UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rafael Alves Chaves

Aspectos e MDA
Criando modelos executáveis baseados em aspectos

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Luiz Carlos Zancanella, Dr.
(orientador)

Florianópolis, Fevereiro de 2004

Aspectos e MDA
Criando modelos executáveis baseados em aspectos

Rafael Alves Chaves

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____
Fernando Austuni Gauthier, Dr.

Banca Examinadora

_____
Luiz Carlos Zancanella, Dr.

_____
Carlos Barros Montez, Dr.

_____
Raul Sidnei Wazlawick, Dr.

_____
Ricardo Pereira e Silva, Dr.

Agradecimentos

À Fabiane e ao Gabriel, esposa e filho, por serem minha grande razão de viver.

À Neusa e ao Luiz Carlos, mãe e saudoso pai, pelo carinho e pela oportunidade de estar aqui.

Ao Zancanella, orientador, pela paciência e confiança; e amigo, pela compreensão e apoio.

Aos membros da banca, que ajudaram a aprimorar o texto.

Aos amigos que leram e criticaram o texto, deram caronas, e/ou comemoraram comigo.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Listagens

Resumo

As principais contribuições deste trabalho consistem em analisar o potencial do uso conjunto das abordagens MDA e orientação a aspectos, e propor extensões à UML para comportar a criação de modelos executáveis usando o paradigma de aspectos.

O resultado demonstra que o benefício da combinação das duas abordagens é maior do que a soma das partes, porque expande o domínio de utilização e maximiza o potencial de adaptabilidade. O conjunto de extensões proposto viabiliza a geração automática de programas completos a partir de especificações com alto grau de abstração e modularidade. Adicionalmente, o uso de XML como base para a linguagem de ações proposta simplifica a implementação de tradutores e a viabiliza como representação intermediária para outras linguagens usadas na programação com ações.

Abstract

The main contributions of this work consist in evaluating the potential benefits of combining the use of the MDA and the aspect-oriented paradigm, and proposing extensions to the UML in order to support the creation of executable models using aspects.

The results show that the benefits of combining those two approaches are greater than the sum of the parts, because this jointly usage expands the application domain and maximizes the potential for adaptability. The proposed extensions allow the automatic generation of complete programs from highly abstract and modular specifications. Moreover, the fact that the proposed action language is based on XML simplifies the construction of translators and makes it suitable as an intermediary representation for other languages used for action-based programming.

1 Introdução

Este trabalho propõe o uso conjunto das tecnologias de aspectos e *Model Driven Architecture* (MDA) em uma abordagem que pretende facilitar a evolução/adaptabilidade em software, no sentido de contemplar mudanças tanto no ambiente computacional como nos requisitos do sistema, gerando sistemas executáveis completos a partir de modelos que são fáceis de compreender, modificar e estender.

1.1 Motivação

Uma das propriedades mais importantes em um sistema de software é a sua adaptabilidade a mudanças tanto nos requisitos a serem satisfeitos como no ambiente computacional no qual se deseja empregá-lo.

No projeto de uma solução computacional, as preocupações (*concerns*)determinam a escolha dos artefatos (como módulos, algoritmos e estruturas de dados) que compõem tal solução. A influência concomitante de duas ou mais preocupações sobre a definição de um mesmo artefato compromete a compreensibilidade, pois obscurece as razões que determinaram sua concepção. Além disso, a rigidez na composição entre duas preocupações fundamentalmente não relacionadas desnecessariamente dificulta a modificação de apenas uma delas.

Já do ponto de vista da implementação, novas preocupações, relativas a idiossincrasias da plataforma alvo, são acrescentadas. Além disso, os artefatos concebidos na fase de projeto precisam ser mapeados/refinados em função dos recursos oferecidos pelo ambiente computacional em questão (como linguagem de programação, bibliotecas de tempo de execução, e chamadas do sistema providas pelo sistema operacional). Tais especificidades também contribuem para obscurecer as intenções por trás de cada artefato.

Nesse contexto, é possível identificarem-se problemas significativos relativos à tolerância a mudanças:

- mudanças nos requisitos do sistema (mesmo que o ambiente computacional seja preservado) podem requerer um grande esforço do desenvolvedor em identificar quais módulos na implementação são afetados e em determinar como o código existente precisa ser modificado para que os novos requisitos sejam apropriadamente atendidos;
- mudanças no ambiente computacional (mesmo que os requisitos de um sistema sejam preservados) podem requerer a re-implementação do sistema - o que

compromete tanto a sua longevidade e reuso (ou de alguns de seus módulos) em outros ambientes computacionais.

Dessa forma, a adequada separação de preocupações (*separation of concerns*) [HÜRSCH95], como provido pela orientação a aspectos [KICZALES97], e o uso de modelos independentes de plataforma, como proposto por MDA (*Model Driven Architecture,* ou Arquitetura Orientada a Modelos) [MDA] colaboram no aumento de compreensibilidade, flexibilidade, portabilidade e conseqüentemente manutenibilidade e tolerância a mudanças em um sistema.

A orientação a aspectos é um paradigma de desenvolvimento de software bastante recente que provê melhor modularização e composição de preocupações intimamente relacionadas. Alterações no sistema em resposta a mudanças nos requisitos são simplificadas, porque as preocupações correspondentes são apropriadamente modularizadas - além de facilmente identificáveis, são menos acopladas entre si, reduzindo o impacto de modificações.

MDA é uma iniciativa do OMG que promove o uso no desenvolvimento de software de modelos independentes de detalhes de implementação. Alterações no ambiente computacional requerem alteração do *modelo de plataforma* (vide seção 2.1) de forma a refletir o novo ambiente, mas não causam qualquer modificação no *modelo independente de plataforma.* Através de transfomações sucessivas, os modelos originais, que tratam das preocupações dominantes, são adornados com artefatos relativos a preocupações relativas à implementação, eventualmente gerando um programa executável completo.

1.2 Objetivos

O objetivo central destre trabalho foi investigar o uso da orientação a aspectos no contexto do desenvolvimento baseado em modelos executáveis com o intuito de prover maior adaptabilidade em software.

Para atingir essa meta, os seguintes objetivos intermediários tiveram que ser alcançados:

i) examinar o desenvolvimento baseado em modelos executáveis, especificamente no contexto de MDA e da semântica de ações da *Unified Modeling Language* (UML).

ii) examinar o paradigma de orientação a aspectos, investigando sua utilização nas fases de programação e modelagem;

iii) apresentar uma proposta de uso do paradigma da orientação a aspectos no desenvolvimento baseado em modelos executáveis de forma a promover maior tolerância a mudanças.

1.3 Organização do trabalho

A estrutura do restante deste documento é a seguinte:

O capítulo 2, "Modelos Executáveis", apresenta uma visão geral do desenvolvimento baseado em modelos executáveis, descrevendo em maiores detalhes MDA e a semântica de ações de UML.

O capítulo 3, "Orientação a Aspectos", descreve os conceitos básicos deste recente paradigma; apresenta AspectJ, uma linguagem de programação orientada a aspectos baseada em Java; provê programas-exemplo que ilustram a orientação a aspectos em ação, e descreve algumas das abordagens existentes para modelagem orientada a aspectos, comparando-as entre si.

O capítulo 4, "Libra: Modelos UML Executáveis Baseados em Aspectos", define uma linguagem de ações UML que provê suporte a aspectos, e que constitui a principal contribuição deste trabalho. O capítulo apresenta ainda exemplos que ilustram o uso da linguagem proposta.

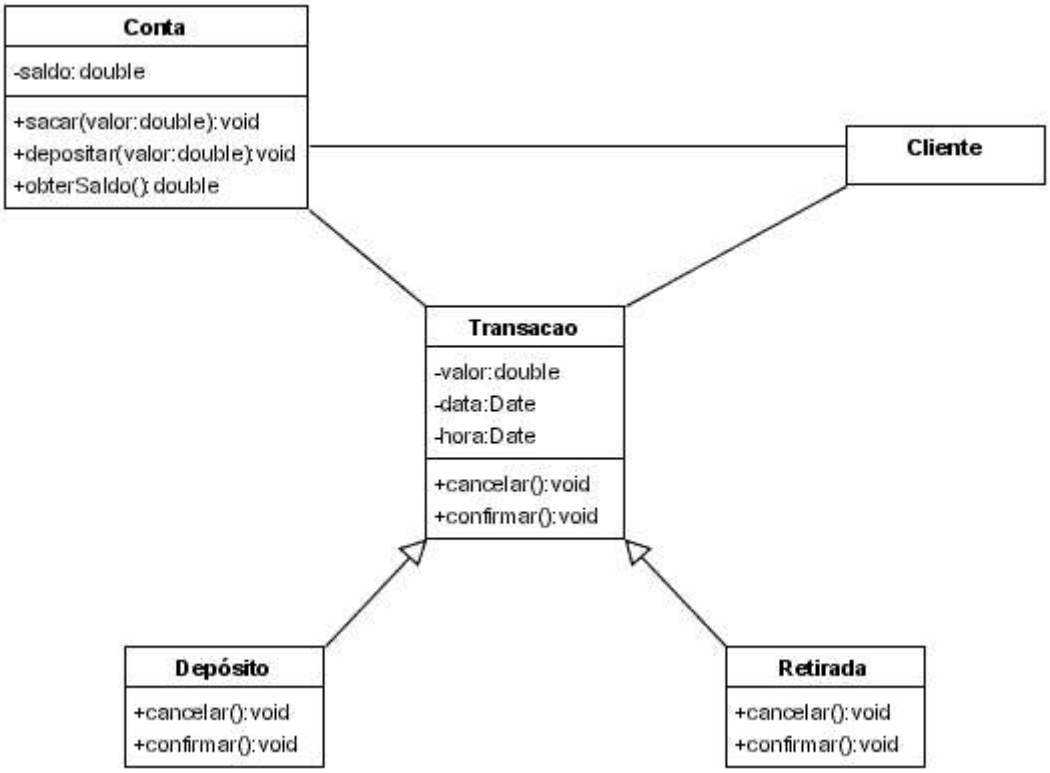
O capítulo 5, "Conclusões e Perspectivas", relata os resultados obtidos e apresenta propostas para a continuidade do desenvolvimento deste trabalho.

Ao final, o trabalho inclui um anexo com o código-fonte dos exemplos apresentados no capítulo 3, e um glossário com a terminologia relativa a aspectos utilizada.

1.4 Aplicação exemplo

Ao longo do presente trabalho, uma aplicação exemplo é utilizada para ilustrar conceitos e idéias sendo descritos: um sistema bancário. O modelo de objetos dessa aplicação está representado na forma de um diagrama de classes na figura 1.

Figura 1: O modelo de objetos da aplicação exemplo



Os objetos primários do domínio do problema são: *Conta*, *Cliente*, e *Transação*. Uma *Conta* apresenta duas operações: *sacar* e *depositar*, além de um atributo *saldo*. Cada *Cliente* pode possuir uma ou mais contas. Uma *Transação* (que pode ser uma *Retirada* ou um *Depósito*) é um histórico para fins de controle que detalha uma operação efetuada.

O modelo aqui apresentado não é definitivo. Eventualmente, dependendo do contexto onde é empregado, modificações são introduzidas, e detalhes omitidos/apresentados.

2 Modelos Executáveis

2.1 Model Driven Architecture

A *Model Driven Architecture* (MDA), ou arquitetura orientada a modelos, [MDA] é uma iniciativa do *Object Management Group* (OMG) [OMG] que promove o uso de modelos no desenvolvimento de software, com o intuito de separar a especificação de um sistema dos detalhes de como sua implementação usa os recursos da plataforma subjacente. MDA promove uma abordagem na qual a especificação do sistema é feita de forma independente de plataforma e, para cada uma das plataformas de interesse, tal especificação pode ser automaticamente transformada em uma implementação correspondente. MDA tem como principais objetivos a portabilidade, a interoperabilidade e a reusabilidade, sendo uma "solução arquitetural para separação de preocupações" [MDA].

As seguintes tecnologias padronizadas pelo OMG servem como base para MDA:

- *Unified Modeling Language* (UML) – notação padrão na indústria para representação gráfica de modelos de software orientado a objetos, UML é bastante indicada para a criação dos modelos em MDA (embora não seja requerida). Uma adição recente a essa linguagem que a torna ainda mais interessante no contexto de MDA é a capacidade de expressão de comportamento no nível de modelos (semântica de ações, vide seção 2.2);
- *Meta-Object Facility* (MOF) – um arcabouço *(framework)* para gerenciamento de meta-dados usado na definição de vários meta-modelos propostos pelo OMG, como UML e o próprio MOF. MOF foi mapeado para tecnologias de amplo uso na atualidade, como *Java Metadata Interface* (JMI) e *XML Metadata Interchange* (XMI).

2.1.1 Modelos e pontos de vista

O modelos ocupam o papel central no desenvolvimento baseado em MDA. É principalmente através da manipulação dos modelos que o sistema sendo desenvolvido pode ser compreendido, projetado, construído, implantado e modificado.

Modelos em MDA descrevem e especificam o sistema através de um ponto de vista específico. MDA define três pontos de vista através dos quais modelos de um sistema podem ser compreendidos/manipulados:

- o ponto de vista independente de computação – o enfoque está no ambiente e nos requisitos do sistema - não provê qualquer detalhamento quanto à estrutura e ao comportamento do sistema. Os modelos independentes de computação (*Computation Independent Models* - CIM) são comumente denominados *modelos do domínio* ou *modelos de negócios*, e a especificação desses modelos é feita usando-se termos familiares aos especialistas no domínio em questão. Tais modelos possuem papel importante na comunicação entre os especialistas do domínio do problema e os desenvolvedores do sistema;
- o ponto de vista independente de plataforma - detalha características estruturais e comportamentais do sistema que são independentes da sua implementação em uma plataforma específica. Os modelos independentes de plataforma (*Platform Independent Models* - PIM) especificam completamente os elementos baseando-se em abstrações computacionais genéricas a uma determinada família de plataformas, como objetos, atributos, mensagens, tipos de dados comuns, operações aritméticas e lógicas, controle de fluxo etc;
- o ponto de vista específico à plataforma - incrementa o ponto de vista independente de plataforma de forma a definir com precisão como os modelos daquele ponto de vista são mapeados para uma implementação em uma plataforma específica. Modelos específicos a plataforma (*Platform Specific Model*s - PSM*)* são o produto final da arquitetura. Exemplos são programas em código-fonte ou objeto, arquivos de configuração e esquemas de banco de dados. Relacionado a isto está o conceito de *modelo de plataforma.* Tal modelo define formalmente quais recursos e serviços estão disponíveis em uma plataforma específica, para serem utilizados em PSMs.

2.1.2 Transformação de modelos

A transformação de modelos em MDA é o processo pelo qual a partir de especificações com um alto nível de abstração e independência de plataforma seja possível gerar, de forma automática, sistemas completos para múltiplas plataformas.

MDA define que um modelo independente de implementação (PIM) seja combinado com informações adicionais através de um processo de transformação de forma a gerar um modelo específico à implementação (PSM). Exatamente que

informações adicionais são essas e a forma como a transformação é efetivamente realizada não são definidos por MDA.

Serviços ubíquos

Serviços ubíquos são serviços disponíveis em uma ampla faixa de plataformas (exemplos são bancos de dados relacionais, serviços de autenticação, sistemas de arquivos e servidores Web). O OMG pretende definir através da MDA modelos independentes de plataforma para tais tipos de serviços. Dessa forma, será possível o uso desses serviços em PIMs sem comprometimento da portabilidade.

2.1.3 O ciclo de desenvolvimento com MDA

O ponto de partida do trabalho com MDA é o modelo independente de computação. O CIM tem as seguintes características:

- representa os requisitos do sistema, definindo sua operação no ambiente para o qual é proposto, mas de forma independente de como o sistema é implementado;
- define um vocabulário comum para uso em outros modelos, de forma que é possível identificar a relação entre os requisitos especificados neste modelo e as correspondentes construções que implementam tais requisitos nos outros modelos;
- por ser um modelo informal, não é diretamente utilizado em transformações automatizadas.

O segundo estágio é a construção do modelo independente de plataforma. Os pontos-chave no PIM são:

- descreve o sistema em detalhes, mas sem definir como os recursos específicos da plataforma serão utilizados;
- pode ser dependente de um determinado estilo arquitetural de plataforma (por exemplo, chamada remota de procedimento, baseado em mensagens assíncronas, processamento em lote, interativo etc).

O terceiro e último estágio é a transformação do modelo independente de plataforma em um ou mais modelos específicos à plataforma. Tal transformação ocorre de acordo com um determinado mapeamento específico. A natureza do mapeamento PIM -> PSM é determinada pelo modelo de plataforma.

2.1.4 Tipos de mapeamentos em MDA

Existem fundamentalmente três formas diferentes de se realizar o mapeamento de modelos em MDA:

- mapeamento baseado em tipos de elementos – o modelo independente de plataforma é construído usando-se uma linguagem que dá suporte a um determinado conjunto de elementos de modelo. Um ou mais mapeamentos definem como cada tipo de elemento no modelo independente de plataforma corresponde a um ou mais tipos de elementos no modelo específico à plataforma. Por exemplo, uma classe definida com o estereótipo <<*Entidade*>> pode ser mapeada para um Entity Bean em *Enterprise JavaBeans* (EJB) [EJB] (que corresponde a um conjunto de interfaces/classes Java). Este tipo de mapeamento possui o inconveniente de não possibilitar ao arquiteto do sistema influenciar no processo, dado que os mapeamentos entre tipos do PIM e do PSM são estáticos.
- mapeamento baseado em instâncias de elementos – nesta abordagem, define-se para cada elemento do modelo independente de plataforma que mapeamento será utilizado. Isso é feito através da anotação do PIM com marcas que dirigem a transformação para o PSM (tal anotação com informações dependentes de plataforma é feita em uma cópia do PIM, de forma a preservar o modelo original).
- mapeamento baseado em tipos *e* instâncias de elementos – na realidade, a forma mais interessante de mapeamento, pois combina as duas técnicas anteriores. Dessa forma, mapeamentos pré-definidos baseados em tipo podem ser adaptados pelo arquiteto, que anota elementos do modelo com marcas apropriadas de forma a conduzir a transformação da maneira desejada.

