. Segundo LIMA (2005), a UML é uma linguagem cheia de recursos, que

permite não apenas capturar as informações, mas também expressá-las de forma clara e objetiva.

Nos tópicos A.1 e A.2 serão apresentados os principais elementos e diagramas da UML considerando-se a sua utilização no presente trabalho.

## A.1 Principais Elementos

### A1.1 Pacotes

Pacotes são elementos responsáveis por agrupar outros elementos. Um pacote em UML pode conter qualquer outro elemento, inclusive outros pacotes. Todos os diagramas da linguagem podem ser organizados em pacotes. Na figura A.1 tem-se um exemplo de pacote.

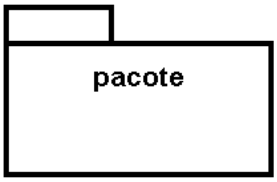


**Figura A.1 Exemplo de pacote em UML**

### A1.2 Classes

Classes definem elementos com estrutura, relacionamentos e comportamento similares. A UML oferece notação para a declaração de classes e suas propriedades assim como para a sua utilização em diversas situações. A figura A.2 representa diferentes classes na linguagem UML.

A notação de classes possui algumas restrições como: classes e operações abstratas devem ter o nome em itálico, todas os atributos e operações devem ter o nome iniciado em letras minúsculas etc.

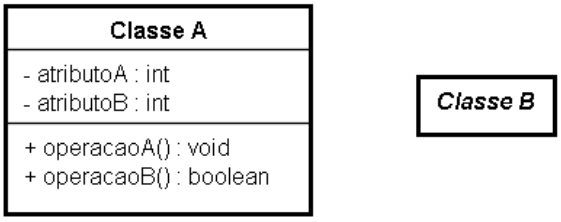


**Figura A.2 Exemplos de classes em UML**

### A1.3 Estereótipos

São elementos utilizados para rotular outros elementos criados. Os estereótipos são criados pelo próprio usuário e devem ser identificados pelo nome inserido entre "<<" e ">>", como visto na figura A.3.

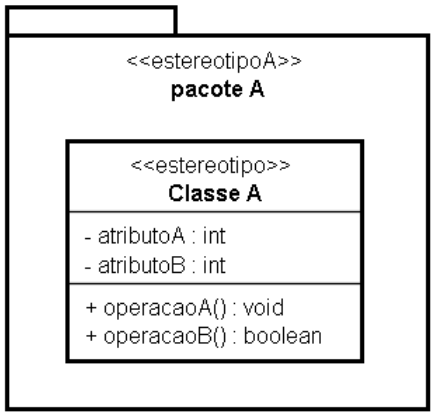


**Figura A.3 Exemplos de pacote e classe estereotipados**

### A1.4 Classes Parametrizadas (*Templates*)

São classes que possuem um ou mais parâmetros formais não associados. Classes parametrizadas não podem ser utilizadas diretamente em diagramas, mas sim classes que façam a vinculação destes parâmetros.

Classes parametrizadas são diferenciadas de outras classes pela presença de um retângulo tracejado no lado direito superior, como visto no exemplo da figura A.4. Este retângulo contém os parâmetros não associados.

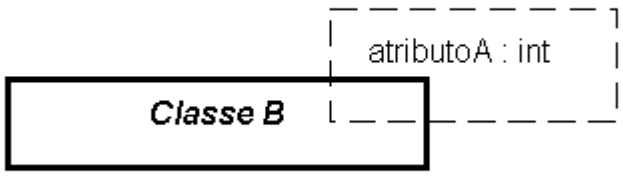


**Figura A.4 Classe parametrizada**

### A1.5 Atores

Representam grupos de entidades, não necessariamente pessoas, que são aptas para realizar uma tarefa determinada por outra entidade. São representados por bonecos como o da figura A.5.


**Figura A.5 Exemplo de Ator**

### A1.6 Caso de Uso

Uma funcionalidade de uma classe, de um sistema ou de outro elemento pode ser representada como um caso de uso. Casos de uso devem possuir um nome capaz de identificá-lo e são apresentados como elipses com este nome em seu centro. Um exemplo desta representação gráfica é apresentado na figura A.6.

