View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

DIOGO VINÍCIUS WINCK

EXTENSÃO PARA UML DESTINADA À MODELAGEM DE VARIABILIDADE TRANSVERSAL EM COMPONENTES ATRAVÉS DA ORIENTAÇÃO A ASPECTOS

FLORIANÓPOLIS – SC
2005

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

DIOGO VINÍCIUS WINCK

EXTENSÃO PARA UML DESTINADA À MODELAGEM DE VARIABILIDADE TRANSVERSAL EM COMPONENTES ATRAVÉS DA ORIENTAÇÃO A ASPECTOS

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

Vitório Bruno Mazzola

Florianópolis, Junho de 2005

EXTENSÃO PARA UML DESTINADA À MODELAGEM DE VARIABILIDADE TRANSVERSAL EM COMPONENTES ATRAVÉS DA ORIENTAÇÃO A ASPECTOS

Diogo Vinícius Winck

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

Professor Raul Sidnei Wazlawick, Dr.

Banca Examinadora

Professor Vitório Bruno Mazzola, Dr. (Orientador)

Professor Mário Antonio Ribeiro Dantas, Dr.

Professor Ricardo Pereira e Silva, Dr.

Professor Luiz Fernando Rust da Costa Carmo, Ph.D.

Dedico ao meu pai e à minha mãe: a origem;

Para Alexandra Lange: o presente;

Ao futuro.

Agradeço a Kugel,
a Utesc,
ao professor Vitório
e aos demais amigos.

Sou humano e nada humano me é alheio.

Terêncio

ÍNDICE

LISTA DE FIGURAS E QUADROS ix
LISTA DE SIGLAS E ABREVIATURAS x
RESUMO xi
ABSTRACT xii
CAPÍTULO 1 - INTRODUÇÃO 12
CAPÍTULO 2 - Re-visitando o Reúso 15
2.1 Motivações para Reúso 16
2.2 Reúso na Engenharia de Software 17
2.3 Tipos de Reúso 18
2.4 Fatores Inibidores para Reúso 19
2.5 Casos de Sucesso de Reúso 20
CAPÍTULO 3 - Componentização 22
3.1 Níveis de Abstração 23
3.2 Repositório de Componentes 23
3.3 Modelos de Desenvolvimento 24
3.3.1 Desenvolvimento Baseado em Componentes 24
3.3.2 Framework 28
3.3.3 Linhas de Produto de Software 30
Considerações 31
CAPÍTULO 4 - Variabilidade, Compartibilidade e Funcionalidade 33
4.1 Funcionalidade 33
4.2 Compartibilidade (Commonality) 34
4.3 Variabilidade (Variability) 35
4.4 Gerenciando a Variabilidade 37
Considerações 38
CAPÍTULO 5 - Orientação a Aspectos 39
5.1 Fundamentos da Orientação a Objetos 39
5.2 Motivação da Orientação a Aspectos 41
5.3 Interesses 44
5.4 Composição de sistemas 45
5.5 Elementos da Orientação a Aspectos 46
Considerações 47
CAPÍTULO 6 - Trabalhos Relacionados 49

6.1 Abordagem de Suzuki e Yamamoto para Representação de Aspectos na UML . 49
6.2 Abordagem de Herrero et al. para Representação de Aspectos na UML 50
6.3 Proposta de Martin Griss para Linhas de Produtos Orientada a Aspectos 51
6.4 Proposta de Spincyzk e Beuche para Linhas de Produto Orientada a Aspectos .. 52
6.5 Proposta de Camargo, Ramos e Masiero para Implementação de Variabilidades em Frameworks Orientados a Aspectos. 53
Considerações 53
CAPÍTULO 7 - Extensão para UML para Modelagem da Variabilidade Transversal em Componentes 55
7.1 Variabilidade em componentes 56
7.2 Motivação para Uso da Orientação a Aspectos na Modelagem Variabilidade Transversal 57
7.3 Mecanismos de Extensibilidade da UML 57
7.4 Modelagem Estrutural e Comportamental 58
7.5 A Proposta para Extensão da UML 58
7.5.1 O Modelo Funcional-Transversal 59
7.5.2 O Modelo Componente-Variações 60
7.5.3 O Modelo de variação 62
7.5.4 O Modelo de Aplicação 63
Considerações 64
CAPÍTULO 8 - Estudo de Caso 65
8.1 Modelo Funcional-Transversal 67
8.2 Modelo Componente-Variações 68
8.3 Modelo de Variação 69
8.4 Modelo de Aplicação 70
CAPÍTULO 9 - CONCLUSÃO 72
9.1 Aplicabilidade e Limitações da Proposta 72
9.2 Trabalhos Futuros 73
REFERÊNCIAS BIBLIOGRÁFICAS 74
ANEXO A - UML 1.4 (*Unified Modeling Language*) 80

LISTA DE FIGURAS E QUADROS

Figura 3.1 – Modelo para desenvolvimento baseado em componente proposto por. 25
Figura 3.2 – Pontos de especialização. 30
Figura 4.1 – Níveis de Abstração 36
Figura 5.1 – Implementação de um aspecto que compõe o interesse principal de uma aplicação. 42
Figura 5.2 - Implementação de um aspecto que compõe o interesse sistêmico de uma aplicação. 43
Figura 5.3 – Composição de um sistema orientado a aspectos. 46
Figura 6.1 – Exemplo da notação de aspecto proposto por SUZUKI E YAMAMOTO..... 50
Figura 6.2 – Simplificação do modelo e processo de derivação. 53
Figura 7.1 – Estereótipo Funcionalidade Transversal. 60
Figura 7.2 – Estereótipo Composição. 60
Figura 7.3 – Estereótipo Funcionalidade 61
Figura 7.4 – Estereótipo Variação 61
Figura 7.5 – Estereótipo Variação Componente.. 62
Figura 7.6 – Estereótipo Aspecto 63
Figura 7.7 – Estereótipo Variante... 64
Figura 8.1 – Modelo do Estudo de caso. 65
Figura 8.2 - Modelagem de Variabilidade de Remessa de Modo Exaustivo. 66
Figura 8.3 – Modelo Funcional-Transversal 67
Figura 8.4 – Modelo de Componente-Variações.. 69
Figura 8.5 – Modelo de Variação 70
Figura 8.6 – Modelo de Aplicação 71
Figura A.1 – Exemplos de Classes 82
Figura A.2 – Exemplos de Objetos 82
Figura A.3 – Itens da UML. 83
Figura A.4 – Tipos de relacionamento da UML.. 84
Figura A.5 – Formas de representar Estereótipos na UML... 87

Quadro A.1 – Diagramas da UML. 85

LISTA DE SIGLAS E ABREVIATURAS

DBC	Desenvolvimento Baseado em Componentes
EA	Engenharia de Aplicação
ED	Engenharia de Domínio
OA	Orientação a Aspectos
OCL	Object Constraint Language
OO	Orientação a Objetos
TCP	Transmission Control Protocol
UML	Unified Modeling Language

RESUMO

O presente trabalho propõe uma extensão para a UML 1.4 (Unified Modeling Language) destinada à modelagem de variabilidade sistêmica utilizando o paradigma orientado a aspectos através da definição de estereótipos. Estes estereótipos podem ser agrupados em quatro perspectivas: a primeira separando características funcionais e transversais, a segunda modelando as possíveis variações de um componente, a terceira apresentando a implementação de cada variação e a quarta demonstrando a uma aplicação resultante da composição das variações. Esta extensão é aplicável em processos como linhas de produto, framework ou desenvolvimento baseado em componentes.

ABSTRACT

The present work proposes an extension for UML 1.4 (Unified Modeling Language) destined to the modeling of systemic variability using the aspect oriented paradigm through the definition of stereotypes. These stereotypes can be grouped in four perspectives: the first one separating to characteristic functionaries and transversals, second shaping the possible variations of a component, third presenting the implementation of each variation and fourth demonstrating to a resultant application of the composition of the variations. This extension is applicable in processes as lines of product, framework or development based on components.

CAPÍTULO 1 - INTRODUÇÃO

Na modelagem de componentes para composição de softwares é necessário não apenas preocupar-se com componentes capazes de executar uma função clara dentro de um domínio mas também se faz necessário atentar para elementos destinados ao suporte da função primordial. Por uma deficiência do paradigma orientado a objetos, também presente em outros paradigmas como o procedural, não é possível tratar de modo isolado características funcionais (lógica de negócio) das transversais (suporte).

Na implementação dos componentes com o paradigma orientando a objetos, não há uma abstração que permita separar o código destinado ao suporte do local onde é útil de modo tão simples quanto o meio pelo qual se separam as características funcionais. A solução comumente utilizada é a replicação pelos vários componentes das implementações de suporte, tornando-as espalhadas e entrelaçado-as, dificultando a reutilização. Em termos práticos, não se pode apenas ignorar funções como controle de exceções, controle de acesso, maneiras de auditoria, persistência de dados, pois estas características influenciam sobre a estrutura, o comportamento e a utilidade do componente. Entretanto, não há meios para, com os paradigmas orientados a objetos e anteriores, isolar estas funções do restante do código.

É comum necessitar implementações diferentes para questões ligadas a suporte, como, por exemplo, poder utilizar diferentes bancos de dados, gerar registros de auditoria (*log*) em arquivos ou em base dados e assim por diante. Os elementos destinados a implementar estas tarefas espalham-se pelos vários componentes do sistema, e por isso as características

funcionais também são chamadas de características transversais. A variabilidade de características transversais é difícil de ser alcançada porque leva a replicação de componentes.

Este trabalho utiliza como base a orientação a aspectos. A orientação a aspectos disponibiliza abstrações que contribuem para minimizar este problema ao possibilitar isolar em módulos, interesses que antes necessitavam ser programados juntos. Outro fator que favorece a orientação a aspectos é a existência, para implementação, de linguagens maduras como o AspectJ, que utiliza o Java como base. Entretanto há poucos trabalhos sobre linguagens para modelagem, ou extensões da UML, aplicáveis a aspectos ou para representar variações.

