UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO VICTORA HECHT

# Análise Automática de Código para Programação Orientada a Aspectos

Dissertação submetida à avaliação, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

**Prof. Dr. Marcelo Soares Pimenta**
**Orientador**

**Porto Alegre, março de 2007**

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Hecht, Marcelo Victora

Análise Automática de Código para Programação Orientada a Aspectos/Marcelo Victora Hecht – Porto Alegre:PPGC da UFRGS, 2007

79p.:iL.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação, Porto Alegre, BR-RS, 2007.

Orientador: Marcelo Soares Pimenta

1. Programação Orientada a Aspectos. 2. Early Aspects. 3.AspectJ. 4. XMI. 5. Refatoração. I. Pimenta, Marcelo Soares. II. Título.

*"All work and no play make Jack a dull boy."*
– PTAH-HOTEP, filósofo egípcio
– CEL. SAITO, "The Bridge over River Kwai"
– JACK TORRANCE, "The Shining"

# AGRADECIMENTOS

À minha esposa Vera, que me ajudou em tantos momentos de dúvida, e me permitiu tentar buscar um caminho diferente.

À minha família, pelo amor e carinho.

Aos meus sogros Alberto e Martina, que fazem mais por mim do que muitos pais fazem por seus filhos biológicos.

Aos professores e colegas do grupo de pesquisas sobre Programação Orientada a Aspectos: meu orientador Prof. Marcelo Pimenta, Prof. Roberto Tom Price, Eduardo Piveta, John Kliff Jochens e Marcelo Czembruski.

Aos meus amigos Christian, que me inspirou a estender meus estudos na pós-graduação, e André, pelas sugestões que contribuíram muito para a elaboração deste trabalho.

À Universidade Federal do Rio Grande do Sul, por me ter proporcionado mais dois anos de ensino público, gratuito e de qualidade.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

O Desenvolvimento de Software Orientado a Aspectos (AOSD) vem se consolidando como uma forma de resolver vários problemas das técnicas convencionais de programação, em particular em sistemas onde diversos interesses se encontram entrelaçados. A popularização dessa tecnologia faz surgir a necessidade de metodologias e ferramentas que facilitem o seu uso, como refatorações que levem em consideração suas características. No entanto as técnicas de modelagem de software disponíveis para AOSD não tem amadurecido no mesmo passo que as de implementação. Assim, para se poder pensar em mecanismos automáticos que trabalhem com a separação de interesses, é preciso verificar se as técnicas de modelagem existentes comportam isso.

Este trabalho propõe uma adaptação da abordagem Theme de modelagem, para que ela permita uma representação mais fiel de sistemas que utilizam orientação a aspectos, em especial os que utilizam a linguagem AspectJ. Essa técnica proposta é utilizada para demonstrar algumas maneiras de detectar *bad smells* em sistemas orientados a aspectos. Também é mostrado como essa modelagem pode ser usada como base para a geração automática de código orientado a aspectos, e como pode ser feita a engenharia reversa de código existente de forma que ele possa ser analisado em forma de modelo.

Palavras-chave: Programação Orientada a Aspectos, Early Aspects, AspectJ, XMI, Refatoração.

**Automatic Source Code Analysis for Aspect-Oriented Programming**

# ABSTRACT

Aspect-Oriented Software Development (AOSD) is increasingly being considered a way to solve several problems in conventional programming methods, particularly in systems with crosscutting concerns. The popularization of this technology brings the need for methodologies and tools to ease its use, such as refactorings that take into account its characteristics. However modeling techniques available for AOSD are not maturing at the same rate as implementation techniques. Thus, in order to be able to devise automatic mechanisms that deal with separation of concerns, it is first necessary to verify if existing modeling techniques support that.

In this work, we propose an adaptation of the Theme modeling approach, so that it represents aspect-oriented systems more closely, especially those using the AspectJ language. This technique is used to demonstrate a few ways of detecting *bad smells* in aspect-oriented systems. It is also shown how this model can be used as a basis for the automatic generation of aspect-oriented code, and how existing code can be reverse-engineered so that its model can be analyzed.

Keywords: Aspect-Oriented Programming, Early Aspects, AspectJ, XMI, Refactoring.

# 1 INTRODUÇÃO

O Desenvolvimento de Software Orientado a Aspectos (AOSD, do inglês *Aspect-Oriented Software Development*)(KICZALES 1997) é uma proposta recente, mas já bastante popular na área de desenvolvimento de sistemas. Isso é comprovado pela quantidade de pesquisas referentes a esse tema desde que ela surgiu (FILMAN 2005). Ele se caracteriza pela preocupação com a *separação de interesses*, a capacidade de isolar os diversos requisitos de um sistema complexo de forma que eles possam ser considerados separadamente.

Uma linha de pesquisa sobre AOSD se ocupa com a separação de interesses ainda na fase de análise e modelagem de sistemas. Os trabalhos nessa área, denominada *Early Aspects*, averiguam como interesses concorrentes podem ser identificados simultaneamente com a investigação dos requisitos de sistema, para verificar se eles podem ser separados, e, se isso não for possível, como eles precisam ser integrados (RASHID 2002). Diversas técnicas já foram propostas para promover essa aplicação, mas uma em particular se destaca pela maturidade: a Theme (CLARKE 2005).

Mas a principal aplicação do conceito de orientação a aspectos tem sido sua utilização para melhorar a modularidade de linguagens de programação, em particular as orientadas a objetos. Isso porque a orientação a aspectos permite concentrar todo o código relativo a um determinado interesse (um agrupamento de requisitos) do sistema em um mesmo lugar, ao mesmo tempo em que separa esse código dos relativos a outros interesses. A implementação mais desenvolvida desse conceito é uma adaptação da linguagem Java para suportar conceitos de orientação a aspectos, denominada AspectJ (ASPECTJ 2006).

Essa separação de interesses, e conseqüente melhora na modularidade, não trazem benefícios apenas no desenvolvimento inicial de um sistema, mas também (e talvez até mais) em sua manutenibilidade. Por isso, há um grande interesse em descobrir como sistemas existentes podem aproveitar a nova tecnologia de orientação a aspectos. Nesse sentido, a Orientação a Aspectos se encontra com outro conceito cuja importância na engenharia de software vem crescendo: a Refatoração de Software (FOWLER 2000). Sistemas existentes podem ser adaptados para aproveitar os benefícios da orientação a aspectos

através de uma técnica de refatoração chamada Extração de Aspectos (MEYER 1997), que busca identificar construções em programas orientados a objetos que seriam melhor expressas através de aspectos.

## 1.1 Motivação

Inicialmente, o foco deste trabalho seria uma continuidade direta de outro trabalho desenvolvido no grupo de estudos de Desenvolvimento de Software Orientado a Aspectos: a detecção de *bad smells* (FOWLER 2000) em código orientado a aspectos. A proposta seria focada em analisar automaticamente software orientado a aspectos através de um modelo em um formato compreensível pelo computador. O interesse principal é na linguagem de programação orientada a aspectos AspectJ (ASPECTJ 2006), por ela ser a mais popular dentro da comunidade de AOSD.

No entanto, logo no início do trabalho detectou-se que as técnicas de modelagem de software orientado a aspectos ainda se encontram em um estágio bastante inicial. Isso não é surpresa, considerando que depois da criação do paradigma de orientação a objetos no final dos anos 60 levou-se quase 20 anos para ser desenvolvida uma técnica de modelagem apropriada para a decomposição do sistema em objetos. Isso criou a necessidade de concentrar a maior parte do esforço do trabalho na investigação e melhoria de técnicas existentes de modelagem de software orientado a aspectos.

As propostas para técnicas de modelagem de software orientado a aspectos existentes em geral baseiam-se na UML (UML 2004), a linguagem mais difundida para a modelagem de software orientado a objetos, inserindo elementos relativos à decomposição do sistema em interesses. Detectou-se que as propostas existentes dividem-se em dois grupos. Algumas fornecem um nível de abstração muito alto (RASHID 2002, CLARKE 2005) e, portanto, não apresentam uma relação trivial com a implementação dos seus modelos, em particular com AspectJ. Outras (PAWLAK 2002, BASCH 2003), inversamente, se limitam a incluir algumas estruturas de AspectJ (*aspects*, *advices*, *pointcuts* e *join points*) na UML, tornando-as dependentes demais da tecnologia de implementação.

A principal carência das técnicas de modelagem orientada a aspectos existentes é, todavia, a carência de recursos computacionais que lhes dêem suporte. Mesmo a proposta mais empregada, Theme (CLARKE 2005), não possui ferramentas que lhe dêem suporte. E a possibilidade de trabalhar com esses modelos em um formato compreensível por um computador é pré-requisito para qualquer análise automática proposta.

## 1.2 Contribuições

Das dificuldades mencionadas acima, surgiu o interesse no desenvolvimento de uma técnica de modelagem que alivie alguns dos problemas detectados nas propostas atuais. A principal contribuição deste trabalho é a modificação da proposta Theme de modelagem orientada a aspectos, com o intuito de atingir os seguintes objetivos:

- Um nível de abstração baixo o suficiente para que características da implementação do sistema sejam visíveis;
- Um nível de abstração alto o suficiente para que o modelo possa ser realizado usando outras linguagens de programação além de AspectJ;
- Utilização de recursos existentes em ferramentas de modelagem UML 2 para representar conceitos referentes a aspectos;
- Armazenamento dos modelos em um formato que seja facilmente compreensível tanto por programas de computador quanto pelos desenvolvedores.

Algumas aplicações foram desenvolvidas sobre esse modelo, e também são contribuições deste trabalho:

- Foram definidas técnicas para detectar alguns *bad smells* característicos de software orientado a aspectos através de análise do modelo;
- Para testar a validade e o poder de expressão do modelo, foi desenvolvido um gerador de código capaz de gerar a estrutura de um sistema orientado a aspectos a partir desse modelo;
- Também foi criada uma ferramenta de engenharia reversa capaz de criar um modelo a partir de código AspectJ.

## 1.3 Estrutura do trabalho

O trabalho se encontra estruturado da maneira definida a seguir.

- No capítulo 2 são apresentados os conceitos relativos a desenvolvimento de software orientado a aspectos: os problemas que ele busca resolver e os mecanismos nele definidos. É feito um breve resumo de como ocorre o processo de desenvolvimento de software usando a decomposição em aspectos. São apresentadas algumas propostas existentes para a modelagem de software orientado a aspectos. Além disso, são apresentados conceitos relativos às aplicações buscadas para a modelagem orientada a aspectos aqui desenvolvida: refatorações, detecção de *bad smells*, e geração de código.
- No capítulo 3 é apresentada a técnica de modelagem proposta para software orientado a aspectos. É feita uma apresentação da proposta

Theme/UML, e são explicitadas suas vantagens e deficiências. São mostradas as mudanças efetuadas sobre ela para torná-la apropriada para a representação, em um formato legível pelo computador, de sistemas reais implementados com o uso de linguagens de programação orientadas a aspectos.

- No capítulo 4 é desenvolvida a idéia de analisar automaticamente código orientado a aspectos, para a detecção de *bad smells*.
- No capítulo 5 são apresentadas outras possibilidades de aplicação do modelo desenvolvido: a geração de estruturas de classes e aspectos a partir do modelo usando XSLT (XSLT 2005), e como sistemas existentes podem ser, através de engenharia reversa, representados no modelo para que possam ser analisados.
- Por fim, no capítulo 6 são apresentadas as conclusões obtidas no desenvolvimento do trabalho, e também as possibilidades de desenvolvimento futuro das idéias aqui apresentadas.

## 1.4 Termos utilizados

Por ser uma área relativamente nova da computação, ainda não existe uma tradução usual para muitos termos utilizados na literatura a respeito de programação orientada a aspectos. A terminologia utilizada neste trabalho é a proposta no Relatório do 1o. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (GARCIA 2004), listada na tabela 1.1.

**Tabela 1.1: Traduções da terminologia de Programação Orientada a Aspectos**

| Termo Original | Tradução |
|---|---|
| Aspect | Aspecto |
| Inter-type declaration | Declaração intertipos |
| Join point | Ponto de junção |
| Pointcut | Conjunto (de pontos) de junção |
| Concern | Interesse |
| Separation of concerns | Separação de interesses |
| Crosscut | Entrecortar |
| Crosscutting concern | Interesse transversal |
| Scattering | Espalhamento |
| Tangling | Entrelaçamento |
| Obliviousness | Inconsciência |
| Weaving | Combinação |
| Aspect weaver | Combinador |
| Advice | Comportamento transversal / Adendo |

Para facilitar a compreensão do texto, já que a grande maioria da literatura sobre o assunto está em inglês, o termo em inglês será repetido ao longo do texto na primeira utilização do termo em português equivalente.

# 2 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS

Na década de 1990, a comunidade do desenvolvimento de sistemas começou a perceber que as técnicas de programação existentes (programação orientada a objetos, técnicas de engenharia de software, linguagens para desenvolvimento rápido de aplicações) não eram suficientes para atender a complexidade crescente dos projetos (ARMOUR 2001). Embora essas técnicas cumpram muito bem o objetivo ao qual elas se propuseram – o de oferecer uma correspondência mais direta entre a estrutura da solução e a estrutura do problema (MEYER 1997) – na percepção de diversos autores, o maior problema estava na representação de *interesses transversais* (*crosscutting concerns*). Esses são requisitos de um sistema que, apesar de não serem o foco principal do sistema, se combinam com outros (como persistência, segurança, rastreamento e tratamento de exceções) (ELRAD 2001b).

As técnicas de separação de interesses tradicionais, como as procedurais e as orientada a objetos, em geral fornecem apenas uma dimensão para a decomposição do problema tratado. No entanto, onde interesses transversais estão presentes, essa decomposição causa 2 grandes problemas: *entrelaçamento de código (code tangling)*, pois a solução dos interesses transversais se encontra combinada à dos requisitos principais da aplicação; e *espalhamento de código (code scattering)*, pois a modelagem ou implementação do mesmo interesse transversal termina espalhada ao longo de vários módulos da aplicação. Por exemplo, um requisito de robustez para o sistema pode exigir que código para tratamento de exceções seja implementado dentro de vários métodos do sistema (espalhamento), misturado ao código do método que é realmente relevante para atender o objetivo do sistema (entrelaçamento). Entre os problemas que isso causa incluem-se (LADDAD 2001):

- *Dificuldade de compreensão do código* – A implementação simultânea de diversos interesses oculta a correspondência entre um interesse e sua implementação, resultando em um mapeamento difícil entre os dois.
- *Baixa produtividade* – A implementação simultânea de múltiplos interesses muda o foco do desenvolvedor do interesse principal para interesses periféricos, levando à baixa produtividade.

- *Menor reuso de código* – Como, devido às circunstâncias, um módulo implementa interesses múltiplos, outros sistemas requerendo funcionalidades similares podem não conseguir utilizar o módulo prontamente, reduzindo ainda mais a produtividade.
- *Qualidade de código baixa* – Entrelaçamento de código produz código com problemas ocultos. Além disso, por atender a diversos interesses de uma só vez, um ou mais desses interesses podem não receber atenção suficiente.
- *Dificuldade de evolução* – Uma visão limitada e recursos restritos frequentemente produzem um projeto que atende apenas interesses atuais. Atender requisitos futuros pode obrigar a um retrabalho na implementação. Como a implementação não é modularizada corretamente, isso implica em modificar muitos módulos. Modificar cada subsistema para tais mudanças pode levar à inconsistências, e também requer um esforço considerável no teste para assegurar que a mudança na implementação não causou *bugs*.

Várias propostas surgiram nos últimos anos para permitir que interesses transversais sejam solucionados isoladamente, entre elas a Separação Multidimensional de Interesses (*Multi-Dimensional Separation of Concerns*) (OSSHER 2001a) e a Programação Orientada a Assuntos (*Subject-Oriented Programming*) (HARRISON 1993). No entanto, a técnica que mais se popularizou, gerando não só a maior quantidade de pesquisas (FILMAN 2005), mas também a ferramenta mais avançada (ASPECTJ 2006) foi a *programação orientada a aspectos* (AOP, do inglês *aspect-oriented programming*) (KICZALES 1997).

Nessa técnica, interesses transversais são implementados em entidades denominadas *aspectos*, que possuem a capacidade de modificar o comportamento das entidades principais do programa, chamadas *componentes* (que no contexto de desenvolvimento orientado a objetos seriam representados por classes), onde os interesses principais do sistema estão implementados. Essa abordagem se popularizou de tal forma que o termo *programação orientada a aspectos* se tornou quase um sinônimo para a separação de interesses transversais, e praticamente todas as outras propostas existentes são comparadas a ela (ELRAD 2001b).

Ela se baseia em dois princípios: quantificação (*Quantification*), a capacidade de implementar funcionalidades uma só vez e aplicá-las em diversos pontos do código, e inconsciência (*Obliviousness*), a capacidade de aplicar aspectos sobre componentes sem que esses últimos tenham de ser preparados especificamente para isso (FILMAN 2005a). Eles podem ser vistos como conceitos opostos aos problemas de entrelaçamento e espalhamento de código. A quantificação indica que interesses precisam ser implementados

apenas uma vez e usados ao longo de todo o sistema, ao contrário do espalhamento de código. A inconsciência promove que a implementação de cada interesse pode evoluir de forma independente das outras, em oposição ao entrelaçamento. Esses são os fatores que fazem com que o desenvolvimento orientado a aspectos melhore a modularidade (CHAVEZ 2004, KICZALES 2005).

Além disso, uma implementação em particular da AOP tem se difundido tremendamente: a linguagem AspectJ (ASPECTJ 2006). Ela permite a implementação de interesses transversais em Java, através de construções como aspectos, pontos de junção e adendos (KICZALES 2001a, KICZALES 2001b). Já tendo sido testada inclusive em projetos comerciais (LADDAD 2005), a linguagem AspectJ, da mesma maneira que a AOP em si, também se tornou o padrão pelo qual todas as outras abordagens são avaliadas (BRICHAU 2005, FILMAN 2005).

## 2.1 Conceitos de Programação Orientada a Aspectos

Antes de apresentar as propostas referentes à modelagem orientada a aspectos, é conveniente fazer uma rápida revisão dos conceitos relacionados a esse tipo de programação, principalmente do ponto de vista da linguagem AspectJ, a implementação mais popular. No entanto, é importante ter sempre em mente que a orientação a aspectos, tanto em termos de programação quanto de modelagem, ainda é uma ciência muito nova, e a forma final que ela tomará pode não ser uma das existentes atualmente (CLARKE 2002).

O mecanismo mais utilizado em AOSD para a implementação de aspectos é o uso de *adendos* (em inglês, *advices*), trechos de código que devem ser inseridos ou substituir trechos na estrutura ou execução de componentes, e *pontos de junção* (em inglês, *join points*), trechos identificáveis dentro da execução de um componente, onde os adendos podem ser inseridos (KICZALES 1997). Esses mecanismos podem ser usados de forma *dinâmica* (em inglês, *dynamic crosscutting*), para a modificação da execução dos componentes, executando adendos antes ou depois de métodos do componente, substituindo parâmetros, ou mesmo trocando a execução de um método do componente por um adendo do aspecto. Ou podem ser usados de forma *estática* (em inglês, *static crosscutting*), alterando a própria estrutura dos componentes: adicionando métodos e propriedades, ou modificando a hierarquia de classes.

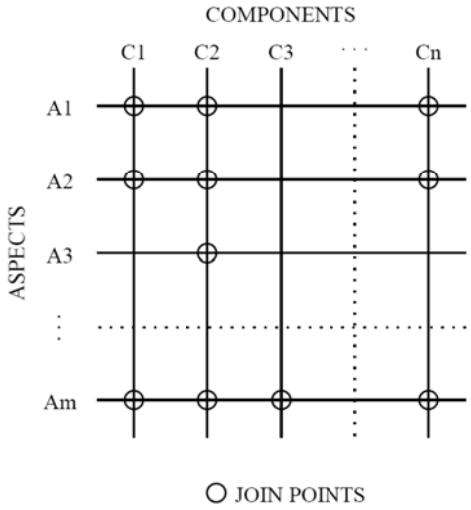


Figura 2.1: Relação entre Aspectos, Componentes e Pontos de Junção (BARDOU 1998)

### 2.1.1 Aspecto (*Aspect*)

Segundo a proposta do Desenvolvimento de Software Orientado a Aspectos (AOSD), as propriedades que devem ser implementadas em um sistema de software podem ser classificadas como (KICZALES 1997):

- Um *componente*, se ela pode ser encapsulada em um procedimento (objeto, método, API) de forma localizada, facilmente acessível e composta quando necessário. Componentes tendem a ser unidades da decomposição funcional do sistema.
- Um *aspecto*, se ela não é uma unidade da decomposição funcional, mas uma propriedade que afeta a performance ou semântica dos componentes.