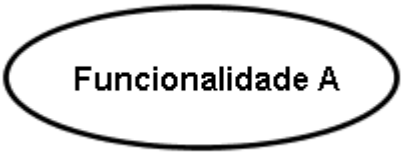


**Figura A.6 Exemplo de Caso de Uso**

## A2. Principais Diagramas

### A2.1 Diagrama de Classes

Representam a estrutura estática de um sistema, incluindo seus componentes, a estrutura interna destes e as relações entre eles. A figura A.7 apresenta um diagrama de classes simplificado.

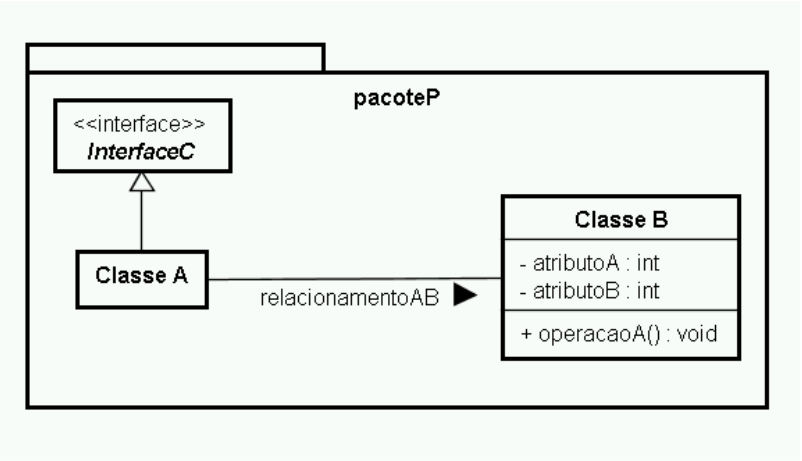


**Figura A.7 Diagrama de classes simplificado**

## A2.2 Diagrama de Caso de Uso

Mostra os relacionamentos entre os atores de um sistema e outras entidades. Estas entidades podem representar funcionalidades de uma classe ou um pacote, por exemplo. A notação deste diagrama é simples e apresenta basicamente os atores relacionados diretamente com as entidades esperadas de acordo com o sistema. Pode-se usar um retângulo para determinar a ligação entre um conjunto de atores e entidades a algum requisito. Na figura A.8 um diagrama de caso de uso é ilustrado.

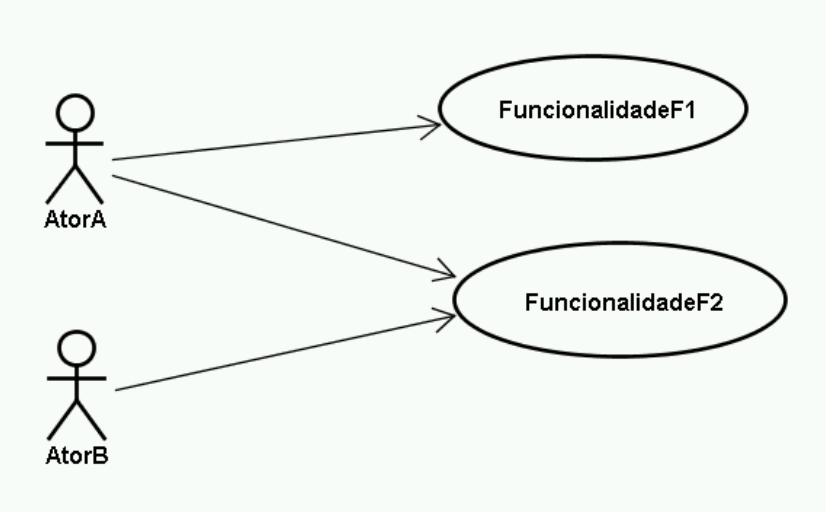


**Figura A.8 Diagrama de caso de uso simplificado**