Neste contexto, propõe-se uma extensão para UML (versão 1.4) capaz de modelar a variabilidade transversal (sistêmica) em componentes, utilizando a separação de interesses disponibilizada pela orientação a aspectos como suporte, de modo a favorecer a reutilização. Como objetivos complementares, pretende-se delimitar o que seriam funcionalidades[1] (*feature*), compartibilidade[2] (*commonality*), variabilidade[3] (*variability*) tanto transversal quanto funcional[4], avaliar a componentização de software e re-visitar o reúso como uma forma de solucionar algumas das dificuldades da engenharia de software.

O trabalho está estruturado da seguinte forma: o primeiro capítulo avaliará o reúso, suas vantagens e dificuldades. O segundo capítulo abordará a componentização e metodologias para desenvolvimento de software como desenvolvimento baseado em componentes e linhas de produto de software. O capítulo subseqüente, de número três, tratará do conceito de funcionalidade, variabilidade e compartibilidade (*commonality*). A orientação a aspectos, sua origem e seus conceitos, é abordada no capítulo quatro. O quinto capítulo é reservado para apresentar alguns dos trabalhos relacionados a este texto. A extensão para

[1] O termo funcionalidade foi aplicado como tradução para o termo *feature*.
[2] O termo compartibilidade foi criado para traduzir o termo *commonality*.
[3] WERNER e BRAGA (2000) sugerem variabilidade como a tradução para o termo *variability*.
[4] É valido ressaltar que o termo funcional qualifica um tipo de funcionalidade, não sendo pleonasmo a expressão "funcionalidades funcionais", visto que existem funcionalidades não-funcionais, sendo elas as funcionalidades transversais.

UML destinada à modelagem da variabilidade transversal através de aspectos é exposta no capítulo seis. O capítulo sete apresenta um estudo de caso. Por fim, seguem a conclusão, referências bibliográficas e os anexos.

CAPÍTULO 2 - Re-visitando o Reúso

Há muito tempo, o reúso é uma das ambições da engenharia de software, tendo sido tema de inúmeras pesquisas. A partir do modelo de computador seqüencial digital, proposto por John von Neumann, a possibilidade de se armazenar instruções na memória do computador, antes de executá-las, permitiu o desenvolvimento das primeiras formas de reúso aplicadas à implementação: linguagens de programação e as bibliotecas de reúso. O termo *reuse* aplicado a software foi posteriormente cunhado por MCILORY (1968). Sua expectativa era transformar o trabalho artesanal da Engenharia de Software em um processo industrial através da comercialização de bibliotecas prontas para o uso.

Em geral, as linguagens tornam a tarefa de programar mais fácil por suportar o reúso de forma inerente. Seleção e interação permitem que instruções possam ser reusadas, executadas novamente sem que sejam reescritas, mas submetidas a condições varáveis; os procedimentos permitem nomear e depois reutilizar conjuntos de instruções em contextos diferentes; os parâmetros permitem que os procedimentos se ajustem às necessidades de cada contexto; esta lista prossegue além do necessário para a ilustração desejada. As bibliotecas estão presentes quase há tanto tempo quanto as próprias linguagens, muitas vezes misturando-se de tal forma que é difícil diferenciar o que compõe as linguagens e o que faz parte de bibliotecas. Isto porque elas são encaradas mais como ferramentas do que como elementos que irão compor o produto final (BIDDLE et al, 2003).

O reúso pode ser conceituado como a aplicação sistêmica de artefatos de software existentes, durante o processo de construção de um novo sistema (uso em um novo

contexto), ou a incorporação física de artefatos em um novo sistema de software (DUSINK & KATWIJK, 1995, p. 138). Em seu cerne está a suposição de que todas as experiências podem ser reutilizadas de alguma forma. Entretanto, um artefato reutilizável possui um certo grau de reusabilidade[5] que é a capacidade de ser aplicado em outros contextos. Para se alcançar altos graus de reusabilidade, são necessários artefatos que permitam adaptações ou modificações de modo a atender de maneiras diferentes determinados requisitos. Por conta disto, é preciso trabalhar o equilíbrio no dualismo generalidade/especificidade, sem, contudo, perder o foco na produtividade, que é a questão original, pois os objetos genéricos demais podem não ser úteis, assim como os muito específicos podem não ser reutilizáveis (BASILI & ROMBACH, 1991, CRNKOVIC & LARSSON, 2002) .

2.1 Motivações para Reúso

Desenvolvedores pouco, ou mesmo muito experientes, procuram através do reúso um meio para evitar a duplicação de código ou a complicação de algumas tarefas já solucionadas, deixando a preocupação apenas em nível de código. Entretanto, a aplicação do reúso não deve ser motivada apenas com o intuito de 'economizar' códigos.

Autores como BROOKS (1987), COX (1990 apud MALAN & WENTZEL, 1993, pág 1), delimitaram uma crise onde a demanda por software estaria crescendo mais rápida do que a capacidade de fornecimento, visto que os meios produtivos disponíveis eram ineficazes. Alguns artigos apresentaram os sintomas típicos da crise (TRACZ, 1987): (i) os projetos eram entregues atrasados, (ii) os requisitos não eram atendidos e (iii) muito da capacidade produtiva era destinada à manutenção de software, podendo chegar, segundo HOROWITZ & MUNSON (1984 apud DUSINK & KATWIJK, 1995, pág 1), a 75% da capacidade de produção.

---

[5] O termo reusabilidade foi criado para traduzir o termo *reusability*.

Nenhuma melhoria isolada no processo ou na tecnologia voltada para o desenvolvimento, segundo eles, seria capaz de solucionar esta questão e, por conta disto, existia uma grande pressão para trocar o desenvolvimento de software individual e artesanal por um processo produtivo que utilizasse reúso e partes intercambiáveis de software, ampliando a produtividade e possivelmente sendo aplicável em todo o ciclo de software. Era preciso criar mecanismos para promover a produtividade de software e, segundo BOEHM (1999), seria possível uma melhoria de 47% no caso de adotar-se a estratégia de evitar o trabalho, através do reúso de artefatos, evitando ou reduzindo desenvolvimentos específicos para projetos. CONSTANTINE (2001) manifestou esta tendência com uma pergunta: "que código é mais barato do que aquele que você não precisa escrever? Com grandes níveis de reúso poder-se-ia duplicar ou triplicar a produção efetiva".

Os níveis apontados por Boehm justificam-se em duas premissas delimitadas por DUSINK & KATWIJK (1995, p.137-149): (i) se um novo software é construído, pode-se reutilizar uma solução existente e o prazo seria mais facilmente cumprido e (ii) se uma solução isolada passe a ser utilizada mais vezes, a qualidade da implementação seria melhorada, conseqüentemente o esforço gasto com manutenção também seria reduzido.

2.2 Reúso na Engenharia de Software

PIETRO-DIAZ (1993, 2001) afirma que a Engenharia de Software difere-se de outras engenharias pelo modo como trata o reúso. Nas demais, o reúso é implícito; natural. Um engenheiro civil utiliza-se de elementos padronizados (tijolos, telhas, janelas, equações para cálculo de resistência, fórmulas para elaboração de concreto, estudos sobre ergonomia, etc.) para construir uma casa. Alguns destes componentes podem sofrer pequenos ajustes para casos específicos e outros precisam ser combinados para se tornarem úteis. Diferentemente, na Engenharia de Software, o reúso é explícito. Mesmo que o engenheiro de software faça uso de elementos da linguagem de programação, do banco de dados e/ou do sistema operacional (como funções, objetos, etc.), é comum ele pensar numa aplicação como um

conjunto de unidades que ele precisa criar. Somente durante a criação é que são percebidas as oportunidades de reúso e, dependendo do caso, aplicadas. Desta forma, perdem-se oportunidades por falta de um planejamento prévio.

Nas outras engenharias, nem todos os requisitos podem ser atendidos. Os engenheiros são treinados para escolherem o melhor conjunto de artefatos, procurando conciliar os requisitos desejáveis do projeto e os artefatos disponíveis e/ou adaptáveis. A Engenharia de Software encara os projetos como um objeto maleável, sendo possível atender a todos os requisitos, mesmo que excludentes entre si, representando uma batalha colina acima. Na visão de PIETRO-DIAZ (2001), o fortalecimento da Engenharia de Software seria obtido através da aproximação com as técnicas utilizadas por outras engenharias. Independente das diferenças, a meta das engenharias é produzir mais, com melhor qualidade e no menor tempo, alcançando altos graus de produtividade e satisfação.

2.3 Tipos de Reúso

O reúso pode ocorrer nas seguintes formas (STAA, 2000. p 6-7):

- verbatim – o elemento reutilizado é referenciado no local de reúso, sem requerer alterações nem cópias. É possível adequar o uso através de parâmetros;
- de extensão – o elemento reutilizado é referenciado e disciplinadamente estendido. Exemplo: redefinição de métodos herdados;
- de cópia – ocorre cópia do elemento reutilizado para os diversos lugares em que é reutilizado, o que leva a multiplicação de código a ser mantido;
- de aproveitamento – o elemento além de copiado é alterado para se adequar aos locais em que é reutilizado. Tem como conseqüência a multiplicação dos códigos a serem mantidos, cada qual com funcionalidades específicas, dificultando a tarefa de manutenção;

- *baseline* – utiliza-se do conceito de item de configuração; um artefato formalmente analisado, revisado e aprovado, e que serve de base para o desenvolvimento posterior. Os artefatos que constituem *baselines* só podem ser alterados por meio de procedimento formal de controle de alterações;
- construto (*build*) – é um conjunto relevante formado por uma coletânea de dois ou mais artefatos. É a versão parcial, porém operacional, de um sistema de software, programa ou componente, que incorpora um subconjunto especificado de características que este sistema, programa ou componente proverá.

2.4 Fatores Inibidores para Reúso

A não aplicação efetiva de reúso muitas vezes é associada a questões apenas tecnológicas, o que é uma concepção equivocada. Fracassos na aplicação podem ter origem em fatores ligados a questões de engenharia, de processo, econômicas ou mesmo de estrutura da organização (ROSSI, 2004, p 23-29).