2.2 Semântica de Ações em UML

A semântica de ações[1] em UML [UML] possibilita ao projetista definir significado para elementos comportamentais em um modelo UML. Para isso, o projetista utiliza uma linguagem de ações, que oferece capacidades semelhantes às linguagens de programação imperativas existentes, mas em um nível de abstração mais alto.

1 Semântica de ações foi adicionado à UML em sua versão 1.5 (que difere da versão 1.4 basicamente pela presença de tal suporte).

UML define a semântica que tal linguagem de ações deve comportar, mas a sintaxe não é especificada. Isso possibilita que cada fornecedor ofereça sua própria linguagem, e também que linguagens existentes possam ser utilizadas (se adaptadas) como linguagens de ações.

A espinha dorsal da semântica de ações em UML é baseada em três conceitos básicos: *procedimentos*, *ações* e *pinos* de entrada e saída:

- procedimentos são peças de comportamento (especificado através de uma ação) que podem ser vinculados a elementos comportamentais de modelos UML, como operações, gatilhos em transições de estado e restrições.
- ações definem o comportamento a ser executado por procedimentos. Ações implementam atividades comuns como ler e escrever dados, enviar mensagens a objetos, criar/destruir objetos, avaliar condições e agrupar/seqüenciar outras ações;
- pinos são vias para transmissão de dados entre ações, ou entre um procedimento e a ação que define seu comportamento. Dados produzidos por uma ação ficam disponíveis em seus pinos de saída, aos quais podem ser conectados os pinos de entrada de outras ações.

2.2.1 Conceitos básicos

A figura 2 ilustra o relacionamento entre os conceitos de procedimentos, ações e pinos, que serão descritos nesta seção.

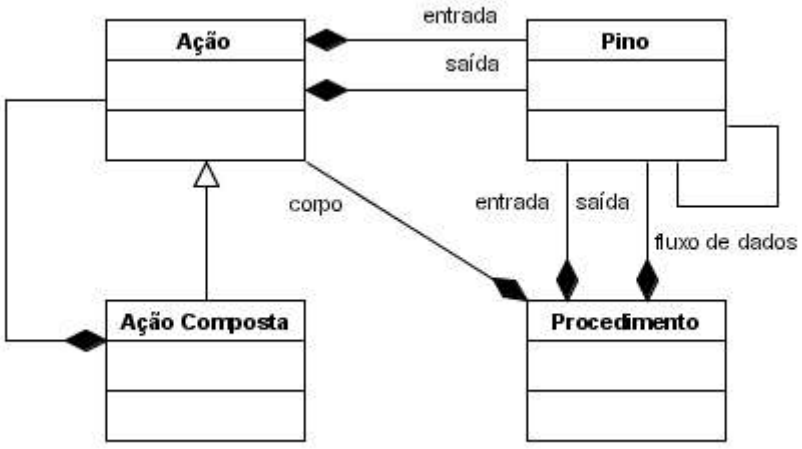


Figura 2: Os conceitos fundamentais da semântica de ações de UML

Procedimentos

Procedimentos são o nível mais alto para composição de ações, e provêem a forma pela qual um conjunto de ações pode ser associado a outros elementos em UML, por exemplo, definindo o corpo de um método ou uma condição para uma transição de estado.

Um procedimento, quando ativado, recebe um objeto requisição e ao final produz um objeto resultado. O comportamento de um procedimento é especificado por meio de uma ação, que é freqüentemente composta.

Um procedimento fornece um contexto para a execução das ações que o compõem. Esse contexto possibilita o compartilhamento de valores entre as ações, em adição àqueles disponíveis através dos pinos de entrada e saída.

Ações

Ações são as unidades elementares para especificação de comportamento. Ações podem aceitar um conjunto de valores de entrada, e produzir um conjunto de valores de saída em função dos valores de entrada.

Ações são inerentemente concorrentes. Em um conjunto de ações. uma ação pode ser executada antes, após ou simultaneamente a outra ação qualquer, exceto quando uma ordem entre elas é definida (explicitamente, através de um fluxo de controle, ou implicitamente, através de um fluxo de dados).

Pinos

As entradas e saídas requeridas por uma ação são denominadas *pinos*. Pinos definem tanto a multiplicidade como os tipos de valores admissíveis/produzíveis.

O contexto para uma ação é completamente definido por seus pinos, inclusive o objeto corrente. Uma ação não pode acessar ou produzir nenhuma informação que não através de seus pinos de entrada e saída (respectivamente).

Fluxos de dados

Fluxos de dados carregam dados de uma ação para outra. Fluxos de dados impõem, implicitamente, um seqüenciamento a ações, de forma que a execução da ação que representa a origem do fluxo de dados precisa ser concluída antes que a ação destino possa iniciar sua execução. Dessa forma, possibilitam a transmissão de dados entre ações

sem que seja necessário armazenamento intermediário.

Fluxos de dados conectam um pino de saída de uma ação aos pinos de entrada de outras, mas um pino de entrada não pode estar conectado a múltiplos pinos de saída.

Um pino de entrada (destino) em um fluxo precisa ser compatível (tanto em tipo como multiplicidade) com o pino de saída a que está conectado.

Fluxos de controle

Um fluxo de controle tem como única finalidade estabelecer um seqüenciamento entre duas ações. A ação sucessora não pode ser executada até que a ação predecessora conclua. Fluxos de controle são úteis quando uma ação precisa acessar uma informação (por exemplo, o atributo de um objeto) somente após outra ação tê-la escrito.

Cadeias de ações conectadas através de fluxos de dados ou controle formam um grafo dirigido acíclico (isto é, uma ação não pode suceder a si mesma, seja direta ou indiretamente).

2.2.2 Tipos de ações

Ações primitivas

Ações primitivas não podem ser decompostas em outras ações mais simples. Tais ações ou realizam computações, ou manipulam dados, ou provêem interações entre objetos. A forma dessas ações especifica os conjuntos de pinos de entrada e saída.

Ações compostas

Ações compostas provêm uma maneira para que um conjunto de ações possa ser executado como se fosse uma única ação. Existem três ações nesta categoria:

- ação de agrupamento *(group)*- o tipo mais simples de ação composta, por ser apenas uma forma de organizar as ações coletivamente, sem prover entradas ou saídas nem encapsular as ações que o compõem. Podem ser utilizadas em fluxos de controle, possibilitando a especificação de seqüenciamento entre dois grupos. Podem também ser usados para definir regiões de exclusão mútua, de forma que apenas uma ação entre as ações contidas seja executada em um determinado instante;

- ação condicional (*conditional*) – uma ação composta que executa uma entre as ações contidas, dependendo do resultado das ações-teste. *Cláusulas* estabelecem a relação entre as ações-teste e as ações contidas. Uma ação contida somente será executada caso o resultado da ação-teste seja o valor lógico verdadeiro. Mesmo que mais de uma cláusula tenha sua ação-teste satisfeita, apenas uma dessas cláusulas terá sua ação-resultado executada;
- ação de iteração (*loop*) – uma ação composta que possibilita a repetição da ações contida enquanto a ação-teste é satisfeita. Diferentemente da ação condicional, a ação de iteração é formada por apenas uma cláusula.

Ações de leitura e escrita

Tais ações obtêm/definem valores de objetos, atributos, variáveis, relacionamentos, ou criam tais elementos. Ações de leitura não modificam os valores que acessam, enquanto que ações de escrita podem criar e destruir objetos e relacionamentos entre objetos.

A tabela 1 enumera algumas ações deste tipo.

Tabela 1: Principais ações para leitura/escrita de valores

Ação	***Descrição***
CreateObject	Cria um novo objeto
DestroyObject	Destrói um objeto existente
ReadAttribute / Variable	Lê os valores de um atributo/variável
AddAttribute / VariableValue	Adiciona um valor ao conjunto de valores de um atributo/variável
RemoveAttribute / VariableValue	Remove um valor do conjunto de valores de um atributo/variável
ClearAttribute/Variable	Remove todos os valores de um atributo/variável
ReadSelf	Obtém o objeto corrente no contexto do procedimento atual
ReadExtent	Obtém todas as ocorrências de um classificador
ReadLink	Obtém o(s) objeto(s) conectados por uma associação
ClearAssociationAction	Desconecta todos os objetos participando da associação
CreateLink	Conecta um objeto a uma associação
DestroyLink	Desconecta um objeto de uma associação

Ações de computação

Ações dentro desta categoria realizam tranformações em um conjunto de valores de entrada, produzindo um determinado conjunto de valores de saída. Tais ações não acessam nem modificam nenhuma outra informação nem dependem da execução de outras ações - são completamente auto-contidas.

A tabela 2 enumera as principais ações desta categoria.

Tabela 2: Principais ações de computação

Ação	***Descrição***
ApplyFunction	Computa uma operação matemática primitiva pré-definida
TestIdentity	Compara se dois objetos são na verdade o mesmo objeto
LiteralValue	Gera um valor literal (de tipo primitivo)
Marshal / Unmarshal	Compõe/decompõe o estado de um objeto em um conjunto de valores
Null	Representa uma"no-operation"

Ações de coleções

Uma ação de coleção aplica uma sub-ação a uma coleção de elementos. O benefício principal deste tipo de ação é ocultar o mecanismo exato para acesso aos elementos de uma coleção. Existem quatro tipos diferentes de ações de coleções:

- filtro (*filter*) - para uma determinada coleção de elementos, seleciona aqueles elementos para os quais a sub-ação resulta no valor lógico verdadeiro, produzindo como resultado final uma coleção que é um subconjunto da coleção original;
- iteração (*iterate*) - a sub-ação é aplicada sucessivamente a cada elemento da coleção, produzindo um resultado final;
- mapeamento (*map*) - resulta em uma coleção com o mesmo número de elementos da coleção original, onde cada elemento é o resultado da execução da sub-ação sobre um dos elementos da coleção original;
- redução (*reduce*) - a sub-ação é aplicada sucessivamente a pares de elementos adjacentes na coleção original, gerando, ao final, um elemento escalar do mesmo tipo dos elementos que compõem a coleção.

Dos quatro tipos de ações de coleções, a natureza dos tipos iteração e redução

implica que as múltiplas execuções da sub-ação acontecem seqüencialmente.

Ações de mensagens

Tais ações proporcionam a troca de mensagens entre objetos. O envio inicial de uma mensagem de um objeto para outro é denominado *requisição,* que pode ser síncrona (*chamada*) ou assíncrona *(envio*). O exato modo pelo qual o objeto destinatário irá processar a mensagem recebida é completamente transparente ao remetente. As informações presentes nas mensagens são sempre passadas por valor. No contexto de um programa orientado a objetos convencional, uma requisição é uma solicitação da execução de uma operação. A ação *CallOperation* constitui no meio mais direto de se fazer isso. A tabela 3 enumera as principais ações de mensagens.

Tabela 3: As principais ações de mensagens

Ação	***Descrição***
SynchronousInvocation	Envia uma mensagem ao objeto destino de forma síncrona
AsynchronousInvocation	Envia uma requisição ao objeto destino de forma assíncrona
CallOperation	Envia uma requisição de execução de uma operação

Ações de desvio

Embora seja possível utilizar apenas ações tradicionais para controle de fluxo (possivelmente aninhadas) quando definindo o comportamento de um procedimento, UML provê ações de desvio similares a construções correspondentes disponíveis nas linguagens de programação convencionais, como exceções, *break* e *continue.* A semântica de ações em UML define duas ações primitivas de suporte à introdução de desvios no fluxo de execução: *Jump*, que provoca a interrupção do fluxo de execução corrente, e *Handler,* que captura um desvio gerado por um *Jump* em uma ação subjacente.

2.2.3 Exemplos

Operação *Conta.sacar*

A descrição em pseudo-código da operação *Conta.sacar* aparece na listagem 1, e é auto-descritiva:

```
operação sacar(valor : double)
  início
    saldo <- saldo - valor
  fim
```

Listagem 1: Descrição em pseudo-código da operação Conta.sacar

A especificação deste método em uma linguagem de ações em UML poderia ser feita conforme ilustrado na figura 3. A figura apresenta um diagrama no qual ações estão representadas por retângulos, e setas definem fluxos de dados entre as ações (no sentido indicado). Detalhes em relação aos pinos envolvidos são omitidos. Segue uma explicação básica de cada ação no contexto apresentado:

- a ação *ReadCurrentObject* é utilizada para obtenção do objeto corrente (da mesma forma como as palavras-chave *this* e *self* são usadas nas linguagens de programação). A saída dessa ação é utilizada como entrada em duas outras ações: *ReadAttribute* e *WriteAttribute*;
- a ação *ReadVariable* obtém o valor da variável local *valor* (um parâmetro), e o disponibiliza na sua porta de saída;
- ação *ReadAttribute* produz como saída o valor do atributo associado (*saldo*) para o objeto atual;
- a ação *ApplyFunction* efetua uma subtração entre os valores fornecidos em suas portas de entrada (o atributo *saldo* e o parâmetro *valor*), disponibilizando o resultado em sua porta de saída;
- por fim, a ação *WriteAttribute* escreve o valor resultante da ação *ApplyFunction* no atributo *saldo* do objeto atual.

É importante lembrar que não há garantias quanto à ordem em que as ações são executadas, exceto quando o fluxo de dados impõe um seqüenciamento entre as ações.

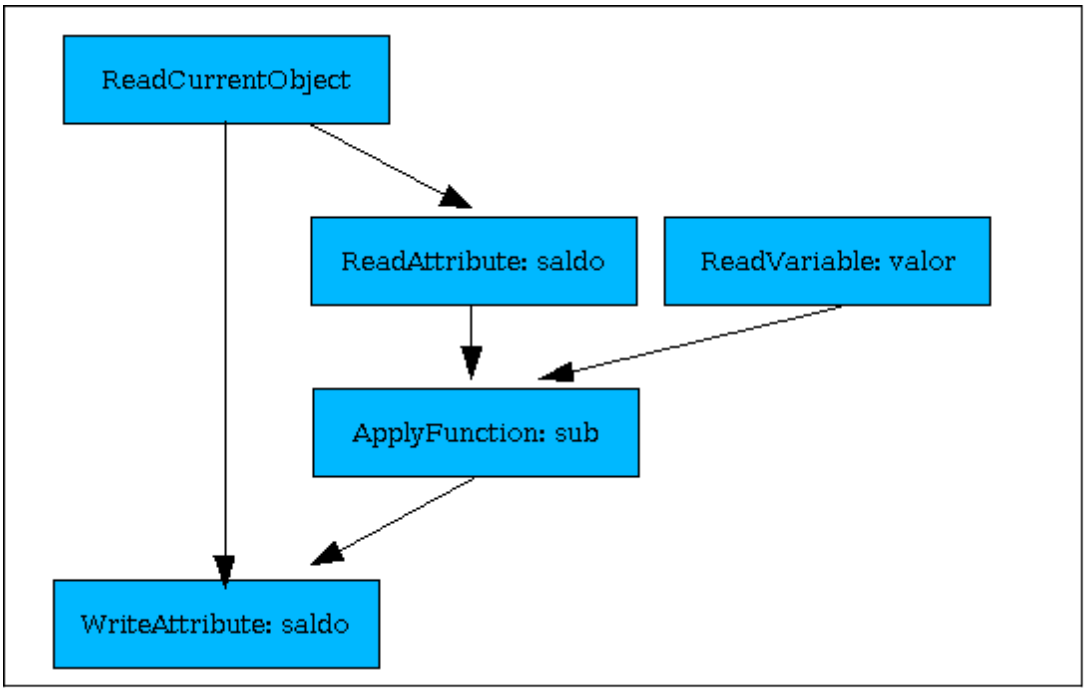


Figura 3: A operação Conta.sacar em uma linguagem de ações UML

3 Orientação a Aspectos

3.1 Conceitos básicos

A orientação a aspectos tem como motivação a constatação de que existem propriedades que um sistema deve implementar que não são modularizáveis segundo os paradigmas atualmente dominantes, como o funcional, o de procedimentos e o de objetos[2]. São preocupações entrecortantes (*crosscutting concerns*) [KICZALES97], precisam ser compostas diferentemente e ainda ser coordenadas. Usualmente, preocupações entrecortantes precisam ser levadas em consideração em vários módulos do sistema, simultaneamente às preocupações específicas de cada módulo. E também são muitas vezes correspondentes a responsabilidades secundárias, não inerentes ao domínio da aplicação. Os exemplos mais comuns de tais preocupações entrecortantes são aquelas referentes a requisitos não-funcionais, tais como a persistência, a distribuição, o paralelismo, a concorrência, a segurança, *logging*, a tolerância a falhas, entre outras.

No caso dos paradigmas convencionais, a composição particular entre propriedades que se entrecortam tem que ser feita manualmente em cada módulo afetado, gerando um emaranhamento entre os trechos de código referentes a cada responsabilidade. A conseqüência desse emaranhamento do código (*code tangling*) [KICZALES97] é um aumento significativo da sua complexidade.

[KICZALES97] separa as propriedades que precisam ser implementadas em um sistema em dois tipos:

- *componentes*, se podem ser facilmente modularizadas segundo o processo de decomposição funcional;
- *aspectos*, em caso contrário. Os aspectos são propriedades que precisam ser satisfeitas em vários componentes de um sistema, como sincronização, persistência, *caching*, etc, ou que não fazem parte das responsabilidades primárias dos componentes.

Assim, o objetivo da orientação a aspectos é possibilitar ao desenvolvedor separar componentes e aspectos de forma bastante clara, e ainda compô-los entre si da maneira apropriada de forma a constituir o sistema desejado.

---

2 Em comum, esses paradigmas têm o fato de se basearem na decomposição funcional como critério principal de projeto, onde cada propriedade a ser implementada é encapsulada em um módulo apropriado.

3.1.1 Composição de um sistema baseado em aspectos

Um sistema baseado em aspectos é composto de: i) uma *linguagem de componentes*, ii) uma (ou mais) *linguagem(ns) de aspectos*, iii) um (ou mais) *programa (s) de componentes*, iv) um ou mais *programas de aspectos*, e v) um *combinador de aspectos* (*aspect weaver*), que é capaz de combinar os programas de componentes e de aspectos de forma a gerar o programa final [PIVETA01].