Um *aspecto* é a implementação de um interesse transversal (ELRAD 2001a) realizada de maneira independente da implementação dos requisitos centrais.

Uma definição, mais informal, de aspectos em termos de programação orientada a objetos é (HIGHLEY 1999):

- Um objeto é alguma coisa.
- Um aspecto não é alguma coisa. Um aspecto é alguma coisa a respeito de alguma coisa.
- Objetos não são dependentes de aspectos.
- Aspectos representam alguma característica ou propriedade de objetos, mas eles não possuem controle sobre objetos.

### 2.1.2 Ponto de Junção (*Join Point*) e Conjunto de Pontos de Junção (*Pointcut*).

*Pontos de junção* são os locais no sistema que são afetados por interesses transversais (OSSHER 1998). Eles são eventos que representam a passagem por locais semanticamente bem definidos no sistema (ELRAD 2001a). Pontos de junção podem ser associados a *operações* do sistema, como passagem de mensagens entre objetos, chamadas de métodos, execução de sub-rotinas e tratamento de exceções; ou podem ser *relativos a comandos*, quando todo comando da linguagem é associado a um ponto de junção. Esse segundo tipo não é usado, pois não necessariamente aumenta o poder da linguagem e é mais invasivo e imprevisível (OSSHER 1998).

O modelo de pontos de junção varia em cada interpretação de programação orientada a aspectos. Ele pode usar nodos de grafos que representam o sistema (LIEBERHERR 2001), elementos da estrutura do sistema como classes e métodos (OSSHER 2001b), passagens de mensagens entre objetos (BERGMANS 2001), ou ações na execução de um programa como chamadas de métodos e construções de objetos (KICZALES 2001a).

Para serem usados, muitas linguagens precisam que eles sejam agrupados em *conjuntos de pontos de junção*, que selecionam os pontos de junção que serão associados a comportamentos (NETINANT 2001).

### 2.1.3 Adendo (*Advice*)

Em linguagens de programação orientadas a aspectos, *adendos* são construções de código que devem ser executadas quando determinados pontos de junção (especificados em conjuntos de pontos de junção) são alcançados. Em geral um adendo é uma construção semelhante a um método ou a uma sub-rotina, e que na ativação de um determinado ponto de junção (por exemplo, em uma chamada de método), será executada antes, depois, ou ao invés dele.

### 2.1.4 Combinador (*Aspect Weaver*)

Combinadores são usados para combinar componentes (o código principal do programa) e aspectos (a implementação dos interesses transversais) (ELRAD 2001b). Para que isso ocorra, é essencial que existam pontos de junção definidos (BRICHAU 2005).

## 2.2 O processo de desenvolvimento usando AOP

O parecer da AOSD é que devem existir mecanismos, tanto de modelagem quanto de programação, que possibilitem a representação direta de interesses transversais (ELRAD 2001a). Isso implica na criação de técnicas de modelagem e de ferramentas de programação que dêem suporte a isso. O processo de desenvolvimento orientado a aspectos é composto de 3 passos principais (LADDAD 2001):

### 2.2.1 Decomposição em Aspectos

O primeiro passo da AOSD isso é a *decomposição em aspectos*: a identificação, na modelagem do sistema, não apenas da estrutura necessária para representar as funcionalidades do sistema em módulos (os *componentes*), mas também dos interesses transversais ao sistema global (os *aspectos*).

No exemplo do sistema gráfico, a decomposição "dominante" seria a da modelagem dos objetos gráficos (pontos e linhas) em *componentes*. A necessidade de atualizar a visualização do sistema seria um *aspecto* do sistema a ser implementado à parte.

### 2.2.2 Implementação dos Interesses

A partir disso é realizada a *implementação de interesses*, onde componentes e aspectos são implementados usando mecanismos que forneçam subsídios para a localização de expressões de um interesse transversal. O simples uso de mecanismos como sub-rotinas ou herança de classes não é suficiente para isso, pois isso muitas vezes implica em que o seu desenvolvimento precise considerar que ela possa ser reutilizada de diversas maneiras, e o reuso requer o conhecimento completo de seu funcionamento. AOSD propõe que existam mecanismos implícitos para a utilização de comportamentos em cujo desenvolvimento não foi contemplada a possibilidade de existirem interesses adicionais (ELRAD 2001b). Para isso, se faz necessário o uso de linguagens de programação orientadas a aspectos.

No sistema exemplo, a abstração de objetos gráficos é feita utilizando uma estrutura de classes tradicional, inconsciente (em inglês, *oblivious* – ver a terminologia na seção "Termos utilizados") dos interesses transversais. A necessidade de atualizar a visualização dos objetos é um *aspecto* do sistema, composto por um adendo (a chamada ao método *displayUpdate*), e um *join point* em cada método que modifica os objetos gráficos (*setX*, *setY*, *setP1*, *setP2*).

### 2.2.3 Combinação de Aspectos

Por fim, é realizada a *combinação* (ou, mais propriamente, *recomposição*) *de aspectos* (ou, em inglês, *aspect weaving*), onde componentes e aspectos são integrados no sistema final. Esse processo é executado por um mecanismo denominado *combinador* (em inglês, *weaver*), e pode ocorrer de diversas maneiras, dependendo da implementação de AOSD utilizada.

A principal diferença entre os mecanismos de combinação está no momento em essa é realizada. Nos mecanismos de *combinação dinâmica* (em inglês, *dynamic weaving*), a junção dos adendos aos componentes é realizada em tempo de execução. Isso pressupõe que a tecnologia utilizada forneça algum subsídio para a alteração em tempo de execução da estrutura de classes, e/ou um método para interceptar as mensagens inter e intra-objetos. Nesse caso, o

combinador é capaz de fazer as modificações necessárias nos componentes do sistema em tempo de execução. Nos mecanismos de *combinação estática* (*static weaving*), o combinador é um sistema transformativo que une os artefatos (em geral, código fonte ou código objeto) dos componentes e dos aspectos do sistema, gerando novos artefatos que serão usados para criar o sistema (POPOVICI 2002).

Aplicando o conceito de combinação estática ao exemplo do sistema gráfico apresentado anteriormente, o combinador integraria à implementação dos pontos de junção (os métodos de alteração dos objetos) a execução do adendo (a chamada ao método *DisplayUpdating*). É interessante notar que o resultado de um combinador estático é praticamente igual ao que seria se o sistema houvesse sido implementado através de técnicas tradicionais. A diferença é que, do ponto de vista do desenvolvedor, os interesses são implementados (e assim podem ser evoluídos e reutilizados) independentemente.

## 2.3 Um exemplo de Desenvolvimento Orientado a Aspectos

Para ilustrar uma aplicação do desenvolvimento orientado a aspectos, será ilustrada aqui uma implementação de um padrão de projeto Observer (GAMMA 1994) sobre um sistema gráfico simplificado, composto de linhas e pontos. Esse exemplo foi usado em (HANNEMANN 2002), para ilustrar os benefícios da orientação a aspectos, e um exemplo semelhante foi usado em (CLARKE 2002) para ilustrar a abordagem Theme.

Padrões de projeto oferecem soluções flexíveis e reutilizáveis para problemas comuns no desenvolvimento de software. No entanto, a sua aplicação traz alguns efeitos colaterais. Eles atribuem papéis para as classes envolvidas em cada padrão, mas muitas vezes essas classes já representam outros papéis, que atendem aos requisitos principais do sistema. Assim, pode-se considerar que padrões de projeto são interesses transversais do sistema. Ao mesmo tempo, embora conceitualmente padrões de projeto possam ser

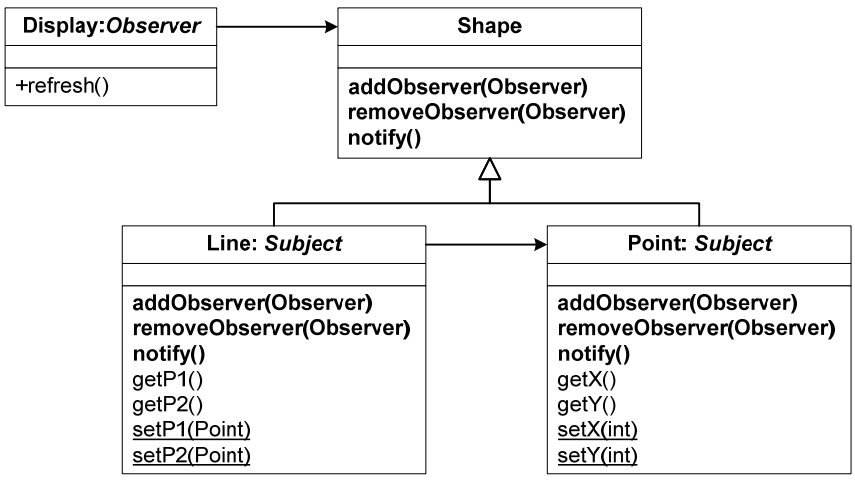


Figura 2.2: Implementação do padrão de projeto Observer sobre um sistema gráfico, usando orientação a objetos

```
public class Point implements ChangeSubject {
    private HashSet observers;

    private int x;
    private int y;

    public Point(int x, int y) {
       this.x = x;
       this.y = y;
       this.observers = new HashSet();
    }

    public int getX() { return x; }

    public int getY() { return y; }

    public void setX(int x) {
        this.x = x;
        notifyObservers();
    }

    public void setY(int y) {
        this.y = y;
        notifyObservers();
    }

    public void addObserver(ChangeObserver o) {
        this.observers.add(o);
    }
    public void removeObserver(ChangeObserver o) {
        this.observers.remove(o);
    }
    public void notifyObservers() {
        for (Iterator e = observers.iterator() ; e.hasNext() ;) {
            ((ChangeObserver)e.next()).refresh(this);
        }
    }
}
```

Figura 2.3: Código da classe Point implementando o papel de Subject no padrão de projeto Observer

reutilizados, sua implementação precisa ser refeita a cada uso, pois ela se encontra entrelaçada com as classes do sistema.

A Figura 2.2 mostra uma implementação do padrão Observer sobre um sistema gráfico, utilizando programação orientada a objetos comum. Os métodos sublinhados precisam ser modificados para a implementação do padrão de projeto, enquanto que os métodos em negrito só existem por causa dela. Para fins de concisão não será incluído aqui o código de cada elemento do sistema, mas fica claro apenas observando o código da classe Point (Figura 2.3) que bastante código repetido precisa existir para implementar o padrão de projeto, incluindo tabelas para armazenar os observers de cada elemento, e iteradores para notificá-los. Esse código se encontra entrelaçado com o que lida com o interesse principal da aplicação – a implementação de um sistema gráfico – espalhado ao longo da aplicação, não apenas dentro dos métodos *set* da classe Point, mas também dentro de todas as outras classes gráficas do sistema. Dessa forma, ele representa um candidato ideal para ser melhorado com o uso de orientação a aspectos.

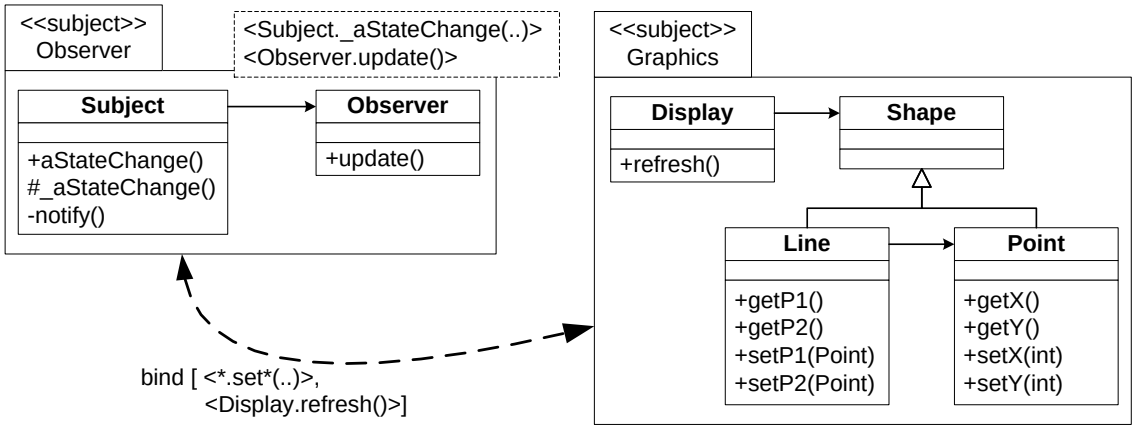


Figura 2.4: Implementação do padrão de projeto Observer sobre um sistema gráfico, usando orientação a aspectos

A modelagem do sistema gráfico usando orientação a aspectos, representada usando Theme/UML (que será vista na seção 2.5), pode ser vista na Figura 2.4. Toda a funcionalidade relativa ao padrão de projeto é extraída do interesse principal do sistema, de forma que possa ser desenvolvida separadamente, e reutilizada em outros projetos. O código dessa solução se encontra na Figura 2.5.

## 2.4 Modelagem Orientada a Aspectos

As técnicas de modelagem apresentadas aqui se dividem basicamente em dois grupos: as que usam elementos da própria UML para expressar aspectos, e as que utilizam novas construções com o propósito específico de modelar o entrecortamento (*crosscutting*) de interesses. A exceção é a abordagem de (RASHID 2002), que se baseia diretamente em engenharia de requisitos, sem entrar no âmbito de UML.

A grande vantagem de usar apenas elementos já existentes em UML é evitar a quebra de compatibilidade com o modelo UML padrão, e, portanto, com todas as ferramentas e processos já preparados para ele (GROHER 2003). Por outro lado, isso obrigatoriamente leva a algumas concessões (CHAVEZ 2002), como uma representação confusa do entrelaçamento. Como se argumenta em (BASCH 2003), assim como novos elementos precisaram ser incluídos em linguagens de programação para suportar aspectos, o mesmo se faz necessário em linguagens de modelagem.

Outro fator de discordância entre as diversas propostas é o modelo preciso de aspectos usado. A maioria se esforça para atender aos requisitos específicos de AspectJ, modelando aspectos, conjuntos de pontos de junção e adendos diretamente. No entanto, como lembram (CHAVEZ 2002, CLARKE 2002), a forma final das linguagens de programação orientada a aspectos pode não ser nenhuma das existentes no momento, e uma linguagem de modelagem

que simplesmente imite as construções de uma delas falha em obter independência de implementação.

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
       this.x=x;
       this.y=y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
}

public abstract aspect ObserverProtocol {
    protected interface Subject  { }
    protected interface Observer { }

    private WeakHashMap perSubjectObservers;
    protected List getObservers(Subject subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }
    public void addObserver(Subject subject, Observer observer) {
        getObservers(subject).add(observer);
    }
    public void removeObserver(Subject subject, Observer observer) {
        getObservers(subject).remove(observer);
    }

    protected abstract pointcut subjectChange(Subject s);
    after(Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
            updateObserver(subject, ((Observer)iter.next()));
        }
    }
    protected abstract void
        updateObserver(Subject subject, Observer observer);
}