No que cabe ao escopo dos fatores de engenharia, estão incluídas tanto as deficiências de tecnologia (falta de componentes ou de ferramentas) quanto às de métodos (dificuldades em identificar elementos que descrevam os requisitos solicitados ou meios para garantir a qualidade dos componentes).

O processo influencia o reúso no que tange à maturidade alcançada pela organização. A reutilização de software só será convertida em vantagem nos casos onde os processos encontram-se estáveis, e os resultados são previsíveis. A repetitividade e o planejamento são essenciais para identificação das oportunidades de reúso.

Os fatores organizacionais influenciam o reúso ao vinculá-lo com a estratégia e com a estrutura da organização. A estratégia delimita como o reúso irá contribuir nos objetivos

firmados. A estrutura determina como a divisão e as relações de trabalho estarão comprometidas para o cumprimento da estratégia e das metas de reúso.

Sob a ótica dos fatores econômicos, o reúso implica em um investimento inicial e o retorno é obtido apenas com passar do tempo. Entretanto, diferente de uma edificação ou um equipamento, bens (ou ativos) intelectuais (componentes de software reusáveis) não aparecem nos balanços contábeis. Não há meios para valorizá-los, dificultando a justificativa do investimento (MALAN & WENTZEL, 1993, pág 15). O desenvolvimento de Software reutilizável, segundo CARPER JONES (1994 apud ROSSI, 2004, pág 28), representa um acréscimo de 50% no custo e 30% no tempo. Estima-se que para ser rentável um componente precisa ser reutilizado pelo menos três vezes em projetos de aplicação, necessitando o tempo médio entre dois e três ciclos de produtos, usualmente representado por três anos.

Há consenso que reúso deve ser integrado a todas as fases do desenvolvimento, inclusive nos projetos de software específico. O desenvolvimento de uma solução peculiar pode dar início uma nova classe de objetos reutilizáveis podendo, assim, atender a outras soluções semelhantes.

2.5 Casos de Sucesso de Reúso

Pietro-Diaz em FRAKES et. al. (1997, pág. 57-59) descreve a aplicação de reúso nas empresas Raytheon, Fujitsu, e GTE Data Services. Na divisão de sistema para mísseis da Raytheon, observou-se que 60% de todos os projetos e códigos de programas eram redundantes e poderiam ser padronizados e reutilizados. Mais de 5000 programas foram analisados e classificados. Desenvolveram-se padrões e uma biblioteca foi criada. Os programadores foram treinados e o reúso tornou-se obrigatório. Após 6 anos, calcula-se que 60% do código de novas aplicações são frutos de reúso e houve um acréscimo de 50% na produtividade.

Na Fujitsu’s Software Development for Electronic, a abordagem foi mais moderada e pragmática. Criou-se um centro de informações de suporte. Este centro é uma biblioteca relacionando informações de domínio, códigos, analistas responsáveis, engenheiros de softwares de cada projeto. A biblioteca mantém também informações sobre cada projeto referentes à evolução, arquivos relacionados e questões comerciais. É obrigatório que cada novo projeto seja registrado nesta biblioteca. Através dela é possível acompanhar como foi o desenvolvimento, recuperar as decisões tomadas assim como as suas circunstâncias. Antes desta biblioteca, apenas 20% dos projetos eram entregues no prazo, subindo para 70% após a sua implantação.

Na GTE foi desenvolvido o programa Data Services’ Asset Management, um programa para gerenciamento do capital intelectual da empresa, destinado a instituir a cultura do reúso. A abordagem baseou-se na avaliação artefatos (projetos, documentos, códigos) criados pela empresa passíveis de reúso, seguido da classificação e catalogação em um sistema de biblioteca automatizado. Foram designados times especializados por manter a biblioteca. O reúso tornou-se obrigatório e promoveu-se um programa de recompensas para funcionários que criassem artefatos que pudessem ser adicionados à biblioteca. No primeiro ano do programa, o nível de reúso chegou a 14 % (representando uma economia de 1,5 milhões de dólares).

Os três casos de sucesso apresentados não comprovam de modo inequívoco que o reúso é uma solução efetiva para produtividade em software. Entretanto, as evidências apresentadas nos casos, adicionadas a questões teóricas, não podem ser simplesmente desconsideradas. Por conta disto, este trabalho não assumirá uma posição extremista perante uma questão parcialmente aberta e considerará que o reúso pode contribuir na melhoria da produtividade, se aplicado em conjunto de outras posições, como, por exemplo, o comprometimento organizacional através da adequação dos processos, empenho dos profissionais envolvidos e disponibilidade de recursos para investimento.

CAPÍTULO 3 - Componentização

A componentização, ilustrada pelo desenvolvimento de software baseado em componente, pelos *frameworks* e principalmente pelas linhas de produto, destaca-se dentro das abordagens para aplicação do reúso.

Um componente é um conjunto de elementos de software que forma um todo coerente e descreve ou realiza uma função específica, estando em conformidade com o modelo e provendo um conjunto de interfaces bem definidas (STAA, 2000, p. 6). É preciso que se disponibilize uma documentação adequada e que esteja inserido no contexto de um modelo que guie a composição deste componente, sendo imperativo que esteja autocontido em termos da funcionalidade que provê. Estão presentes em todos os níveis de abstração do processo de desenvolvimento, podendo ser dividido em dois grupos:

- O primeiro, representado pelos componentes de negócio. Eles disponibilizam serviços que o domínio, por si, justifica.
- O segundo, constituído por componentes de infra-estrutura que disponibilizam serviço de suporte aos componentes de negócio, tais como processo de segurança, auditoria, tratamento de mensagens de erro, etc.

Os componentes são definidos pelas suas interfaces. Elas podem ser agrupadas em interfaces fornecedoras e interfaces solicitantes. As interfaces fornecedoras definem os serviços fornecidos pelo componente. As interfaces solicitantes especificam os serviços utilizados pelo componente e precisam ser disponibilizadas a partir do sistema (SOMMERVILLE, 2003, p. 263).

3.1 Níveis de Abstração

Os componentes podem representar níveis de abstração bastante diferentes, desde uma sub-rotina até uma aplicação completa. Há cinco níveis diferentes de abstração, segundo Meyer (1999 apud SOMMERVILLE, 2003, p 264):

- Abstração funcional: o componente implementa uma única função, conseqüentemente, a interface fornecida é a própria função;
- Agrupamentos casuais: o componente é uma coleção de entidades inadequadamente relacionadas;
- Abstração de dados: o componente representa uma abstração de dados. A interface fornecida será composta pelas operações de criar, modificar e acessar a abstração de dados;
- Abstração de agrupamento: o componente é um grupo de classes/objetos relacionados que trabalham em conjunto. Podem representar um *framework*;
- Abstração de sistema: o componente é um sistema inteiramente autocontido. O reúso de abstrações em nível de sistema é, às vezes, chamado de reúso de produto.

3.2 Repositório de Componentes

Um repositório de componentes é análogo a um banco de dados, mas preparado para armazenar e recuperar componentes. No que diz respeito à recuperação efetiva, é importante o armazenamento de informações adicionais relativas ao componente (WERNER & BRAGA, 2000). TRACZ (1990 apud DUSINK & KATWIJK, 1995, p 140) propôs um modelo para descrição de componentes chamado 3C: conceito, conteúdo e contexto. A proposta é atribuir um valor semântico ao componente facilitando a sua localização.

O conceito de um componente é uma descrição detalhada da interface acrescida de um contexto de pré e pós-condições. O conceito deve comunicar o que o componente faz. O conteúdo detalha a implementação da interface do componente. Esta informação deve ficar oculta de usuários casuais, sendo necessária apenas aos interessados em testar ou modificar o componente. O contexto situa o componente dentro do seu domínio de aplicabilidade, especificando características conceituais tanto operacionais quanto de implementação.

Paradoxalmente, o grau de reutilização em um ambiente de desenvolvimento não melhora linearmente com o aumento de componentes disponíveis como seria de se esperar, principalmente em repositórios centralizados de componentes genéricos. A solução para este problema seria organizar os componentes por domínio (e sub-domínios) de aplicação. Esta abordagem diminuiria o escopo de componentes a serem pesquisados (WERNER & BRAGA, 2000). Outras soluções seriam o desenvolvimento de um mecanismo de busca eficiente para determinar qual o componente mais adequado à determinada solução, ou reduzir o número de componentes através do fornecimento de componentes capazes de atenderem mais contextos, diminuindo a necessidade de novos.

3.3 Modelos de Desenvolvimento

Entre os modelos que fazem uso de componentes, três se destacam e se entrelaçam: o desenvolvimento baseado em componentes, *frameworks* e linhas de produto de software. Todos os três serão abordados nos próximos tópicos.

3.3.1 *Desenvolvimento Baseado em Componentes*

O desenvolvimento baseado em componentes (DBC) surgiu no final da década de 1990 como uma abordagem promissora baseada em reúso. O DBC foi uma das respostas à frustração surgida pelo uso da orientação a objetos, que falhava ao disponibilizar um elemento fundamental, o par classe/objeto, com uma granulariedade ainda muito fina. E

mesmo que a O.O representava um grande passo em relação aos modelos anteriores, ainda exigia conhecimento detalhado sobre a estrutura e a relação das classes. Assim, o DBC propõe a criação de aplicações a partir de partes já existentes; os componentes (SOMMERVILLE, 2003, p. 263).

O modelo de desenvolvimento baseado em componentes incorpora características do modelo espiral. Ele é dividido em duas etapas simultâneas: a Engenharia de Domínio e a Engenharia de Aplicação. O modelo proposto por PRESSMAN (2002, p. 707) pode ser observado na figura 3.1.