A linguagem de componentes normalmente é uma linguagem de programação que conforma-se a um dos paradigmas de desenvolvimento de software dominantes, como Java, Smalltalk, C++, Pascal, C, Lisp, etc.

Algumas linguagens de aspectos são genéricas quanto à sua aplicação, como AspectJ [ASPECTJ] e AspectC. Outras são voltadas a preocupações específicas, e nesses casos podem ser usadas múltiplas linguagens de aspectos, uma para cada preocupação. Um exemplo do uso de múltiplas linguagens de aspectos aparece em D [LOPES97], onde são usadas Ridl (para distribuição) e COOL (para concorrência).

Embora em sistemas legados isso normalmente não ocorra, idealmente, o código do programa de componentes deve ser escrito tendo-se em mente que as responsabilidades entrecortantes devam ser deixadas para os aspectos. Dessa forma, os benefícios da orientação a aspectos (abordados adiante) ficam mais claros.

Os programas de aspectos, que podem estar implementados em múltiplas linguagens de aspectos, implementam as responsabilidades entrecortantes. O código do aspecto torna explícito o comportamento que será integrado ao código dos componentes, e em que contextos tal integração ocorre: são os chamados *pontos de junção* (*join points*), que são elementos semânticos da linguagem de componentes com as quais os programas de aspectos se coordenam. Exemplos de pontos de junção comuns são: invocações de métodos (chamadas ou recebimentos), geração de exceções, criação de objetos, entre outros.

O combinador de aspectos identifica nos componentes pontos de junção onde os aspectos se aplicam, produzindo o código final da aplicação, que implementa tanto as propriedades definidas pelos componentes como aquelas definidas pelos aspectos. Combinadores podem atuar em tempo de compilação ou de execução. Implementações de combinadores em tempo de execução têm a possibilidade interessante de permitir a adição / exclusão de aspectos com a aplicação em pleno funcionamento.

3.1.2 Modificações contribuídas por aspectos

Aspectos efetivamente alteram os componentes sobre os quais atuam, podendo contribuir com propriedades entrecortantes de natureza estrutural e/ou comportamental.

Exemplos de *propriedades entrecortantes estruturais* (quando componentes são classes) são a adição de operações e atributos a classes existentes, ou a criação de um relacionamento (associação, herança) entre uma classe e outra.

Exemplos de *propriedades entrecortantes comportamentais* são a modificação de um comportamento existente de forma a incluir aquele que é especificado pelo aspecto, ou até mesmo a total substituição de um comportamento existente por outro definido pelo aspecto.

3.1.3 Associações entre aspectos e componentes

Em [KERSTEN99], são definidas quatro possibilidades de associação entre um aspecto e um componente que seja afetado pelo aspecto: *fechada* (o aspecto e o componente não se referenciam), *aberta* (aspecto e componente se referenciam mutuamente), *dirigida ao componente* (somente o aspecto referencia o componente) e *dirigida ao aspecto* (somente o componente referencia o aspecto).

Em linhas gerais, o acoplamento entre aspectos e componentes compromete a compreensão do código e o reuso. As associações aberta e dirigida ao aspecto têm conseqüências indesejáveis, porque não permitem aos componentes afetados operarem na ausência do aspecto. Não que, na prática, a funcionalidade do componente não possa ser dependente daquela provida pelo aspecto (por exemplo, considere o aspecto de autenticação em um sistema bancário). Mas essa dependência deveria se limitar ao nível semântico, e não à sintaxe. Dessa forma, seria possível a aplicação do componente em cenários mais simples, como durante a prototipação ou testes de unidades.

Além disso, é preferível que a associação entre aspectos relacionados a requisitos não-funcionais e componentes seja sempre fechada. A especificação de quais componentes, especificamente, o aspecto irá afetar deveria ser uma decisão do usuário do aspecto, e não do desenvolvedor. Assim, a reusabilidade dos aspectos seria preservada.

3.1.4 Benefícios da orientação a aspectos

Como principais vantagens da adoção da orientação a aspectos, o código do sistema

como um todo se torna mais fácil de escrever, compreender, reutilizar e manter. As propriedades entrecortantes não aparecem no texto dos componentes, ficando devidamente encapsuladas nos aspectos. Os componentes e aspectos ficam mais simples, e a maneira como eles se compõem fica mais clara.

Durante a criação de um programa com aspectos, o desenvolvedor de componentes pode desconsiderar quaisquer preocupações que não aquelas pertinentes às responsabilidades primárias dos componentes (seus requisitos funcionais), simplificando bastante a tarefa. E como os aspectos fatoram código que normalmente estaria espalhado e replicado por vários locais da aplicação, o tamanho do código-fonte também é reduzido.

A compreensão do código dos componentes fica facilitada, já que o código dos componentes tende a ser mais limpo e simples. A compreensão do código dos aspectos também é mais fácil, porque o código referente a cada responsabilidade entrecortante está adequadamente encapsulado em um aspecto correspondente. E os pontos de junção aos quais os aspectos se aplicam usualmente ficam mais explícitos.

O reuso do código na programação baseada em aspectos é sensivelmente mais efetivo, dado que o componente torna-se reutilizável em uma maior variedade de contextos, pois é isento de dependências relativas a requisitos adicionais impostos pela aplicação para a qual foi originalmente concebido.

A manutenção é também menos propensa a erros, já que a integração com código referente a responsabilidades adicionais é feita pelo combinador, através da análise automática dos pontos de junção afetados por cada aspecto, e não mentalmente pelo desenvolvedor. Por exemplo, o desenvolvedor que mantém o código não precisa lembrar de confirmar a transação corrente no banco de dados toda vez que escrever um método que altere um objeto persistente, já que a persistência e a transacionalidade dos objetos é implementada por aspectos específicos. Caso os seletores de pontos de junção forem implícitos, pontos de junção adicionados posteriormente serão automaticamente incluídos no domínio de atuação do aspecto.

3.1.5 Problemas da composição de aspectos

Existe um problema intrínseco da orientação a aspectos, relacionado à interação entre dois ou mais aspectos diferentes que se aplicam a um mesmo ponto de junção. Em [PAWLAK01] são identificadas algumas diferentes categorias de problemas de

composição de aspectos, das quais duas são destacadas aqui:

- problemas de compatibilidade - os aspectos que se compõem podem ser incompatíveis. Por exemplo, os aspectos implementam propriedades relacionadas, o que pode gerar redundância (os aspectos implementam a mesma funcionalidade desnecessariamente) ou inconsistência (a funcionalidade de um aspecto invalida a do outro);
- problemas de ordenamento - os aspectos implementam propriedades que, embora sejam compatíveis, possuem restrições de precedência entre si.

A problemática da composição de aspectos é maximizada quando se considera um ambiente onde diversos aspectos de diferentes fornecedores são combinados em uma aplicação. Idealmente, deve existir um mecanismo que identifique e proíba a combinação de aspectos incompatíveis, e que possibilite a definição de regras de precedência entre aspectos quando necessário.

3.1.6 Considerações

Parece seguro crer que a orientação a aspectos tem méritos suficientes para se estabelecer como uma ferramenta auxiliar no desenvolvimento de software orientado a objetos, melhorando as potencialidades de abstração, encapsulamento e reuso desse paradigma.

Enquanto é evidente que os proponentes da orientação a aspectos acreditam que as preocupações entrecortantes, para as quais o novo paradigma foi criado, não se limitem aos requisitos não-funcionais, é bastante clara a importância da sua aplicação nesse contexto.

3.2 Programação orientada a aspectos em AspectJ[3]

AspectJ é uma linguagem para programação orientada a aspectos de uso geral criada pelo laboratório Palo Alto Research Center, da Xerox, e hoje mantida pelo projeto de código aberto Eclipse [ECLIPSE].

Segundo [KICZALES01], de forma a facilitar a sua adoção pela comunidade de desenvolvedores, AspectJ foi concebida como uma extensão à Java, sendo um superconjunto desta linguagem. Além disso, todo programa em AspectJ, ao ser

3 Esta seção é referente à versão 1.0 da linguagem.

compilado, é passível de execução em qualquer máquina virtual Java (as extensões são limitadas à sintaxe da linguagem).

Assim, em uma aplicação orientada a aspectos em AspectJ, os componentes são implementados usando a sintaxe padrão de Java, e os aspectos são implementados usando a sintaxe de AspectJ.

3.2.1 Modelo de pontos de junção

O modelo de pontos de junção de AspectJ é baseado em um grafo dirigido de envio de mensagens a objetos [KICZALES01]. Os nós são os pontos onde as classes e objetos recebem uma invocação de método ou têm um atributo acessado. As arestas representam as relações de fluxo de controle entre os nós (orientadas do chamador para o chamado). O fluxo de controle, na realidade, ocorre nos dois sentidos: no sentido definido pela aresta, quando o sub-nó será ativado, e no sentido contrário, quando a computação realizada pelo sub-nó estiver concluída.

3.2.2 Selecionando pontos de junção

Em AspectJ, um *ponto de atuação* (*pointcut*) é definido através de um PCD (*pointcut designator*, ou designador de ponto de atuação). Existem diversos PCDs pré-definidos na linguagem (PCDs primitivos), mas o programador pode elaborar novos PCDs (nomeados ou não) a partir dos existentes.

Os PCDs primitivos dividem-se entre aqueles que selecionam pontos de junção de um determinado tipo (como, por exemplo, recepções de mensagem, leituras de atributo, escritas de atributo), aqueles que selecionam pontos de junção que satisfazem uma determinada propriedade (em função da classe do objeto onde o ponto de junção ocorre, por exemplo), e os de composição, que permitem criar PCDs mais complexos em função dos PCDs existentes . Seguem tabelas identificando todos os PCDs primitivos de AspectJ, dividindo-os nos três grupos:

Tabela 4: PCDs baseados no tipo do ponto de junção

PCD	**Pontos de junção selecionados**
call*(Operação)*	Quando o método é chamado
execution(*Operação*)	Quando o método é executado
get(*Atributo*)	Quando o atributo é acessado
set(*Atributo*)	Quando o atributo é alterado

handler(*TipoExceção*)	Quando a exceção é tratada
initialization(*Construtor*)	Quando o construtor é executado
staticinitialization(*Tipo*)	Quando a inicialização de classe é executada

Tabela 5: PCDs baseados em propriedades do ponto de junção

PCD	**Pontos de junção selecionados**
within(*Tipo*)	Qualquer PJ que ocorra na classe
withincode(*Método*)	Qualquer PJ que ocorra no método/construtor
cflow(*PCD*)	Qualquer PJ que ocorra no contexto de um PJ selecionado pelo PCD
cflowbelow(*PCD*)	Idem ao anterior, excluindo os PJs selecionados pelo próprio PCD
this(*Tipo*)	Qualquer PJ que ocorra em um objeto da classe
target(*Tipo*)	Quando o objeto alvo do call/get/set é da classe
args(*Tipo, ...*)	Quando os argumentos são do tipo especificado
if(*ExpressãoLógica*)	Quando a expressão é verdadeira

Tabela 6: PCDs de composição

PCD	**Pontos de junção selecionados**
!*PCD*	Qualquer PJ não selecionado pelo PCD
PCD1 && PCD2	Qualquer PJ selecionado por ambos os PCDs
PCD1 || PCD2	Qualquer PJ selecionado por ao menos um dos PCDs

Os PCDs aceitam curingas na especificação dos pontos de junção a serem selecionados. Por exemplo, a expressão abaixo seleciona os pontos de junção onde quaisquer métodos que retornarem *String* sejam invocados.

```
call(String *..*(..))
```

Já a expressão a seguir seleciona os pontos de junção onde quaisquer atributos de objetos da classe Cliente ou derivadas sejam alterados.

```
set(* modelo.Cliente+.*)
```

O usuário pode criar seus próprios PCDs, de forma a possibilitar seu reuso. A palavra-chave *pointcut* é usada para definir um PCD:

```
[abstract] pointcut Id([ListaParametros]) [: PCD];
```

Um PCD pode ser abstrato, quando deve omitir a sua definição (que segue os dois pontos). A existência de um PCD abstrato obriga o aspecto a ser abstrato. Deve haver então um sub-aspecto concreto que especialize o aspecto abstrato e forneça uma definição para os PCDs abstratos. Tanto PCDs primitivos como de usuário podem ser usados na

definição dos PCDs de usuário.

3.2.3 Acrescentando comportamento

Como dito anteriormente, os PCDs identificam os pontos de atuação de um aspecto. Já os *atuadores* definem em que momento e como o aspecto irá atuar no ponto de junção. Atuadores em AspectJ são blocos de código não-identificados que definem o comportamento a ser executado pelo aspecto nos pontos de junção associados.

Atuadores são definidos em AspectJ como segue:

```
TipoAtuador([ListaParâmetros]) : PCD {
    Corpo
}
```

Como tipos de atuadores, AspectJ provê as palavras-chave *before*, *after*, e *around*, que definem a relação do atuador em relação ao ponto de junção. Dessa forma, um atuador pode ser definido como executando, respectivamente, antes, após, ou ao invés de um determinado conjunto de pontos de junção definidos pelo PCD associado.

Os atuadores com designador *around* são os que têm maior poder de influência sobre o código dos componentes, pois substituem o comportamento original do ponto de junção. Para ativar o comportamento substituído, a linguagem oferece o comando *proceed*`()`. Assim, além de substituir completamente o comportamento original, esses atuadores têm como possibilidade executar comportamentos adicionais antes *e* após o comportamento original do ponto de junção.

Um atuador pode declarar parâmetros, que devem corresponder a parâmetros presentes no PCD. Esses parâmetros ficam disponíveis para manipulação pelo corpo do atuador.

Para um determinado ponto de junção, todos os atuadores cujos PCDs são satisfeitos são executados antes e/ou após o ponto de junção (de acordo com o tipo de cada atuador). A ordem em que os vários atuadores associados são executados é definida em função de fatores como tipo do atuador, especificidade do PCD, e existência de regras de precedência explícitas entre os aspectos que contêm os PCDs.

3.2.4 Alterando a estrutura

As *introduções* possibilitam aos aspectos modificarem classes presentes no sistema, acrescentando métodos, atributos e classes internas, e alterando a hierarquia de herança de

classes e interfaces.

A seguir, vê-se um exemplo que ilustra as diferentes formas de se usar a introdução em AspectJ:

```
...
// declara uma interface para uso pelo aspecto
interface Proxy {
     void setID(Object id);
     Object getID();
}
// todas as classes do pacote "modelo" implementarão a interface
declare parents: modelo.* extends Proxy;

// introduz um atributo privado nas classes do pacote modelo
Object Proxy+.id;

// introduz implementação para os métodos da interface Proxy
public void Proxy+.setID(Object id) {
  this.id = id;
}
public Object Proxy+.getID() {
  return id;
}
...
```

Neste exemplo pode-se observar que os atributos introduzidos nas classes dos componentes estão disponíveis para uso pelos métodos também introduzidos. O exemplo também ilustra que uma interface adicionada pelo aspecto pode ser usada em PCDs para selecionar as classes nas quais tal interface foi declarada como implementada.

3.2.5 Encapsulamento de aspectos

Aspectos na linguagem AspectJ são construções similares a classes, que encapsulam, além dos atributos e métodos convencionais, atuadores, PCDs de usuário, e introduções. Embora o comportamento especificado pelos atuadores e as introduções declaradas modifiquem o sistema ortogonalmente, eles sempre são localizados no código-fonte dos aspectos.

Os aspectos têm a seguinte forma geral:

```
[privileged] [Modificadores] aspect [Id] [ModoInstanciação]
[extends Tipo]
[implements ListaDeTipos]
[dominates ListaDeTipos]
{ Corpo }
```

3.2.6 Herança de aspectos

A herança de aspectos é contemplada em AspectJ, sendo que:

- um aspecto pode derivar de outro aspecto ou de uma classe Java regular;
- um aspecto pode implementar interfaces Java regulares;
- apenas aspectos abstratos (com PCDs abstratos) podem ser especializados.

O mecanismo de herança em AspectJ provê um modo limitado de se implementar aspectos que desconheçam a base de código que entrecortam, ao possibilitar delegar a um aspecto derivado a tarefa de definir concretamente PCDs declarados como abstratos.

3.2.7 Instanciação de aspectos

Por padrão, existe apenas uma instância para cada aspecto. Mas é possível definir-se uma associação por instância de objeto ou ponto de junção selecionados por um PCD, usando os modificadores *perthis, pertarget, percflow, percflowbelow.*

Basicamente, *perthis* e *pertarget* criam uma instância do aspecto para cada objeto que for o objeto corrente nos pontos de junção especificados.

Já *percflow* e *percflowbelow* o fazem sempre que um nó no grafo do fluxo de controle selecionado pelo PCD for visitado.

3.2.8 Compondo aspectos

A linguagem fornece um mecanismo elementar para definição de precedência entre aspectos, através da palavra-chave `dominates`. Exemplo:

```
aspect X dominates Y { ... }
```

Dessa forma, nos pontos de junção em que ambos aspectos se aplicarem, os atuadores do aspecto X serão prioritários sobre os de Y.

Não há qualquer suporte ao controle de compatibilidade entre aspectos.

3.2.9 Considerações

A linguagem AspectJ teve sua versão 1.0 liberada em novembro de 2001. Nessa implementação inicial de AspectJ, verificavam-se as seguintes limitações:

- o combinador opera somente em tempo de compilação. Não é possível incluir e excluir aspectos em tempo de execução;
- o combinador de aspectos opera exclusivamente sobre código-fonte, tanto para aspectos como componentes, o que impede que se tenha bibliotecas pré-compiladas de aspectos e que se aplique aspectos a classes de terceiros distribuídas na forma de *bytecodes*;
- embora seja possível acrescentar novos membros (métodos, atributos, classes internas, etc) às classes existentes, a linguagem não possibilita a criação de novas classes em função das classes existentes. Por exemplo, as tecnologias de objetos distribuídos requerem que um objeto remoto declare uma interface para acesso pelos clientes. Seria interessante que um aspecto pudesse genericamente declarar interfaces para acesso remoto para os métodos públicos de classes de uma aplicação;
- no modelo de aspectos de AspectJ, o aspecto embute tanto a funcionalidade a ser combinada com o código base, como os pontos de junção em que tal combinação se dará. Em um ambiente onde aspectos são desenvolvidos por terceiros, a definição exata dos pontos de junção deveria ser uma decisão de implantação, e não de projeto dos aspectos. A definição de PCDs e aspectos abstratos fornece apenas um suporte mínimo a tal necessidade.