public aspect CoordinateObserver extends ObserverProtocol{
       declare parents: Point  implements Subject;
       declare parents: Screen implements Observer;
       protected pointcut subjectChange(Subject subject):
              (call(void Point.setX(int)) ||
               call(void Point.setY(int)) ) && target(subject);
       protected void
          updateObserver(Subject subject, Observer observer) {
             ((Screen)observer).display("Screen updated ");
       }
}
```

Figura 2.5: Código da classe Point e dos aspectos que implementam o padrão de projeto Observer

### 2.4.1 UML (Unified Modelling Language)

Praticamente todas as propostas de modelagem contemplando separação de interesses se baseiam na linguagem UML (UML 2004), modificando-a de uma maneira ou de outra (RASHID 2005). Daí a utilidade de realizar uma breve revisão dos conceitos de UML.

A linguagem UML não é um processo de desenvolvimento, mas sim uma linguagem para a especificação e visualização de artefatos de um sistema de software que pode ser utilizada em diversas metodologias. Ela é composta por uma série de diagramas, muitos deles apresentando as mesmas entidades sob pontos de vista diferentes. Os principais diagramas utilizados na UML são os de:

- Casos de uso;
- Classes;
- Estados;
- Interação (seqüência e colaboração);
- Implementação (componentes e implantação).

A idéia é que a partir desses diagramas seja possível definir com o mínimo de ambigüidade a forma com que o sistema deve ser implementado.

A UML prevê uma série de mecanismos para sua extensão. Ela própria é definida sobre uma arquitetura denominada MOF (*MetaObject facility*) (MOF 2006), capaz de representar não apenas modelos UML como outros definidos pelo usuário. Entre os mecanismos de extensão contidos na própria UML, encontram-se:

- *Estereótipos (stereotypes) e Marcações (tagged values)* – valores de texto associados a elementos do diagrama, que, apesar de não possuírem semântica dentro da UML, podem ser usados para fornecer informações para ferramentas;

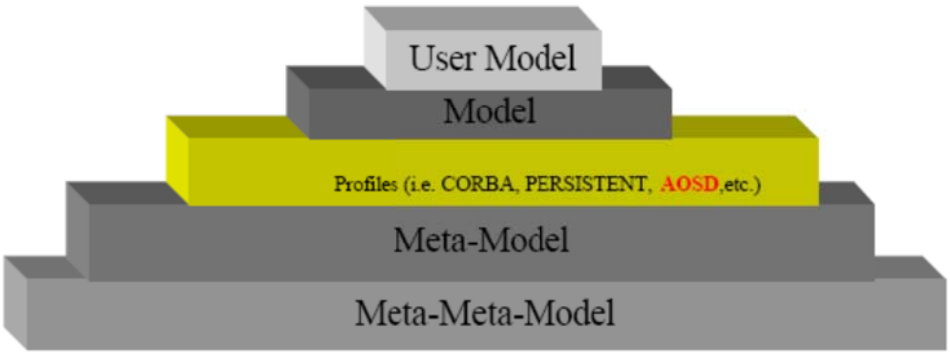


Figura 2.6: Arquitetura UML/MOF

- *Restrições (constraints)* – Regras que podem modificar a semântica de UML.

Esse modelo também contempla a especificação XMI (*XML Metadata Interchange*) (XMI 2005), que pode ser usada para o intercâmbio de modelos entre diversas ferramentas (e que pode, como será visto na seção 4.2, ser usado por geradores de código). Diversas ferramentas de modelagem UML atualmente suportam exportar seus modelos para XMI (apesar de diferirem bastante em suas interpretações da especificação da linguagem). Entre elas, podemos citar as ferramentas Rational (RATIONAL 2006), ArgoUML (ARGOUML 2006), Poseidon (POSEIDON 2006) e MVCase (MVCASE 2006).

Outro conceito introduzido a partir da UML é a MDA (*Model-driven architecture*) (MDA 2006), que lida com a possibilidade de transformar modelos com um nível suficiente de detalhamento diretamente em programas executáveis. Esse conceito será visto em mais detalhes na seção 4.2.

### 2.4.2 Extending UML with Aspects: Aspect Support in the Design Phase (SUZUKI 1999)

Esta proposta, bastante antiga, mas de grande relevância (RASHID 2005), é a única a incluir uma representação XML de aspectos, mas em uma linguagem proprietária denominada UXL. Ela traz semelhanças com (PAWLAK 2002) nos elementos adicionados ao metamodelo UML:

- *Aspect*: um elemento derivado de *classifier* (como *class*, *interface* e *component*) representando um aspecto.
- *Aspect-class relationship*: indica a aplicação de um aspecto a uma classe.
- *Woven class*: representa uma classe já combinada com o aspecto.

No entanto, o modelo (Figura 2.7) proposto não traz um nível de detalhamento suficiente para que possa levar diretamente à implementação. Não há indicação, por exemplo, de que tipo de combinação será feita entre os aspectos e as classes.

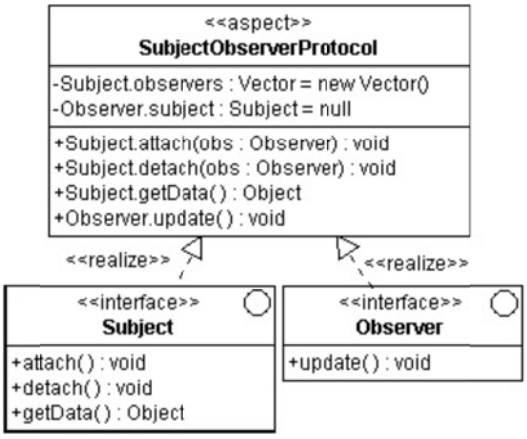


Figura 2.7: Aspectos e relacionamento com classes (SUZUKI 1999).

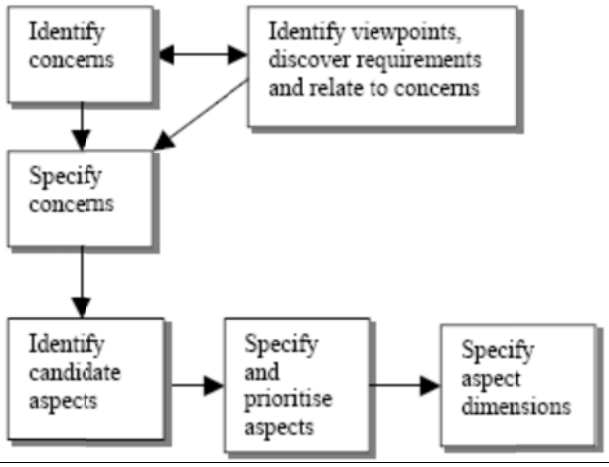


Figura 2.8: Modelo de Engenharia de Requisitos Orientada a Aspectos (RASHID 2002).

### 2.4.3 Early Aspects: A Model for Aspect-Oriented Requirements Engineering (RASHID 2002)

Este trabalho sugere um método para a engenharia de requisitos orientada a aspectos, afirmando que é essencial contemplar interesses transversais o mais cedo possível, ou eles podem inibir a adaptabilidade do sistema.

### 2.4.4 Aspect-Oriented Requirements with UML (ARAÚJO 2002)

Este artigo propõe uma expansão de UML para permitir o tratamento de interesses transversais durante a análise de requisitos do sistema. Ele divide o processo UML em 3 partes principais: interesses funcionais, interesses não-funcionais e interesses compostos dos dois anteriores. O artigo também mostra como modelos de casos de uso e de seqüência podem ser usados em conjunto para especificar a interação de aspectos.

### 2.4.5 A Metamodel for Aspect-Oriented Modeling (CHAVEZ 2002)

Esta proposta consiste em uma extensa adição de elementos ao metamodelo de UML para permitir a representação de aspectos (usando o modelo AspectJ). Ela não se preocupa em reutilizar elementos existentes do UML (como classes e relacionamentos), definindo cerca de 20 elementos específicos para AOP.

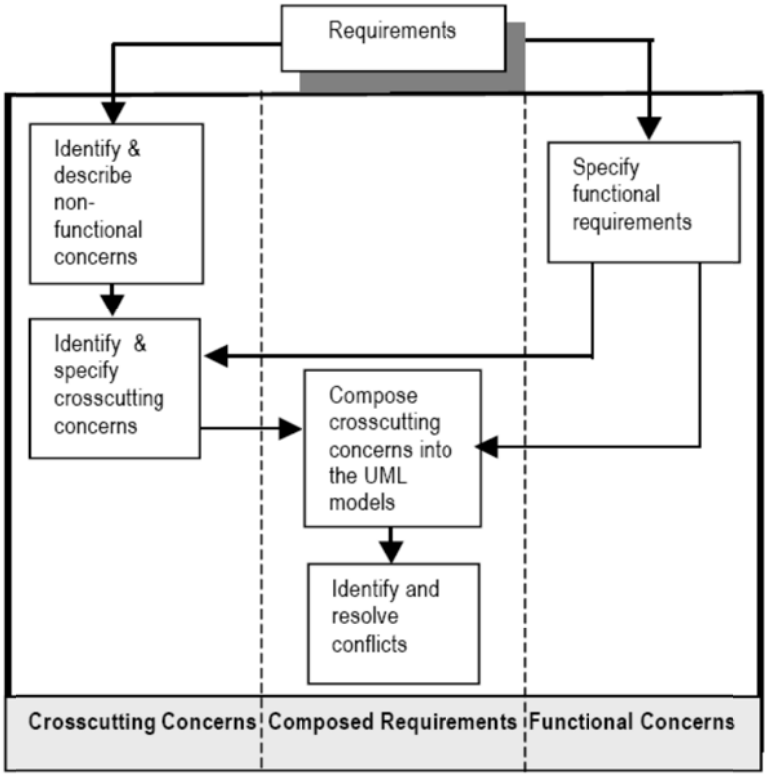


Figura 2.9: Um modelo para requisitos em orientação a aspectos (ARAÚJO 2002)

### 2.4.6 An UML-based Aspect-Oriented Design Notation (STEIN 2002)

Esse artigo apresenta métodos para a representação de programas em AspectJ usando elementos do UML padrão:

- Pontos de junção são representados através de *links* (associações) em diagramas de classe, e mensagens de invocação de "pseudo-operações" em diagramas de interação.
- Conjuntos de pontos de junção são representados através de operações associadas a um estereótipo *«pointcut»*. Essas operações possuem parâmetros (apenas de saída), e sua "implementação" é na verdade a definição dos pontos de junção.
- Adendos também são representados como operações, com o estereótipo *«advice»*.
- Introduções (modificações na estrutura estática do sistema) são representadas por colaborações (geralmente usadas para representar a implementação de um padrão de projeto (GAMMA 1994)).
- Por fim, Aspectos são representados por classes com o estereótipo *«aspect»*, que contém as construções definidas acima.

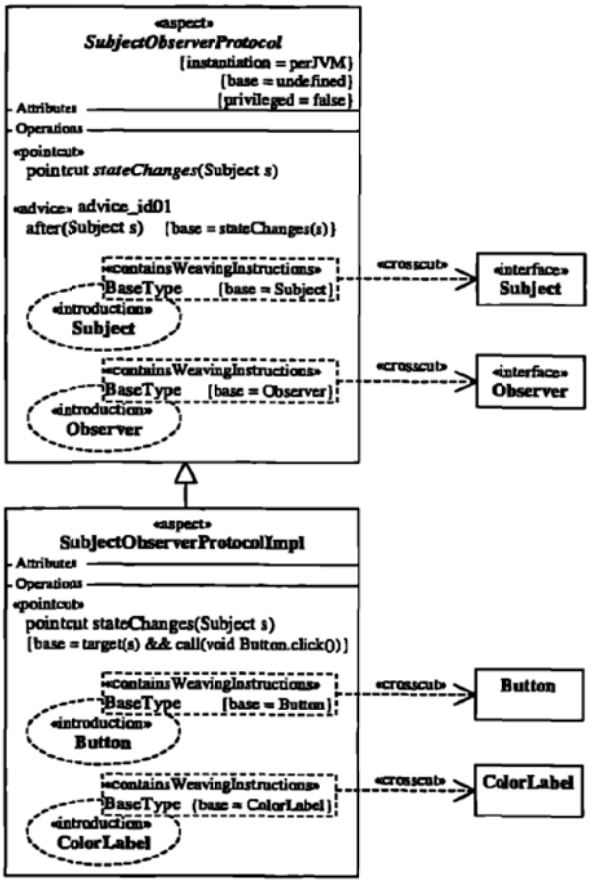


Figura 2.10: Modelo orientado a aspectos (STEIN 2002)

O sistema ainda utiliza diagramas de casos de uso para definir como a combinação de aspectos deve ser feita. No geral, o sistema parece bastante completo, embora de difícil visualização.

### 2.4.7 A Toolkit for Weaving Aspect Oriented UML Designs (HO 2002)

Esta sugestão não utiliza elementos novos para modelar aspectos, mas modifica a semântica de elementos existentes da UML como colaborações em diagramas de classes, e usa o diagrama de estados para indicar como as combinações devem ser feitas. Ele também não é centrado em AspectJ, mas em um framework próprio para a combinação de modelos.

A técnica apresentada, no entanto, não parece suficientemente detalhada para permitir a implementação de sistemas definidos nela.

### 2.4.8 A UML Notation for Aspect-Oriented Software Design (PAWLAK 2002)

Esta proposta utiliza 3 novos elementos no modelo de classes para representar aspectos em UML:

- *Grupos*: Um conjunto de objetos heterogêneos, mas que será afetado pelos mesmos aspectos;

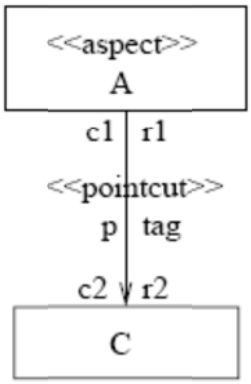


Figura 2.11: Aspectos e Relacionamento de Ponto de Junção (PAWLAK 2002).

- *Aspectos*: Semelhante a classes, mas seus métodos podem (através de estereótipos como «before», «after», etc.) ser usados para representar adendos;
- *Relacionamentos de pontos de junção*: Indica a aplicação de um aspecto à classe que ele modifica. Os papéis do relacionamento possuem uma semântica especial que representa o tipo de ponto de junção representado.

No geral, é um método que implica em poucas mudanças na linguagem UML padrão (basicamente acrescenta semântica nova a elementos existentes), e poderia ser facilmente implementado em ferramentas. No entanto, ele é voltado totalmente para AspectJ, e, para representar um ponto de junção que afeta vários métodos, exigiria uma seta de relacionamento para cada um, o que poderia rapidamente tornar o modelo incompreensível visualmente.

Um ponto interessante dessa proposta é a sugestão de criar um combinador de aspectos que atue no nível do modelo, capaz de transformar o modelo UML estendido com aspectos em um modelo UML padrão com aspectos já combinados.

### 2.4.9 Incorporating Aspects into the UML (BASCH 2003)

Mais um sistema que adiciona elementos ao modelo UML, e focado especificamente para a linguagem AspectJ. Ele varia, no entanto, ao permitir que conjuntos de pontos de junção sejam modelados diretamente, através de um elemento gráfico próprio nos diagramas.

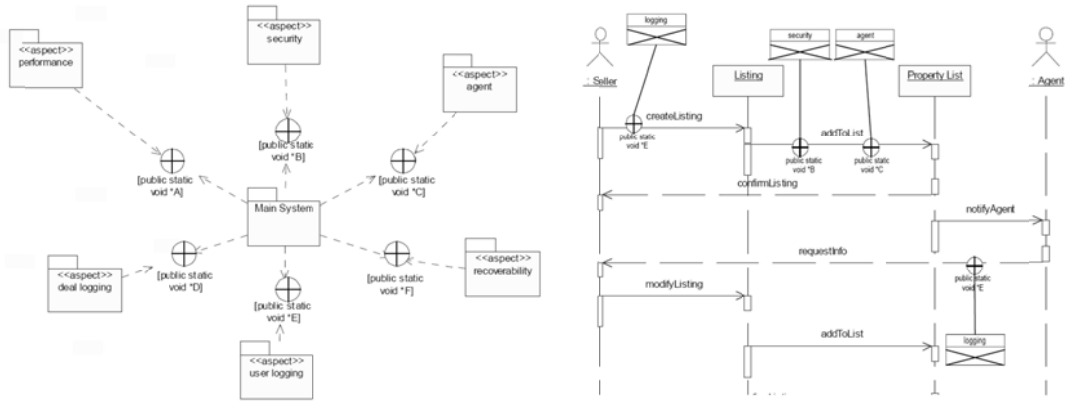


Figura 2.12: Pontos de junção em pacotes de aspectos e diagrama de seqüência

Esse mesmo elemento pode ser usado em diagramas de pacotes, classes, seqüência, estado e atividades. Combinados, eles seriam suficientes para mapear qualquer comportamento desejado de AspectJ.

### 2.4.10 Aspect-Orientation from Design to Code (GROHER 2004)

Este modelo usa apenas UML padrão, e apresenta como poderia ser feito seu mapeamento para linguagens de programação. É apresentado como um de seus objetivos o suporte a várias linguagens de programação, e a possibilidade de criar mapeamentos simples entre o modelo e essas linguagens. Essa capacidade é demonstrada, no entanto, apenas sobre AspectJ.

A proposta separa o sistema em no mínimo 3 pacotes: o pacote "base" , que contém a implementação dos requisitos funcionais do sistema; um ou mais pacotes de aspectos; e um ou mais pacotes de conexão, que indicam os mapeamentos entre os aspectos e as classes.

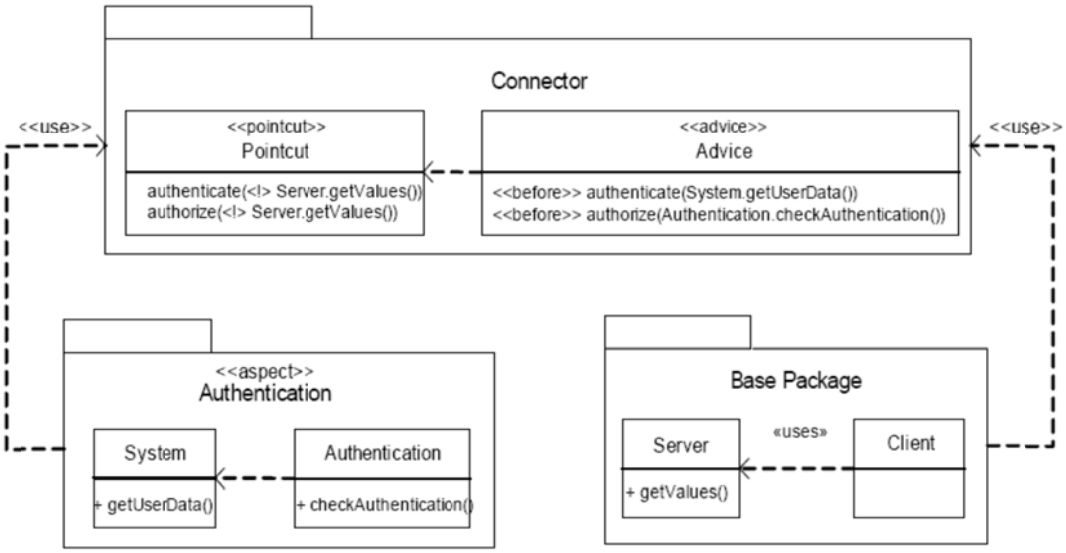


Figura 2.13: Modelo UML de (GROHER 2004)

O artigo também possui uma seção sobre a utilização de XMI para gerar código orientado a aspectos a partir dos modelos, mas a execução da conversão se encontra relacionada como trabalho futuro.

### 2.4.11 Theme (CLARKE 2004)

O trabalho de Clarke sobre padrões de composição (*composition patterns* ou CP) é o mais fundamentado conceitualmente, sendo desenvolvido continuamente desde 1998 (RASHID 2005). Apesar de ter sido iniciado focado em *subject-oriented programming*, ele evoluiu para se tornar o mais genérico e independente de implementação possível, e ao mesmo tempo compatível com diversas formas de implementação de separação de interesses (CLARKE 2001b, CLARKE 2002), apesar de críticas de que a forma utilizada para expressar suas integrações não seria suficiente para modelar todas as características disponibilizadas por AspectJ (STEIN 2002).

Devido a isso, a abordagem Theme, e em particular Theme/UML, que é a parte que trata especificamente da modelagem, foi escolhida para servir de base para este trabalho. Ela será tratada em detalhes no próximo capítulo.

## 2.5 Theme

A abordagem Theme propõe que um sistema pode ser dividido em *temas*, representando cada um dos seus interesses. Temas podem ser básicos (*base themes*) ou transversais (*aspect themes*). Um sistema pode conter diversos *base themes* e *aspect themes*. Por exemplo, uma aplicação de compartilhamento de arquivos poderia ter comunicações e armazenamento como temas básicos, e segurança e rastreamento como temas transversais.

Ela se divide em duas partes: Theme/Doc, para a engenharia de requisitos, e Theme/UML, para o projeto de sistemas, ambas considerando sempre a separação de interesses em todos os níveis. Apesar do interesse para este trabalho ser nessa última, serão apresentados rapidamente alguns conceitos referentes à primeira.

### 2.5.1 Theme/Doc

Theme/Doc é a parte de Theme que se preocupa com a detecção de interesses transversais durante a fase de análise de requisitos do sistema. Basicamente, ela define técnicas para classificar os requisitos em básicos ou transversais de acordo com os interesses do sistema aos quais eles estão relacionados.

Um interesse transversal é disparado a partir de outro tema. Além disso, ele também normalmente é disparado em várias situações diferentes, e sua descrição se encontra irreparavelmente entrelaçada com a de outros interesses. A identificação de temas consiste em identificar a quais interesses cada

requisito da aplicação está associado, de forma que um requisito não se encontre associado a mais de um interesse.

Para os requisitos que não possam ser associados a apenas um interesse, se faz necessária uma análise adicional. Ele será um aspecto se ele atender às seguintes condições:

- O requisito não pode ser dividido em partes independentes; caso contrário, ele não indica comportamento entrelaçado, devendo apenas ser dividido.
- Dentre os interesses associados ao requisito, existe um que predomina sobre os outros, e é transversal aos outros interesses associados ao requisito.
- O tema dominante é acionado pelos outros temas descritos no requisito.
- O tema dominante é acionado em situações diversas.

Depois de finalizada a etapa de análise, os *themes* detectados são modelados usando Theme/UML.

### 2.5.2 Theme/UML

A construção adicionada à UML pelo Theme é o *composition pattern* (ou *theme*), que é semelhante ao conceito de *templates* em UML 2.0, mas aplicados a pacotes. Os parâmetros de CPs podem incluir operações que podem ter seu comportamento modificado quando o CP for composto com classes.

Nos diagramas de Theme/UML, *themes* são representados como pacotes na UML tradicional, e marcados com o estereótipo «*theme*» (ou «*subject*», dependendo da versão de Theme utilizada). A diferença é que alguns *themes* podem possuir parâmetros, como se fossem *templates* da UML 2.0. Esses parâmetros serão usados para indicar quais elementos contidos no pacote podem ser ligados a elementos de outros pacotes, e dessa forma tem uma semântica semelhante ao de *pointcuts*. Os *themes* que possuem parâmetros são chamados *aspect themes*.

A ligação dos *aspect themes* com os *base themes* – aqueles que não possuem parâmetros, e portanto podem existir independentemente de outros *themes*, se dá através de *bindings*. Estes são representados através de dependências entre pacotes, com uma anotação *bind*, indicando, em ordem, os elementos contidos no *base theme* que se relacionam com cada parâmetro do *aspect theme*.

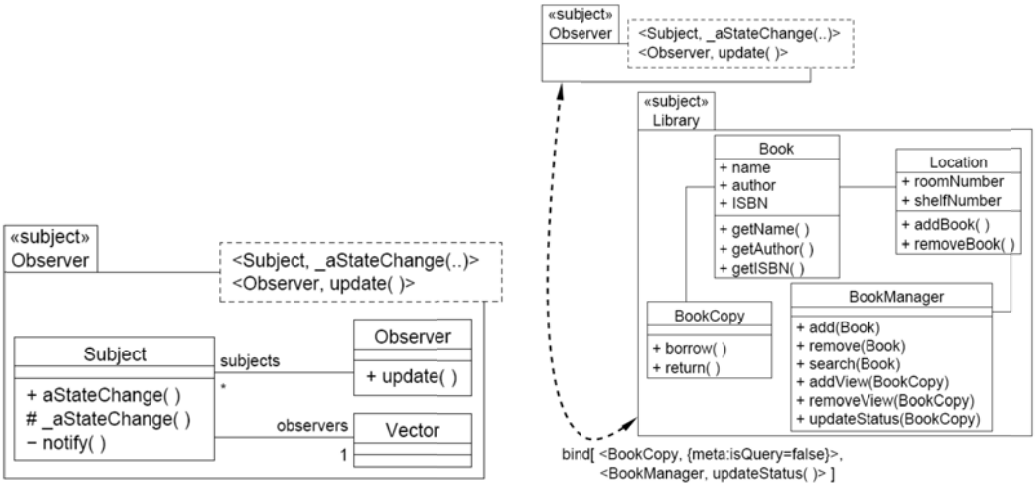


Figura 2.14: Diagrama de classes utilizando *Composition Patterns*

A combinação se dá através da substituição dos parâmetros do CP quando esse é ligado a classes, como visto na Figura 2.14, e a forma como adendos serão integrados é especificada pelos diagramas de interação, como na Figura 2.15. Em (CLARKE 2002), é especificado como esses modelos podem ser transformados em código AspectJ. Esse assunto será retomado na seção 4.2.

O modelo de classes dos *Composition Patterns* é semelhante ao conceito de *templates*, introduzido na versão 2.0 da UML. *Templates* são pedaços de modelos com partes específicas que podem ser substituídas, definidas através de parâmetros. Para serem usados, os *templates* são ligados a elementos concretos do modelo, substituindo seus parâmetros. Esse tipo de funcionalidade já existia em linguagens de programação como C++, e está presente nas últimas versões das linguagens Java e C#.

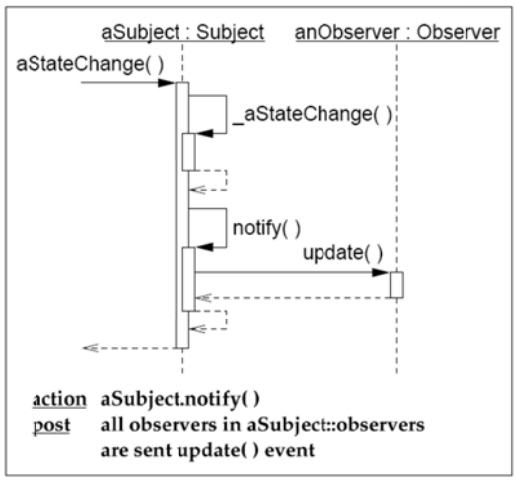


Figura 2.15: Diagrama de seqüência mostrando a interação entre *composition pattern* e classe modificada

O sistema de padrões de composição tem a vantagem de ser a mais genérica entre as propostas apresentadas para a representação de separação de

interesses na modelagem de sistemas. Por exemplo, além da implementação em AspectJ já mencionada, (CLARKE 2001b) demonstra a sua aplicabilidade para modelar as construções da linguagem Hyper/J, baseada na *separação multidimensional de interesses*, uma proposta concorrente à AOP (OSSHER 2001a).

Esta flexibilidade não vem sem um custo, no entanto. As construções de Theme/UML não possuem um paralelo direto com as construções usadas em AspectJ, e boa parte dos processos de implementação demonstrados dependem mais de uma interpretação humana dos modelos, por parte de um programador versado em ambos os domínios, do que de um processo bem-definido, capaz de ser executado automaticamente por um computador. Devido a isso, por razões que serão vistas na seção seguinte, Theme/UML não é um sistema trivial para ser usado como base para a representação de código de um sistema existente.

## 2.6 Refatoração no contexto de Orientação a Aspectos

Refatorações são transformações de código-fonte utilizadas para melhorar o projeto de um software sem modificar o comportamento da aplicação. Elas consistem em um conjunto de passos bem definidos, incluindo testes freqüentes, que garantem que a mudança no trecho específico sendo melhorado não afetará o restante do sistema (FOWLER 2000).

Existem alguns problemas recorrentes que surgem durante o desenvolvimento de software: trechos de código que foram abandonados em um módulo e não estão mais sendo usados, duplicações de código, classes com poucas responsabilidades ou atribuições em demasia (FOWLER 2000). Estes problemas normalmente dificultam o reuso em todas as fases de um processo de desenvolvimento (BIGGERSTAFF 1998) e podem ser minimizados através da identificação de seus sintomas e da remoção das causas destes problemas. Estes sintomas (denominados de *bad smells* por Fowler (FOWLER 2000)) podem ser vistos como sinais ou alertas de que problemas existem no software (ELSSAMADISY 2002), e podem ser corrigidos através da aplicação de refatorações definidas para cada *bad smell*.

As refatorações são verificadas através de um contrato, normalmente especificado através de um conjunto de pré e pós condições, na tentativa de preservar o comportamento observável da aplicação. Por exemplo, sempre que uma refatoração de renomeação de método (*Rename Method*) for executada, deve ser garantido que todas as referências a este método também sejam renomeadas. Normalmente, este processo é auxiliado por ferramentas de refatoração e apoiado por um conjunto de testes unitários, facilitando que os efeitos da refatoração sejam desfeitos caso necessário.

(FOWLER 2000) define diversas refatorações utilizadas em sistemas orientados a objetos. Dentre estas existem refatorações para a renomeação de

classes, atributos e métodos, encapsulamento de escrita e leitura de atributos, movimentação de membros de uma classe entre a hierarquia de classes etc.

No contexto de sistemas orientados a aspectos, também existe a necessidade de refatorações que permitam a manipulação de código na presença de aspectos. Mais especificamente, refatorações que tratem de sistemas orientados a aspectos devem possibilitar mover código implementado em classes para aspectos, manipular código de aspectos para aspectos e mover código de aspectos para classes.

Algumas refatorações foram propostas de forma a possibilitar a manipulação de código na presença de aspectos (vide (GARCIA 2004, HANENBERG 2003, IWAMOTO 2003, MONTEIRO 2004, MEYER 1997)). Estas refatorações auxiliam nas atividades de remoção de bad smells em aspectos. A seguir, na tabela 2.1, são listadas algumas refatorações orientadas a objetos e orientadas a aspectos, juntamente com suas fontes.

**Tabela 2.1: Refatorações para sistemas orientados a aspectos (PIVETA 2005)**

| **Refatoração** | **Descrição** | **Fonte** |
|---|---|---|
| Extract Method | Remove um trecho de código para um novo método | (FOWLER 2000) |
| Combine Pointcut | Mescla os predicados de vários conjuntos de junção | (HO 2002) |
| Pull Up Field | Move um atributo para uma das super-classes da classe atual | (FOWLER 2000) |
| Pull Up Method | Move um método para uma das super-classes da classe atual | (FOWLER 2000) |
| Pull Up Advice | Move um adendo para uma das super-classes ou super-aspectos do aspecto atual | (GARCIA 2004) |
| Pull Up Pointcut | Move um conjunto de junção para uma das superclasses ou super-aspectos do aspecto atual | (GARCIA 2004) |
| Pull Up Inter-Type Declaration | Move uma declaração intertipos para uma das super-classes ou super-aspectos do aspecto atual | (GARCIA 2004) |
| Extract Pointcut | Extrai uma definição de um conjunto de junção de um adendo | (HO 2002) |
| Extract Class | Cria uma nova classe e move atributos e métodos relevantes da classe antiga para a nova classe | (FOWLER 2000) |
| Extract Aspect | Extrai um aspecto a partir de código constante em classes | (MEYER 1997) |
| Extract Sub-Aspect | Cria um sub-aspecto contendo um sub-conjunto de funcionalidades | (PIVETA 2005) |
| Collapse Aspect Hierarchy | Une uma hierarquia de aspectos | (GARCIA 2004) |
| Inline Aspect | Insere o código de aspectos nas classes que ele afeta | (PIVETA 2005) |
| Remove Advice Parameter | Remove um parâmetro de um adendo | (PIVETA 2005) |
| Rename Aspect | Renomeia um aspecto e todas as referências a este | (HANENBERG 2003) |
| Rename Pointcut Definition | Renomeia um conjunto de junção e todas as definições a este | (GARCIA 2004) |
| Move Pointcut | Move um conjunto de junção de uma classe/aspecto para outra classe/aspecto | (PIVETA 2005) |

### 2.6.1 Melhores Práticas de Programação Orientada a Aspectos

O objetivo da final da refatoração, conforme definido por (FOWLER 2000), não é fazer o sistema *funcionar* melhor, pois na verdade toda refatoração parte do princípio de que o sistema original já está funcionando. O objetivo é, na verdade, melhorar o *estilo* do código, ou seja, tornar o código mais compreensível, reutilizável, e manutenível.
Por isso, para definir refatorações no contexto de desenvolvimento orientado a aspectos, é necessário saber qual o estilo desejado para um programa que usa essa tecnologia. Devido à juventude do paradigma, esse estilo ainda não se encontra tão bem definido quanto para a orientação a objetos. Mesmo assim, algumas melhores práticas vêm sendo sugeridas (CHAVEZ 2004, GARCIA 2005).

- *Utilizar aspectos abstratos, com pontos de junção que serão especificados na sua concretização.* O aspecto precisa conhecer apenas seu próprio funcionamento (adendos e introduções), mas não os pontos de junção específicos que serão usados. Isso é atingido com o uso de conjuntos de pontos de junção abstratos, que serão definidos na concretização dos aspectos. Isso melhora a modularidade do sistema, e consequentemente sua reusabilidade. Tanto as implementações de padrões de projeto (GAMMA 1994) utilizando AspectJ (HANENBERG 2003), quanto o algoritmo para a transformação de Theme/UML para AspectJ (CLARKE 2002, CLARKE 2005) utilizam esta técnica.
- *Não usar conjuntos de pontos de junção anônimos.* Pontos de junção associados diretamente a adendos não podem ser declarados como abstratos.
- *Não utilizar o mesmo conjunto de pontos de junção para vários adendos.* Isso garante que os adendos podem ser compostos de forma independente.
- *Usar nomes relevantes para conjuntos de pontos de junção.*
- *Não se basear em convenções de nomes para a definição de conjuntos de pontos de junção; ao invés disso, utilizar o mecanismo de anotações.* Convenções de nomes nem sempre são seguidas corretamente, e mesmo uma mudança na caixa dos prefixos (por exemplo, *DaoProvider* e *DAOClient*) podem dificultar a obtenção dos pontos de junção corretos. Da mesma forma, pontos de junção sem relação com os que se deseja obter podem por acaso atender ao mesmo critério de nomenclatura.
- *Aspectos devem atender a apenas um interesse do sistema.* Aspectos servem para representar interesses independentes da aplicação, e seria um contra-senso que dentro deles houvesse os mesmos problemas de espalhamento e entrelaçamento que eles se propõe a resolver. Por isso, aspectos que tentam solucionar mais de um interesse devem ser

divididos. Theme, tanto na parte de análise quanto na de modelagem, tenta garantir que os requisitos do sistema sejam bem distribuídos dentro dos aspectos.

- *Aspectos concretos devem ser sempre vazios.* Aspectos concretos não podem sofrer herança em AspectJ. Por isso, a concretização de aspectos deve sempre ser o último passo antes da utilização do aspecto.

## 2.7 Detecção de *Bad Smells*

*Bad smells* são propostos por Fowler (FOWLER 2000) de forma a possibilitar a identificação de problemas em artefatos de software pré-existentes. Isto é feito através da sugestão de possíveis sintomas que podem aparecer nestes artefatos, indicando áreas que normalmente podem ser melhoradas, através de refatorações. O uso de refatorações possibilita que estes sintomas sejam minimizados e excluídos, atacando as causas dos problemas.