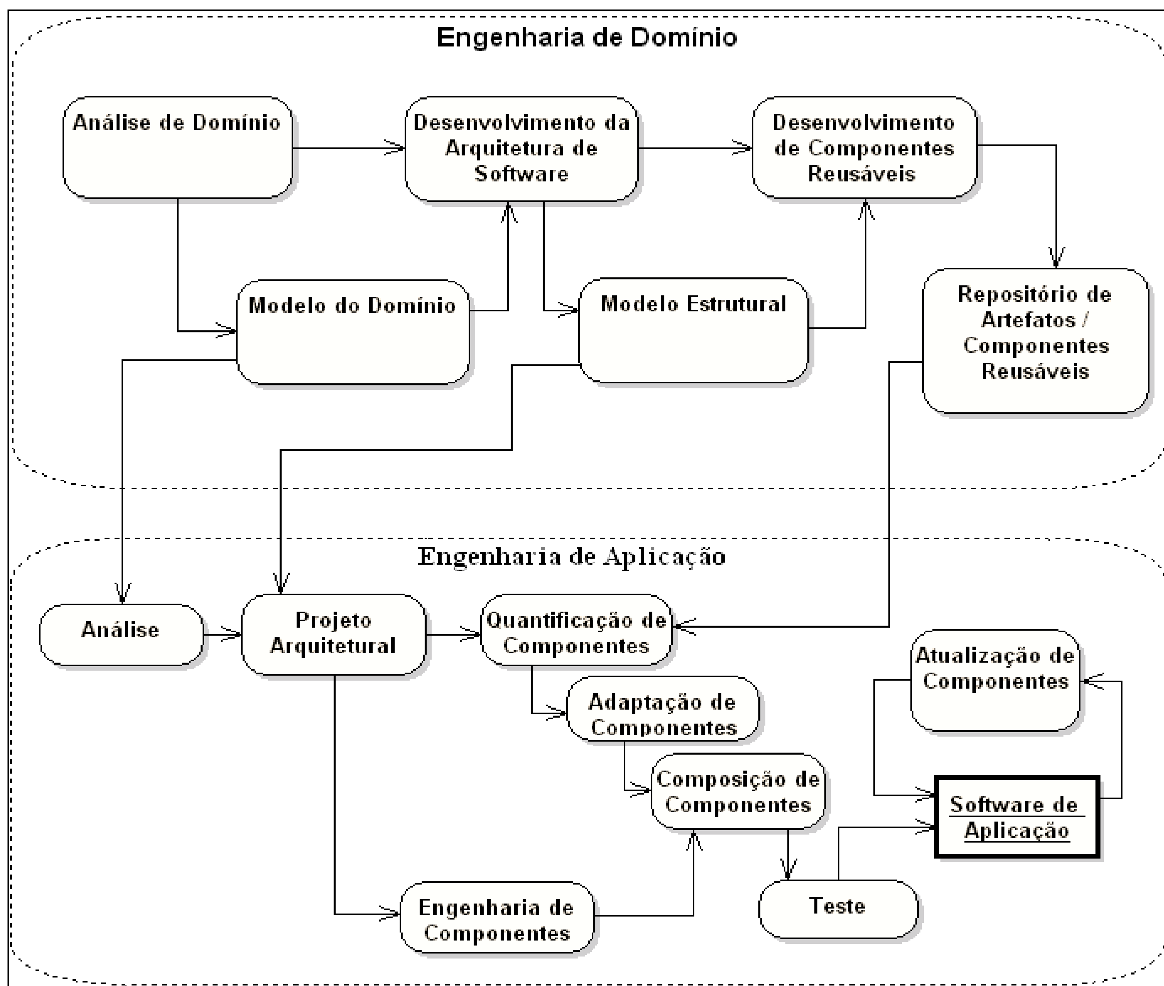

Figura 3.1 – Modelo para desenvolvimento baseado em componente proposto por PRESSMAN (2002, p. 707).

A Engenharia de Domínio (ED) tem como objetivo fornecer ao desenvolvedor mecanismos para compreender os conceitos do domínio da aplicação, representando-os de maneira adequada e genérica, permitindo que possam ser (re) utilizados no processo de desenvolvimento de software (OLIVEIRA et. al., 2001). O escopo da ED é identificar, construir, catalogar e distribuir um conjunto de artefatos de software que possam ser utilizados em aplicações existentes e futuras de um domínio em particular. Ela procura sistematizar a reutilização de artefatos, garantindo a homogeneidade entre os elementos reutilizados nas diversas fases do desenvolvimento do software (WERNER & BRAGA, 2000).

A primeira etapa da ED é a análise de domínio. Seu objetivo é identificar e categorizar os elementos que compõem uma solução genérica de uma classe de problemas. Um dos resultados é a elaboração de uma linguagem do domínio.

A criação do modelo de domínio faz uso da linguagem elaborada na análise para criar a representação de uma solução genérica. O modelo procura mostrar a relação entre os elementos sem atentar a detalhes, podendo ser utilizado linguagens gráficas, como UML, para execução desta tarefa.

O desenvolvimento da arquitetura caracteriza-se pela criação de um esqueleto elementar representando o modelo de um domínio, formado por contratos criados na forma de interfaces requeridas e fornecidas. Nesta etapa inicia-se a criação de um *framework* caixa-branca, conceito abordado em detalhes no próximo tópico.

O modelo estrutural consiste em um pequeno número de elementos estruturais que manifestam padrões claros de interação Ele determina um conjunto de padrões estruturais previsíveis que determinam: interface usuário-sistema, persistência de dados, modelos numéricos e não-numéricos para manipulação de dados, mecanismos para extração de dados e mecanismos para adaptação da aplicação ao usuário (PRESSMAN 2002, p 711). São definidas pré-condições, pós-condições e *invariantes*; escritos em OCL ou linguagem natural.

Depois de definido o modelo estrutural, inicia-se o desenvolvimento dos componentes reutilizáveis aplicáveis ao modelo. Nesta etapa são criadas as variações de componentes que cumpram os requisitos definidos pelo modelo estrutural e obedeçam aos contratos estipulados pela arquitetura do modelo.

A última etapa da Engenharia de Domínio é a armazenagem dos componentes no repositório, item abordado em tópico anterior. A Engenharia de Aplicação (EA) acontece em paralelo a ED. Seu objetivo não é a criação de algo aplicável a todo uma classe de problemas, mas desenvolver de uma solução endereçada a uma situação. O primeiro passo da EA é a análise dos requisitos específicos da solução. Neste momento, o analista avalia quais os elementos da solução que diferem do modelo do domínio. As diferenças são descritas utilizando as convenções e a linguagem de domínio proposta no processo de análise do domínio.

O próximo passo é a criação de projeto arquitetural da aplicação, baseado nos requisitos identificados no passo anterior e no modelo estrutural originado na ED. Este projeto é elaborado em um formato que permita, no próximo passo, verificar os componentes disponíveis capazes de atender os requisitos específicos.

A partir do projeto arquitetural é feita a quantificação dos componentes. Neste processo é avaliado, a partir das informações presentes na biblioteca, a disponibilidade de componentes que supram determinado requisito. Havendo componentes que atendam ou que através de adaptações supram a necessidade, estes são selecionados. No caso do requisito estar parcialmente atendido através de um componente abstrato, então, o fluxo é direcionado para o processo de Engenharia de Componentes. Na Engenharia de Componentes ou cria-se ou adquire-se um componente para atender o requisito não presente (ou abstrato) no repositório.

Como a ED e EA ocorrem em paralelo, deve-se tomar cuidado para evitar que um requisito recém identificado não seja ao mesmo tempo implementado como um componente reutilizável pela ED e como componente específico, através da Engenharia de Componentes. A composição de componentes é o processo responsável por estruturar os

componentes de modo a trabalharem juntos. Nas soluções baseadas em frameworks, é criado um roteiro para distribuição da aplicação que garanta a entrega e instanciação dos componentes corretos. A fase de testes pode, e deve, ser auxiliada por ferramentas de testes. Ela é responsável por identificar eventuais conflitos entre componentes e inconsistências no projeto. Depois de testado, tem-se a aplicação final.

3.3.2 *Framework*

Framework é uma aplicação incompleta de software pertencente a um domínio, que possibilita o reúso de grandes estruturas de software (alta granulariedade) por permitir a criação mais de várias aplicações através da complementação de sua estrutura (BRAGA & MASIERO, 2001, COAD, 1992, FRANCA & STAA, 2001, GIMENES & TRAVASSOS, 2002). A sua utilização otimiza o desenvolvimento por disponibilizar a infra-estrutura comum.

Ele promove o reúso (MALDONADO et al, 2001):

- de elementos da análise – ao incorporar a descrição dos tipos de objetos e ao descrever como um problema maior pode ser dividido e
- de elementos do projeto – por conter algoritmos abstratos e por descrever as interfaces que o programador deve implementar, e por conter as restrições a serem satisfeitas na implementação.

Em um framework, pode-se identificar locais onde é possível representar um aspecto variável de um domínio de aplicação. A eles é dado o nome de ponto de especialização, podendo ser de dois tipos: (i) caixa-branca ou (ii) caixa-preta.

Nos pontos de especialização caixa-branca, as variações estão representadas através de componentes abstratos normalmente representados por interfaces, classes abstratas ou incompletas. Através do uso da derivação implementa-se a variação. Um exemplo de ponto de especialização caixa-branca pode ser observado na figura 3.2, onde é representada uma aplicação para vendas na Internet. A classe Promoção possui um dos seus métodos não

implementado, simbolizado por um retângulo marcado com um traço na diagonal. A implementação deste método ocorre na classe derivada FreqShopping. Este mecanismo é conhecido de *overriding* de métodos, podendo ser implementado, no caso da linguagem C++, através do uso de métodos virtuais e, em Java, pode ser implementado através do uso de métodos abstratos ou interface. Nesta abordagem, o desenvolvedor precisa ter conhecimento tanto da aplicação que está sendo criada quanto do *framework*, tornando a documentação crucial para o sucesso do projeto e para diminuir a curva de aprendizado (FONTOURA et al, 2002, FRANCA & STAA, 2001).

Os pontos de especialização caixa-preta representam variações através da composição de componentes prontos para uso, que são acoplados ao *framework*, podendo ser modificados através de alteração de parâmetros (FONTOURA et al, 2002, FRANCA & STAA, 2001). Na figura 3.2, a relação entre as classes Payment, EMoney e CreditCard representa esta abordagem. As duas classes fornecem uma completa implementação para a operação que necessita variação.