A versão 1.1, lançada em novembro de 2002, entre outras mudanças, acrescentou suporte à combinação de aspectos e classes em formato binário (*bytecodes*).

3.3 Exemplos

Nesta seção, serão apresentados exemplos de usos da orientação a aspectos na adição a uma aplicação Java pura (que lide unicamente com requisitos funcionais) de provisão de responsabilidades não-funcionais com AspectJ. Para tal, foi desenvolvida uma classe de objetos que não leva em consideração qualquer responsabilidade adicional além da representação de um determinado conceito do mundo real, e uma classe cliente que demonstre formas de utilização usuais da primeira.

3.3.1 A aplicação base

A aplicação existente é composta por duas classes:

- uma classe de negócios, que representa a entidade Pessoa, com as propriedades nome, nascimento e idade, e um método que simula um processamento demorado qualquer (Pessoa.java);
- uma classe cliente (que contém o ponto de entrada da aplicação) chamada *TestDriver*, que instancia um objeto da classe Pessoa e faz manipulações básicas sobre este objeto, definindo / acessando propriedades e invocando seus métodos (TestDriver.java).

Para facilitar a compreensão, o esqueleto da classe *Pessoa* aparece a seguir[4].

```
public class Pessoa {
     private String nome;
     private Date nascimento;
     public String getNome() ...
     public void setNome(String nome) ...
     public void processamentoPesado() ...
     public Date getNascimento() ...
     public void setNascimento(Date nascimento)...
     public void setNascimento(String nascimento)...
     public int getIdade()...
}
```

E o corpo do método principal da classe *TestDriver*:

4 Listagens de todas as classes e aspectos mencionados nesta seção aparecem no Anexo.

```
    public static void main(String args[]) throws Exception {

        // instancia a classe Pessoa
        Pessoa p = new Pessoa();

        // define a propriedade nome
        p.setNome("Fabiane");

        // define a propriedade nascimento
        p.setNascimento("26/01/1974");

        // realiza um processamento demorado
        p.processamentoPesado();

        // apresenta ao usuário o valor da propriedade nome
        System.out.println("Nome: " + p.getNome());

        // apresenta ao usuário o valor da propriedade idade
        System.out.println("Idade: " + p.getIdade());
    }
```

O objetivo então é demonstrar que preocupações ligadas a requisitos não–funcionais como assincronismo e distribuição são mais bem representadas através do conceito de aspectos, possibilitando a combinação de tais aspectos com uma aplicação existente, de forma a torná-la automaticamente assíncrona e distribuída, sem qualquer modificação direta do seu código.

3.3.2 Exemplo: um aspecto para rastreio do fluxo de execução

Para demonstrar a forma como os aspectos produzidos no experimento alteram o comportamento da aplicação, será usada uma das aplicações clássicas de aspectos: o rastreio do fluxo de execução (*tracing*). Aplicando-se um aspecto para rastreamento das invocações de métodos (inclusive construtores) na classe *Pessoa* (Log.aj), o resultado da aplicação seria:

```
[20:02:37'870] Iniciando modelo.Pessoa()
[20:02:37'870] Concluindo modelo.Pessoa()
[20:02:37'870] Iniciando void modelo.Pessoa.setNome(String)
[20:02:37'870] Concluindo void modelo.Pessoa.setNome(String)
[20:02:37'920] Iniciando void modelo.Pessoa.setNascimento(String)
[20:02:37'920] Concluindo void modelo.Pessoa.setNascimento(String)
[20:02:37'920] Iniciando void modelo.Pessoa.processamentoPesado()
[20:02:40'940] Concluindo void modelo.Pessoa.processamentoPesado()
[20:02:40'940] Iniciando String modelo.Pessoa.getNome()
[20:02:40'940] Concluindo String modelo.Pessoa.getNome()
Nome: Fabiane
[20:02:40'940] Iniciando int modelo.Pessoa.getIdade()
[20:02:41'0] Concluindo int modelo.Pessoa.getIdade()
Idade: 28
```

Como se pode observar no histórico de rastreamento acima (onde cada linha informa o momento em que foi emitida), o primeiro ponto de junção capturado trata-se da instanciação da classe *Pessoa.* Em seguida, no objeto recém criado, são invocados os métodos *setNome(String), setNascimento(String), processamentoPesado(), getNome()* e *getIdade().* O nome e idade obtidos são ecoados na saída padrão.

3.3.3 Exemplo: um aspecto para assincronismo

Este exemplo demonstra a transformação de uma aplicação convencional em assíncrona. Para isso, criou-se um aspecto que captura os pontos de junção de invocação do método *processamentoPesado* da classe *Pessoa* (na verdade, de classes em seu pacote), usando o PCD primitivo *call.* A este PCD associou-se um atuador do tipo *around*, que intercepta todas as invocações selecionadas, iniciando um fluxo de execução em separado para atendê-las, retornando o controle para o chamador logo em seguida, possivelmente antes mesmo que o processamento invocado tenha sido concluído.

Segue o resultado da aplicação incluindo o aspecto de assincronismo (Async.aj):

```
[20:04:58'920] Iniciando modelo.Pessoa()
[20:04:58'920] Concluindo modelo.Pessoa()
[20:04:58'920] Iniciando void modelo.Pessoa.setNome(String)
[20:04:58'920] Concluindo void modelo.Pessoa.setNome(String)
[20:04:58'970] Iniciando void modelo.Pessoa.setNascimento(String)
```

```
[20:04:58'970] Concluindo void modelo.Pessoa.setNascimento(String)
[20:04:58'970] Iniciando void modelo.Pessoa.processamentoPesado()
[20:04:58'970] Iniciando String modelo.Pessoa.getNome()
[20:04:59'30] Concluindo String modelo.Pessoa.getNome()
Nome: Fabiane
[20:04:59'30] Iniciando int modelo.Pessoa.getIdade()
[20:04:59'30] Concluindo int modelo.Pessoa.getIdade()
Idade: 28
[20:05:01'990] Concluindo void modelo.Pessoa.processamentoPesado()
```

Observa-se que a conclusão do método *processamentoPesado()* se dá *após* a conclusão do fluxo normal da aplicação.

3.3.4 Exemplo: um aspecto para distribuição

Neste exemplo, a idéia é tornar uma aplicação distribuída, fazendo que a classe *Pessoa* seja apta à distribuição. A aplicação resultante compreenderá então classes tanto locais como remotas, sem que seja necessário levar em consideração que uma infraestrutura para o suporte de objetos distribuídos está em uso, como é necessário quando se desenvolve aplicações em CORBA (que requer a publicação e obtenção explícitas de objetos em um serviço de nomes, declaração de interfaces com IDL, etc) e Java RMI (que requer que se defina uma interface derivada de *java.rmi.Remote,* que a classe servidora derive de *UnicastRemoteObject,* etc).

Segue o resultado da aplicação combinada com o aspecto de distribuição (Remoto.aj), na parte cliente:

```
[20:12:56'160] Iniciando modelo.Pessoa()
[20:12:56'160] Concluindo modelo.Pessoa()
[20:12:57'760] Iniciando void modelo.Pessoa.setID(Object)
[20:12:57'760] Concluindo void modelo.Pessoa.setID(Object)
[20:12:57'760] Iniciando Object modelo.Pessoa.getID()
[20:12:57'760] Concluindo Object modelo.Pessoa.getID()
[20:12:57'810] Iniciando Object modelo.Pessoa.getID()
[20:12:57'810] Concluindo Object modelo.Pessoa.getID()
[20:12:57'870] Iniciando Object modelo.Pessoa.getID()
[20:12:57'870] Concluindo Object modelo.Pessoa.getID()
[20:13:00'940] Iniciando Object modelo.Pessoa.getID()
[20:13:00'940] Concluindo Object modelo.Pessoa.getID()
Nome: Fabiane
[20:13:01'0] Iniciando Object modelo.Pessoa.getID()
[20:13:01'0] Concluindo Object modelo.Pessoa.getID()
Idade: 28
```

E na parte servidora:

```
[20:12:57'700] Iniciando modelo.Pessoa()
[20:12:57'700] Concluindo modelo.Pessoa()
[20:12:57'810] Iniciando void modelo.Pessoa.setNome(String)
[20:12:57'810] Concluindo void modelo.Pessoa.setNome(String)
[20:12:57'870] Iniciando void modelo.Pessoa.setNascimento(String)
[20:12:57'870] Concluindo void modelo.Pessoa.setNascimento(String)
[20:12:57'920] Iniciando void modelo.Pessoa.processamentoPesado()
[20:13:00'940] Concluindo void modelo.Pessoa.processamentoPesado()
[20:13:01'0] Iniciando String modelo.Pessoa.getNome()
[20:13:01'0] Concluindo String modelo.Pessoa.getNome()
[20:13:01'50] Iniciando int modelo.Pessoa.getIdade()
[20:13:01'50] Concluindo int modelo.Pessoa.getIdade()
```

Observa-se que nesse caso, no processo cliente, apenas o construtor e os métodos *getID()* e *setID(Object)* da classe *Pessoa* são executados. As invocações aos métodos são redirecionadas para o servidor, sendo lá efetivamente executadas.

Para implementação desse exemplo foi construído um protocolo bastante rudimentar para invocações remotas de métodos, usando Java RMI como base. Isso foi necessário porque (como visto anteriormente nesta seção), com RMI, seria necessária a criação de uma interface especial para acesso remoto, o que atualmente não é possível se fazer automaticamente com AspectJ.

3.3.5 Considerações

Apesar dos exemplos aqui apresentados sejam de natureza bastante simples, pode-se observar o quão elegantemente a orientação a aspectos é capaz de lidar com o problema da separação de preocupações. Ao longo deste seção, viu-se o programa exemplo ser gradualmente incrementado com funcionalidades anteriormente não existentes, através da combinação com aspectos, sem qualquer alteração no código da aplicação base.

3.4 Modelagem orientada a aspectos

Embora inicialmente o foco no desenvolvimento orientado a aspectos fosse a programação, na atualidade existem diversas propostas para o uso de aspectos nas fases de análise e projeto.

A grande maioria dos trabalhos enfoca a extensão de UML (a notação orientada a objetos mais popular na indústria) para inclusão do suporte a aspectos.

Esta seção descreve sumariamente as características básicas de algumas das abordagens existentes que se baseiam em UML, dando destaque para duas das mais

completas: Theme/UML e AODM.

3.4.1 Theme/UML

Theme/UML [CLARKE02a] é uma proposta de modelagem orientada a aspectos que deriva do paradigma de programação orientada a assuntos (*subject-oriented programming*).

Cada assunto é um modelo correspondente a um requisito individual do sistema, contendo apenas os elementos de um modelo de objetos relativos àquele requisito. Por exemplo, uma mesma classe pode aparecer em vários assuntos diferentes, mas diferentes características serão definidas em cada assunto.

A composição dos diferentes assuntos pode se dar através de combinação ou redefinição. Na combinação, os assuntos correspondem a requisitos diferentes e todos precisam estar presentes no modelo resultante. Na redefinição, um novo assunto provê modificações a um assunto existente.

Notação

A construção primária definida por Theme/UML para comportar a separação de preocupações é o *padrão de composição.* Padrões de composição são pacotes parametrizados (gabaritos) que utilizam o estereótipo <<*subject*>> e declaram um diagrama de classes relacionadas incompleto, com elementos a serem substituídos através da instanciação do gabarito. Com o provimento de elementos concretos, o padrão de composição é instanciado, impondo modificações aos elementos envolvidos.

No caso do exemplo da figura 4, um padrão de composição é usado para a aplicação do padrão de projeto *Observador* [GAMMA00].

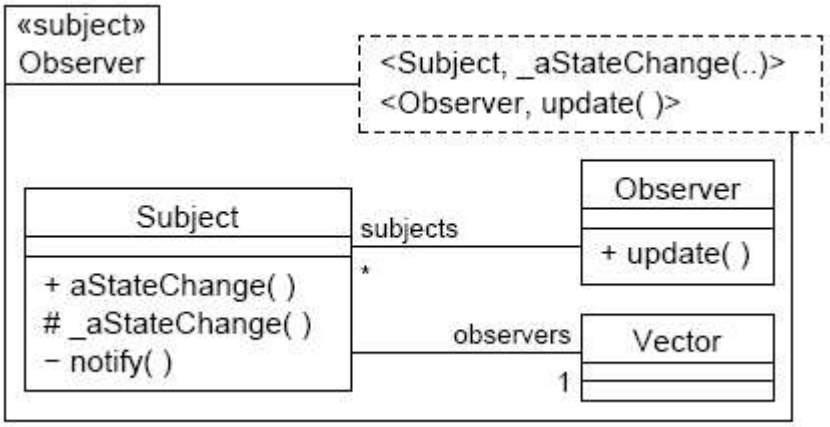


Figura 4: Padrão de composição para Observer [CLARKE02b]

O padrão do exemplo acima têm como parâmetros dois pares de elementos, o primeiro formado por uma classe que assume o papel de sujeito e uma operação que representa uma mudança de estado, e o segundo por uma classe que assume o papel de observador e uma operação nessa classe que reage a mudanças no sujeito.

Padrões de composição podem então ser combinados com pacotes que contém os objetos do nível base através de um relacionamento de vinculação (uma instanciação do gabarito).

Definindo propriedades entrecortantes

Propriedades estruturais

Propriedades estruturais entrecortantes em Theme/UML são definidas separadamente em padrões de composição. A composição dos diferentes assuntos que compõem o sistema dá origem ao sistema completo.

Propriedades comportamentais

Padrões de composição também são contêineres para comportamento entrecortante. O padrão de composição declara uma redefinição para uma operação definida em outros assuntos (representada como um parâmetro), e a redefinição da operação pode invocar a operação original.

Considerações sobre Theme/UML

Um dos destaques de Theme/UML está em fornecer um modelo para especificação

de propriedades entrecortantes de maneira independente de linguagem. No entanto, [OVLINGER03] identifica conflitos no mapeamento de Theme/UML para AspectJ. Tais conflitos são derivados das diferenças conceituais entre as linguagens de modelagem (baseada na orientação a assuntos) e programação (baseada na orientação a aspectos).

Uma característica fraca da abordagem é sua incapacidade de compor módulos (componentes/aspectos) através de expressões regulares. Tal limitação força o projetista a especificar a composição entre aspectos e componentes de maneira explícita, o que tende a ser mais trabalhoso e difícil de manter.

3.4.2 Aspect Oriented Design Model

Aspect-Oriented Design Model [STEIN02] é uma abordagem que enfatiza a fase de projeto de aplicações orientadas a aspectos, estendendo UML de forma a contemplar os conceitos desse paradigma de acordo como encontrados em AspectJ.

AODM utiliza apenas os mecanismo de extensão a UML convencionais, e por isso não requer suporte específico nas ferramentas CASE para contemplar aspectos.

Além disso, o modelo define um modelo de projeto orientado a aspectos que pode ser transformado em um modelo de objetos UML convencional.

Notação

Em AODM, aspectos são representados por classes utilizando o estereótipo <<*aspect*>>. Como variações do conceito de classes, aspectos podem conter operações e atributos próprios e participar de associações e hierarquias de herança com outras classes.

Valores etiquetados (*tagged values*) são usados nos aspectos para definir a forma de instanciação (uma instância por objeto afetado, por fluxo de controle afetado ou uma única instância) e a visibilidade dos elementos privados da classe afetada a partir do aspecto.

A dominância entre aspectos é representada através de um relacionamento de dependência entre os aspectos usando o estereótipo <<*dominates*>>.

Seletores de pontos de junção (PCDs) são definidos como operações sinalizadas com o estereótipo <<*pointcut*>>.

Definindo propriedades entrecortantes

Propriedades estruturais

AODM utiliza gabaritos de colaboração para representar propriedades entrecortantes estruturais.

Colaborações descrevem como um determinado conjunto de objetos interage de forma a atingir um objetivo mútuo. Colaborações fornecem uma visão de um sistema maior, apresentando apenas os objetos envolvidos e as associações entre eles [UML].

Gabaritos são elementos do modelo UML que são parametrizados. Gabaritos em si não são elementos UML válidos, mas através do fornecimento de argumentos (também elementos UML) na instanciação do gabarito, um novo tipo de elemento é criado.

Assim, um gabarito de colaboração apresenta uma ou mais classes relacionadas que podem ser parametrizadas.

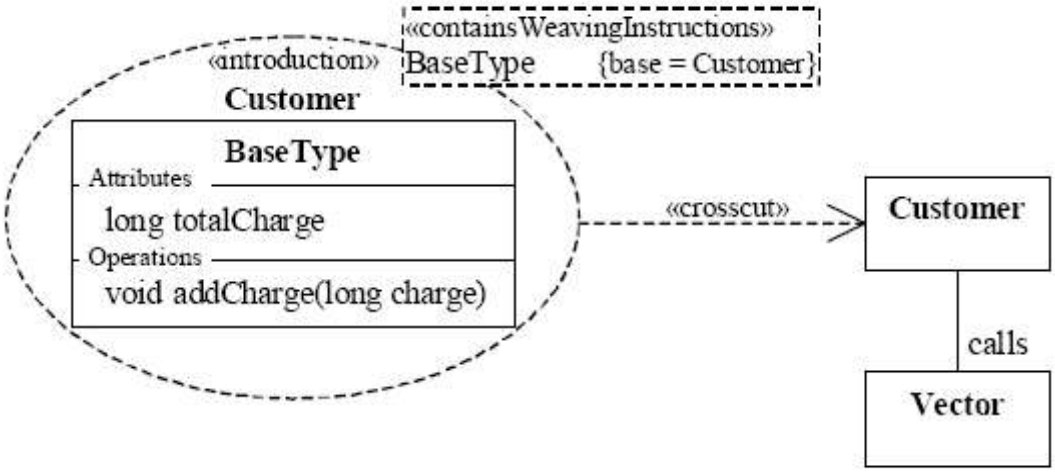


Figura 5: Exemplo de gabarito de colaboração [STEIN02]

Propriedades comportamentais

Do ponto de vista dinâmico, os pontos de junção em modelos UML, segundo AODM, são as ligações que representam a existência de transmissão de estímulos entre objetos (por exemplo, o envio de uma mensagem). De forma a preservar a correspondência direta com AspectJ, ações internas que não são relacionadas à comunicação entre objetos (como acesso a atributos, a inicialização de objetos e a execução de métodos) precisam ser modeladas como interações, usando estereótipos como <<*get*>>, <<*set*>> (para acesso a atributos), <<*initialize*>> (para a inicialização de

objetos), e <<*execute*>> (para execução de métodos).