O artigo (PIVETA 2005) descreve uma série de *bad smells* que podem ser detectados especificamente em sistemas orientados a aspectos, e alguns outros que foram inicialmente detectados em sistemas orientados a objetos, mas ganham um novo significado no contexto de separação de interesses. A seguir, serão listados alguns desses *bad smells*, para oferecer uma idéia dos tipos de análises que podem ser realizadas automaticamente. Mais detalhes sobre esses *bad smells*, incluindo exemplos de ocorrência e sugestões de refatorações para

**Tabela 2.2: Bad Smells e Refatorações associadas (PIVETA 2005)**

| **Bad Smell** | **Refatoração** |
|---|---|
| Duplicação de Código | Extract Method<br>Combine Pointcut<br>Pull Up Field<br>Pull Up Method<br>Pull Up Advice<br>Pull Up Pointcut<br>Pull Up Inter-Type Declaration |
| Mudanças Divergentes | Extract Pointcut |
| Definição Anônima de Conjunto de Junção | Extract Pointcut |
| Aspecto Extenso | Extract Class<br>Extract Aspect<br>Extract Sub-Aspect |
| Definição de Conjunto de Junção Extensa | Extract Pointcut |
| Aspecto com Poucas Responsabilidades | Collapse Aspect Hierarchy<br>Inline Aspect |
| Generalidade Especulativa | Remove Advice Parameter<br>Collapse Aspect Hierarchy<br>Inline Aspect<br>Rename Aspect<br>Rename Pointcut Definition |
| Feature Envy | Move Pointcut |
| Introdução de Método Abstrato | – |

sua remoção, podem ser vistas no artigo original.

Os bad smells apresentados são sumarizados na Tabela 2.2: .

### 2.7.1 Duplicação de Código

Diminuir a quantidade de código duplicado é uma das motivações do desenvolvimento de software orientado a aspectos. Ao prover mecanismos de abstração para a modularização de interesses multi-dimensionais, a tendência é que a duplicação seja reduzida, dado que interesses antes espalhados ao longo das abstrações de uma aplicação podem ser encapsulados em um único aspecto ou em um conjunto pequeno de aspectos. Dessa forma, refatorações que transferem responsabilidades do sistema para dentro de aspectos devem melhorar esse problema.

Mesmo assim, durante o desenvolvimento dos aspectos, pode ocorrer de permanecer código duplicado, ou mesmo de o problema ser criado novamente. Refatorações específicas para aspectos, como *Pull Up Advice* e *Pull Up Pointcut* (GARCIA 2004) podem oferecer uma solução.

### 2.7.2 Mudanças Divergentes

Outro *bad smell* que pode ser identificado ocorre quando a definição de vários conjuntos de junção é praticamente igual, só diferindo pelos seus modificadores ou pequenas partes do predicado. Toda vez que um conjunto de junção é modificado, o mesmo deve ser feito em todos os outros. O uso de uma refatoração de extração (*Extract Pointcut* (IWAMOTO 2003)) resolveria o problema, definindo semântica para o trecho do conjunto de junção que se repete ao longo dos conjuntos.

### 2.7.3 Definição Anônima de Conjunto de Junção

Como adendos não possuem nomes, às vezes é necessário recorrer à descrição contida no conjunto de junção para obter uma idéia dos pontos afetados pelo adendo. O uso da definição do conjunto de junção diretamente no adendo pode diminuir a legibilidade e ocultar a intenção deste.

Para definir claramente a intenção de um conjunto de junção, deve ser definido um nome para o conjunto e este utilizado em quaisquer adendos que afetem os pontos.

A refatoração denominada *Extract Pointcut* (IWAMOTO 2003) pode ser utilizada para extrair definições de conjuntos de junção declaradas diretamente no adendo.

### 2.7.4 Aspecto Extenso

Quando um aspecto tenta lidar com mais de um interesse, este deve ser dividido em tantos aspectos quantos houver interesses. Normalmente isto

ocorre com a definição de adendos com propósitos diferentes ou outros tipos de estruturas, como atributos, declarações inter-tipo etc.

Se o aspecto possui membros que correspondem a interesses diferentes do interesse principal sendo tratado no aspecto, e estes possam ser extraídos para uma classe, deve ser utilizada a refatoração *Extract Class* (FOWLER 2000). Caso estes membros sejam estruturas exclusivas de aspectos, deve ser aplicada a refatoração *Extract Aspect* (MONTEIRO 2004). Quando os diferentes interesses podem ser separados através de herança, dada a afinidade dos interesses tratados, pode ser utilizada a refatoração *Extract Sub-Aspect* (PIVETA 2005).

### 2.7.5 Aspecto com poucas responsabilidades

Este *bad smell*, inicialmente definido em (MONTEIRO 2005) e estendido em (PIVETA 2005), ocorre quando um aspecto tem poucas responsabilidades e sua eliminação poderia proporcionar benefícios na etapa de manutenção. Muitas vezes esta diminuição de responsabilidade ocorre depois de uma refatoração, ou de uma modificação em função de alterações que foram planejadas/previstas, mas que não ocorreram. À medida que refatorações são efetuadas, certas classes e/ou aspectos podem perder responsabilidades.

Caso as responsabilidades de um aspecto não sejam suficientes para justificar sua existência, a refatoração *Collapse Aspect Hierarchy* (GARCIA 2004) pode ser utilizada.

Esta refatoração visa reduzir a árvore de hierarquia, movendo membros de um aspecto para suas sub-classes ou das sub-classes para o super-aspecto. Aspectos vazios podem ser removidos com o uso de *Inline Aspect* (PIVETA 2005).

### 2.7.6 Generalidade Especulativa

Algumas vezes classes e aspectos são criados para lidar com requisitos futuros. Como aspectos tornam mais simples postergar algumas decisões do projeto, é possível remover as funcionalidades que não são usadas de alguma maneira.

De forma a remover parâmetros que não estão sendo usados no adendo, usar a refatoração *Remove Advice Parameter* (PIVETA 2005). Para remover aspectos que não estão sendo usados, usar as refatorações *Collapse Aspect Hierarchy* (GARCIA 2004) e *Inline Aspect* (PIVETA 2005). Aspectos e conjuntos de junção com nomes genéricos podem ser renomeados para a semântica atual usando as refatorações *Rename Aspect* (HANENBERG 2003) ou *Rename Pointcut* (GARCIA 2004).

### 2.7.7 Feature Envy

Em AspectJ é permitido usar conjuntos de junção em classes. Caso este conjunto de junção seja utilizado por apenas um aspecto, é interessante que este seja movido para o aspecto que o utiliza. Este mesmo problema pode ocorrer em classes, em situações tais que um método de uma classe faz referência aos atributos e métodos de outra classe ao invés de referenciar os membros da classe que o contém.

Uma refatoração de movimentação pode ser utilizada, de forma a movimentar o conjunto de junção da classe para o aspecto. Esta refatoração, denominada de *Move Pointcut* (HANENBERG 2003), pode ser utilizada também para mover conjuntos de junção entre aspectos.

### 2.7.8 Introdução de Método Abstrato

**Aspectos podem ser utilizados de forma a adicionar estado e** comportamento a classes existentes. Isto é feito através de um mecanismo chamado de declaração intertipos (intertype declaration). Este mecanismo permite que métodos e/ou atributos sejam inseridos nas classes afetadas pelo aspecto. Entretanto, o uso desta funcionalidade pode causar problemas caso sejam inseridos métodos abstratos nas classes da aplicação.

Esta introdução força o desenvolvedor a prover implementações concretas para os métodos introduzidos em todas as classes e/ou sub-classes afetadas, aumentando de forma desnecessária o acoplamento entre o aspecto e as classes afetadas.

A introdução de métodos abstratos através de uma declaração inter-tipos (*intertype declaration*) deve ser evitada, pois implica que cada vez que uma sub-classe de uma classe afetada é criada, implementações para os métodos abstratos inseridos pelo aspecto devem ser providas.

## 2.8 Geração de Código

Geração de código é o processo de transformação de artefatos em um nível mais alto (mais próximo do domínio do problema) em outros de nível mais baixo (mais próximo da arquitetura da solução) (KRUEGER 1992). Os exemplos mais comuns de geração são:

- Conversão de código em uma linguagem de alto nível em código executável binário (compiladores);
- Combinação e modificação de componentes reutilizáveis através de uma linguagem de especificação própria (o conceito de *generative programming* (BIGGERSTAFF 1998));

- Transformação de uma especificação (na forma de modelos de alto nível, por exemplo, UML) em código (de alto nível ou binário), evitando etapas custosas de programação manual.

Nesse ponto de vista, o próprio combinador de aspectos (*aspect weaver*) pode ser considerado um gerador de código. Por exemplo, o combinador de AspectJ leva de uma especificação de nível mais alto – o código definindo classes e aspectos – a um código de nível mais baixo – código Java padrão, com aspectos e classes combinados. Contudo, nesta seção iremos considerar apenas o último tipo de geração de código: a transformação de modelo em código.

O gerador de código aqui descrito foi desenvolvido para testar o modelo, e verificar sua capacidade de representar construções de AspectJ fielmente. Como geradores de código exigem modelos bem-formados e livres de ambigüidades (conforme será visto na seção 2.8.1), eles são excelentes testes para modelos de software. Seu objetivo é gerar estruturas de classes e aspectos, não chegando ao ponto de gerar o corpo dos métodos (mesmo quando eles se encontram descritos em diagramas de seqüência). Ele é explicado com mais detalhes em (HECHT 2006). Nesse artigo, contudo, foi usada uma versão mais antiga do modelo proposto neste trabalho.

### 2.8.1 Requisitos da geração de código

Existem alguns requisitos para se construir um gerador de código. Em primeiro lugar, é necessário como ponto de partida um modelo do sistema desejado, definido sobre um metamodelo suficientemente detalhado e livre de ambigüidades para fornecer a estrutura que deve ser gerada. Também é preciso uma especificação igualmente detalhada de quais elementos serão compostos para formar o resultado final, e regras (baseadas no conteúdo do metamodelo) que especificam como será feita essa composição (ver Figura 2.16).

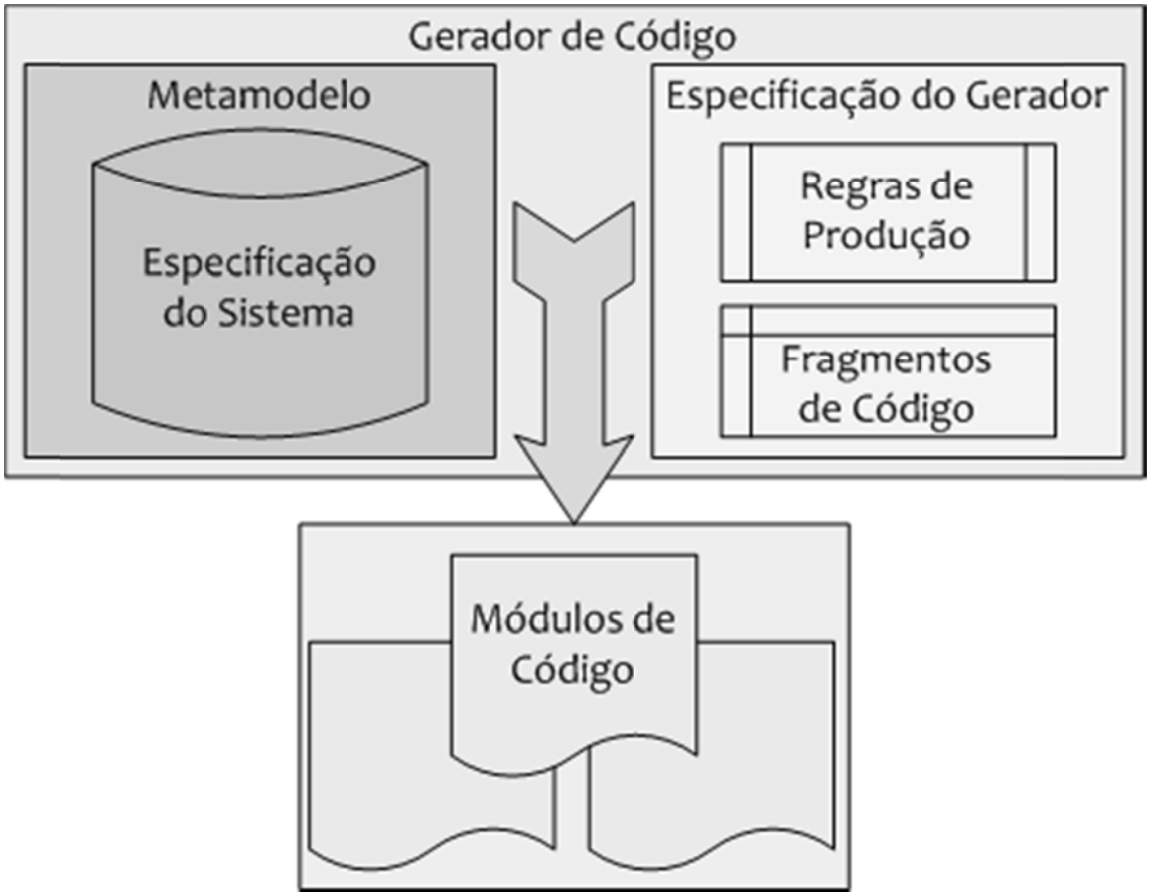


Figura 2.16: Estrutura típica de um gerador de código.

O metamodelo é um modelo de dados capaz de armazenar outros modelos de sistemas de forma compreensível pelo gerador. Ele precisa ser específico para o objetivo do gerador: por exemplo, o metamodelo usado por um gerador de interfaces precisa conter informações sobre telas, campos e mensagens, enquanto que um metamodelo de programas orientados a objetos contém especificações de pacotes, classes, propriedades e métodos (como na Figura 2.17).

Figura 2.17: Metamodelo para especificação de uma estrutura de classes.

### 2.8.2 Especificação do Gerador

A especificação de um gerador de código é dividia em duas partes principais: as *regras de produção* e os *fragmentos de código*.

Fragmentos de código são trechos de programas, na sintaxe da linguagem de saída do gerador, que são montados para gerar os arquivos de código fonte do programa final. Esses fragmentos serão intercalados com algumas informações contidas no metamodelo (como nomes de elementos). Por exemplo, em um gerador de classes Java, haveria fragmentos de código representando a declaração de classes, campos e métodos. Nesses fragmentos, haveria a previsão de onde deveriam ser colocadas as informações dos nomes específicos das classes e dos tipos de métodos.

As regras de produção determinam como a estrutura definida no metamodelo será convertida, nos arquivos de saída, em uma estrutura formada pelos fragmentos de código. Ela é composta principalmente de elementos alternativos (decidindo se um ou outro fragmento será usado), iterativos (repetindo um fragmento) e recursivos (criando estruturas reentrantes). No exemplo do gerador Java, elementos alternativos seriam usados na escolha do tipo de dados de métodos e atributos, iterativos na geração de atributos e métodos da classe, e recursivos na geração de classes aninhadas. Pode-se notar que são exigidas das regras de produção funcionalidades semelhantes às de uma linguagem de programação procedural.

Geradores de código mais simples (aqueles desenvolvidos para uso pessoal ou de uma equipe pequena, para solucionar um problema pontual, como a criação de classes de acesso a um banco de dados) em geral não possuem uma distinção tão clara entre as duas partes: fragmentos de código são strings embrenhadas no meio das regras de produção, que são feitas em uma linguagem de programação tradicional. Geradores mais avançados possuem uma distinção mais clara, e muitas vezes possuem uma linguagem própria para a especificação das regras de produção.

Existem diversos geradores de código comerciais disponíveis, em geral associados a ferramentas de modelagem (RATIONAL 2006, ARGOUML 2006, POSEIDON 2006). Esses geradores normalmente possuem uma estrutura fixa ou possibilitam modificações limitadas. Também existem frameworks para a criação de geradores de código (CODEGEN 2006, VELOCITY 2006, AUTOGEN 2006, CODESMITH 2006). Praticamente todos combinam uma linguagem de script para a programação das regras com algum tipo de armazenamento para fragmentos de código. Uma outra possibilidade é a criação de bibliotecas de classes para auxiliar a programação direta de geradores de código usando linguagens de programação tradicionais.

Uma tendência que vem surgindo, com a popularização da metalinguagem XML de intercâmbio de dados, é o uso dela tanto para definir

fragmentos de código como para a especificação de modelos, e o uso de programas XSLT (uma linguagem especializada na transformação de documentos em XML) como regras de produção (DODDS 2003).

### 2.8.3 Geração de Código Orientado a Aspectos

Nos últimos anos, alguns trabalhos vêm tentando investigar as colaborações possíveis entre desenvolvimento de software orientado a aspectos e geração de código.

Em (AMAYA 2005), é apresentada uma abordagem para lidar com modelagem orientada a aspectos e MDA (MDA 2006). Nessa abordagem, interesses transversais típicos são considerados diferentes perspectivas do mesmo sistema, modelado através de UML. Essas perspectivas serão mantidas desde o modelo independente de computador (CIM) até o modelo específico para plataforma (PIM). Também é proposto o emprego de casos de uso para a análise de requisitos, e o de padrões de composição para o projeto do sistema.

A abordagem Libra (CHAVES 2004) descreve os benefícios potenciais na combinação entre o desenvolvimento de sistemas guiado por modelos (Model-Driven Development - MDD) e o desenvolvimento orientado a aspectos. Os autores propõem o uso de um novo modelo de pontos de junção dinâmicos, representando a semântica de ações na UML.

(KULESZA 2004) fornecem uma versão preliminar de uma forma de geração de código orientado a aspectos. Ela inclui uma descrição das adaptações necessárias no método de engenharia de domínios para acomodar tecnologias orientadas a aspectos.

(GONZÁLEZ 2005) discute princípios para a análise de software orientado a aspectos no contexto da MDA, através da identificação das dependências entre as propriedades do sistema. Não é incluída nenhuma tentativa de implementar esses princípios.

Finalmente, Cam/DAOP (PINTO 2005) busca combinar a engenharia de software baseada em componentes com conceitos de orientação a aspectos, com o uso de uma linguagem de descrição de arquitetura (ADL - Architecture Description Language) baseada em XML. Essa linguagem, chamada DAOP-ADL, contempla a composição de aspectos.

# 3 USANDO THEME/UML PARA A MODELAGEM DE PROGRAMAS ASPECTJ

O interesse principal durante o desenvolvimento deste trabalho era investigar análises e conversões possíveis de ser feitas utilizando modelos e linguagens de programação orientadas a aspectos. Entretanto, as técnicas de modelagem orientada a aspectos ainda se encontra muito atrás das linguagens de programação, que tem recebido muito mais interesse. Assim sendo, fez-se necessário desenvolver uma linguagem de modelagem para software orientado a aspectos, capaz de atender melhor as necessidades definidas.

Mesmo assim, dentre as propostas existentes para a modelagem orientada a aspectos, existe uma mais difundida e pesquisada do que as outras: a Theme (CLARKE 2005). Partindo do respeitado princípio do reuso de artefatos – o objetivo fundamental da engenharia de software (MEYER 1987) – decidiu-se não tentar criar desde o início uma técnica de modelagem, correndo os riscos de encontrar os mesmos problemas existentes nas outras linguagens. Ao invés disso, desenvolveu-se uma técnica baseada em Theme, com adaptações e extensões para torná-la mais apropriada aos objetivos desejados.

## 3.1 Deficiências da Theme/UML

Apesar de toda a sua maturidade, a Theme/UML apresenta algumas deficiências, em particular na relação dos modelos representados nela com a sua implementação em linguagens de programação orientadas a aspectos. A seguir são listadas algumas dessas deficiências, e nas seções seguintes são apresentadas as soluções encontradas para elas.

### 3.1.1 Descasamento entre as linguagens de modelagem e de implementação

Theme/UML oferece mecanismos para representar apenas dois tipos de junção entre *base themes* e *aspect themes*: a execução de métodos (entrelaçamento dinâmico) e a introdução de elementos (entrelaçamento estático). No entanto, linguagens de programação orientada a aspectos identificam várias outras formas de entrelaçamento.

Por exemplo, AspectJ é capaz de identificar uma série de pontos de junção dinâmicos como acesso a propriedades, instanciação de objetos, e

ocorrência de exceções, que não possuem representação em Theme. Mesmo para a chamada de métodos, AspectJ reconhece dois tipos possíveis: *call*, quando o método é chamado (ainda dentro do método que o chamou) e *execute* (quando ocorre a execução propriamente dita do método chamado). Apenas o segundo pode ser representado em Theme. Outras linguagens de programação orientada a aspectos também possuem construções de pontos de junção semelhantes (ASPECT# 2006, JBOSSAOP 2006).

### 3.1.2 Dificuldade de representação em ferramentas de modelagem UML

Os elementos introduzidos por Theme na UML não estão disponíveis em nenhuma ferramenta de modelagem UML atualmente disponível. Devido a isso, se torna impossível usar essas ferramentas para criar modelos Theme.

### 3.1.3 Dificuldade de compreensão dos modelos por computadores

Os modelos Theme/UML possuem um nível de abstração bastante elevado. Apesar de bastante eficazes para representar interesses transversais em um sistema, não oferecem uma representação totalmente livre de ambigüidades no mecanismo de composição entre *themes*, deixando boa parte da interpretação do modelo como responsabilidade de um leitor humano.

## 3.2 Representando outros tipos de pontos de junção nos *bindings*

Devido ao problema exposto na seção 3.1, existe uma limitação visível para Theme/UML representar fielmente a semântica de programas orientados a aspectos. A solução encontrada para isso foi adicionar formas de representar outros tipos de *pointcuts* primitivos em Theme, e também de permitir a composição deles.

Para representar *pointcuts* primitivos, a sintaxe do *bind* de Theme foi estendida. Por exemplo, ao invés de usar a sintaxe `bind[<*.set*(..)>, <Display.refresh()>]`, se torna possível definir `bind[call<*.set*(..)>, declaration<Display>]`.

Os tipos de *pointcuts* disponíveis estão listados a seguir. Os listados nas seções 3.2.1 a 3.2.6 podem ser úteis mesmo na fase de modelagem do sistema, enquanto que os listados nas seções 3.2.7 a 3.2.10 provavelmente só fazem sentido em uma implementação AspectJ. Mesmo assim, eles foram incluídos para completude. A maioria dos *pointcuts* aceita como parâmetro o nome do elemento sobre o qual ele será aplicado. Quando o nome do parâmetro terminar por *Pattern*, o parâmetro aceita o coringa * na sua definição.

### 3.2.1 *Pointcuts* relativos a métodos

AspectJ oferece um par de *pointcuts* que capturam métodos.

- `call(`*`MethodPattern`*`)`

- `execution(MethodPattern)`

O *pointcut* **call** captura a chamada a um método (antes do fluxo de execução passar para o controle do método chamado), enquanto execution captura o momento em que o controle de fluxo é passado para o método. O conteúdo de *methodpattern* é um nome de método, opcionalmente contendo o coringa *.

A Theme/UML oferece uma relação de composição que captura a execução de métodos. Entretanto, ela não deixa clara a semântica específica do momento em que esta captura é feita, da forma como AspectJ permite. Nos exemplos contidos em (CLARKE 2005), a relação de composição **bind** corresponde, em AspectJ, ao *pointcut* **execution**. Dessa forma, embora essa semântica não esteja bem especificada, assumimos que, em sua forma pura, **bind** captura a execução. Pode-se interpretar, desta forma, **bind** como uma forma abreviada de expressar **bind execution**.

### 3.2.2 *Pointcuts* relativos a campos

O acesso a campos de uma classe (variáveis de instância e de classe) é capturado pelos seguintes *pointcuts*:

- `get(FieldPattern)`
- `set(FieldPattern)`

**get** e **set** dizem respeito, respectivamente, à leitura e à atribuição do(s) campo(s) em questão. É importante não confundir, devido ao nome dos *pointcuts*, o acesso ao campo com o uso de métodos de acesso, os quais devem ser capturados por **execution** e **call**, pertinentes à captura de métodos.

### 3.2.3 *Pointcuts* relativos à criação de objeto

Semelhantes aos *pointcuts* referentes a métodos, estes *pointcuts* se distinguem daqueles por receberem um padrão que se refere a um construtor, não a um método (o padrão não oferece tipo de retorno). As variantes **call** e **execution**, em verdade, somente diferem dos seus respectivos equivalentes referentes a métodos por terem como parâmetro um construtor ao invés de um método. Seu funcionamento é semelhante ao já mencionado.

- `call(ConstructorPattern)`
- `execution(ConstructorPattern)`
- `initialization(ConstructorPattern)`
- `preinitialization(ConstructorPattern)`

Além dos *pointcuts* **call** e **execution**, temos também **initialization** e **preinitialization**, que diferem no momento em que ocorre a captura em relação à chamada ao construtor. Os *pointcuts* ocorrem na seguinte ordem, conforme o objeto da classe em questão é inicializado: **preinitialization**, **initialization**, **call** e **execution**.

### 3.2.4 *Pointcuts* relativos à criação de classe

Além dos *pointcuts* referentes à instanciação de um objeto, é possível capturar a inicialização estática de uma classe (que ocorre uma única vez para a classe, na primeira vez em que ela é carregada). Isto é realizado pelo seguinte *pointcut*:

- `staticinitialization(`*`TypePattern`*`)`

### 3.2.5 *Pointcuts* relativos ao tratamento de exceções

O disparo de exceções é capturado em AspectJ com o seguinte *pointcut*:

- `handler(`*`TypePattern`*`)`

Este *pointcut* captura o momento em que a exceção é tratada. É importante não confundir com o momento em que a exceção é criada. Esse *pointcut* possivelmente estaria relacionado à criação do objeto da classe Exception.

### 3.2.6 *Pointcuts* relativos a adendos

A execução de um adendo também é um *join point* válido para AspectJ. O seguinte *pointcut* o captura:

- `adviceexecution(`*`AdvicePattern`*`)`

Um *pointcut* referente à execução de um adendo é de grande utilidade para garantir, por exemplo, que não ocorra uma seqüência infinita de capturas de *join points*, causando uma recursão. Ele só faz sentido quando usado em um *binding* entre dois *aspect themes*.

### 3.2.7 *Pointcuts* baseados em estado

Dizem respeito a informações dinâmicas (de tempo de execução) a respeito dos tipos dos objetos sobre os quais está se operando.

- `this(`*`Type`* `or` *`Id`*`)`
- `target(`*`Type`* `or` *`Id`*`)`
- `args(`*`Type`* `or` *`Id`* `or ".."`, `...)`

### 3.2.8 *Pointcuts* relativos ao fluxo de controle

Dizem respeito ao fluxo de controle durante a execução do aplicativo. São relativos a outros *pointcuts*, que precisam ser declarados como parâmetros.

- `cflow(`*`Pointcut`*`)`
- `cflowbelow(`*`Pointcut`*`)`

O *pointcut* **cflow** indica todos os métodos chamados a partir do momento em que o *pointcut* informado como parâmetro é atingido, incluindo o próprio método, enquanto **cflowbelow** exclui o próprio método. Por exemplo, a declaração a seguir indica todos os métodos chamados pelo método draw (e

todos os métodos chamados por este método, recursivamente), mas não o método *draw* em si.