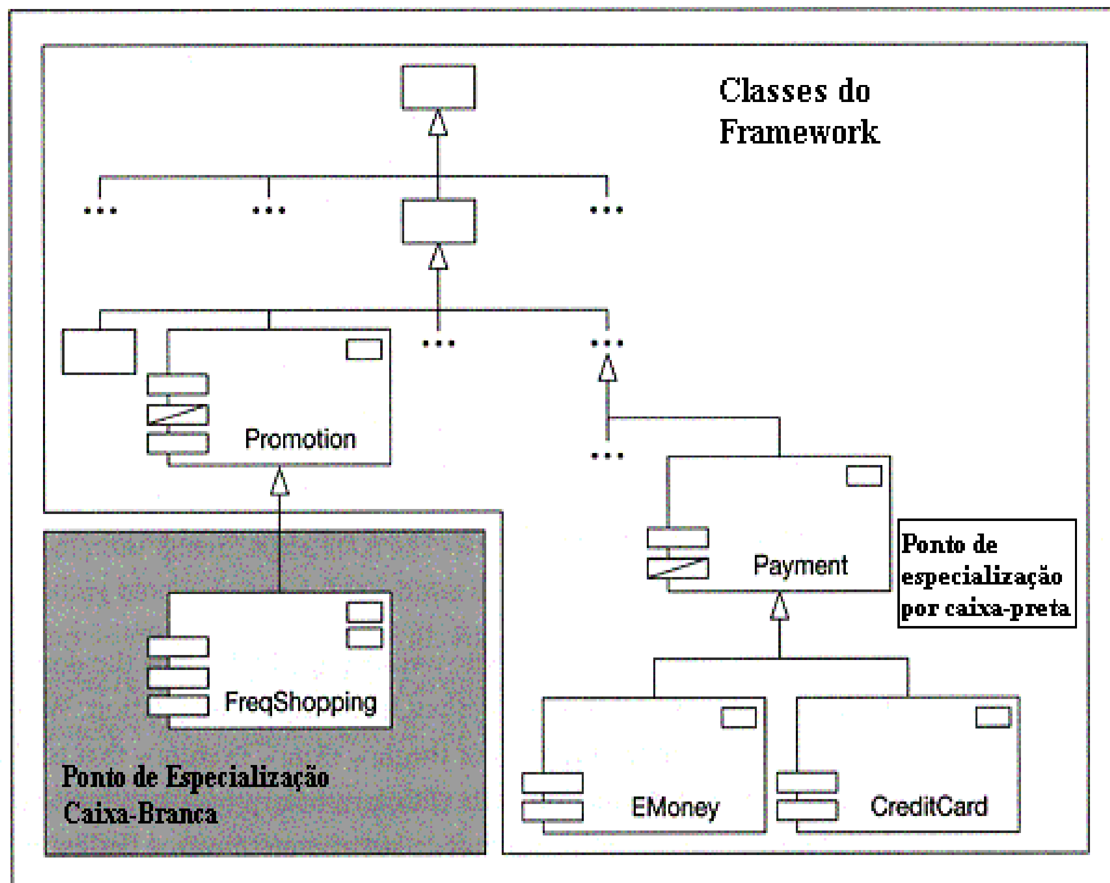

Figura 3.2 – Pontos de especialização (FONTOURA et al, 2002).

3.3.3 *Linhas de Produto de Software*

A evolução das idéias originadas com o desenvolvimento orientado a objetos e componentização, na procura da Engenharia de Software pelo reúso, forneceram os substratos para justificar a origem das linhas de produtos de software. Em 1976, Parnas deu o primeiro passo ao delimitar o conceito de famílias de produto (CLEMENTS & NORTHROP, 1996).

Uma linha de produto é um conjunto de produtos que compartilham uma coleção de requisitos comuns, mas que também apresentam requisitos variantes. Pode-se explorar a compartibilidade e a variabilidade de um conjunto de produtos tratando-os como uma família e decompondo o projeto e a implementação (GRISS - 1, 2000). A abordagem

primeiro analisa o produto para determinar as funcionalidades comuns e variáveis, para então desenvolver a estrutura e a estratégia de implementação que expresse a compartibilidade nos termos de componentes reusáveis (GRISS - 2, 2000). A variabilidade e a compartibilidade serão temas do próximo capítulo.

Por exemplo, uma aplicação para automatização comercial para varejo pode compartilhar os elementos para comunicação com operadoras de cartão de crédito, independente do ramo de atuação. Entretanto, ao lidar com controle de estoque, ou estratégias para cálculo de comissões, pode utilizar abordagens diferentes dependente do ramo de atuação (farmácia, confecção, mercados, etc.)

As linhas de produto possibilitam o reúso mais efetivo de componentes, pois os mais comuns são utilizados múltiplas vezes, aplicando o reúso de extensão, *baseline* e construto. O uso múltiplo permite que correções de defeitos e melhorias no produto possam ser rapidamente propagadas para outros membros da linha de produto (GRISS - 2, 2000, p.3).

Considerações

A componentização é uma abordagem utilizada de maneira natural pelas engenharias e é uma promessa para engenharia de software, somente em parte aplicada. Utilizar componente caixa-preta permite representar as variações de modo extremamente dinâmico, possibilitando adaptações de modo simples e possibilitando que o tempo de desenvolvimento seja reduzido. Ela permite tratar em reúso vários níveis de abstração; desde o nível procedimental até aplicações completas.

Entretanto, a viabilização do processo depende de meios para identificar componentes capazes de atender necessidades. Nesta linha, os repositórios permitem criar mecanismos eficientes para selecionar os componentes apropriados. Entretanto, independente da abordagem, existe um paradoxo entre a quantidade de componentes e capacidade de reúso,

pois o que efetiva o reúso não é apenas a disponibilidade de um componente apto à representação de requisitos de uma situação, mas a capacidade de identificar, selecionar e adequar os disponíveis. Por isso, um número excessivo de componentes pode tornar difícil à identificação de um componente apropriado, levando muitas vezes ao re-desenvolvimento. Deste modo, é preferível um número reduzido de componentes capazes de representar um número maximizado de requisitos.

Dois pontos pesam de forma negativa sobre o uso de componentes. O primeiro é o enrijecimento do modelo para atendimento de requisitos. Este passa a ser limitado pela existência (ou não) de componentes para o atendimento dos requisitos, ou ao desenvolvimento de novos componentes, processo este que se torna burocrático, pois é preciso atender os contratos estipulados (interfaces). O segundo ligado ao desenvolvimento dos componentes, pois existem situações em que a implementação de determinados requisitos pode estar espalhada em vários componentes, o que ocasiona duplicação de código e dificulta a manutenção.

CAPÍTULO 4 - Variabilidade, Compartibilidade e Funcionalidade

As aplicações desenvolvidas com linhas de produto, *frameworks* ou apenas componentes, podem ser expressas através de características compartilhadas ou específicas. Conceitos como: variabilidade, compartibilidade e funcionalidade serão tratados nos próximos tópicos.

4.1 Funcionalidade

Uma funcionalidade é uma característica a qual os usuários e clientes consideram como importante na descrição e distinção dos membros de uma linha de produto. Uma funcionalidade pode ser (GRISS - 2, 2000, p.3).:

- um requisito específico ou
- uma seleção entre requisitos opcionais ou alternativos, relativo a características claras de produtos como funcionamento, usabilidade e desempenho ou características de implementação como tamanho, plataforma de execução ou padrões de compilação

Para facilitar a compreensão sobre funcionalidade, pode-se categorizá-las nos seguintes grupos (GURP & BOSCH, 2001, pag. 4):

- Externas: são funcionalidades oferecidas pela plataforma do sistema alvo. Quando não fazem parte diretamente do sistema, elas são importantes porque o sistema usa e é dependente delas. Um exemplo seria em um cliente de e-mail, a capacidade de efetuar conexões TCP a outros computadores é essencial, mas não é parte do cliente;
- Obrigatórios: são as funcionalidades que identificam o produto. No mesmo exemplo, a capacidade de enviar uma mensagem a um servidor de envio é essencial;
- Opcionais: existem funcionalidades que, quando disponíveis, acrescentam algum valor às funcionalidades externas e obrigatórias. No exemplo do cliente de e-mail, um exemplo seria a capacidade de adicionar uma assinatura em cada mensagem.
- Variante: uma funcionalidade variante é uma abstração para uma coleção de funcionalidades relativas (opcionais, obrigatórias e externas). Um exemplo de funcionalidade variante no contexto de um cliente de e-mail seria a possibilidade de escolher qual o editor utilizar para digitar as mensagens.

4.2 Compartibilidade (Commonality)

Um artefato de software possui dois grandes conjuntos de funcionalidade. O primeiro representando os elementos comum de cada aplicação ou componente. No contexto das linhas de produto, compartibilidade representa as estruturas comuns entre produtos. Quando aplicada aos artefatos são os elementos inalterados compartilhados a todos os seus possíveis usos (WEISS, 97, pág 5).

Compartibilidades são as funcionalidades comuns a um grupo de aplicações de um domínio. O nível de compartibilidade está diretamente relacionado ao nível de reúso, pois um requisito compartilhado entre aplicações de um domínio possui uma implementação

única. Entretanto, como será tratado nos próximos tópicos, em virtude de algumas restrições da orientação a objetos, e por conseqüência dos componentes, isto nem sempre é possível manter implementações únicas.

4.3 Variabilidade (Variability)

O outro grupo de requisitos representa o elemento variável de cada componente ou aplicação. No contexto das linhas de produto, variabilidades são diferenças tangíveis entre produtos (GIMENES & TRAVASSOS, 2002). Quando aplicada aos artefatos, é a capacidade de adaptar-se a contextos diferentes (WEISS, 1997, pág 5). Um nível alto de variabilidade permite ao artefato ser utilizado em mais contextos, contribuindo para a capacidade reúso e por isso é uma característica desejável. Variabilidade é a habilidade de ser mudado ou personalizado através de configuração ou substituição de partes (componentes).