A especificação de comportamento entrecortante é feita em diagramas de interação. Tal comportamento é a realização de um determinado atuador. Atuadores, assim como em AspectJ, definem, além do comportamento a ser executado, os pontos de atuação e a ocasião de atuação (por exemplo, antes, após, ou ao invés da ação original no ponto de junção selecionado).

Considerações sobre AODM

Os pontos fortes e fracos de AODM derivam principalmente da forte ligação dessa abordagem com o modelo conceitual de AspectJ. Por um lado, AODM herda uma fundação conceitual sólida do projeto de AspectJ. Por outro, peca pelo excesso de detalhes relativos a implementação, o que compromete sua utilidade como uma notação de uso geral (independente de linguagem) para modelagem orientada a aspectos. No entanto, [STEIN02a] demonstra que o modelo de pontos de junção de AODM pode ser mapeado para os modelos correspondentes nas técnicas de filtros de composição, programação adaptativa e hiper-espaços.

Não relacionado com a proximidade com AspectJ, uma característica relevante de AODM é a sua compatibilidade com UML e a preocupação com a preservação da semântica original dos elementos estendidos.

3.4.3 Outras propostas pra modelagem orientada a aspectos em UML

Basch e Sanchez

Também bastante voltada à implementação em AspectJ, a proposta de [BASCH03] baseia-se em dois princípios:

- aspectos devem ser apropriadamente modularizados, separados do sistema como um todo;
- pontos de junção são cidadãos de primeira classe, definidos separadamente dos aspectos e componentes (associação aspecto-componente é fechada).

Tal trabalho demonstra o uso de conceitos orientados a aspectos em diagramas de classes, seqüência e estados. Para isso, baseia-se fortemente na definição de elementos específicos para a representação de aspectos e pontos de junção. Se, por um lado, tal

abordagem consegue uma expressividade maior para os conceitos relacionados a aspectos, o não uso dos mecanismos convencionais para extensão de UML compromete a sua aceitação como uma notação padrão.

Zakaria et al.

Outra abordagem que compartilha da fundação conceitual de AspectJ, a proposta de [ZAKARIA02] objetiva a modelagem orientada a aspectos de nível básico durante a fase de projeto. Características dessa abordagem são:

- aspectos são classes com o estereótipo *<<aspect>>*, que podem ser abstratos;
- aspectos são anotados com as marcas *{Active aspect}* e *{Passive aspect}* caso modifiquem o comportamento original nos pontos de junção afetados ou apenas interceptem tais eventos para fins de análise/depuração, respectivamente;
- aspectos e as classes afetadas são vinculados através do uso de associações convencionais UML que utilizam valores etiquetados para descrever a natureza da vinculação:
  - *{control}* – o aspecto controla o comportamento da classe afetada;
  - *{track}/{report}* – aspecto apenas captura pontos de junção para fins de análise/depuração;
  - *{customize}* – aspecto customiza a classe afetada, configurando-a para um cenário específico;
  - *{validate}* – aspecto faz validação através do acréscimo de pré-condições;
  - *{save}* – aspecto responsável por auferir persistência à classe;
  - *{handleError/Exception}* – aspecto responsável por tratar condições de erro no sistema.
- a seleção de pontos de junção de interesse é feita através de uma classe com estereótipo *<<pointcut>>*, que é ligada ao aspecto e à classe afetada através de associações *<<hasPointcut>>*;
- é possível definir precedência entre aspectos através de uma associação entre eles com o estereótipo *<<dominates>>;*
- atuadores são operações com os estereótipos *<<after>>*, *<<before>>*, *<<around>>*, *<<after returning>>*, *<<after throwing>>*, com a mesma semântica de AspectJ, embora não seja descrito como atuadores são relacionados

a PCDs.

Nessa proposta, a modelagem de aspectos visa o mapeamento para uma linguagem de programação orientada a aspectos. A combinação entre aspectos e componentes deve ocorrer no nível da liguagem de programação.

Um ponto questionável do trabalho é a inclusão de elementos relativos a semântica de preocupações entrecortantes específicas, como persistência e depuração. Não é claro que benefícios são obtidos com a inclusão de tais idiossincrasias.

Kandé et al.

A proposta de [KANDÉ01] tem como principais características:

- aspectos são definidos como classes com estereótipo <<*aspect*>>*;*
- operações e atributos declarados no aspecto são contribuições às classes afetadas;
- atuadores aparecem em um quarto compartimento da classe (além dos compartimentos para nome da classe/estereótipo, atributos e operações);
- os pontos de atuação de um aspecto (*pontos de conexão*, na nomenclatura do trabalho) são representados por classes com estereótipo <<*pointcut*>>; tais classes são agregadas ao aspecto que pertencem e conectadas aos componentes afetados por relacionamentos que utilizam o estereótipo <<*binding*>>.

Aldawud et al.

A proposta de [ALDAWUD02] visa a criação de um perfil UML para contemplar a orientação a aspectos. Um perfil UML é um conjunto de extensões à UML relacionado ao suporte à modelagem em um domínio específico. Características da proposta:

- aspectos são classes usando o estereótipo <<*aspect*>>;
- aspectos anotados com a marcação *{synchronous=true}* controlam o comportamento dos componentes afetados. *{synchronous=false}* denota um aspecto que atua nos pontos de junção de interesse sem interferir no comportamento original;
- aspectos síncronos devem definir operações especiais denominadas *Preactivation* e *Postactivation;*
- associações com estereótipo <<*crosscut*>> identificam que componentes são afetadas pelos aspectos.

Uma das contribuições mais interessantes desse trabalho é a definição da natureza de sincronismo na atuação de aspectos.

Groher e Schulze

A proposta de [GROHER03] tem como uma das principais metas o reuso do código dos aspectos e dos componentes, separadamente, bem como o suporte ao mapeamento automático de um modelo orientado a aspectos em uma implementação orientada a aspectos.

- aspectos são representados como pacotes – a justificativa seria que uma preocupação muitas vezes envolve várias classes;
- a composição entre aspectos e componentes é feita através de relacionamentos entre pacotes usando o estereótipo <<*use*>>, orientado no sentido aspecto-componente;
- *conectores* vinculam aspectos e componentes de forma independente da tecnologia de aspectos subjacente utilizada (por exemplo, PCDs, atuadores, e introduções são providos pelo conector AspectJ).

Entre as contribuições mais importantes dessa proposta, está a clara separação entre a representação de componentes, aspectos, e da composição entre eles. Além disso, o uso de um mecanismo abstrato para composição (conectores) é bastante interessante, no sentido que viabiliza a utilização das idéias básicas da modelagem combinadas com o suporte específico à linguagem de implementação para definição das regras de composição.

Mellor

O trabalho de [MELLOR03], um especialista da área de modelos UML executáveis, inicia uma investigação sobre a união de modelos e aspectos, e a combinação de modelos contendo aspectos através de funções de mapeamento que compreendam pontos de junção.

Sunyé et al.

Em [SUNYÉ01], analisa-se o potencial do uso de ações para manipulação do meta-nível. O trabalho explora, além de aspectos, aplicações como *refactoring* e padrões de

projetos. No contexto da orientação a aspectos, as ações são usadas para definir regras para combinação de modelos via transformação.

3.4.4 Considerações

As propostas descritas acima variam em termos de abordagem, finalidade e base conceitual. Mesmo assim, é interessante analisá-las em conjunto, destacando semelhanças e diferenças.

Nível de composição (modelo x implementação)

A principal vantagem da composição ainda no nível de modelagem é que ela permite a transformação para linguagens de programação convencionais, dado que construções relativas a preocupações entrecortantes desaparecem durante a composição. Tanto [STEIN02] como [CLARKE02b] fazem uso de gabaritos para implementar a composição no nível de modelos. Outras abordagens preferem delegar o processo de composição para o nível da linguagem de programação.

Representação de aspectos (classes x pacotes)

[CLARKE02b] e [GROHER03] representam aspectos como pacotes, enquanto que as outras abordagens (muitas delas derivadas de AspectJ) representam aspectos como classes.

Especificação da composição entre aspectos e componentes

A separação da especificação da composição entre aspectos e componentes dos aspectos e componentes propriamente ditos é essencial para possibilitar o reuso dos mesmos em contextos diferentes. [GROHER03], [BASCH03] e, de certa forma, [STEIN02] provêem construções em separado para representação das regras de composição.

Correspondência entre linguagem de modelagem e implementação

Uma linguagem de modelagem, para ter utilidade geral, deve prover mapeamento para diferentes linguagens de programação compatíveis com o mesmo paradigma. [CLARKE02b] e [GROHER03] têm como premissas básicas a independência de

linguagem de programação. A maioria das outras abordagens apresentadas nesta seção sobre modelagem orientada a aspectos é claramente voltada à implementação em AspectJ. É importante notar que, embora a base conceitual em [STEIN02] seja explicitamente derivada de AspectJ, o fato da composição poder ocorrer no nível da modelagem (o que elimina quaisquer construções relativas a aspectos) viabiliza o mapeamento do modelo resultante para puro Java, ou até mesmo para outras linguagens orientadas a objetos.

Compatibilidade com UML

Essencial para possibilitar adoção em larga escala, a compatibilidade com UML é claramente uma preocupação central na proposta de [STEIN02]. A preservação da compatibilidade requer grande cuidado na extensão e/ou emprego dos elementos existentes de forma a respeitar as restrições semânticas relativas impostas pela notação.

4 Libra: Modelos UML Executáveis Baseados em Aspectos

Este capítulo apresenta Libra, proposta de um conjunto de linguagens para criação de modelos UML executáveis com suporte a aspectos. Libra oferece uma linguagem de ações com capacidades reflexivas para especificação de comportamento, e extensões à UML que contemplam os conceitos relacionados a aspectos na especificação estrutural. Além disso, o conjunto oferece uma estratégia para combinação de aspectos e componentes em modelos visando a tradução para outras linguagens e/ou a execução direta.

4.1 A linguagem de ações

A linguagem de ações de Libra expõe com fidelidade os conceitos definidos pela semântica de ações em UML (vide seção 2.2), como procedimentos, ações e pinos de entrada e saída.

A semântica de ações de UML provê todas as funcionalidades comuns às linguagens orientadas a objeto em geral, como envio de mensagens síncronas ou assíncronas, criação de objetos, controle do fluxo de execução básico (execução condicional e iterativa), exceções, operações aritméticas etc.

A linguagem de ações em Libra é utilizada para definir o comportamento de elementos tanto do modelo de objetos como do de aspectos.

4.1.1 Representação em XML

Procedimentos na linguagem de ações de Libra são definidos usando-se elementos XML [XML]. A escolha de tal formato de representação se deve aos seguintes fatores:

- a construção de tradutores (compiladores, interpretadores, geradores) compatíveis com a linguagem é consideravelmente simplificada, dada a grande disponibilidade de ferramentas para análise/transformação/síntese de documentos XML;
- provê uma representação intermediária universal para procedimentos escritos em linguagens imperativas convencionais.

Embora a legibilidade do código em seu formato nativo seja certamente diminuída, isto não constitui um problema real, dado que, na prática, espera-se que os usuários manipulem uma representação mais amigável, como diagramas ou a linguagem de programação de preferência.

4.1.2 Sintaxe

A sintaxe da linguagem de ações de Libra possibilita a criação de procedimentos. Como visto na seção 2.2, procedimentos são cápsulas de comportamento não-identificadas vinculáveis a elementos do modelo UML. Procedimentos possuem um pino de entrada (argumento), um de saída (resultado), e uma ação que o define, todos potencialmente compostos.

As listagens 2 e 3 ilustram como as operações *Conta.sacar* e *Conta.obterSaldo* poderiam ser descritas utilizando-se a linguagem de ações de Libra.

```
<!-- Conta#sacar -->
<procedure xmi.idref="127-0-0-1-61a907:f92d65a288:-7ff3">
  <input name="valor" parameter="valor"/>
  <ouput name="resultado" parameter="return"/>
  <body>
    <group>
      <write-attribute name="saldo">
        <data-flow input="target">
          <read-self/>
        </data-flow>
        <data-flow input="value">
          <apply-function function="subtract">
            <data-flow input="argument">
              <read-attribute name="saldo">
                <data-flow input="target">
                  <read-self/>
                </data-flow>
              </read-attribute>
            </data-flow>
            <data-flow input="argument">
              <read-variable name="valor"/>
            </data-flow>
          </apply-function>
        </data-flow>
      </write-attribute>
    </group>
  </body>
</procedure>
```

Listagem 2: Um procedimento que descreve a operação Conta.sacar

```
<!-- Conta#obterSaldo -->
<procedure xmi.idref="127-0-0-1-61a907:f92d65a288:-7fec">
  <ouput name="resultado" parameter="return"/>
  <body>
    <group>
      <write-variable name="resultado">
        <data-flow input="value">
          <read-attribute name="saldo">
            <data-flow input="target">
              <read-self/>
            </data-flow>
          </read-attribute>
        </data-flow>
      </write-variable>
    </group>
  </body>
</procedure>
```

Listagem 3: Um procedimento que descreve a operação Conta.obterSaldo

Como se pode observar, os parâmetros da ação são declarados como pinos de entrada e o resultado da ação como pinos de saída.

O elemento <*body*> delimita o conjunto de ações que define o comportamento do procedimento.

O fluxo de dados entre ações é representado através do uso do elemento <*data-flow*>, cujos subelementos provêm valores para o pino de entrada identificado pelo atributo *input*. Por exemplo, se a saída de uma ação *A* é usada como entrada para o pino *x* de uma ação *B*:

```
<B>
  <data-flow input="x">
    <A/>
  </data-flow>
</B>
```

A natureza hierárquica de XML impede que uma mesma instância de uma ação apareça como entrada de múltiplas ações[5] (o que é permitido na semântica de ações de UML). Uma maneira simples de se contornar essa limitação é através da repetição da ação que provê a saída desejada aninhada dentro de cada ação que a tem como entrada (como ocorre nos exemplos anteriores com a ação *read-self*). Por exemplo, se a saída de *A* é utilizada como entrada para *B* e *C*:

5 Tal incapacidade de representar grafos de fluxo de dados arbitrários é certamente comum às linguagens de programação baseadas em texto em geral.

```
<B>
  <data-flow input="x">
    <A/>
  </data-flow>
</B>
<C>
  <data-flow input="y">
    <A/>
  </data-flow>
</C>
```

Para casos onde múltiplas execuções de uma determinada ação não é desejada (por exemplo, a ação é não-idempotente, ou seu custo de execução é alto), a solução é utilizar variáveis locais para armazenar temporariamente a informação a ser compartilhada. Por exemplo, veja a listagem 4.

```
<procedure>
  <local>
    <pin name="valorDeA"" type="integer"/>
  </local>
  <body>
    <!-- ações internas são seqüenciadas -->
    <group sequential="true">
      <write-variable name="valorDeA">
        <data-flow input="value">
          <A/>
        </data-flow>
      </write-variable>
      <!-- ações internas NÃO são seqüenciadas -->
      <group>
        <B>
          <data-flow input="x">
            <read-variable name="valorDeA"/>
          </data-flow>
        </B>
        <C>
          <data-flow input="y">
            <read-variable name="valorDeA"/>
          </data-flow>
        </C>
      </group>
    </group>
  </body>
</procedure>
```

Listagem 4: Um procedimento com variáveis locais e seqüenciamento

A seção *local* acima declara pseudo-pinos, um conceito particular à linguagem de ações de Libra, que são variáveis locais, visíveis somente no procedimento onde declaradas.

Como o uso compartilhado de uma variável (ou atributo) não impõe seqüenciamento como ocorre quando há fluxo de dados entre ações, é necessário que se

faça o seqüenciamento explicitamente, através de um fluxo de controle. A instrução *<group sequential="true">* representa um fluxo de controle entre as instruções que estão *imediatamente* aninhadas, na ordem em que aparecem.

O seqüenciamento é necessário de forma que as ações *B* e *C* só sejam executadas após a variável *valorDeA* ter seu valor definido. Mas não há interesse em se impor seqüenciamento entre as ações *B* e *C*. Para se evitar o seqüenciamento indesejado, uma nova sub-ação de agrupamento (não seqüencial) deve ser utilizada.

4.1.3 Vinculação de procedimentos a elementos do modelo

A vinculação de procedimentos com os elementos correspondentes no modelo UML pode ser feita de diversas maneiras. Idealmente, o texto do procedimento é embutido no próprio modelo. Caso a ferramenta CASE em uso não suporte a definição de procedimentos para elementos comportamentais, uma alternativa é manter os procedimentos em separado. Por exemplo, caso o modelo seja armazenado no formato XMI [XMI], a vinculação pode ser feita através do uso de uma referência ao identificador único global correspondente ao elemento, o *xmi.uuid*, a partir do elemento *<procedure>* (como aparece na listagem 3).

4.2 Extensões à UML para suporte a aspectos

Libra baseia-se simultaneamente em Theme/UML e AODM (e, por conseqüência, AspectJ) para prover extensões a UML de forma a contemplar aspectos e a combinação de aspectos com componentes (i.e., classes) no nível de modelagem.

O exemplo na figura 6 ilustra a maioria dos elementos relativos a aspectos que aparecem em diagramas de classes conforme propostos em Libra.