```
cflowbelow(call<*.draw(..)>)
```

### 3.2.9 *Pointcuts* relativos ao texto

Dizem respeito ao lugar (léxico) no qual sua estrutura está definida, ou seja, dentro da declaração dos tipos ou métodos.

- `within(TypePattern)`
- `withincode(MethodPattern)`
- `withincode(ConstructorPattern)`

### 3.2.10 *Pointcuts* baseados em expressões

Diz respeito a uma expressão dinâmica que é avaliada em tempo de execução

- `if(BooleanExpression)`

### 3.2.11 Declarações Intertipos (entrelaçamento estático)

Entrelaçamento estático é utilizado extensivamente na literatura de Theme/UML, e é suportado na versão original da linguagem. No entanto, a forma com que ele é apresentado não permite uma definição precisa de como o *crosscutting* deve ocorrer. A princípio, como será visto na seção 4.2, todo método que não inicia um diagrama de seqüência de um aspecto (e que dessa forma representa um *pointcut*) deve ser introduzido na classe modificada. Isso impede a modelagem de estruturas permitidas em linguagens orientadas a aspectos tais como classes e aspectos, que podem possuir seus próprios métodos e armazenar estado.

Por isso, a forma de representar declarações intertipos em Theme foi modificada para explicitar a forma que a composição deve ser realizada. Os métodos e campos que se desejam introduzir devem ser marcados com o estereótipo ***«themeDeclaration»*** (que será visto na seção 3.3.1). Assim, toda classe que possui pelo menos um campo ou método marcado com esse estereótipo se torna candidata a ser introduzida em classes dos *base themes*.

O parâmetro do *aspect theme* que representa essa classe passa a indicar não o método de classe que se deseja introduzir, mas a classe inteira. Isso elimina uma ambigüidade possível no Theme/UML padrão, onde dois métodos da mesma classe podiam figurar como parâmetros separados, e nesse caso não ficaria claro como essa composição deveria ser feita. O *bind* desse parâmetro também agora deve indicar a classe onde a composição deverá ocorrer, e não um método específico da classe. Para indicar o entrelaçamento estático, foi definido um *bind* especial denominado `declaration<TypePattern>`.

### 3.2.12 Representação de Pointcuts Compostos

A composição de pointcuts primitivos em AspectJ se dá com o uso de conetivos lógicos, sendo comum construções da forma:

```
pointcut a():
  (call(* *(..)) && !target(Point)) || this(Display);
```

A mesma estrutura pode ser utilizada na representação da relação de composição em Theme/UML, indicando composições entre pointcuts primitivos, por exemplo, da seguinte forma:

```
bind[<(call(* *(..)) && !target(Point)) || this(Display)>]
```

## 3.3 Modelando Theme/UML usando UML2.0

Mais algumas adaptações se fazem necessárias de forma a tornar Theme compatível com as ferramentas de modelagem que utilizam UML 2.0 pura.

Nos estágios iniciais deste projeto (incluindo o artigo sobre geração de código (HECHT 2006)), a forma utilizada para representar os elementos adicionados por Theme à UML – parâmetros e *bindings* – foi criar *tags* especificamente para esse objetivo, e adicioná-las manualmente a um arquivo XMI gerado por alguma ferramenta de *design* UML. Essa forma, no entanto, não era prática, pois, depois de realizadas essas modificações, os arquivos não podiam mais ser editados na ferramenta.

A solução encontrada foi utilizar uma feature de UML chamada *tagged values*. Cada *tagged value* é um par nome-valor que pode ser associado a qualquer elemento de um modelo UML. O objetivo dessa feature, é justamente oferecer um mecanismo de extensão para a linguagem UML padrão. Este método, apesar de dificultar a interpretação do arquivo por outras ferramentas (como a de geração de código), permite que o arquivo do modelo seja modificado sem problemas. Em conjunto com o uso de estereótipos, essa funcionalidade foi suficiente para representar todas as características de Theme/UML desejadas.

Um efeito colateral benéfico dessas adaptações é que, com o uso de UML padrão para representar modelos Theme, todas as aplicações desenvolvidas sobre esse modelo (que serão vistas nos capítulos seguintes) funcionam automaticamente sobre modelos orientados a objetos tradicionais.

A seguir, serão explicadas as construções que foram utilizadas para representar os conceitos de Theme usando UML 2.0 pura. A ferramenta usada para a modelagem foi a ArgoUML (ARGOUML 2005).

### 3.3.1 *Theme Parameters*

Como explicado na seção 2.5.2, parâmetros de *themes* representam *pointcuts* abstratos (ou seja, representados de forma independente dos *joinpoints*

que eventualmente serão associados a eles). Um *package* que representa um *theme* possui uma série de características.

Em primeiro lugar, não importando se ele é um *base theme* ou um *aspect theme*, ele recebe dentro da ferramenta um estereótipo ***«theme»***. Como no XMI gerado pelo Argo todos os elementos do modelo recebem IDs únicos, que podem mudar de modelo para modelo, esse estereótipo será referenciado pelo seu ID.

Em segundo lugar, apenas para os *aspect themes*, os métodos dentro do *package* recebem estereótipos para indicar o tratamento que devem sofrer na composição. Métodos que representam adendos recebem o estereótipo ***«themeAdvice»***. Métodos que representam pontos de junção recebem o estereótipo ***«themeParameter»***. E métodos e campos que serão introduzidos através de declarações intertipos são associados ao estereótipo ***«themeDeclaration»***.

Por último, é preciso referenciar quais elementos do modelo sofrerão composição através de *binding*. Isso é feito usando *tags* ***themeParameter***, cujo valor associado é o nome do método ou campo que corresponde ao conjunto de pontos de junção ou à introdução representados. A associação é feita pelo nome, no formato *classe.método* para pointcuts ou apenas *classe* para introduções. Isso se deve ao ID único ser criado apenas no momento da gravação do XMI, e, portanto, não estar disponível dentro da ferramenta de UML. Por isso, existe a necessidade de um tratamento extra na interpretação do arquivo, que pode ser facilmente implementado utilizando XSLT.

### 3.3.2 *Bindings*

Os *bindings* de Theme indicam quais *join points* concretos os *pointcuts* abstratos declarados no *theme* devem monitorar. Eles são um relacionamento entre *themes* (entre um *base theme* e um *aspect theme*, ou entre dois *aspect themes*), e, portanto, a estrutura de UML padrão que mais se assemelha a eles são relacionamentos.

Para representar *bindings* em XMI, a representação de relacionamentos foi estendida com a indicação de quais elementos no destino do relacionamento se relacionam com cada parâmetro do theme. O relacionamento em si é associado com o estereótipo ***«themeBinding»*** e recebe *tagged values* de nome ***themeParameterBinding*** com a definição dos *join points* que serão associados a cada *pointcut/theme parameter*.

### 3.3.3 Diagramas de Seqüência

Como visto na seção 2.5.2, diagramas de seqüência são usados em Theme/UML para indicar onde deve ocorrer a combinação entre o adendo e o código principal. Theme/UML não adiciona nenhum elemento aos diagramas

da UML padrão, e assim este elemento não precisa sofrer nenhuma modificação. A única informação obtida desse modelo é se o aspecto deve ser aplicado antes, depois ou ao invés do método sendo modificado. Em AspectJ, isso corresponde aos modificadores *before*, *after* e *around*.

### 3.3.4 Criando modelos em uma ferramenta de modelagem

Para testar o modelo utilizando os conceitos apresentados aqui foi utilizada a ferramenta ArgoUML (ARGOUML 2006), uma ferramenta gratuita e multiplataforma de modelagem UML. Além dessas vantagens, essa ferramenta foi selecionada porque o XMI gerado por ela segue um formato bastante legível, o que será explorado na próxima seção.

## 3.4 Armazenando Theme/UML em XMI

Para que o modelo possa ser utilizado por outras ferramentas, como o analisador automático de *bad smells*, ele precisa estar em um formato que possa ser compreendido por algoritmos de computador. O formato mais natural para que isso ocorra é o XMI (XML Metadata Interchange), um padrão definido pelo OMG para o intercâmbio de metadados de uma MOF (Meta Object Facility) usando XML (XMI 2005). Na prática, o metamodelo mais comum definido usando MOF é a UML.

Embora um dos objetivos dessa especificação fosse definir como deveria ser feito o intercâmbio de metadados entre diversas aplicações, não existe uma especificação do formato preciso em que um modelo UML deva ser armazenado, nem um esquema comum para documentos XMI. Assim, não existe garantia de que o XMI gerado por uma determinada ferramenta de modelagem UML seja compatível com outra ferramenta. O que na prática se encontra são modelos de difícil interpretação, onde a mesma *tag* é usada para representar elementos diferentes, ou que incluem as informações gráficas dos diagramas (a posição e tamanho de elementos na tela) misturadas com as informações sobre o modelo em si.

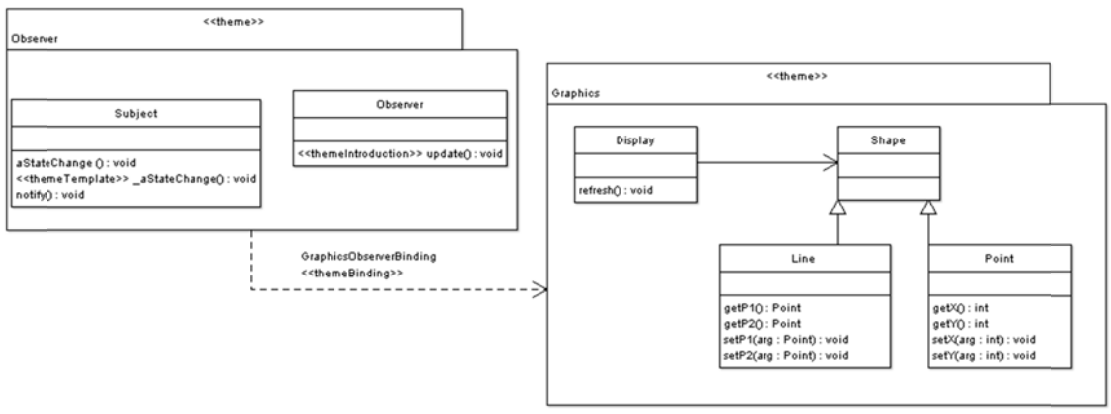


**Figura 3.1:** Modelo Theme/UML usando a ferramenta Argo

Entretanto, o XMI gerado pela ferramenta Argo é bastante adequado. Ele segue um formato bastante legível, com nomes de *tags* que representam claramente seu conteúdo. Além disso, essa ferramenta mantém as informações sobre o modelo em si em arquivos diferentes, separados das informações sobre a sua apresentação gráfica em diagramas, o que facilita ainda mais sua compreensão e manipulação.

A estrutura simplificada de um arquivo XMI gerado pela ferramenta Argo pode ser vista no apêndice A. Essa estrutura já contém os *tags* utilizados para representar os elementos de Theme/UML definidos nesse capítulo.

# 4 REENGENHARIA DE SOFTWARE ORIENTADO A ASPECTOS

A reengenharia de software consiste na análise e alteração de um sistema existente, para reconstruí-lo de forma a apresentar mais qualidade ou novas funcionalidades. Podem-se identificar três tipos de reengenharia, de acordo com o nível de abstração dos artefatos utilizados:

- Refatoração: conforme apresentada na seção 2.6, consiste em reestruturar partes de um sistema, de forma a melhorar seus atributos de qualidade, sem, contudo, modificar seu nível de abstração.
- Engenharia Reversa: processo de analisar o sistema existente, identificado seus componentes e representando-os em um nível mais alto de abstração.
- Engenharia Avante: tradicional processo de desenvolvimento de software. A engenharia avante requer a tradução das abstrações, modelo lógico e projeto para implementação física (codificação).

A existência de um modelo detalhado para software orientado a aspectos fornece uma base para se desenvolver formas de utilizar os conceitos de reengenharia para melhorar o processo de desenvolvimento de software. No caso de engenharia reversa, é possível utilizar as técnicas desenvolvidas para a avaliação e refatoração de modelos em programas orientados a aspectos existentes, extraindo o modelo a partir deles. No caso de engenharia avante, um modelo suficientemente detalhado pode ser usado como base para um gerador de código capaz de saltar etapas custosas na implementação de sistemas. Essas possibilidades são exploradas no restante deste capítulo.

## 4.1 Análise Automática de Código Orientado a Aspectos

A adição de aspectos ao arsenal disponível para o desenvolvimento de sistemas pode ajudar muito em melhorar fatores de qualidade do sistema, como reusabilidade e manutenibilidade. No entanto, ela também cria uma nova dimensão onde podem surgir problemas tradicionais do desenvolvimento de software, como trechos de código duplicados, abandonados, ou estruturas que

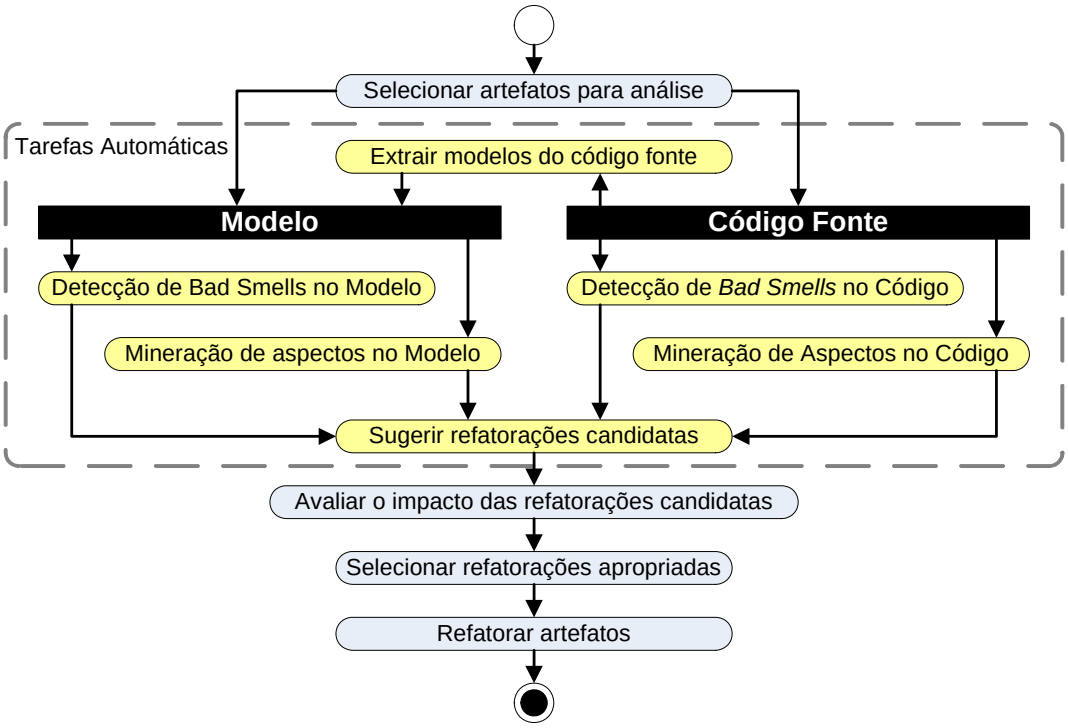


**Figura 4.1:** Processo de análise automática de código orientado a aspectos.

faziam sentido no início do processo de criação do sistema, mas agora funcionariam melhor se fossem compostas de maneira diferente.

Estes problemas são os mesmos que as refatorações de software tentam resolver. Portanto, é natural querer aplicar os conceitos de refatoração ao desenvolvimento orientado a aspectos. Algumas maneiras em que isso pode ser feito já foram descritas na seção 2.6. E, como extensão dessa idéia, tendo disponível o modelo orientado a aspectos descrito no capítulo 3, é possível pensar em maneiras de detectar alguns *bad smells* descritos anteriormente de forma automática, através do processo mostrado na Figura 4.1.

Há, no entanto, alguns empecilhos à implementação prática dessa idéia. Mesmo com as mudanças propostas, o modelo desenvolvido ainda não chega a representar a implementação de métodos. Por isso, alguns *bad smells* (como Duplicação de Código e Aspecto Extenso) são por enquanto impossíveis de ser detectados. Outros, como Definição Anônima de Conjuntos de Junção, simplesmente não podem ser modelados usando a técnica proposta, pois todo conjunto de pontos de junção é determinado por um parâmetro de *theme*.

Isso não impede, mesmo assim, que alguns *bad smells* possam ser detectados facilmente através do modelo, desde que ele seja descrito com detalhamento suficiente. Por exemplo, o *bad smell* Generalidade Especulativa pode ser detectado se forem criados diagramas de seqüência explicitando a funcionalidade dos adendos, e Feature Envy pode ser detectado se as dependências entre as classes do *aspect theme* e do *base theme* forem explicitadas.

Se for usada uma ferramenta de engenharia reversa para criar o modelo a partir de código existente, seria dela a responsabilidade de chegar a esse nível de detalhamento. Isso permitiria o uso do modelo para avaliar código existente.

A seguir apresentamos duas maneiras através das quais os modelos podem ser analisados, e uma forma de refatoração automática.

### 4.1.1 Consultas XQuery

XQuery (XQUERY 2006) é uma linguagem de consulta e seleção de nodos em XML, semelhante ao SQL para bancos de dados relacionais. Alguns *bad smells*, como Mudanças Divergentes e Aspecto com Poucas Responsabilidades, podem ser detectados apenas com uma simples consulta.

A Figura 4.2 mostra um exemplo de consulta XQuery que detecta Mudanças Divergentes.

### 4.1.2 Programas em linguagens imperativas

Apesar de a linguagem XQuery ter sido projetada especificamente para a consulta de documentos XML, e portanto tenha uma série de facilidades nesse sentido, algumas análises sobre o modelo podem exigir funcionalidades não disponíveis nela, como o uso de listas e testes através de expressões regulares.

Para esses testes mais complexos, o uso de recursos de linguagens de programação pode se fazer necessário para realizar a investigação. A maioria das linguagens de programação moderna possui recursos nativos para trabalhar com documentos XML, e dessa forma estão aptas a ler com facilidade os arquivos XMI usados no modelo proposto. Outra vantagem de utilizar linguagens imperativas aparece para programadores que já são proficientes em alguma delas, e não tem interesse em aprender uma nova linguagem.

### 4.1.3 Refatorações usando XSLT

A linguagem XSLT (XSLT 2005) de transformação de documentos XML

```
xquery version "1.0";
declare namespace UML = "org.omg.xmi.namespace.UML";
<repeatedNodes>
{
for $a in (//UML:Dependency/UML:ModelElement.taggedValue/UML:TaggedValue)
for $b in ($a/following-sibling::*)
for $c in ($a/@xmi.id)
where $a/UML:TaggedValue.dataValue/text() =
$b/UML:TaggedValue.dataValue/text()
return
  <repeatedNode>
    <firstNode xmi.idref="{$a/@xmi.id}"/>
    <secondNode xmi.idref="{$b/@xmi.id}"/>
  </repeatedNode>
}
</repeatedNodes>
```

**Figura 4.2:** Consulta XQuery que detecta Mudanças divergentes.

pode ser usada para a refatoração dentro dos modelos armazenados em XMI.

A Figura 4.3 lista uma transformação XSL que é capaz de realizar a refatoração Move Field from Class to Intertype Declaration (MONTEIRO 2004). Embora na listagem os nomes dos elementos estejam fixos, eles poderiam ser passados como parâmetros para a transformação (por exemplo, utilizando a saída de uma consulta de detecção de *bad smells* como a da seção 4.1.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:UML="org.omg.xmi.namespace.UML">
       <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
       <xsl:template match="*">
              <xsl:copy>
                     <xsl:copy-of select="@*"/>
                     <xsl:apply-templates/>
              </xsl:copy>
       </xsl:template>
       <xsl:template
 match="UML:Package[@name='Graphics']/UML:Namespace.ownedElement
/UML:Class[@name='Display']/UML:Classifier.feature
/UML:Attribute[@name='updates']"/>
       <xsl:template
 match="UML:Package[@name='ObserverDP']/UML:Namespace.ownedElement
/UML:Class[@name='Observer']/UML:Classifier.feature">
              <xsl:copy>
                     <xsl:copy-of select="@*"/>
                     <xsl:apply-templates/>
                     <xsl:apply-templates
 select="//UML:Package[@name='Graphics']/UML:Namespace.ownedElement
/UML:Class[@name='Display']/UML:Classifier.feature
/UML:Attribute[@name='updates']" mode="insertAttribute"/>
              </xsl:copy>
       </xsl:template>
       <xsl:template
 match="UML:Package[@name='Graphics']/UML:Namespace.ownedElement
/UML:Class[@name='Display']/UML:Classifier.feature
/UML:Attribute[@name='updates']"
 mode="insertAttribute">
              <xsl:copy>
                     <xsl:copy-of select="@*"/>
                     <xsl:apply-templates/>
                     <xsl:element name="UML:ModelElement.stereotype">
                            <xsl:element name="UML:Stereotype">
                                   <xsl:attribute name="xmi.idref">
<xsl:value-of select="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement
/UML:Stereotype[@name='themeDeclaration']/@xmi.id"/></xsl:attribute>
                            </xsl:element>
                     </xsl:element>
              </xsl:copy>
       </xsl:template>
</xsl:stylesheet>
```

**Figura 4.3:** Transformação XSLT capaz de executar a refatoração Move Field from Class to Intertype Declaration.

## 4.2 Geração de Código Orientado a Aspectos

O gerador aqui descrito foi implementado usando XSLT (Extensible Stylesheet Language Transformations), versão 2.0 (XSLT 2005). A escolha dessa linguagem se deu por ser uma linguagem padrão, com extenso suporte de ferramentas, e possuir a maior parte dos requisitos de um gerador de código.

É bom ressaltar, no entanto, que o uso de XSLT para geração de código só é viável para a criação de *geradores passivos* (BELL 2006), aqueles onde uma nova geração do mesmo código acarreta na perda de qualquer modificação feita manualmente na versão anterior do código. Se houver a necessidade de manter alterações realizadas no código gerado e ao mesmo tempo permitir regerações, é necessário considerar outras alternativas. Existe, mesmo assim, a possibilidade de utilizar as próprias facilidades da programação orientada a aspectos para a alteração do comportamento das classes geradas, sem modificá-las diretamente.

Uma explicação detalhada sobre a sintaxe da linguagem XSLT foge do escopo desse trabalho. Mesmo assim, é interessante ressaltar um recurso, adicionado pelo XSLT versão 2.0, que simplifica bastante sua utilização como gerador de código: a diretiva `xsl:result-document`, que permite que diversos documentos sejam criados pelo mesmo arquivo XSLT. Sem ela, a geração de um arquivo para cada classe ou aspecto gerado (um requisito da linguagem Java) seria dificultada. Além disso, diversas outras características (diretivas para importar outros XSLs, permitindo uma melhor modularização; funções para tratamento de texto, permitindo formatação de strings; possibilidade de leitura de constantes de texto a partir de outros documentos XML) tornam a linguagem XSLT 2.0 uma candidata viável para a criação de geradores de código.

- Cada composition pattern CP se torna um aspecto abstrato A;
- Para cada classe em CP:
  - Declarar uma interface I dentro de A;
  - Para cada operação template que não implementa comportamento suplementar, declarar um método abstrato correspondente em I;
  - Para cada operação template com comportamento suplementar:
    - Declarar um conjunto de pontos de junção abstrato, com parâmetros formais para a captura do objeto alvo (do tipo I), mais um para cada parâmetro formal especificado na operação;
    - Declarar um adendo sobre esse ponto de junção, conforme o comportamento suplementar definido;
  - Para cada operação não-template, introduzir (usando entrecortamento estático) um método que a implemente sobre a interface I
  - Para cada associação não-template da classe para outra classe fora do CP, introduzir um campo (usando entrecortamento estático) em I;
- Para cada classe fora do CP, implementá-la diretamente se ainda não existir.

**Figura 4.4:** Algoritmo para a transformação de um modelo Theme/UML em código AspectJ (CLARKE 2002)

- Cada *package* T que esteja marcado com o estereótipo ***«theme»*** e possua *tagged values* ***themeParameter*** se torna um aspecto abstrato A;
- Para cada classe em T:
  - Declarar uma interface I dentro de A;
  - Para cada operação que esteja marcada com o estereótipo ***«themeAdvice»***, esteja referenciada nos *tagged values* ***themeParameter*** do pacote, e possua um diagrama de seqüência referenciando uma operação marcada com o estereótipo ***«themeParameter»*** :
    - Declarar em A um conjunto de pontos de junção abstrato, com parâmetros formais para a captura do objeto alvo (do tipo I), mais um para cada parâmetro formal especificado na operação;
    - Declarar em A um adendo sobre esse conjunto de pontos de junção, conforme o comportamento suplementar definido no diagrama de seqüência.
  - Para cada associação da classe para outra classe fora do CP, introduzir um campo (usando entrecortamento estático) em I;
  - Se a classe está referenciada apenas pelo nome nos *tagged values* ***themeParameter*** de T, para cada operação ou campo marcado com o estereótipo ***«themeDeclaration»***, introduzi-lo (usando entrecortamento estático) na interface I.
- Cada *package* que esteja marcado com o estereótipo ***«theme»*** mas não possua *tagged values* ***themeParameter*** é gerado como um *package* Java comum;
- Cada dependência D entre packages que esteja marcada com o estereótipo ***«themeBinding»*** e possua *tagged values* ***themeParameterBinding*** se torna um aspecto concreto C.