Ampliar a variabilidade em um sistema implica em facilitar determinados tipos de mudança e prever possíveis variações. Entretanto, tanto a modelagem quanto a implementação da variabilidade, por não ser não suportada de forma trivial pela orientação a objetos, pode complicar a criação dos modelos e do código-fonte, dificultando tanto a legibilidade quanto a manutenibilidade dos mesmos. Através das Linhas de Produto, a arquitetura de um sistema é fixada inicialmente, mas os detalhes de uma implementação de uma solução são postergados até a sua implementação. Nos pontos onde são postergadas, as decisões de projetos são chamados de pontos de variação e podem ser introduzidos em diferentes níveis de abstração (GURP & BOSCH, 2001, pág 6), como pode ser observado na figura 4.1:

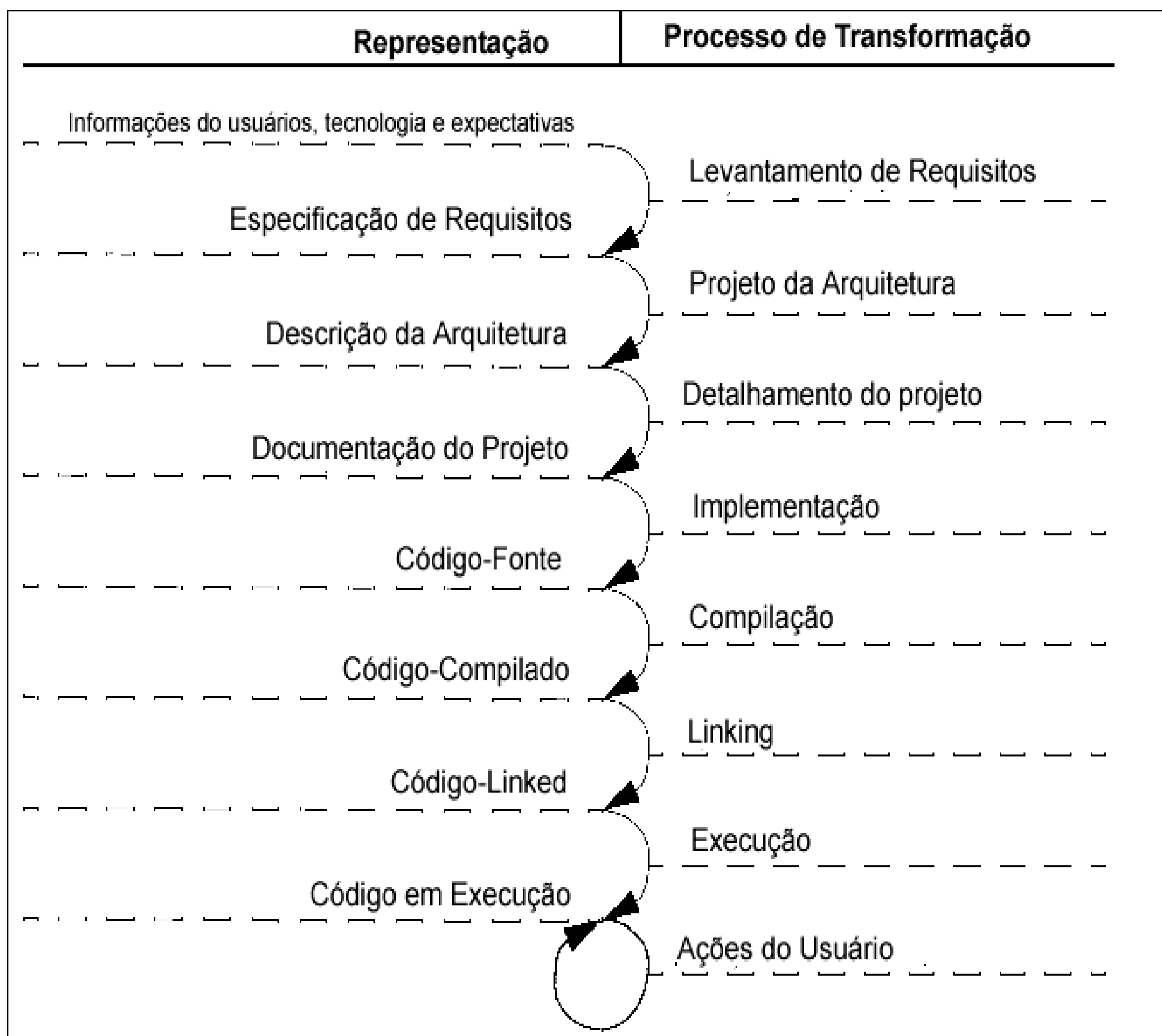

Figura 4.1 – Níveis de Abstração (GURP & BOSCH, 2001, pág 6).

De acordo com o nível no qual a variabilidade é introduzida, pode ser distinguido três estados para os pontos de variação em um sistema (GURP et al, 2000, pág 5):

- Implícito: se a variabilidade é introduzida em um nível particular de abstração, isto significa que nos níveis de abstração superiores, ela também está presente, mas de maneira implícita;
- Projetado: Assim que um ponto de variabilidade é explicitamente introduzido, ele é denominado projetado;

- Delimitado: O propósito de um ponto de variação projetado é tornar possível a definição posterior destes pontos em uma aplicação variante. Quando isto ocorre este ponto de variação é delimitado.

Os mecanismos pelos quais a variabilidade é introduzida sempre operam com a representação do sistema. E variabilidade pode ser introduzida, mantida e delimitada através de alguns poucos padrões recorrentes (GURP et al, 2000).

4.4 Gerenciando a Variabilidade

Tornar o projeto flexível permite o acréscimo de novos requisitos sempre que isto for necessário. Os pontos de variação mais importantes devem ser antecipados de modo a favorecer a sua implementação, isto evitaria problemas na adaptação da arquitetura para suportá-los. Em GURP et al (2000) é proposto um método para gerenciar os pontos de variação, composto pelas seguintes tarefas:

- Identificar da variabilidade. esta é a fase inicial no desenvolvimento de uma linha de produtos, onde os desenvolvedores confrontam os requisitos para um certo número de produtos com os requisitos que serão provavelmente incorporados em produtos futuros;
- Inserir da variabilidade no sistema. a inserção resulta em um ponto de variação em uma das representações do sistema e no mecanismo que será usado para implementar/projetar a variação;
- Colecionar as variantes. Resulta em um conjunto de variantes associadas a um ponto de variação. A coleção de variantes pode ser tanto implícita quanto explícita. No caso das implícitas, não há primeiramente representação da coleção, delegando ao desenvolvedor ou usuários fornecer a variante adequada quando questionado;

- Vincular o ponto de variação de uma solução a uma variante. Isto resulta em uma relação entre pontos de variação com uma de suas variações. Pode ser tanto interna quanto externa. A vinculação interna implica em que o sistema possui um mecanismo para o tal. Quando a vinculação é externa, o sistema delega a outras ferramentas, como, por exemplo, sistemas de gerenciamento de configuração de software (SCM – *Software Configuration Management*).

Considerações

Ao tratar as aplicações como membros de um domínio; analisando, projetando e implementado em termos de características compartilhadas ou variáveis, eleva o desenvolvimento a um nível superior na capacidade de atender requisitos específicos das soluções de cada clientes.

Entretanto, a adaptação de aplicações para funcionalidades específicas do cliente, quando efetivada de maneira não direcionada para o gerenciamento da variabilidade, normalmente representa um custo superior ao valor da própria aplicação. Isto porque se cobra de maneira integral do cliente, pois sem o gerenciamento, a replicação das alterações para produtos de outros (ou novos) clientes, o que possibilitaria a diluição do custo, normalmente não é possível.

Além disto, sem o gerenciamento, as aplicações que sofreram personificações, ficam amarradas a uma versão da aplicação original. No caso de atualizações na aplicação base, a extensão para as aplicações adaptadas ou é processo inviável, ou um tanto quanto complicado.

CAPÍTULO 5 - Orientação a Aspectos

A programação orientada a aspectos foi criada em 1997 nos laboratórios da Xerox em Palo Alto que, estendendo outras paradigmas, como a programação orientada a objetos ou programação orientada a procedimentos, propõe não apenas uma decomposição funcional mas também transversal de um problema. A programação orientada a aspectos permite que na implementação de um sistema separem-se requisitos funcionais e não-funcionais (SOARES & BORBA, 2003), disponibilizando além das abstrações presentes no paradigma-base (como: funções, variáveis, classes, objetos, etc.) o aspecto para a decomposição de interesses sistêmicos.

Para a compreensão da orientação a aspectos e a sua relação com paradigmas, como o orientado a objetos, são necessários revisar alguns conceitos relacionados com a orientação a objetos. Este tema será abordado no próximo tópico.

5.1 Fundamentos da Orientação a Objetos

As primeiras linguagens orientadas a objetos foram criadas na década de 1960, mas somente a partir de 1990 que a orientação a objetos (OO) iniciou a ser amplamente utilizada. O termo orientação a objeto remete à tentativa de simular realidade, trazendo para o computador, abstrações do cotidiano em um nível conceitual sobre o que um objeto faz.

A idéia básica da OO. é a decomposição da solução em objetos definidos através de classes. Um objeto é uma construção de software que encapsula estado e comportamento permitindo que se modele software em termos reais e abstrações. Eles podem ser de vários tipos como entidades físicas: nota-fiscal, cliente, aluno, etc.; ou abstratas: listas, pilhas, vetores, etc. (FURLAN, 1998, pág 16, SCHWARTZBACH, 1994).

Na OO pode-se agrupar objetos pelos seus comportamentos e atributos comuns, formando uma classe de objetos. É através das classes que são implementadas as responsabilidades de um sistema, são descritas as características e os comportamentos. Os objetos de um certo tipo ou classificação compartilham os mesmos comportamentos e atributos (SINTES, 2002).

O estado de um objeto é determinado pelo conjunto dos valores dos atributos. Os atributos são representações de características de um objeto sobre os quais se podem executar ações. Para favorecer o encapsulamento, os atributos devem ser apenas manipuláveis por operações do próprio objeto, evitando que acessos indevidos levem a estados incoerentes (PAGE-JONES, 2001).