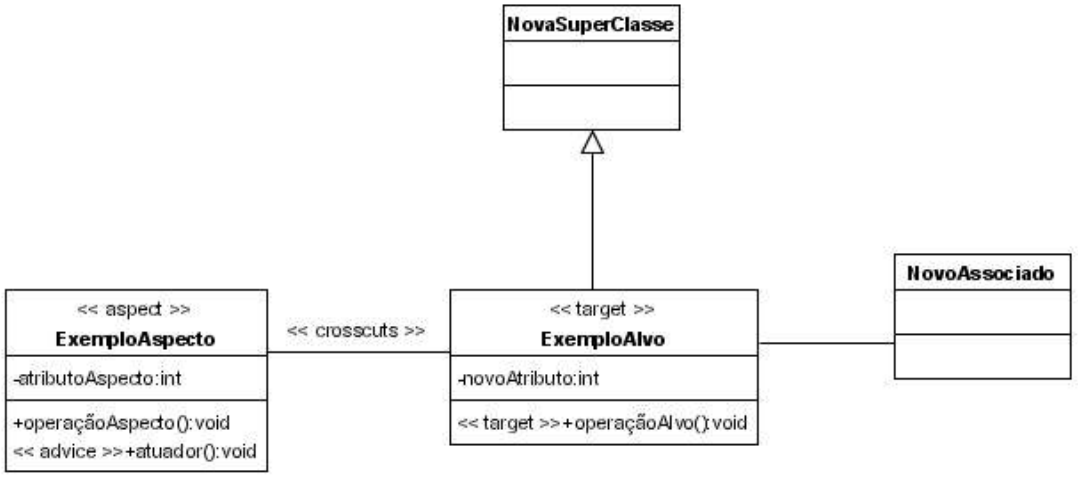


Figura 6: Exemplo de um aspecto em Libra

Um aspecto é uma classe UML com o sugestivo estereótipo <<*aspect*>>.

As associações entre aspecto e componentes são representadas por associações UML com o estereótipo <<*crosscuts*>>.

O estereótipo <<*target*>> sinaliza elementos no modelo de aspectos que representam elementos do modelo de classes sobre o qual o aspecto contribuirá propriedades entrecortantes. Isto possibilita ao aspecto referenciar os elementos do modelo que afeta sem requerer conhecimento das classes envolvidas (a associação aspecto-componente é dita *fechada* - vide seção 3.1). Em tempo de transformação, o combinador determina que elementos do modelo de componentes correspondem a que elementos alvo, e as modificações declaradas no modelo de aspectos são efetivamente aplicadas.

Conforme visto ainda na mesma seção 3.1, existem dois tipos principais de modificações contribuídas por aspectos ao modelo de objetos definido pelo usuário: as modificações estruturais e as comportamentais.

Do ponto de vista das modificações estruturais, aspectos podem introduzir novos atributos, operações, e relacionamentos (tanto de herança como de associação) às classes alvo. Tais modificações são representadas pela simples declaração dos elementos nas classes alvo. Por exemplo, como ilustrado na figura 6, o aspecto *ExemploAspecto* declara um atributo, uma operação, uma super classe e uma associação com uma outra classe.

Já do ponto de vista da contribuição de modificações comportamentais, os elementos alvo permitem que expressões de seleção de pontos de junção sobre os quais

um determinado atuador opera (mais sobre pontos de junção em Libra a seguir) sejam independentes do modelo de componentes. No entanto, pontos de junção podem também referenciar elementos específicos do modelo se desejado (embora normalmente não recomendado, pois compromete a reusabilidade).

4.2.1 Modelo de pontos de junção

O modelo de pontos de junção de Libra é bastante simples, possuindo apenas um tipo de designador: a execução de uma ação.

Aspectos podem definir atuadores que executam antes, depois, ao invés da ação ou assincronamente a partir da execução da ação.

A exata forma como a seleção das ocorrências de interesse de uma determinada ação é especificada depende do tipo de elemento do modelo UML sobre o qual ela atua. Por exemplo, para uma ação que atua sobre um atributo (como *ReadAttribute* ou *WriteAttribute*), o atributo é usado na seleção. O mesmo se aplica a ações que atuam sobre operações (como *CallOperation* ou *CallProcedure*), ou sobre classes (como *CreateObject* ou *Handler*).

A diversidade e a granularidade das ações primitivas disponíveis na semântica de ações de UML, cobrindo praticamente toda a funcionalidade necessária em um programa orientado a objetos, faz com que o modelo de pontos de junção de Libra seja bastante rico (pois todas as ações são expostas).

Comparando-se com o modelo de pontos de junção de AspectJ, é possível traçar-se uma correspondência entre a maioria dos designadores de pontos de junção primitivos daquela linguagem com o designador de pontos de junção universal de Libra aplicado a uma determinada ação. A tabela 7 relaciona alguns dos designadores de pontos de junção de AspectJ e as correspondentes ações UML a serem capturadas pelo designador universal de Libra.

Tabela 7: PCDs AspectJ e as correspondentes ações em UML

PCD em AspectJ	***Ação correspondente em UML***
call(<operação>)	CallOperation ou CreateObject (quando operação é “new”)
execution(<operação>)	CallProcedure
get(<atributo>)	ReadAttribute
set(<atributo>)	WriteAttribute

PCD em AspectJ	*Ação correspondente em UML*
handler(<tipo>)	Handler

4.3 O processo de combinação de aspectos e componentes

As modificações estruturais em Libra são realizadas através da superposição de uma ou mais classes providas pelo aspecto a classes selecionadas no modelo de componentes, acrescentando operações, atributos, superclasses e associações.

Já as modificações comportamentais são implementadas através de atuadores contribuídos pelos aspectos que designam comportamento que complementa, ou mesmo substitui, o existente em pontos de junção selecionados.

Em ambos os casos, cabe ao usuário definir a quais classes do seu modelo de objetos os aspectos devem atuar. Tal informação é representada em Libra na forma de um modelo auxiliar, que orienta o processo de combinação.

4.3.1 Associando aspectos e componentes

Como visto no capítulo 3, as associações entre aspectos e componentes devem ser feitas de forma que haja um alheamento recíproco. Além disso, é interessante que o mecanismo destinado a associar aspectos e componentes possibilite a definição das associações de maneira tão genérica quanto possível. Assim, reduzem-se as chances de que mudanças no modelo de objetos requeiram atualização da especificação das associações.

Tome-se como exemplo o mecanismo de designação de pontos de junção de AspectJ, que provê o uso de curingas. A expressão *package1.*.get** especifica todos os métodos *get* de todas as classes do pacote *package1*. Caso um novo método *get* ou uma nova classe no pacote *package1* sejam criados, eles serão automaticamente incluídos na seleção.

Além disso, um mesmo aspecto pode declarar múltiplas classes-alvo, e isso introduz uma complexidade adicional no processo de combinação. Tal cenário exige que se identifique, para cada ocorrência do aspecto no modelo de objetos, que classe corresponde a que classe-alvo. Por exemplo, considere a aplicação do padrão de projeto comportamental *Observer* [GAMMA00] usando aspectos. Em cada ocorrência desse aspecto, uma classe assumirá o papel de Sujeito, enquanto outra será o Observador. Mas

tais papéis são restritos ao contexto daquela ocorrência. As várias ocorrências do aspecto estabelecem, de maneira inequívoca, a correspondência entre os pares sujeito-observador.

Libra não determina um mecanismo concreto para associação de aspectos e componentes – ao invés disso, especifica, de maneira abstrata, um *modelo de combinação* (vide figura 7), que é responsável por fornecer meta-informações a respeito das associações, como:

- para um dado aspecto, obter todas as suas ocorrências;
- para cada ocorrência de um aspecto, obter as classes do modelo de componentes afetadas (e obter *a* classe que corresponde a uma determinada classe-alvo);
- para cada classe do modelo de componentes afetada em um aspecto, obter todas as características-alvo (como atributos-alvo e operações-alvo).

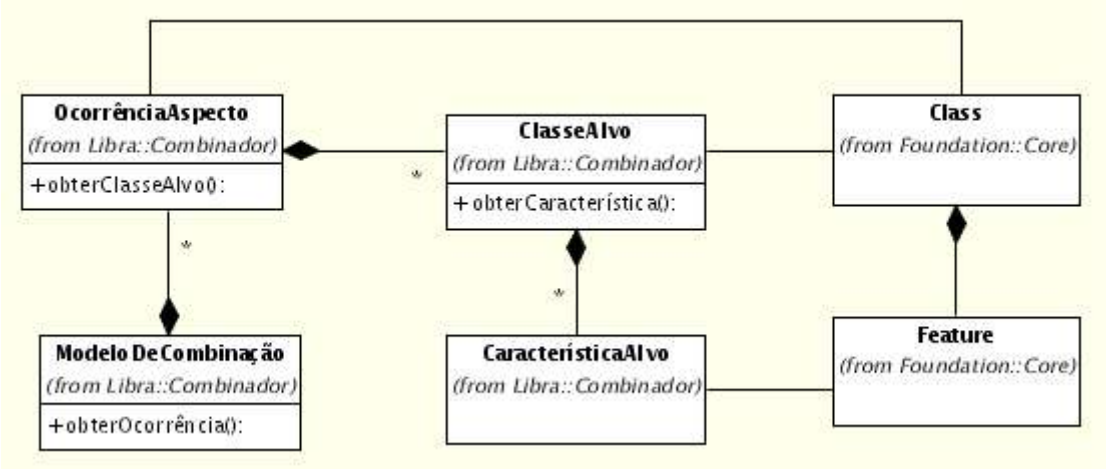


Figura 7: O modelo de combinação de Libra

Assim, um modelo de combinação é uma maneira genérica pela qual Libra obtém acesso a informações relativas a como aspectos e componentes devem ser associados. Descrever a forma exata como tal modelo poderia ser implementado não é objetivo deste trabalho.

4.3.2 O processo de combinação como uma transformação MDA

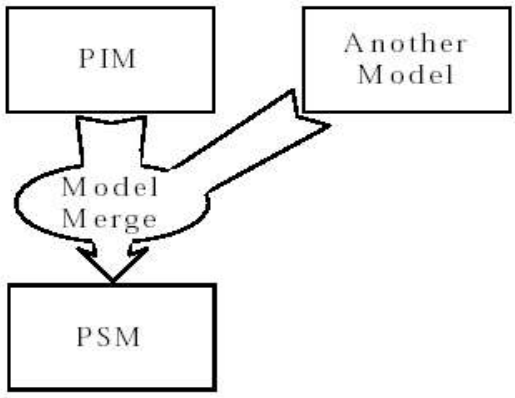


Figura 8: Transformação com combinação de modelos [MDA]

O processo de combinação em Libra segue o formato básico da transformação por combinação de modelos de MDA (vide figura 8). O modelo de componentes e o modelo de aspectos são combinados, e tal processo ocorre segundo orientação de um modelo auxiliar, o modelo de combinação (que não está ilustrado).

O modelo final é o modelo de componentes enriquecido com as propriedades entrecortantes contribuídas pelo modelo de aspectos aos elementos alvo conforme especificado através do modelo de combinação.

4.3.3 Modificações estruturais

As modificações estruturais são realizadas através da sobreposição de uma classe alvo declarada por um aspecto a cada classe correspondente do modelo de componentes.

Cada um dos elementos (atributos, operações, relacionamentos) declarados na classe alvo é acrescentado à classe correspondente. Isso exclui elementos que aparecem vinculados à classe mas que tem significado especial, como aqueles com os estereótipos *<<entrecorta>>* e *<<alvo>>*.

Tais modificações são efetuadas pela manipulação direta do meta-modelo de UML.

4.3.4 Modificações comportamentais

Os pontos de junção na linguagem de ações de Libra são a execução de ações. A seleção dos pontos de junção de interesse em um atuador é feita especificando-se:

- a ação de interesse;

- os parâmetros de interesse para a ação;
- a modalidade de atuação: pré-atuação, pós-atuação, preempção, ou assíncrona.

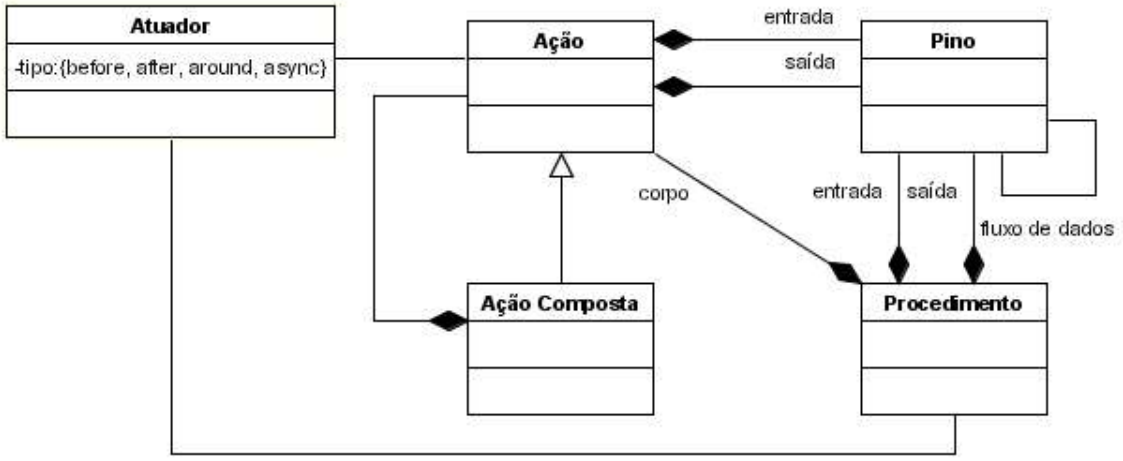


Figura 9: O modelo de semântica de ações de UML modificado

A figura 9 descreve como o modelo de semântica de ações de UML pode ser modificado de forma a comportar atuadores. Para cada ação, estão potencialmente associados múltiplos atuadores. Da mesma forma, um mesmo atuador pode estar conectado a múltiplas ações. O comportamento para um atuador é especificado através de um procedimento (que não pertence ao domínio do atuador).

Assim, do ponto de vista das modificações comportamentais, o combinador da linguagem de ações de Libra tem como função conectar atuadores a ações. Durante a execução do modelo (ou sua transformação), a presença de atuadores altera a forma como as ações são executadas (ou transformadas).

Como exemplo, considere um simples aspecto que conte o número de vezes que é ativado. O procedimento que especifica comportamento poderia ser descrito como na listagem 5.

```
<procedure ...>
  <body>
    <group>
      <write-attribute name="contador">
        <data-flow input="target">
          <read-self/>
        </data-flow>
        <data-flow input="value">
          <apply-function name="add">
            <data-flow name="argument">
              <read-attribute name="saldo">
                <data-flow name="target">
                  <read-self/>
                </data-flow>
              </read-attribute>
            </data-flow>
            <data-flow name="argument">
              <literal-value value="1" type="integer"/>
            </data-flow>
          </apply-action>
        </data-flow>
      </write-attribute>
    </group>
  </body>
</procedure>
```

Listagem 5: Um procedimento descrevendo o comportamento de um atuador

4.4 Exemplos

4.4.1 Exemplo 1: rastreio de execução com aspectos

Neste primeiro exemplo, pretende-se demonstrar o uso de aspectos na sua aplicação mais simples: rastreio de mensagens enviadas entre objetos (vide figura 10).

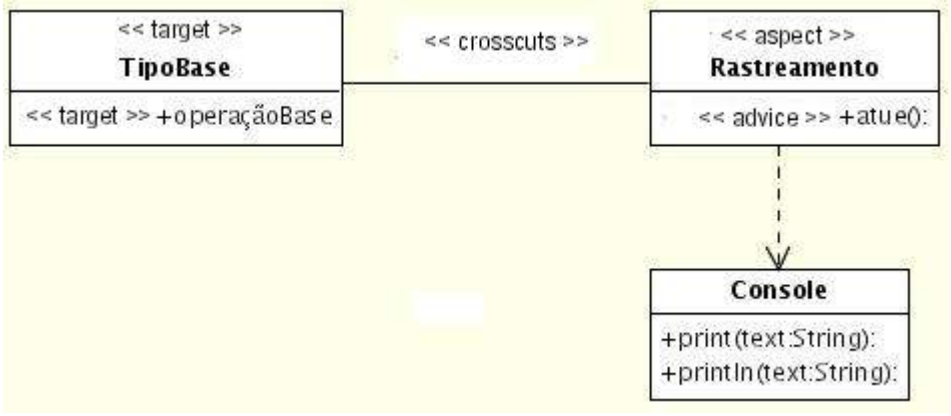


Figura 10: Um aspecto para rastreamento

Apenas uma classe-alvo é afetada, denominada *TipoBase* no modelo. Nessa classe-

alvo, uma operação-alvo é declarada *(operaçãoBase*), representando quaisquer operações do modelo de componentes que o usuário queira rastrear.

O aspecto, denominado *Rastreamento*, declara um atuador, anotado com um valor nomeado *{mode=before}* (não aparece no diagrama).

A implementação do atuador imprime a invocação atual no console.

```
<procedure ...>
  <body>
    <group>
      <call-operation name="println">
        <data-flow input="target">
          <get-classifier name="Console"/>
        </data-flow>
        <data-flow input="argument">
          <call-operation name="toString">
            <data-flow input="target">
              <current-join-point/>
            </data-flow>
          </call-operation>
        </data-flow>
      </call-operation>
    </group>
  </body>
</procedure>
```

Listagem 6: Comportamento do atuador Rastreamento.atue

A ação *currentJoinPoint* é definida por Libra, e provê acesso ao ponto de junção atual. O ponto de junção descreve a ação correspondente (já que pontos de junção em Libra são sempre relativos a execução de uma ação).

A ação *getClassifier* também é definida por Libra, sendo um exemplo de ação que provê acesso ao meta nível de UML. O resultado da ação é uma referência ao classificador dado.

UML não define a semântica de ações quando aplicadas a características (como atributos e operações) cujo escopo é o próprio classificador (como a classe) ao invés de uma instância do classificador. Libra provê tal suporte. Através da ação *getClassifier*, o procedimento pode ter acesso a um classificador arbitrário. E ações que atuam sobre atributos e operações comportam classes como sendo o alvo da ação.