**Figura 4.5:** Algoritmo para a transformação do modelo Theme/UML modificado em código AspectJ

É importante observar, todavia, que os paradigmas empregados na programação usando XSLT, baseados em percorrimento de árvores e laços, são bastante diferentes dos que programadores que usam linguagens imperativas estão habituados. Ela também é uma linguagem difícil de depurar, no caso de a saída do gerador não corresponder ao planejado. Ela também é uma linguagem que só prevê extensões fornecidas pelo processador de transformações, e assim, se uma transformação necessária para o gerador não existir como função já definida na linguagem, não existe alternativa (exceto em implementações fora do padrão de processadores XSLT, por exemplo, (XALAN 2006)). Portanto, apesar da linguagem ser apropriada para a criação de alguns geradores, ela nem sempre será a melhor escolha.

O processo sugerido em (CLARKE 2002) para a transformação de um modelo Theme/UML para a linguagem AspectJ é descrito na Figura 4.4.
Esse algoritmo, apesar de bastante direto para ser executado por um humano, deixa de fora algumas decisões importantes, como o tipo de adendo que deve ser declarado (*before*, *after*, ou *around*), o que só pode ser decidido verificando o diagrama de seqüência definido no CP. Nos casos aonde a "operação *template*" (que irá se tornar um ponto de junção futuramente) não é nem a primeira nem a última no diagrama, a não ser que a implementação completa do adendo esteja definida no diagrama de seqüência, será difícil para um gerador decidir

```
<xsl:template match="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement">
  <xsl:apply-templates select="UML:Package[
    UML:ModelElement.stereotype/UML:Stereotype/@xmi.idref = $themePackageId
    and
    count(UML:ModelElement.taggedValue/UML:TaggedValue/UML:TaggedValue.type
      /UML:TagDefinition[@xmi.idref=$themeParameterId])>0]"
    mode="T2AJAspect"/>
  <xsl:apply-templates select="UML:Dependency[
      UML:ModelElement.stereotype/UML:Stereotype/@xmi.idref
        = $themeBindingId]"
    mode="T2AJBinding"/>
  <xsl:apply-templates select="UML:Package[
    UML:ModelElement.stereotype/UML:Stereotype/@xmi.idref = $themePackageId
    and
    count(UML:ModelElement.taggedValue/UML:TaggedValue/UML:TaggedValue.type
      /UML:TagDefinition[@xmi.idref=$themeParameterId])=0]"
    mode="T2AJClass"/>
</xsl:template>
```

**Figura 4.6:** Seção de uma transformação XSLT que seleciona o tipo de elemento a ser processado.

exatamente como realizar a composição. Além disso, obviamente, o algoritmo não considera as modificações propostas para Theme neste trabalho. Devido a isso, foi desenvolvido um algoritmo próprio para a conversão (Figura 4.5). Trechos da implementação desse algoritmo em XSLT estão mostrados na Figura 4.6 e na Figura 4.7.

```
<xsl:template match="UML:Package" mode="T2AJAspect">
  <xsl:variable name="uri" select="@name"/>
  <xsl:result-document href="{$uri}.aj" format="textFormat">
    <xsl:call-template name="lowercase">
      <xsl:with-param name="name" select="@name"/>
    </xsl:call-template>;
    <xsl:value-of select="@visibility"/> aspect <xsl:call-template
name="capitalize">
      <xsl:with-param name="name" select="@name"/>
    </xsl:call-template> {
    <xsl:apply-templates select="UML:Namespace.ownedElement/UML:Class"
mode="Interface"/>
    }<!---->
  </xsl:result-document>
</xsl:template>

<xsl:template match="UML:Class" mode="T2AJAspect Interface">
  interface I<xsl:value-of select="@name"/> {
  <xsl:apply-templates select="UML:Classifier.feature/UML:Operation"/>
  }
</xsl:template>

<xsl:template match="UML:Operation" mode="T2AJAspect Interface">
  <xsl:if test="substring(@name, 1, 1) != '_' and
not(exists(../UML:Operation[@name = concat('_', current()/@name)]))">
    <xsl:text>
    </xsl:text><xsl:call-template name="methodSig"/>;<!---->
  </xsl:if>

</xsl:template>
```

**Figura 4.7:** Seção de uma transformação XSLT que converte os pacotes de um documento XMI em aspectos AspectJ.

## 4.3 Engenharia Reversa de Código Orientado a Aspectos

A implementação da engenharia reversa de código AspectJ para o modelo aqui proposto foi realizada pelo aluno Marcelo Czembruski, do II-UFRGS, como trabalho de conclusão de curso.

A Figura 4.8 mostra todas as etapas utilizadas pelo protótipo para gerar o código XMI.

- ASTParser: transforma a representação de um código fonte em uma árvore de sintaxe abstrata (AST), que é uma visão estruturada do código.
- Visitor: transforma a representação AST em uma estrutura de dados intermediária.
- Gerador XMI: transforma essa estrutura de dados intermediária em um documento XMI, o qual representa um modelo gráfico baseado na UML.

  Essas etapas são detalhadas a seguir.

### 4.3.1 ASTParser

Existe um parser já pronto, disponibilizado nas AspectJ Development Tools (ECLIPSE 2006) no pacote jdt.core.dom, o que simplificou bastante esta etapa. O parser recebe uma *string* com o código fonte, e retorna uma AST que representa o código AspectJ.

É interessante notar que, como qualquer programa Java válido é também um programa AspectJ válido, é possível utilizar o mesmo ASTParser para geração de ASTs tanto de aspectos quanto de classes.

### 4.3.2 Visitor

O AJDT fornece uma classe, AjASTVisitor, que implementa o padrão de projeto Visitor (GAMMA 1994), e fornece métodos de acesso a qualquer estrutura sintática disponibilizada por AspectJ. A intenção do Visitor é executar uma operação em todos os elementos de uma estrutura de objetos – no caso em questão, na árvore que representa o programa AspectJ.

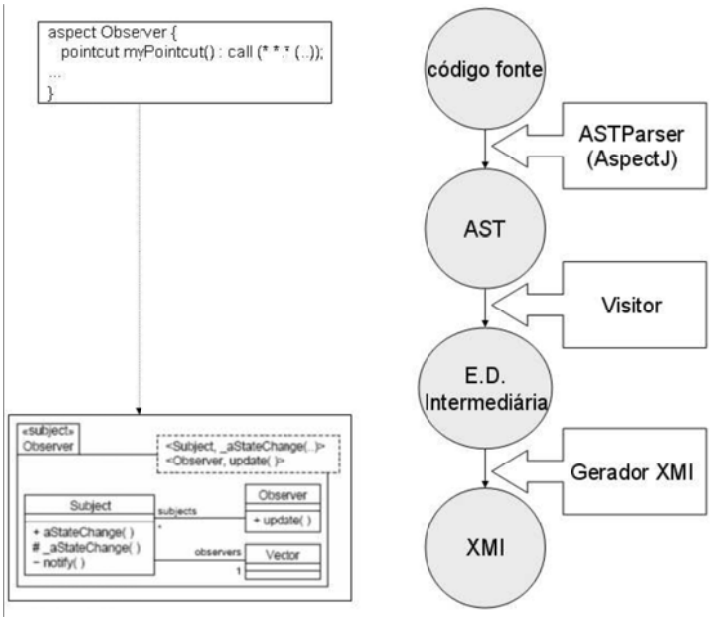


**Figura 4.8:** Visão abstrata da funcionalidade do Protótipo

O protótipo utiliza duas classes (AspectVisitor e ClassVisitor) que estendem a funcionalidade do AjASTVisitor de modo a gerar a estrutura de dados intermediária para classes e aspectos. Essa estrutura de dados intermediária contém classes, com seus métodos, atributos, construtores, etc., e aspectos, que contém conjuntos de junção, adendos, introduções, métodos, atributos, etc.

Nesse ponto, estão identificados todos os adendos, bem como os conjuntos de pontos de junção que os disparam. Estão identificados também todos os pontos de junção das classes. Um elemento denominado Matcher se encarrega de comparar todos os pontos de junção das classes com os conjuntos de pontos de junção definidos nos aspectos, de forma a identificar onde deve ocorrer combinação de comportamento.

### 4.3.3 Geração do XMI

A partir da representação das classes, dos aspectos e os pontos onde eles se combinam, é gerado o documento XMI de acordo com os novos elementos definidos no capítulo 3.

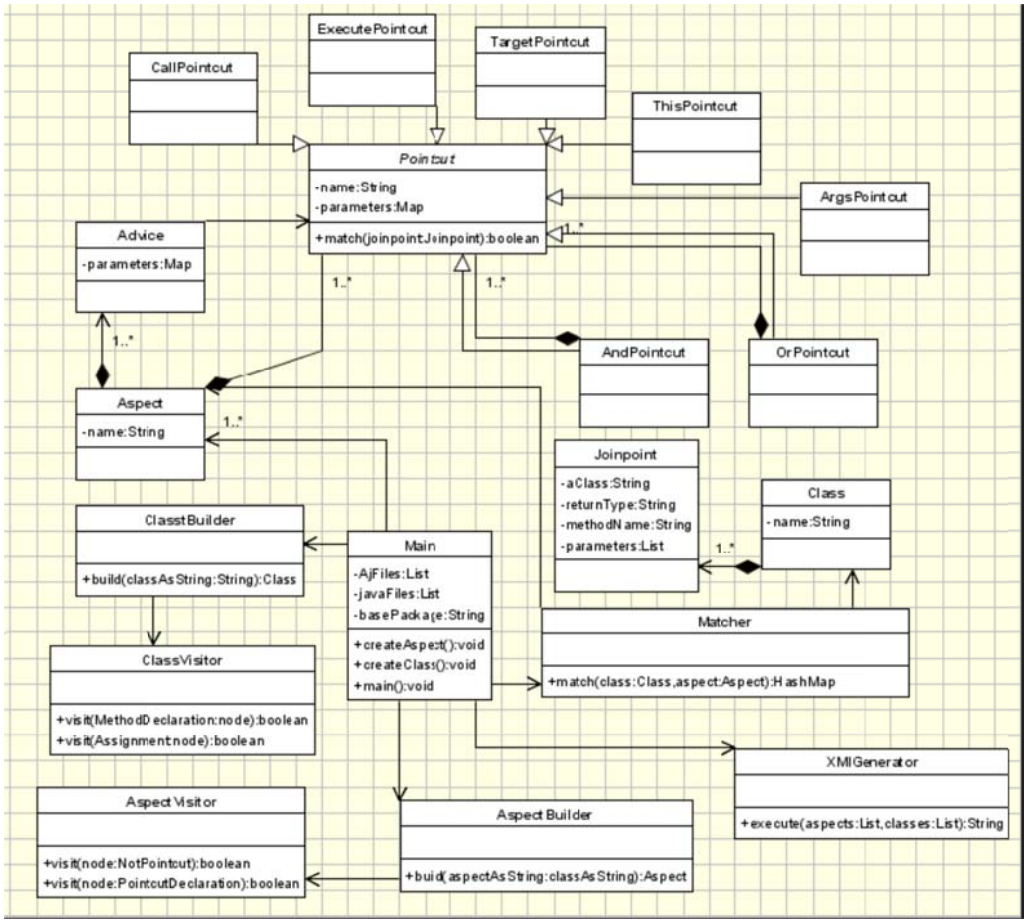


**Figura 4.9:** Diagrama de classes do protótipo

## 4.4 Transformações de ida-e-volta (*round-trip engineering*)

O termo *round-trip engineering* refere-se ao ato de executar em seqüência uma engenharia reversa e uma engenharia avante, ou vice-versa.

Por causa da forma como Theme modela programas AspectJ, nem sempre o sistema gerado a partir da engenharia reversa de um sistema existente é exatamente igual ao sistema original. Embora semanticamente Theme seja capaz de representar boa parte das construções de AspectJ, várias delas não podem ser representadas sintaticamente da mesma forma.

Como um exemplo, AspectJ permite que aspectos concretos sejam associados diretamente a classes, sem o uso da dupla aspecto abstrato/aspecto concreto utilizada em Theme. Contudo, como o uso de conjuntos de pontos de junção concretos associados diretamente a adendos é considerado uma má utilização de programação orientada a aspectos, nesse caso a transformação de ida e volta teria como efeito colateral uma refatoração que melhora o estilo do sistema.

# 5 CONCLUSÃO E TRABALHOS FUTUROS

Desenvolvimento de software orientado a aspectos é uma mudança de paradigma com o potencial de alterar drasticamente a forma com que sistemas são criados (CLARKE 2005). Isso é demonstrado pelo interesse que essa tecnologia vem despertando, o que é aparente pela grande quantidade de pesquisas sendo realizadas a esse respeito, e a variedade de implementações que tem surgido (BRICHAU 2005).

Os efeitos de considerar a separação de interesses afetam todas as etapas do desenvolvimento de software: modelagem, análise, implementação e manutenção. Daí a relevância de se investigar como modelos de software orientado a aspectos podem ser avaliados em busca de possíveis problemas (os *bad smells*). Da mesma forma, é interessante verificar maneiras de aproximar os modelos das implementações dos sistemas, através dos mecanismos de geração e engenharia reversa de código.

O grande problema ainda está na relativa imaturidade da tecnologia de orientação a aspectos. Tanto as linguagens de implementação quanto de modelagem ainda não se encontram em um estado definitivo, e por isso ainda ocorre muita divergência entre elas. Mesmo assim, em cada uma dessas áreas existe uma tecnologia que se destaca das outras pela maturidade e popularidade. Na área de implementação, essa tecnologia é o AspectJ da IBM, uma adaptação da linguagem Java suportando orientação a aspectos. Na área de modelagem, a abordagem Theme utiliza uma variante de UML que suporta a separação de interesses em seus diagramas. Essas duas tecnologias, no entanto, utilizam alguns conceitos de AOSD de forma diferente, o que torna difícil sua integração. Além disso, no caso da linguagem de modelagem Theme/UML, não existem ferramentas computacionais que lhe dêem suporte.

Nesse trabalho, foi apresentada uma proposta para resolver esses problemas, através de duas modificações: algumas extensões à Theme/UML, de forma que ela permitisse representar mais fielmente a implementação um sistema real utilizando AspectJ, e adaptações de conceitos de Theme a elementos nativos da UML 2.0, de forma que ela possa ser modelada em ferramentas de edição de diagramas UML existentes. Com isso, foi possível criar uma representação de um modelo Theme utilizando a tecnologia XML.

Esse modelo serviu de base para algumas utilizações interessantes. Foi possível utilizá-lo como base para realizar algumas análises de qualidade sobre o sistema representado, e detectar refatorações que tentam melhorar essa qualidade. Ele pôde também ser usado como fonte para um protótipo de gerador de código, capaz de gerar toda a estrutura de um sistema orientado a aspectos, incluindo a definição de *join points* (pontos de junção) e *pointcuts* (conjuntos de pontos de junção). Por fim, ele serviu como base para uma ferramenta de engenharia reversa que pode ser usada para extrair o modelo de uma aplicação AspectJ existente.

O modelo aqui desenvolvido ainda deixa muito espaço para evolução. Alguns conceitos de AspectJ, como pontos de junção relativos ao fluxo de código (*cflow* e *cflowbelow*), apesar de suportados no sistema atual, podem não estar representados da melhor forma possível. Além disso, seria interessante estender o modelo para que ele possa gerar todo o código da aplicação, e não apenas o esqueleto. Isso poderia ser feito através de diagramas de seqüência, ou através de constantes no diagrama UML. Esse modelo estendido também permitiria uma detecção de alguns *bad smells* que não dependem apenas da estrutura de código, como Duplicação de Código.

As atividades que podem ser desenvolvidas ao redor do modelo proposto estão representadas na Figura 5.1.

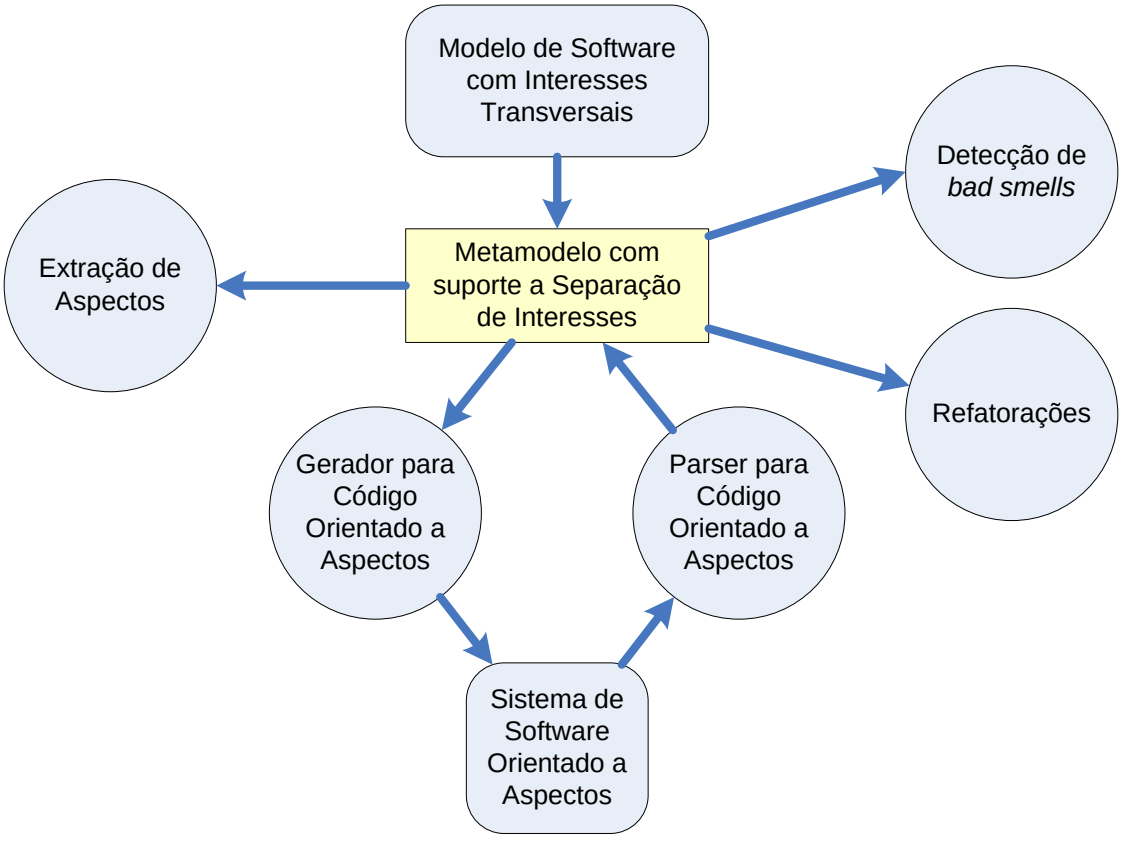


**Figura 5.1:** Atividades que podem ser desenvolvidas sobre um metamodelo que suporte separação de interesses.

O trabalho aqui desenvolvido se encaixa no grupo de pesquisas de Programação Orientada a Aspectos da Engenharia de Software da UFRGS, que já obteve a publicação de alguns artigos (PIVETA 2005, PIVETA 2006, HECHT 2006), e irá prosseguir na evolução das idéias desenvolvidas neste trabalho.