O comportamento é uma ação executada por um objeto em resposta a alguma mensagem ou mudança de estado. É algo que um objeto faz sendo implementado através de operações. As operações são procedimentos que acessam os atributos internos e/ou que alteram o estado de um objeto. O conjunto das operações públicas de um objeto forma a lista de serviços disponibilizados, também chamada de interface do objeto. A interface é um contrato com o mundo exterior; ela informa o que um objeto pode fazer, mas não como será feito (SINTES, 2002).

Uma característica importante da OO é o encapsulamento. Ele é anterior a OO, tendo surgido junto com o conceito de sub-rotina. Através dele é conhecido o que é feito, mas ignora-se como é feito. Na OO, encapsulamento é o pacote formado pelas operações e pelo conjunto de atributos de um objeto. O encapsulamento protege a estrutura interna de cada objeto contra a utilização arbitrária, que fuja dos objetivos propostos pelo projeto da classe (PRESSMAN, 2002). As entidades de software passam a ser encaradas como uma caixa-

preta, sendo conhecida apenas a perspectiva externa, sem se preocupar com questões internas. O solicitante de determinada operação não se atenta a detalhes da implementação.

A herança é um mecanismo para reúso de código que possibilita a criação de uma classe baseada em outra previamente existente. A nova classe irá receber as características da classe-base, utilizando as partes comuns e especializando os pontos onde a classe-derivada necessita de um comportamento mais específico (PAGE-JONES, 2001, p. 33). Ela traz, para o desenvolvimento de sistema, o conceito aristotélico de classificação. Esta concepção aumenta a inteligibilidade do sistema, pois permite fazer inferências sobre os objetos a partir da sua classificação.

Umas das características marcantes da OO é o polimorfismo. Polimorfismo é a capacidade de uma mesma referência como, por exemplo, um ponteiro para objetos, poder referenciar implementações diferentes, selecionada por algum mecanismo automático. Assim, permite-se que um único nome expresse comportamentos diferentes possibilitando que se utilize um mesmo nome de método em mais de uma classe e ainda assumir implementações diferentes em cada classe (PAGE-JONES, 2001, p.39). Ele está presente no cotidiano quando se designa o verbo fechar para operações tão diferentes quanto fechar uma porta, um livro, uma caixa ou uma conexão com a Internet.

5.2 Motivação da Orientação a Aspectos

O paradigma orientado a objeto disponibiliza um conjunto de abstrações (classe, objeto, interface, atributo, método) para representar os elementos do sistema. Este conjunto mostrou-se eficaz ao lidar com a complexidade, fazendo um ótimo trabalho ao representar as unidades funcionais que compõem a solução.

Entretanto, este conjunto apresenta limitações ao tratar comportamentos que se espalham pelas diversas classes. Além disso, a complexidade ampliou-se não apenas no que se refere aos elementos que compõem a solução, paralelamente, o uso de novas tecnologias

acrescentou novos elementos à solução do problema. Partes destes entraves podem ser superadas através do uso de padrões de projeto ou através do uso de algumas extensões da orientação a objetos como é o caso da programação orientada a aspectos (HOLMES & SCHILDT, 2004, pág 55, KICZALES et al., 1997, SOARES & BORBA, 2003).

O modo de abstração presente na orientação a objetos é direcionado na representação de elementos do domínio, habitualmente representando-o de forma natural. Os elementos que compõem o domínio formam um interesse. O par classe/objeto consegue encapsular todo o código referente à determinada característica do sistema em uma ou em poucas classes. Um exemplo disto seria a implementação de uma característica que compõe o domínio como as classes Cliente e ClienteEspecial em um sistema comercial. A figura 5.1 representa a implementação de um aspecto que compõe o interesse. As áreas escuras representam as linhas de código envolvidas na implementação do interesse (KICZALES et. al., 1997).

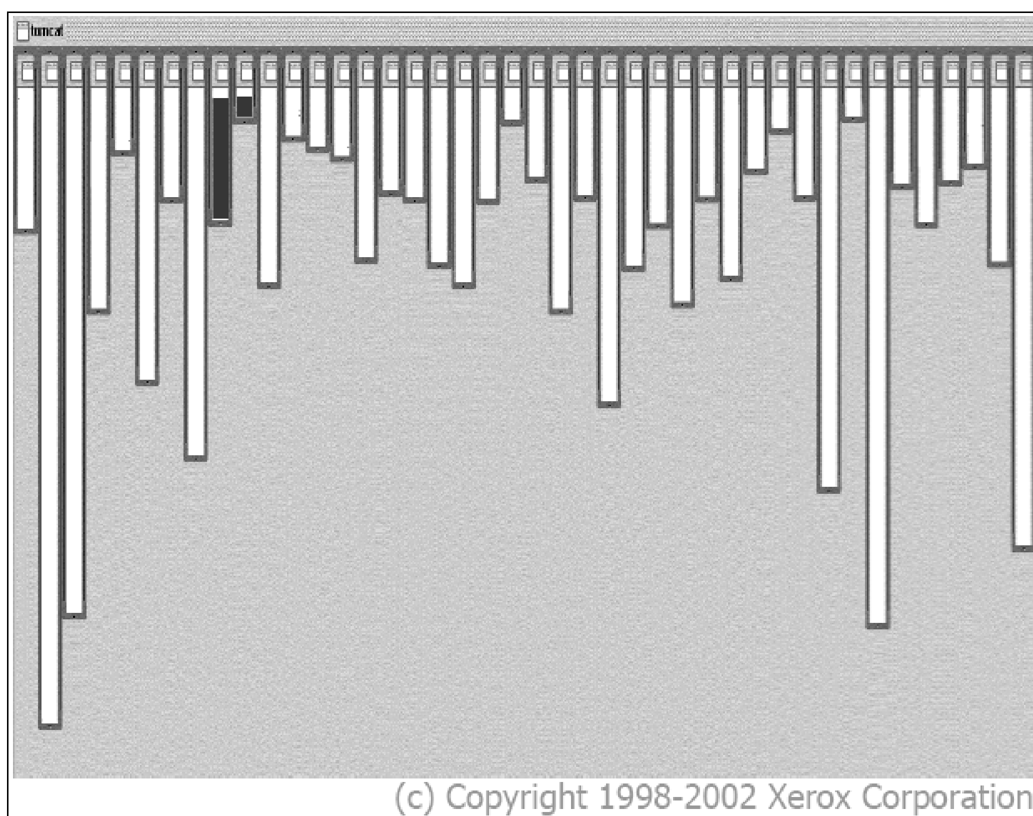

Figura 5.1 – Implementação de um aspecto que compõe o interesse principal de uma aplicação (CHAVEZ et al, 2003).

Entretanto, há um conjunto de características que prestam suporte para a resolução do problema principal, mas que a usa representação no modelo OO é difícil. Este conjunto está, normalmente, ligado a características sistêmicas da aplicação, que dão suporte à solução do problema como, por exemplo, comunicação de dados ou tratamento de exceções.

Normalmente a implementação destas características é misturada com o código responsável pela representação do aspecto principal, sendo incorporadas ao sistema através da replicação de código pelas várias classes do sistema e, muitas vezes, em vários métodos de uma mesma classe. A figura 5.2 representa a implementação de um interesse sistêmico. As áreas escuras representam as linhas de código envolvidas na implementação do interesse. Kiczales (1997) justifica a dificuldade por capturar estes interesses por elas atravessarem o código (*cross-cut*).

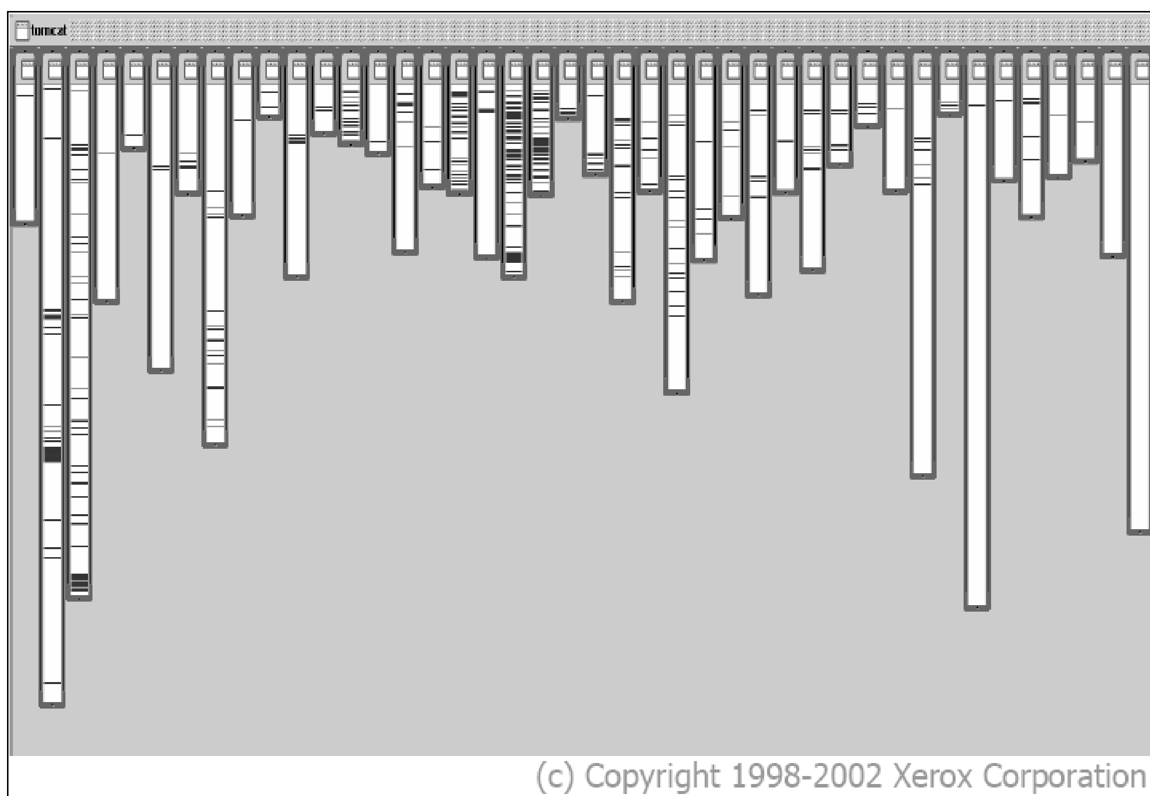

Figura 5.2 - Implementação de um aspecto que compõe o interesse sistêmico de uma aplicação (CHAVEZ at al, 2003).