4.4.2 Exemplo 2: aplicando o padrão *Observer* com aspectos

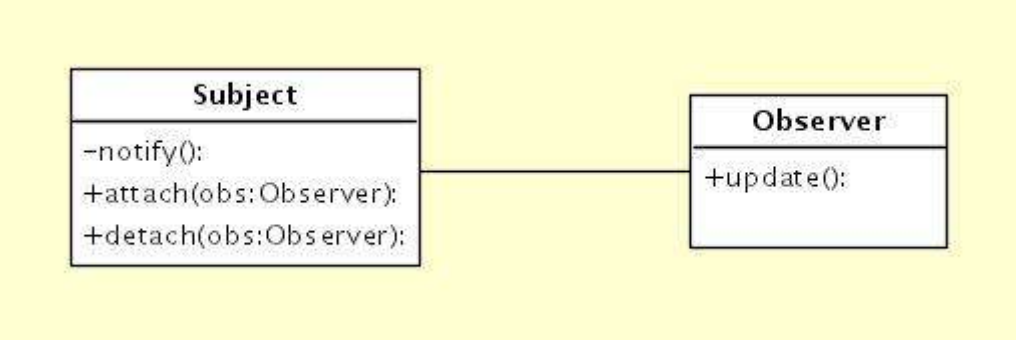


Figura 11: O padrão de projeto *Observer*

A aplicação do padrão *Observer* é exemplo recorrente do uso da orientação a aspectos com padrões de projeto [CLARKE02a] [PIVETA03a]. Este padrão de colaboração descreve um cenário geral onde um objeto (o Observador) precisa reagir a mudanças no estado de um outro objeto (o Sujeito), conforme ilustrado na figura 11.

O participante Sujeito oferece operações para o registro de observadores (*attach*/*detach*) e possui uma operação para uso interno que provoca a notificação de todos os observadores registrados (*notify*). Mudanças no estado interno do Sujeito provocam a notificação dos observadores, que reagem de acordo.

Na aplicação do padrão *Observer* a um modelo de objetos existente, identificam-se os seguintes requisitos:

1. alguns objetos do modelo receberão o papel de observador ou sujeito;
2. o protocolo especificado para cada um dos participantes precisa ser provido pelas classes que assumem tais papéis;
3. mudanças no “estado interessante” de cada sujeito deve causar a notificação dos correspondentes observadores.

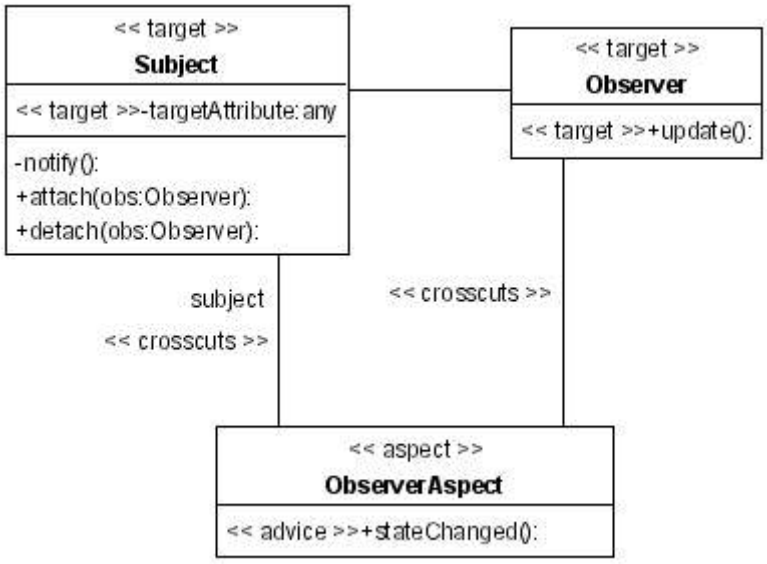


Figura 12: Modelo de aspectos para o padrão *Observer*

A figura 12 descreve um modelo de aspectos que pretende atender a esses requisitos.

Neste exemplo, o aspecto atua sobre duas classes alvo: *Subject* e *Observer.*

Na classe alvo *Subject*, são declaradas as operações *notify*, *attach* e *detach*, que implementam o protocolo de registro de observadores e a notificação. Além dessas operações introduzidas, um atributo-alvo é declarado. Tal atributo representa quaisquer atributos em classes do modelo de componentes representadas por *Subject* cuja alteração deva causar notificação dos observadores.

Na classe alvo *Observer*, a operação-alvo *update* é declarada. Essa operação representa a operação em cada classe do modelo de componentes representada por *Observer* que deva ser invocada no caso em que uma mudança no estado de interesse do *subject* ocorra.

No aspecto *ObserverAspect*, o atuador *stateChanged* é declarado. Tal atuador tem como especficação de pontos de atuação a ação *WriteAttribute* aplicada ao atributo *targetAttribute*, com modo de ativação posterior à execução da ação (os observadores devem ser notificados somente *após* a mudança ter ocorrido).

A implementação do atuador invoca a operação *Subject.notify*, que causa a notificação dos observadores associados.

```
<procedure ...>
  <body>
    <group>
      <call-operation name="notify">
        <data-flow input="target">
          <call-operation name="getCurrentObject">
            <data-flow input="target">
              <current-join-point/>
            </data-flow>
          </call-operation>
        </data-flow>
      </call-operation>
    </group>
  </body>
</procedure>
```

Listagem 7: Comportamento do atuador ObserverAspect.stateChanged

A operação *JoinPoint.getCurrentObject* retorna uma referência ao objeto atual no contexto do ponto de junção atual. No caso, uma instância de uma classe do modelo de componentes que teve seu estado alterado. Tendo tal referência em mãos, a operação *notify* é invocada, provocando a notificação dos observadores.

No modelo de objetos da aplicação exemplo, um uso interessante para o padrão *Observer* seria na criação de uma nova entrada no histórico de transações bancárias toda vez que uma operação alterasse o saldo da conta corrente (depósito ou saque). Dessa forma:

- a classe Conta seria o sujeito, que notificaria seus observadores (para fins de simplificação, apenas um) toda vez que o saldo fosse alterado;
- o observador seria responsável por acrescentar novos objetos Transação ao histórico descrevendo a natureza da operação, o momento em que ocorreu, o valor movimentado e o saldo resultante.

4.5 Considerações

Libra, o conjunto de linguagens proposto pelo autor neste capítulo, tira proveito simultaneamente da notação (referente à modelagem estática) e da semântica de ações de UML para comportar o paradigma de aspectos no contexto de modelos executáveis.

É interessante analisar Libra segundo os mesmos critérios utilizados durante o estudo de outras abordagens para modelagem orientada a aspectos existentes (vide seção 3.4):

- nível de composição (modelo x implementação): a composição em Libra ocorre

no nível da modelagem. Os modelos resultantes do processo de composição são isentos de quaisquer características relativas a aspectos, o que possibilita o mapeamento para plataformas que não suportem aspectos;

- representação de aspectos (classes x pacotes): aspectos em Libra são representados por classes. Tal escolha é influência da proposta evolutiva de AspectJ, onde aspectos são classes com a habilidade de especificar características entrecortantes;
- especificação da composição entre aspectos e componentes: em Libra, a composição entre aspectos e componentes é completamente separada da definição dos dois tipos de módulos – o que significa que aspectos e classes podem ser mais facilmente recombinados. Libra especifica de maneira bastante abstrata como tal especificação é feita. Tal independência em relação ao mecanismo de composição, embora diminua a utilidade imediata de Libra, oferece oportunidades interessantes na especialização do modelo de composição como, por exemplo, o suporte aos PCDs de AspectJ;
- correspondência entre linguagem de modelagem e implementação: a linguagem de ações e a notação gráfica de Libra extendem UML para comportar aspectos, mas não impõem quaisquer conceitos relativos a uma linguagem específica. Teoricamente, modelos em Libra podem ser mapeados para qualquer linguagem (desde que haja um modelo de plataforma provendo a transformação para essa linguagem);
- compatibilidade com UML – neste quesito, a abordagem de Libra é bastante modesta: Libra delega a implementação do modelo de composição para uma entidade externa, e a composição dos modelos de aspectos e componentes é realizada através de reflexão sobre o modelo de objetos de UML. Tais características retiram grande parte da responsabilidade na composição de modelos da notação de UML (contrário do que ocorre com [CLARKE02b], por exemplo).

5 Conclusões e Perspectivas

O presente trabalho demonstra o uso do paradigma de aspectos no contexto de MDA, de forma a obter um maior grau de suporte à separação de preocupações e, por conseqüencia, melhor tolerância a mudanças nos requisitos do sistema e no ambiente computacional. Para esse fim, Libra, um conjunto de linguagens que provê extensões à UML e uma especificação concreta (baseada em XML) para a semântica de ações dessa notação, foi proposto. Libra possibilita a definição completa (dos pontos de vista estrutural e comportamental) de modelos de aspectos e componentes, e a composição posterior desses modelos.

O uso de MDA possibilita a especificação completa de sistemas no nível de modelagem, sem criar quaisquer vínculos com um ambiente computacional específico.

Já o paradigma de aspectos permite que cada requisito da aplicação seja satisfeito de forma modular, dando grande liberdade na forma como os módulos são compostos de forma a gerar o sistema final.

Libra herda características de várias abordagens existentes para modelagem orientada a aspectos em UML. O que Libra provê em adição é o suporte a especificação completa de comportamento. A adoção da semântica de ações de UML ocasionou a criação de um modelo de pontos de combinação ao mesmo tempo simples (um designador de pontos de atuação único) e poderoso (todas as ações estão expostas à introdução de comportamento entrecortante).

5.1 Trabalhos Futuros

A proposta do presente trabalho oferece diversas oportunidades para a continuidade do seu desenvolvimento e a criação de uma gama de ferramentas compatíveis. Exemplos são:

- suporte à especificação de modelos de composição nas ferramentas existentes – a especificação de modelos de combinação concretos é um ponto em que Libra deixa em aberto. Uma abordagem interessante seria a de estender uma ferramenta de modelagem em UML existente de forma a possibilitar a definição das regras de combinação de maneira visual, baseando-se na seleção de elementos dos modelos e a vinculação entre eles;
- uma notação gráfica para especificação de comportamento em Libra – a sintaxe de Libra não foi projetada para favorecer a legibilidade. A semântica de ações de

UML permite a criação de redes complexas de ações, que poderiam ser mais facilmente construídas e compreendidas através de uma ferramenta visual que possibilitasse a seleção de ações e a criação de conexões entre elas de forma a estabelecer o comportamento de um método;

- tradutores de outras linguagens orientadas a objetos e aspectos para Libra – atualmente, não muitos desenvolvedores estariam à vontade usando uma linguagem visual. O uso de linguagens textuais seria a maneira mais provável de favorecer a adoção de Libra em larga escala. Se a sintaxe básica de linguagens existentes, como C, Pascal e Smalltalk pudesse ser utilizada, ainda melhor (quem precisa de outra linguagem de programação?);
- criação de mapeamentos que possibilitariam a transformação de PIMs em Libra para PSMs compatíveis com ambientes computacionais diversos;
- enriquecer o conjunto de ações provido de forma a comportar o desenvolvimento de aplicações reais sobre Libra.

Anexo: código-fonte

Este anexo apresenta o código-fonte completo para os exemplos descritos na seção 3.3. Os exemplos foram compilados e executados com AspectJ versão 1.0 e Java 2 *Software Development Kit* versão 1.3.

Listagem 1. TestDriver.java

```
import java.text.*;
import modelo.*;

/**
 * Classe usuária que demonstra a utilização da classe de
 * negócios Pessoa.
 */

class TestDriver {

    // fluxo principal da classe TestDriver
    public static void main(String args[]) throws Exception {

        // instancia a classe Pessoa
        Pessoa p = new Pessoa();

        // seta a propriedade nome
        p.setNome("Fabiane");

        // seta a propriedade nascimento
        p.setNascimento("26/01/1974");

        // realiza um processamento demorado
        p.processamentoPesado();

        // apresenta ao usuário o valor da propriedade nome
        System.out.println("Nome: " + p.getNome());

        // apresenta ao usuário o valor da propriedade idade
        System.out.println("Idade: " + p.getIdade());
    }
}
```

Listagem 2. Pessoa.java

```
package modelo;

import java.util.Date;

/**
 * Classe de negócios que representa a entidade Pessoa.
 */
public class Pessoa {

/** um formatador para datas */
private static DateFormat format = new SimpleDateFormat("dd/MM/yyyy");

      /** milissegundos em um ano. */
      private final static long MILISSEGUNDOS_POR_ANO =
         1000L * 60 * 60 * 24 * 365;

      /** atributo nome */
      private String nome;

      /** atributo nascimento */
      private Date nascimento;

      /** acessa a propriedade nome */
      public String getNome() {
            return nome;
      }

      /** acessa a propriedade nascimento */
      public Date getNascimento() {
            return nascimento;
      }

      /** simula a execução de um processamento pesado. */
      public void processamentoPesado() {
            try {
          // provoca um retardo de 3 segundos
                Thread.sleep(3000);
            } catch (InterruptedException ie) {
                  ie.printStackTrace();
            }
      }

      /** modifica a propriedade nome */
      public void setNome(String nome) {
            this.nome = nome;
      }

      /** modifica a propriedade nascimento */
      public void setNascimento(Date nascimento) {
            this.nascimento = nascimento;
      }

      /** modifica a propriedade nascimento */
      public void setNascimento(String nascimento) throws ParseException {

            this.nascimento = format.parse(nascimento);
      }

      /** acessa a propriedade idade */
      public int getIdade() {
```

```
            if (nascimento == null) {
                throw new IllegalStateException (
                   "Data de nascimento indefinida."
              );
            } else {
                   // idade = data atual - data de nascimento
                long agora = new Date().getTime();
                return (int)
                      ((agora - nascimento.getTime()) /
                      MILISSEGUNDOS_POR_ANO);
            }
      }

      /** cria uma representação String útil para depuração.*/
      public String toString() {
            return
                "Nome: [" + getNome() + "]" +
                "\nNascimento: [" + getNascimento() + "]" +
                "\nIdade: [" + getIdade() + "]";
      }
}
```

Listagem 3. IMRProtocol.java

```
package imr;

import java.rmi.*;
import java.lang.reflect.*;

/**
 * Protocolo para invocação de métodos em objetos remotos,
 * baseado em RMI.
 */
public interface IMRProtocol extends Remote {

      /** solicita a criação de um objeto remoto. */
      Object create(String className, Object args[])
         throws RemoteException;

      /** invoca uma operação em um objeto remoto. */
      Object invoke(Object id,String methodName,Object args[])
         throws RemoteException;
}
```

Listagem 4. Remoto.aj

```
import org.aspectj.lang.*;
import imr.*;
/**
```

```
 * Este aspecto torna classes existentes em classe aptas à
 * distribuição. As classes selecionadas (no caso, do
 * pacote modelo) podem ser instanciadas remotamente, e o
 * uso desses objetos (acesso a métodos) será feito sempre
 * remotamente. A localização dos objetos é definida por um
 * arquivo de propriedades "imr.ini" como a seguir:
 *
 * # configurações deste servidor
 * imr.localPort=6060
 * imr.name=Server1
 *
 * # classes remotas e respectivas localizações
 * modelo.Pessoa=//localhost:6060/Server1
 * modelo.Banco=//venus:7080/Server2
 * modelo.Empresa=//localhost:6060/Server1
 *
 */

aspect Remoto {

    /** interface auxiliar que representa um objeto remoto */
      interface Proxy {
             void setID(Object id);
             Object getID();
      }

    /**
     * faz com que todas as classes de interesse implementem
     * Proxy
     */
    declare parents: modelo.* extends Proxy;

    /**
     * introdução de um atributo privado nas classes do
     * pacote modelo
     */
    Object Proxy+.id;

    /* introdução dos métodos da interface declarada */
    public void Proxy+.setID(Object id) {
        this.id = id;
    }
    public Object Proxy+.getID() {
        return id;
    }

    /**
     * instanciação de uma classe do pacote modelo que não
     *  seja a partir de IMRServer
     */
    pointcut instanciacao() :
        call(Proxy+.new(..))
        && !cflow(within(IMRServer));

    after() returning (Proxy s) : instanciacao() {
        IMRClient imrClient = IMRClient.getInstance();
        Signature signature = thisJoinPoint.getSignature();
        if (imrClient.isClassRemote(
            signature.getDeclaringType().getName()
        )) {
            s.setID(imrClient.create(
                signature.getDeclaringType().getName(),
```

```
                thisJoinPoint.getArgs()
            ));
        }
    }

    /**
     * invocação de um método em uma classe que não seja
     * apartir de IMRServer
     */
    pointcut invocacao(Proxy s) :
        call(* Proxy+.*(..))
        && target(s)
        && !call(* Proxy+.getID())
        && !call(void Proxy+.setID(Object))
        && !within(IMRServer);

    Object around(Proxy s) : invocacao(s) && args(..) {
        IMRClient imrClient = IMRClient.getInstance();
        Signature signature = thisJoinPoint.getSignature();
        if (imrClient.isClassRemote(
            signature.getDeclaringType().getName()
        )) {
            return imrClient.invoke(
                s.getID(),
                signature.getName(),
                thisJoinPoint.getArgs()
            );
        } else {
            return proceed(s);
        }
    }
}
```

Listagem 5. Async.aj

```
/**
 * Este aspecto determina que invocações de métodos sem
 * valor de retorno sejam executados em uma thread própria
 * (sendo então assíncronos)
 */
aspect Async {
    /**
     * intercepta chamadas a métodos void em objetos de
     * classes do pacote modelo.
     */
    pointcut invocacao() : execution(void modelo..*.processamentoPesado(..));

    /**
     * envolve a chamada original de forma a executá-la
     * em uma thread diferente (gerando assincronismo).
     */
    void around() : invocacao() {
        new Thread() {
            public void run() {
                proceed();
            }
        }.start();
    }
}
```

Listagem 6. Log.aj

```
import java.text.*;
import java.util.Date;

/**
 * Um aspecto que faz um log das invocações a objetos de classes do pacote
 * modelo.
 */
public aspect Log {
    private static final DateFormat horaFmt =
        new SimpleDateFormat("HH:mm:ss''S");

    pointcut recepcao() :
        execution(* modelo..*.*(..)) ||
        execution(modelo..new(..));

    before() : recepcao() {
        System.out.println (
            getTimestamp() + " Iniciando " + thisJoinPoint.getSignature()
        );
    }

    after() : recepcao() {
        System.out.println (
            getTimestamp() + " Concluindo " + thisJoinPoint.getSignature()
        );
    }

    public String getTimestamp() {
        return "["+horaFmt.format(new Date())+"]";
    }

}
```