# APÊNDICE: LISTAGEM DE UM DOCUMENTO XMI

A listagem a seguir é a versão simplificada de um documento XMI gerado usando ArgoUML, representando um modelo em Theme a partir da notação sugerida neste trabalho.
As principais diferenças entre esta listagem e o documento original são:

- Identificadores de *tags* (os atributos `xmi.id`) que não eram referenciados em outros pontos do documento (através de atributos `xmi.idref`) foram removidos.
- A maior parte das *tags* representando conceitos já demonstrados em outros *tags* foi removida. A exceção foram *tags* cujos IDs eram importantes para a demonstração de outros trechos.
- *Tags* que não eram relevantes para a compreensão geral do modelo foram removidas.
- Os valores dos identificadores de *tags* foram simplificados para facilitar a compreensão. Os identificadores originais possuem 45 caracteres.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Nov
20 18:41:14 BRST 2006'>
  <XMI.header></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '77B' name = 'GraphicsObserver'>
      <UML:Namespace.ownedElement>
        <UML:DataType xmi.id = '783'/>
        <UML:Stereotype xmi.id = '781' name = 'theme' />
        <!-- Definições dos estereótipos usados por Theme -->
        <UML:Stereotype xmi.id = '7A8' name = 'themeBinding' />
        <UML:TagDefinition xmi.id = '7A5' name = 'themeParameter' />
        <!-- Tagged values usados para definir -->
        <UML:TagDefinition xmi.id = '7AD' name = 'themeParameterBinding' />
        <!-- os parâmetros dos Aspect Themes -->
        <UML:Package xmi.id = '77E' name = 'Observer'>
          <UML:ModelElement.clientDependency>
<!-- Dependências (na verdade bindings) associadas a este aspect theme -->
            <UML:Dependency xmi.idref = '7A3'/>
          </UML:ModelElement.clientDependency>
          <UML:ModelElement.stereotype>
            <UML:Stereotype xmi.idref = '781'/>
            <!-- Indica que este package deve ser tratado como um theme -->
          </UML:ModelElement.stereotype>
          <UML:ModelElement.taggedValue>
        <!-- Especificação dos elementos do modelo que indicam parâmetros -->
            <UML:TaggedValue>
              <UML:TaggedValue.dataValue>
                Subject.aStateChange
              </UML:TaggedValue.dataValue>
              <UML:TaggedValue.type>
                <UML:TagDefinition xmi.idref = '7A5'/>
              </UML:TaggedValue.type>
            </UML:TaggedValue>
            <UML:TaggedValue>
              <UML:TaggedValue.dataValue>
                Observer
              </UML:TaggedValue.dataValue>
              <UML:TaggedValue.type>
                <UML:TagDefinition xmi.idref = '7A5'/>
              </UML:TaggedValue.type>
            </UML:TaggedValue>
          </UML:ModelElement.taggedValue>
          <UML:Namespace.ownedElement>
            <UML:Class xmi.id = '77F' name = 'Subject'>
              <UML:Classifier.feature>
                <UML:Operation xmi.id = '784'
                  name = 'aStateChange ' visibility = 'public' />
                <UML:Operation xmi.id = '786'
                  name = '_aStateChange' visibility = 'protected' />
              </UML:Classifier.feature>
            </UML:Class>
            <UML:Collaboration>
            <!-- Representa o diagrama de seqüência associado -->
              <UML:Namespace.ownedElement>
                <UML:Collaboration.representedClassifier>
                  <UML:Class xmi.idref = '77F'/> <!-- Package Observer -->
                </UML:Collaboration.representedClassifier>
                <UML:Collaboration.representedOperation>
                  <UML:Operation xmi.idref = '784'/>
                  <!-- Subject.aStateChange() -->
                </UML:Collaboration.representedOperation>
                <UML:ClassifierRole xmi.id = 'D82A' name = 'aStateChange()' />
                 <!-- Métodos sendo representados neste diagrama -->
                <UML:ClassifierRole xmi.id = 'D82D' name = '_aStateChange()'/>
                <!-- Métodos sendo representados neste diagrama -->
```

```
            </UML:Namespace.ownedElement>
            <UML:Collaboration.interaction>
              <!-- Chamada do método _aStateChange() por aStateChange() -->
              <UML:Interaction>
                <UML:Interaction.message>
                  <UML:Message>
                    <UML:Message.sender>
                     <UML:ClassifierRole xmi.idref = 'D82A'/>
                    </UML:Message.sender>
                    <UML:Message.receiver>
                      <UML:ClassifierRole xmi.idref = 'D82D'/>
                    </UML:Message.receiver>
                  </UML:Message>
                </UML:Interaction.message>
              </UML:Interaction>
            </UML:Collaboration.interaction>
          </UML:Collaboration>
        </UML:Namespace.ownedElement>
      </UML:Package>
      <UML:Package xmi.id = '782' name = 'Graphics' >
        <UML:ModelElement.stereotype>
          <UML:Stereotype xmi.idref = '781'/>
        </UML:ModelElement.stereotype>
      </UML:Package>
      <UML:Dependency xmi.id = '7A3' name = 'GraphicsObserverBinding'>
        <UML:ModelElement.stereotype>
          <UML:Stereotype xmi.idref = '7A8'/> <!-- themeBinding -->
        </UML:ModelElement.stereotype>
        <UML:Dependency.client> <!-- Aspect theme do binding -->
          <UML:Package xmi.idref = '77E'/>
        </UML:Dependency.client>
        <UML:Dependency.supplier> <!-- Base theme do binding -->
          <UML:Package xmi.idref = '782'/>
        </UML:Dependency.supplier>
        <UML:ModelElement.taggedValue>
          <!-- Valores associados com cada parâmetro do Aspect theme -->
          <UML:TaggedValue xmi.id = 'F827'>
            <UML:TaggedValue.dataValue>
              call[*.set*(..)]
            </UML:TaggedValue.dataValue>
          </UML:TaggedValue>
          <UML:TaggedValue xmi.id = 'F828'>
            <UML:TaggedValue.dataValue>
              declaration[Display]
            </UML:TaggedValue.dataValue>
          </UML:TaggedValue>
        </UML:ModelElement.taggedValue>
      </UML:Dependency>
    </UML:Namespace.ownedElement>
  </UML:Model>
</XMI.content>
</XMI>
```

# REFERÊNCIAS

AMAYA, P.; GONZALEZ, C.; MURILLO, J. M. Towards a Subject-Oriented Model-Driven Framework. In: AKSIT, M.; BEZIVIN, J.; ROUBTSOVA, E. **Aspect-Based and Model-Based Separation of Concerns in Software Systems.** Enschede: Centre for Telematics and Information Technology, University of Twente, 2005.

ARAÚJO, J.; MOREIRA, A.; BRITO, I.; RASHID, A. Aspect-Oriented Requirements with UML. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, 2., 2002, University of Twente, Enschede, Netherlands. **Proceedings…** [S.l:s.n.], 2002.

ARGOUML [Ferramenta]. Disponível em <http://argouml.tigris.org>. Acesso em: 2005.

ARMOUR, P. Zeppelins and Jet Planes: a metaphor for modern software projects. **Communications of the ACM**, New York, v. 44, n. 10, p. 13-15, Oct. 2001.

ASPECTJ DEVELOPMENT TOOLS FOR ECLIPSE (AJDT) [Ferramenta]. Disponível em: <http://www.eclipse.org/ajdt/>. Acesso em: Dec. 2006

ASPECTJ [Ferramenta]. Disponível em: <http://eclipse.org/aspectj/>. Acesso em: Dec. 2006.

AUTOGEN [Ferramenta]. Disponível em: <http://www.gnu.org/software/autogen/>. Acesso em: Sep. 2005.

BARDOU, D. Roles, Subjects and Aspects: how do they relate? In: WORKSHOP ON OBJECT-ORIENTED TECHNOLOGY, ECOOP 98, 12., 1998, Brussels. **Proceedings…** Berlin: Springer-Verlag, 1998. p. 418 – 419. (Lecture Notes In Computer Science; v. 1543).

BASCH, M.; SANCHEZ, A. Incorporating Aspects into the UML. In: ASPECT ORIENTED MODELING WORKSHOP AT AOSD, Boston, Massachusetts 2003. **Proceedings…** [S.l:s.n.], 2003.

BELL, P. **An Introduction to Code Generation**. Disponível em <www.pbell.com/index.cfm/2006/10/5/An-Introduction-to-Code-Generation>. Acesso em Dec. 2006

BERGMANS, L.; AKŞIT, M. Composing Crosscutting Concerns Using Composition Filters. **Communications of the ACM**, New York, v. 44, n. 10, p. 51-57, Oct. 2001.

BIGGERSTAFF, T. J. A perspective of generative reuse. **Annals of Software Engineering**, Dordrecht, v. 5, p. 169-226, 1998.

BOEHM, B. W.; SULLIVAN, K. J. Software economics: a roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2000, Limerick, Ireland. **Proceedings...** New York: ACM, 2000, p. 319–343.

BRICHAU, J.; HAUPT, M. Survey of aspect-oriented languages and execution models. In: AOSD-EUROPE, 2005, Glasgow, UK (Technical Report AOSD-Europe-VUB-01). Disponível em <http://www.aosd-europe.net/deliverables/d12.pdf>. Acesso em Sep. 2005.

CASTLE PROJECT: ASPECT# [Ferramenta]. Disponível em <http://www.castleproject.org/aspectsharp/index.html>. Acesso em: May 2006.

CHAVES, R. **Aspects and MDA**: Creating aspect-based executable models. 2004. Master Thesis.

CHAVEZ, C.; LUCENA, C. A Metamodel for Aspect-Oriented Modeling. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, 2., 2002, University of Twente, Enschede, Netherlands. **Proceeedings…** [S.l:s.n.], 2002.

CHAVEZ, C.; LUCENA, C. Guidelines for Aspect-Oriented Design. In: BRAZILIAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 2004, **Anais…** Brasília, DF, Brasil.

CLARKE, S.; BANIASSAD, E. **Aspect-Oriented Analysis and Design**: The Theme Approach. Upper Saddle River: Addison-Wesley, 2005

CLARKE, S.; BANIASSAD, E. Theme: An Approach for Aspect-Oriented Analysis and Design. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, **Proceedings…**, New York: ACM, 2004.

CLARKE, S.; WALKER, R. J. **Separating Crosscutting Concerns across the Lifecycle**: From Composition Patterns to AspectJ and Hyper/J. Dublin: Trinity College, 2001 (Technical Report TCD-CS-2001-15).

CLARKE, S.; WALKER, R. J. Towards a standard design language for AOSD. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD'2002), 2002, Enschede, Netherlands. **Proceedings…,** New York: ACM, 2002.

CODEGEN [Ferramenta]. Disponível em <http://forge.novell.com/modules/xfmod/project/?codegen>. Acesso em Set. 2005.

CODESMITH [Ferramenta]. Disponível em <http://www.ericjsmith.net/codesmith/>. Acesso em Set. 2005.

DODDS, L. **Code generation using XSLT.** Disponível em: <http://www-128.ibm.com/developerworks/edu/x-dw-codexslt-i.html>. Acesso em Oct. 2006

ELRAD, T. et al. Discussing Aspects of AOP. **Communications of the ACM**, New York, v. 44, p. 33-38, 2001.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-Oriented Programming: Introduction. **Communications of the ACM**, New York, v. 44, p. 29-32, 2001.

ELSSAMADISY, A.; SCHALLIOL, G. Recognizing and responding to bad smells in extreme programming. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 24., 2002. **Proceedings…**, New York: ACM, 2002.

FILMAN, R. E. **A Bibliography of Aspect-Oriented Programming, Version 2.0**. Disponível em <http://home.comcast.net/~refilman/text/oif/aosd-bibliography.pdf >. Acesso em Nov. 2005.

FILMAN, R. E.; FRIEDMAN, D. P. **Aspect-Oriented Programming Is Quantification and Obliviousness.** Disponível em <www.riacs.edu/research/technical_reports/TR_pdf/TR_01.12.pdf>. Acesso em Nov. 2005.

FOWLER, M. **Refactoring:** improving the design of existing code. Boston: Addison-Wesley Longman, 2000.

GAMMA, E. et al. **Design Patterns**. Boston: Addison-Wesley Longman, 1994.

GARCIA, A. et al. Aspects @ PUC-Rio: Poster Session. In: BRAZILIAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, Brasília, Brazil, 2004. **Proceedings…**, [S.l:s.n.], 2004.

GARCIA, A. et al. Modularizing Design Patterns with Aspects: a quantitative study. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD-2005), 4., 2005. **Proceedings…**, New York: ACM, 2005.

GONZÁLEZ, C.; MURILLO, J. M.; AMAYA, P. A. Aspect-Oriented Analysis: a MDA based approach. In: TEKINERDOGAN, B. **Early Aspects**: Aspect-Oriented Requirements Engineering and Architecture Design. Enschede: Centre for Telematics and information Technology, University of Twente, 2004

GROHER, I.; BAUMGARTH, T. Aspect-Orientation from Design to Code. In: TEKINERDOGAN, B. **Early Aspects**: Aspect-Oriented Requirements

Engineering and Architecture Design. Enschede: Centre for Telematics and information Technology, University of Twente, 2004

GROHER, I.; SCHULZE, S. Generating Aspect Code from UML Models. In: ASPECT ORIENTED SOFTWARE DEVELOPMENT MODELING WITH UML WORKSHOP, 2003, **Proceedings…**, San Francisco, California, Mar. 2003.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of aspect-oriented software. In: ANNUAL INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED AND INTERNET-BASED TECHNOLOGIES, CONCEPTS, AND APPLICATIONS FOR A NETWORKED WORLD (NET.OBJECTDAYS), 2003. **Proceedings…**, New York: ACM, 2003.

HANNEMANN, J.; KICZALES, G. Design pattern implementation in Java and AspectJ. In: ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 17., 2002. **Proceedings…**, New York: ACM, 2002.

HANNEMANN, J.; KICZALES, G. **Aspect-Oriented Design Pattern Implementations**. Disponível em: <http://www.cs.ubc.ca/~jan/AODPs/>. Acesso em Oct. 2006.

HARRISON, W.; OSSHER, H. Subject-Oriented Programming — A Critique of Pure Objects. In: ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 1993. **Proceedings…**, New York: ACM, 1993.

HECHT, M. V.; PIVETA, E. K.; PIMENTA, M. S.; PRICE, R. T. Aspect-Oriented Code Generation. In: SIMPÓSIO BRASIELIRO DE ENGENHARIA DE SOFTWARE, 20., SBES, 2006, Florianópolis, Brasil. **Anais...**, p. 209-223, Florianópolis:SBC, 2006.

HIGHLEY, T. J.; LACK, M.; MYERS, P. **Aspect Oriented Programming:** a critical analysis of a new programming paradigm**. Technical Report CS-99-29**. Charlottesville: University of Virginia, 1999.

HO, W. M.; JÉZÉQUEL, J.-M.; PENNANEAC'H, F.; PLOUZEAU, N. A Toolkit for Weaving Aspect Oriented UML Designs. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD'2002), 2002, Enschede, Netherlands. **Proceedings…,** p. 99-105, New York: ACM, 2002.

IWAMOTO, M.; ZHAO, J. Refactoring aspect-oriented programs. In: AOSD MODELING WITH UML WORKSHOP, 4., 2003, **Proceedings…** San Francisco, California, USA, Mar. 2003.

JBOSS ASPECT-ORIENTED PROGRAMMING [Ferramenta]. Disponível em <http://www.jboss.org/products/aop>. Acesso em Dec. 2006.

KICZALES, G. et al. An overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP 2001), 15., 2001. **Proccedings…**, London:Springer-Verlag, 2001.

KICZALES, G. et al. Getting Started with AspectJ. **Communications of the ACM**, New York, v. 44, p. 59-65, 2001

KICZALES, G. et al Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP-97), 11., 1997. **Proceedings…**, London:Springer-Verlag, 2001.

KICZALES, G.; MEZINI, M. Aspect-oriented programming and modular reasoning. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 27., 2005, St. Louis, MO. **Proceedings…**, New York:ACM, p. 49-58, 2005.

KRUEGER, C. W. Software Reuse. **ACM Computing Surveys**, New York, v. 24, p. 131-183, 1992.

KULESZA, U.; LUCENA, A. G. C. Towards a Method for the Development of Aspect-Oriented Generative Approaches. . In: TEKINERDOGAN, B. **Early Aspects**: Aspect-Oriented Requirements Engineering and Architecture Design. Enschede: Centre for Telematics and information Technology, University of Twente, 2004

LADDAD, R. **An AOP success story from the real world**. Disponível em <http://ramnivas.com/blog/wp-trackback.php/19>. Acesso em Nov. 2005.

LADDAD, R. **I want my AOP!** Disponível em <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>. Acesso em Mar. 2005

LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. Aspect-Oriented Programming with Adaptive Methods. **Communications of the ACM**, New York, v. 44, p. 39-41, 2001

MEYER, B. Reusability: The case for Object-Oriented design. **IEEE Software**, Los Alamitos, CA, v. 4, n. 2, p. 50-64, 1987.

MEYER, B. **Object-Oriented Software Construction**, 2nd ed, Upper Saddle River:Prentice-Hall, 1997.

MONTEIRO, M. P.; FERNANDES, J. M. Object-to-aspect refactorings for feature extraction. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD'2004, 3., 2004. **Proceedings…**, New York: ACM Press, 2004.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a Catalog of Aspect-Oriented Refactorings. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED

SOFTWARE DEVELOPMENT, AOSD'2005, 4., 2005. **Proceedings…**, New York: ACM, 2005.

MVCASE [Ferramenta]. Disponível em <https://mvcase.dev.java.net/>. Acesso em Oct. 2005.

NETINANT, P.; ELRAD, T.; FAYAD, M. E. A Layered Approach to Building Open Aspect-Oriented Systems: A Framework for the Design of On-Demand System Demodularization. **Communications of the ACM**, New York, v. 44, p. 83-85, 2001.

OMG MDA (MODEL DRIVEN ARCHITECTURE) APPLICATION GUIDE. Disponível em <http://www.omg.org/mda/>. Acesso em Nov. 2005.

OMG MOF (METAOBJECT FACILITY) SPECIFICATION 2.0. Disponível em <http://www.omg.org/spec/MOF/2.0/PDF/>. Acesso em Dec. 2006

OSSHER, H.; TARR, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: SYMPOSIUM ON SOFTWARE ARCHITECTURES AND COMPONENT TECHNOLOGY, 2001. **Proceedings…**, London:Kluwer, 2001.

OSSHER, H.; TARR, P. Operation-Level Composition: a case in (join) point. In: WORKSHOP ON ASPECT ORIENTED PROGRAMMING, 1998, **Proceedings…**, Brussels, Belgium, July 1998.

OSSHER, H.; TARR, P. The Shape of Things To Come: using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. **Communications of the ACM**, New York, v. 44, p. 43-50, 2001

PAWLAK, R. et al. A UML Notation for Aspect-Oriented Software Design. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2002, Enschede, Netherlands. **Proceedings…,** New York: ACM, 2002.

PINTO, M.; FUENTES, L.; TROYA, J. M. A Component and Aspect Dynamic Platform. **The Computer Journal**, Oxford: Oxford University Press, v. 48, n. 4, p. 401-420, 2005.

PIVETA, E. K.; HECHT, M. V.; PIMENTA, M. S.; PRICE, R. T. Bad Smells em Sistemas Orientados a Aspectos. In: XIX SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 2005, Uberlândia. **Anais...**, v. 1. p. 184-199, 2005.

PIVETA, E. K.; HECHT, M.; PIMENTA, M. S.; PRICE, R. T. Detecting Bad Smells in AspectJ. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 10. , Itatiaia, 2006. **Anais…**, Itatiaia:SBC, 2006.

POPOVICI, A.; GROSS, T.; ALONSO, G. Dynamic Weaving for Aspect-Oriented Programming. In: INTERNATIONAL CONFERENCE ON ASPECT-

ORIENTED SOFTWARE DEVELOPMENT, AOSD'2002, 2002, Enschede, Netherlands. **Proceedings…,** p. 141-147, New York: ACM, 2002.

POSEIDON [Ferramenta]. Disponível em <http://gentleware.com/>. Acesso em Oct. 2005.

RASHID, A. et al. Survey of Aspect-Oriented Analysis and Design Approaches. In: AOSD-EUROPE, 2005, Glasgow, UK (Technical Report AOSD-Europe-VUB-01). Disponível em <http://www.aosd-europe.net/deliverables/d11.pdf>. Acesso em Apr. 2006.

RASHID, A.; SAWYER, P.; MOREIRA, A.; ARAÚJO, J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In: IEEE JOINT INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING**,** Los Alamitos:IEEE, 2002. **Proceedings…**, p. 199-202, 2002.

RATIONAL SOFTWARE [Ferramenta]. Disponível em <http://www-306.ibm.com/software/rational/>. Acesso em Nov. 2005.

STEIN, D.; HANENBERG, S.; UNLAND, R. An UML-based Aspect-Oriented Design Notation. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD'2002, 2002, Enschede, Netherlands. **Proceedings…,** New York: ACM, 2002.

SUZUKI, J.; YAMAMOTO, Y. Extending UML with aspects: aspect support in the design phase. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED PROGRAMMING, ECOOP, 1999, Lisboa, Portugal, **Proceedings…**, p299-300.UML: UNIFIED MODELING LANGUAGE SPECIFICATION, VERSION 2.0. Disponível em <http://www.omg.org/technology/documents/formal/uml.htm>. Acesso em Dec. 2005.

VELOCITY [Ferramenta]. Disponível em <http://jakarta.apache.org/velocity/>. Acesso em Sep. 2005.

XALAN-JAVA VERSION 2.7.0 [Ferramenta]. Disponível em <http://xml.apache.org/xalan-j/>. Acesso em Jun. 2006

XMI MAPPING SPECIFICATION, v2.1. Disponível em <http://www.omg.org/technology/documents/formal/xmi.htm>. Acesso em Dec. 2005.

XQUERY 1.0: AN XML QUERY LANGUAGE. Disponível em <http://www.w3.org/TR/xquery/>. Acesso em Dec. 2006

XSLT (EXTENSIBLE STYLESHEET LANGUAGE TRANSFORMATIONS) SPECIFICATION VERSION 2.0. Disponível em <http://www.w3.org/TR/xslt20/>. Acesso em Dec. 2005.