Listagem 7: IMRServer.java

```
package imr;

import java.util.*;
import java.io.*;
import java.lang.reflect.*;
import java.rmi.server.*;
import java.rmi.*;
import java.rmi.registry.*;

public class IMRServer extends UnicastRemoteObject implements IMRProtocol {
      private Map objects;
      private int localPort;
      private String name;

      public IMRServer() throws Exception {
            File configFile = new File("imr.ini");
            if (!configFile.exists()) {
                  throw new FileNotFoundException("Arquivo de configuracao
<" + configFile + "> nao encontrado");
            }

            Properties prop = new Properties();

            InputStream configIS = new FileInputStream(configFile);
            prop.load(configIS);
```

```
            configIS.close();

            this.objects = new HashMap();
            this.localPort        =        Integer.parseInt(prop.getProperty
("imr.localPort"));
            this.name = prop.getProperty("imr.name");
      Registry registry = LocateRegistry.createRegistry(localPort);
      registry.bind(name,this);
      }
      /**
       * Cria um objeto, retornando um ID ao cliente. Registra o objeto para
invocações
       * posteriores.
       */
      public   Object   create(String   className,   Object   args[])   throws
RemoteException {
            try {
                  Class[] argTypes = new Class[args.length];
                  for (int i = 0; i < argTypes.length; i++) {
                        argTypes [i] = args[i].getClass();
                  }
                Constructor    constructor    =    Class.forName(className).
getDeclaredConstructor(argTypes);
                  Object obj = constructor.newInstance(args);
                  Object id = getID(obj);
                  objects.put(id,obj);
                  return id;
            } catch (Exception e) {
                  e.printStackTrace();
                  throw   new   RemoteException("Erro   instanciando   objeto
remoto",e);
            }
      }
      /**
       * Invoca uma operação em um objeto previamente instanciado.
       */
      public Object invoke(Object id,String methodName,Object args[]) throws
RemoteException {
            try {
            Object obj = objects.get(id);

                  Class[] argTypes = new Class[args.length];
                  for (int i = 0; i < argTypes.length; i++) {
                        argTypes [i] = args[i].getClass();
                  }
               Method   method      =      obj.getClass().getDeclaredMethod
(methodName,argTypes);

                  Object ret = method.invoke(obj,args);
                  return ret;
            } catch (Exception e) {
                  e.printStackTrace();
                  throw    new    RemoteException("Erro    invocando    metodo
remoto",e);
            }
    }
    /**
     * Calcula o id para o objeto especificado.
     * O formato atual e' <class-name>#<hash-code>.
     */
    public Object getID(Object obj) {
      return obj.getClass().getName() + "#" + obj.hashCode();
```

```
    }

    public static void main(String args[]) throws Exception {
      IMRServer imr = new IMRServer();
    }

}
```

Listagem 8: IMRClient.java

```
package imr;

import java.util.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import java.rmi.*;

/**
 * Componente cliente do protocolo IMR. Realiza invocações em objetos
locais/remotos.
 */
public class IMRClient {

      /** coleção de pares (classe,URL) */
      private Map classes;

      /** coleção de pares (id,URL) */
      private Map objects;

      /** referência ao singleton */
      private static IMRClient ref;

      /** método para criação/obtenção de uma referência ao singleton
IMRClient. */
      public static IMRClient getInstance() {
            if (ref == null) {
                  try {
                        ref = new IMRClient();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }
            return ref;
      }


      private IMRClient() throws IOException {
            File configFile = new File("imr.ini");
            if (!configFile.exists()) {
                  throw new FileNotFoundException("Arquivo de configuracao
<" + configFile + "> nao encontrado");
            }

            Properties prop = new Properties();

            InputStream configIS = new FileInputStream(configFile);
            prop.load(configIS);
            configIS.close();

            this.classes = prop;
            this.objects = new HashMap();
      }

      public Object create(String className, Object[] args) {
            String url = (String) classes.get(className);
            if (url == null) {
                  return localCreate(className,args);
            } else {
                return remoteCreate(url,className,args);
          }
```

```
      }
      public Object invoke(Object ref, String methodName, Object[] args) {
            String url = (String) objects.get(ref);
            if (url == null) {
                  return localInvoke(ref,methodName,args);
            } else {
                return remoteInvoke(url,ref,methodName,args);
          }
    }
    private Object localCreate(String className,Object[] args) {
      try {
                  Class[] argTypes = new Class[args.length];
                  for (int i = 0; i < argTypes.length; i++) {
                        argTypes [i] = args[i].getClass();
                  }
                Constructor     constructor = Class.forName(className).
getDeclaredConstructor(argTypes);
                  return constructor.newInstance(args);

      } catch (Exception e) {
            e.printStackTrace();
            return null;
      }
    }
    private Object remoteCreate(String url,String className,Object[] args) {
      try {
            IMRProtocol imr = (IMRProtocol) Naming.lookup(url);
            Object id = imr.create(className,args);
            objects.put(id,url);
            return id;
      } catch (Exception e) {
            e.printStackTrace();
            return null;
      }
    }
    private Object localInvoke(Object obj,String methodName,Object[] args) {
      try {
        Class[] argTypes = new Class[args.length];
        for (int i = 0; i < argTypes.length; i++) {
            argTypes [i] = args[i].getClass();
        }
        Method method = obj.getClass().getDeclaredMethod
(methodName,argTypes);
       return method.invoke(obj,args);
      } catch (Exception e) {
        e.printStackTrace();
        return null;
      }
    }
    private Object remoteInvoke(String url,Object id,String methodName,
Object[] args) {
      try {
            IMRProtocol imr = (IMRProtocol) Naming.lookup(url);
            return imr.invoke(id,methodName,args);
      } catch (Exception e) {
            e.printStackTrace();
            return null;
      }
    }
    public boolean isClassRemote(String className) {
      return classes.containsKey(className);
    }
```

```
}
```

Glossário

"Venha provar meu 'brunch', saiba que eu tenho 'approach'
Na hora do 'lunch', eu ando de 'ferryboat'"
Samba do Approach - Zeca Baleiro

Como a pesquisa em orientação a aspectos é bastante recente, a terminologia utilizada ainda está em constante evolução, e não raro dá margem a interpretações distintas por parte de diferentes autores.

A situação é ainda mais confusa em se tratando da tradução dos termos para o português. Como conseqüência, muitos autores brasileiros preferem utilizar os termos como cunhados originalmente, sem tradução, ou até mesmo utilizam-se exclusivamente da língua inglesa na produção científica [KON02], o que é ainda mais lamentável.

Embora tal estratégia simplifique a vida dos autores, é opinião deste autor que existem, sim, traduções óbvias para muitos dos termos normalmente importados, embora as versões em português possam causar estranheza ao leitor ou simplesmente não tenham o mesmo "*sex-appeal"* dos originais em inglês.

Dessa forma, ao longo deste trabalho, o autor fez uso de termos em português mesmo quando o original em inglês já goza de alguma popularidade no círculo acadêmico brasileiro. Não é, de forma alguma, pretensão do autor propor uma terminologia definitiva para orientação a aspectos em Língua Portuguesa. Mas parece razoável acreditar que a insistência na adoção de traduções para o português (ainda que eventualmente não muito felizes) nas publicações científicas nacionais é importante para fortalecer o uso da combalida Língua Portuguesa no contexto da área de orientação a aspectos, e mais generalizadamente, da Ciência da Computação.

Por isso, este glossário, que tem dois objetivos: definir com precisão o significado dos termos utilizados no contexto do presente trabalho, e possibilitar ao leitor relacionar os termos traduzidos com os originais em inglês.

***

atuador (*advice*) - unidade de comportamento relativo a uma *preocupação entrecortante,* contribuída por um aspecto em *pontos de junção* de interesse de forma modificar o comportamento original. Vide *ponto de atuação.*

combinação de aspectos (*aspect weaving*) - processo de mescla do código de aspectos com o de componentes, introduzindo modificações estruturais e comportamentais nos componentes, de forma a instrumentá-los com código relativo a preocupações entrecortantes.

combinador de aspectos (*aspect weaver*) - um tradutor de programas que realiza *combinação de aspectos.*

emaranhamento de código *(code tangling)* – fenômeno provocado pela inabilidade das linguagens de programação convencionais em modularizar apropriadamente preocupações entrecortantes – o resultado é que o código referente a tais tipos de preocupações fica espalhado por diversos módulos do sistema, misturando-se a código relativo à preocupação dominante e a outras preocupações entrecortantes.

pontos de atuação *(pointcut) – ponto de junção* sobre o qual um determinado aspecto atua. Sempre relativo a um aspecto ou atuador.

ponto de combinação *(join point)* - mesmo que *ponto de junção.*

ponto de junção (*join point*) – ponto do fluxo de execução ou da estrutura estática do sistema onde a composição com aspectos pode ocorrer.

preocupação (*concern*) - responsabilidade de um sistema, requisito. O termo denomina uma abstração relevante em uma solução computacional para um determinado problema. Preocupações são relativas a requisitos (funcionais ou não-funcionais), e aparecem tanto no nível conceitual como na implementação. Muitas vezes, as preocupações em um sistema são dependentes umas das outras (por exemplo, a efetiva redução do saldo durante uma retirada em uma conta bancária e a autenticação do usuário em tal operação), embora normalmente uma delas tenda a ser mais significativa do que as outras (no exemplo, a redução do saldo da conta bancária). Também freqüentemente, algumas preocupações não possuem qualquer relacionamento entre si.

preocupação entrecortante *(crosscutting concern)* - responsabilidade do sistema que afeta múltiplos módulos simultaneamente, em adição às preocupações dominantes de cada módulo.

6 Bibliografia

[ALDAWUD02] ALDAWUD, Omar, ELRAD, Tzilla, BADER, Atef. **UML Profile for Aspect-Oriented Software Development.** In *3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03*. Boston, Massachussets, March, 2003.

[ASPECTJ] **The AspectJ Programming Guide.** The AspectJ Team. 2003. http://eclipse.org/aspectj.

[BASCH03] BASCH, Mark, SANCHEZ, Arturo. **Incorporating Aspects into the UML.** In *3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03*. Boston, Massachussets, March, 2003.

[BEIER02] BEIER, Georg, KERN, Markus. **Aspects in UML Models from a Code Generation Perspective.** In *2nd Workshop on Aspect-Oriented Modeling with UML, UML 2002*. Dresden, Germany, September 30, 2002.

[CHAVES02] CHAVES, Rafael. **Aspectos e Middleware.** Relatório de pesquisa. Universidade Federal de Santa Catarina. Programa de Pós-graduação em Ciência da Computação. Agosto, 2001.

[CHAVEZ02] CHAVEZ, Christina, LUCENA, Carlos. **A Metamodel for Aspect Oriented Modeling.** In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02*. Netherlands, April, 2002.

[CLARK03] CLARK, Tony, EVANS, Andy, KENT, Stuart. **Aspect-Oriented Metamodelling.** The Computer Journal. Volume 46, Issue 5, p.p. 566-577. Oxford Journals, September, 2003.

[CLARKE02a] CLARKE, Siobhán. **Extending standard UML with model composition semantics**. Science of Computer Programming, Volume 44, Issue 1, pp. 71-100. Elsevier Science, July, 2002.

[CLARKE02b] CLARKE, Siobhán, WALKER, Robert. **Towards a Standard Design Language for AOSD**. In *proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April, 2002.

[CLARKE01] CLARKE, Siobhán, WALKER, Robert. **Composition Patterns: An Approach to Designing Reusable Aspects.** In *proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Canada, May 2001.

[DLPO] **Dicionário de Língua Portuguesa On-Line.** Priberam Informática, 2003. http://www.priberam.pt/dlpo.

[ECLIPSE] **Eclipse.org.** 2003. http://www.eclipse.org.

[EJB] **Enterprise JavaBeans Specification, Version 2.0**. Linda G. De Michiel (specification lead). Sun Microsystems, 2001. http://java.sun.com/ejb.

[ELRAD01] ELRAD, Tzilla, AKSIT, Mehmet, KICZALES, Gregor, et al. **Discussing Aspects of AOP**. Communications of the ACM. New York, USA, Volume 44, Issue 10, pages 33-38, October 2001.

[GAMMA00] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph et al. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos.** Tradução de Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000. Tradução de: Design patterns – elements of reusable object-oriented software.

[GROHER03] GROHER, Iris, SCHULZE, Stefan. **Generating Aspect Code from UML Models.** In *3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03*. Boston, Massachussets, March, 2003.

[HARRISON02] HARRISON, William, TARR, Peri, OSSHER, Harold. **A Position On Considerations In UML Design of Aspects**. In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02*. Netherlands, April, 2002.

[HERRMANN02] HERRMANN, Stephan. **Composable Designs with UFA**. In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02*. Netherlands, April, 2002.

[HO02] HO, Wai-Ming, JEZEQUEL, Jean-Marc, PENNANEAC'H, François, et al. **A Toolkit for Weaving Aspect Oriented UML Designs**. In 1st

International Conference on Aspect-Oriented Software Development. Enschede, Netherlands, April, 2002.

[HÜRSCH95] HÜRSCH, Walter, LOPES, Cristina. **Separation of Concerns.** Technical Report NU-CCS-95-03, Northeastern University, 1995.

[KANDÉ01] KANDÉ, Mohamed, KIENZLE, Jörg, STROHMEIER, Alfred. **From AOP to UML – A Bottom-Up Approach.** In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02.* Netherlands, April, 2002.

[KATARA02] KATARA, Mika. **Superposing UML Class Diagrams.** In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02.* Netherlands, April, 2002.

[KC ASL] **UML ASL Reference Guide.** WILKIE, Ian, KING, Adrian, CLARKE, Mike, et al. Kennedy Carter, 2001. http://www.kc.com.

[KERSTEN99] KERSTEN, Mik, MURPHY, Gail. **Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming.** In OOPSLA'1999. 1999.

[KICZALES97] KICZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag et al. **Aspect-Oriented Programming.** In ECOOP'97. 1997.

[KICZALES01] KICZALES, Gregor, HILSDALE, Eric, HUGUNIN, Jim et al. **An Overview of AspectJ.** In ECOOP'01. 2001.

[KON02] KON, Fabio. **Tupi or not Tupi, That's not the Question.** Computação Brasil, SBC, 5 de Novembro de 2002, página 8.

[LOPES97] LOPES, Cristina, KICZALES, Gregor. **D: A Language Framework for Distributed Programming.** Technical Report. Xerox PARC, February, 1997.

[MDA] **MDA Guide Version 1.0.1.** MILLER, Joaquin, MUKERJI, Jishnu (ed.). Object Management Group. http://www.omg.org/mda, 2003.

[MELLOR03] MELLOR, Stephen. **A Framework for Aspect-Oriented Modeling.** In *4th Workshop on Aspect-Oriented Modeling with UML, UML 2003.*

San Francisco, October, 2003.

[OMG] **Object Management Group.** 2003. http://www.omg.org.

[OSSHER02] OSSHER, Harold, TARR, Peri. **Multi-dimensional Separation of Conceerns and the Hyperspace Approach**. *In* Mehmet Aksit(ed.) *Software Architectures and Component Technology.* Norwel, Massachussetts, Kluwer, 2002.

[OVLINGER03] OVLINGER, Johan. **From Aspect-Oriented Model to Implementation: Watch out for Impedance Mismatch**. In *3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03.* Boston, Massachussets, March, 2003.

[PAWLAK01] PAWLAK, Renaud, SEINTURIER, Lionel, DUCHIEN, Laurence et al. **Dynamic Wrappers: Handling the Composition Issue with JAC.** In *Proceedings of TOOLS USA'2001.* 2001.

[PIVETA01] PIVETA, Eduardo. **Um Modelo de Suporte a Programação Orientada a Aspectos.** Dissertação de Mestrado. Universidade Federal de Santa Catarina. Programa de Pós-graduação em Ciência da Computação. 2001.

[PIVETA03] PIVETA, Eduardo, ZANCANELLA, Luiz Carlos. **Architecture of an XML-based aspect weaver**. In *Correctness of Model-based Software Composition (CMC), ECOOP 2003.* Darmstadt, July 2003.

[PIVETA03a] PIVETA, Eduardo, ZANCANELLA, Luiz Carlos. **Observer Pattern using Aspect-Oriented Programming.** In *Third Latin American Conference on Pattern Languages of Programming.* Porto de Galinhas, Brazil, 2003.

[POOLE01] POOLE, John. **Model-Driven Architecture: Vision, Standards and Emerging Technologies.** Position paper submitted to ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models.

[SIMONYI95] SIMONYI, Charles. **The Death of Computer Languages, The Birth of Intentional Programming**. Technical Report, Microsoft Research, 1995.

[STEIN02] STEIN, Dominik, HANENBERG, Stefan, UNLAND, Rainer. **Designing Aspect-Oriented Crosscutting in UML**. In *Workshop on Aspect-Oriented Modeling with the UML, AOSD'02*. Netherlands, April, 2002.

[STEIN02a] STEIN, Dominik, HANENBERG, Stefan, UNLAND, Rainer. **On Representing Join Points in the UML**. In *2nd Workshop on Aspect-Oriented Modeling with UML, UML 2002*. Dresden, Germany, September 30, 2002.

[SUNYÉ01] SUNYÉ, Gerson, PENNANEAC'H, François, HO, Wai-Ming, et al. **Using UML Action Semantics for executable modeling and beyond.** In *13th Conference on Advanced Information Systems Engineering*. Interlaken, Switzerland, 2001.

[UML] **UML specification v1.5.** Object Management Group, March, 2003. http://www.uml.org.

[XMI] **OMG XML Metadata Interchange (XMI) Specification**. Version 1.2. January 2002. http://www.omg.org/technology/xml.

[XML] **Extensible Markup Language (XML) 1.0 (Second Edition)**. World Wide Web Consortium. October 2000. http://www.w3.org/XML.

[ZAKARIA02] ZAKARIA, Aida, HOSNY, Hoda, ZEID, Amir. **A UML Extension for Modeling Aspect-Oriented Systems.** In *2nd Workshop on Aspect-Oriented Modeling with UML, UML 2002*. Dresden, Germany, September 30, 2002.