5.3 Interesses

Interesses são as características relevantes de uma aplicação. Um interesse pode ser dividido em uma série de aspectos que representam os requisitos. Os aspectos que podem ser agrupados no domínio da aplicação compõem os interesses funcionais, sendo também chamados como lógica de negócio.

Existe uma série de aspectos, como por exemplo, o tratamento de exceções, que não fazem parte daquilo que pode ser chamado de lógica de negócio. São características de suporte. No caso do tratamento de exceção, quando implementado utilizando a OO, o código responsável por este interesse, é inseparável do código onde a exceção poderá ser gerada, impedindo o agrupamento deste interesse em uma unidade elementar. O código fica disperso na implementação. Os interesses que apresentam esta característica são chamados interesses sistêmicos (KICZALES, 1997).

Os interesses sistêmicos podem também ser chamados de ortogonais ou transversias. Piveta (2001) afirma que quando duas propriedades devem ser compostas de maneira diferente e ainda se coordenarem, é dito que elas são ortogonais entre si. A tentativa de implementar um interesse sistêmico tem como resultado códigos que se espalham por todo o programa. Para isto, é dado o nome de código espalhado (*Scattering Code*). O código espalhado pode ser classificado em: (i) bloco duplicado e (ii) bloco complementar (LADDAD, 2003, pág 17). A implementação de vários interesses sistêmicos e funcionais em um mesmo módulo resulta no código chamado de emaranhado (*Tangled Code*) (LADDAD, 2003, pág 16).

Os problemas resultantes desta descentralização do código são:

- Aumento da dificuldade da compreensão, tanto do contexto geral do programa quanto do interesse ortogonal;
- Replicação de código: Um mesmo tratamento pode ser necessário em classes diferentes, que não compõe uma mesma estrutura de heranças;

- Dificuldade em manutenção: A cada alteração, ou correção, a implementação dos interesses ortogonais implicará em uma varredura em todos os locais onde há lógica destinada à implementação;
- Redução da capacidade de reutilização de código: Um tratamento destinado a implementação de um interesse sistêmico em uma classe genérica, pode impedir o uso do método genérico em uma classe especializada.

5.4 Composição de sistemas

Os sistemas desenvolvidos com o paradigma orientado a aspectos, diferentemente dos sistemas desenvolvidos com o paradigma estruturado ou o paradigma orientado a objetos, passam por um processo de combinação. Conforme descrito em PIVETA (2001, p 7), um sistema que utiliza a programação orientada a aspectos é composto pelos seguintes elementos (figura 5.3):

- Linguagem de componentes: com ela são implementadas as funcionalidades básicas do sistema, normalmente ligadas ao domínio do problema;
- Linguagem de aspectos: disponibiliza suporte para a implementação de características normalmente desvinculadas do domínio do problema ou de interesses que atravessam o sistema, fornecendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem;
- Combinador Aspectual: é o responsável por combinar os componentes escritos em linguagem de componentes com os elementos escritos em linguagem de aspectos. O resultado, caso fosse criado um combinador é um código que precisa ser compilado.

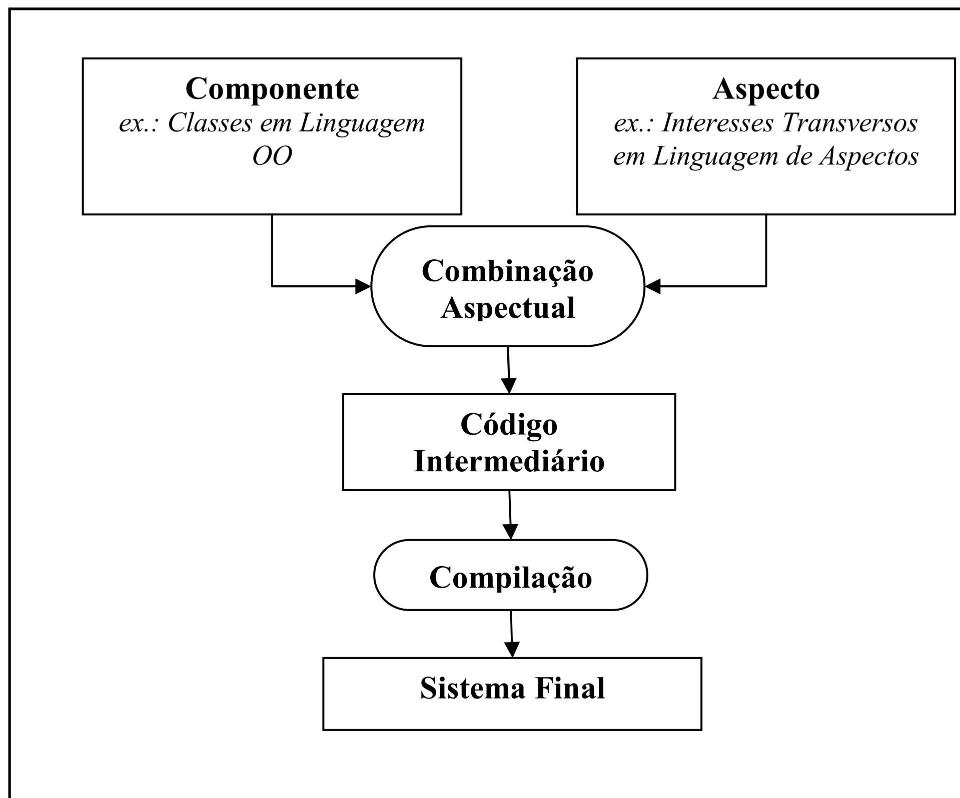

Figura 5.3 – Composição de um sistema orientado a aspectos.

5.5 Elementos da Orientação a Aspectos

Há quatro conceitos elementares na orientação a aspectos: aspecto (*Aspect*), ponto de junção (*Join Point*), ponto de atuação (*Pointcuts*) e adendo (*Advice*).

Aspecto é uma unidade de código, similar uma classe, que encapsula os diversos pontos de atuação e adendos de um programa. Os aspectos de um programa são definidos através de comandos armazenados em um arquivo específico para esta finalidade. Antes, porém, faz-se importante à definição de alguns destes termos.

Os pontos de junção são locais bem definidos da execução de um programa, como, por exemplo, uma chamada a um método ou a ocorrência de uma exceção, dentre muitos

outros. A partir dos pontos de junção são criadas as regras que darão origem aos pontos de atuação.

Os pontos de atuação assemelham-se a regras criadas para especificar o local onde poderá ocorrer uma alteração no comportamento do objeto. Os pontos de atuação existem para que possam ser criadas regras genéricas para definir os eventos que serão considerados pontos de junção, para que não seja necessário definir cada ponto de junção individualmente (o que tornaria a programação orientada a aspectos quase sem sentido). Outra função dos pontos de atuação, além de definir os eventos considerados como pontos de junção, é apresentar dados do contexto de execução de cada ponto de junção, que serão utilizados pela rotina disparada pela ocorrência do ponto de junção em questão.

Adendo é o código que será executado nos pontos de junção quando alguma regra definida pelo ponto de atuação tornar-se válida. No adendo é determinada uma condição temporal, atribuindo-lhe características de evento, possibilitando que a alteração no fluxo ocorra, por exemplo, antes ou depois, do ponto de junção

É importante observar que os pontos de atuação e os adendos são elementos reais, representados fisicamente por trechos de código. Já os pontos de junção não existem fisicamente. Eles são apenas os “pontos de entrada” mentalizados, como marcadores imaginários de pontos onde se deseja executar porções extras de código.

Considerações

A orientação a objetos acrescentou um nível de abstração ao desenvolvimento software. No que se refere à criação de aplicativos comercias, aproximou o desenvolvimento dos problemas que originam a necessidade pelo software. Os conceitos de classe, objeto e relacionamento, inerentes ao modelo OO, possibilitam o reúso de granulariedade maior se comparada à programação estruturada. O modelo orientado a procedimento permitia o

reúso de funções, algo nem sempre fácil de ser alcançado em virtude da interdependência entre elas. No modelo OO, as classes encapsulam a interdependência, facilitando o reúso.

A orientação a objetos não garante o sucesso de projetos e, possivelmente, o seu mau uso é mais desastroso se comparado à orientação a procedimentos. Entretanto, ela serve como fundamento para a abordagem baseada em componentes, uso de *framework* e para programação orientada a aspectos; os próximos tópicos deste trabalho.

Um dos complicadores para o reúso no desenvolvimento baseado em componentes é o paradoxo entre crescimento do número de componentes disponíveis para serem reusados e as oportunidades de reúso efetivadas. A origem deste problema está na dificuldade de identificar o componente apropriado para determinada situação, mesmo com o uso de repositórios para auxiliar nesta atividade.

CAPÍTULO 6 - Trabalhos Relacionados

Este capítulo destina-se a apresentar alguns trabalhos relacionados com a proposta de ser apresentada no próximo tópico. Dois grupos de trabalhos forma avaliados. O primeiro, relacionado com a representação através da UML de aspectos, pontos de junção, pontos de atuação e adendos. O segundo grupo apresenta modelos de implementação de linhas de produtos e *frameworks* utilizando orientação a aspectos.

6.1 Abordagem de Suzuki e Yamamoto para Representação de Aspectos na UML

Esta proposta (SUZUKI & YAMAMOTO, 1999) é baseada na criação de um novo item estrutural – o aspecto, que pode ser observado na figura 6.1. O aspecto é similar à classe e relaciona-se com elas utilizando a relação de realização. Também são criados novos relacionamentos derivados da dependência, sendo eles: requerido, abstração, uso e permissão. Como observaram STEIN et al (2002, pág 111), esta proposta implica em duas dificuldades.

Primeiro, a abordagem limita-se a apresentar uma notação que pode ser utilizada para projetar o que será incorporado. Entretanto, não fica claro como os pontos de atuação ou os adendos deveriam ser projetados com a UML e como os seus efeitos sobre comportamento da classe base deveria ser representado.

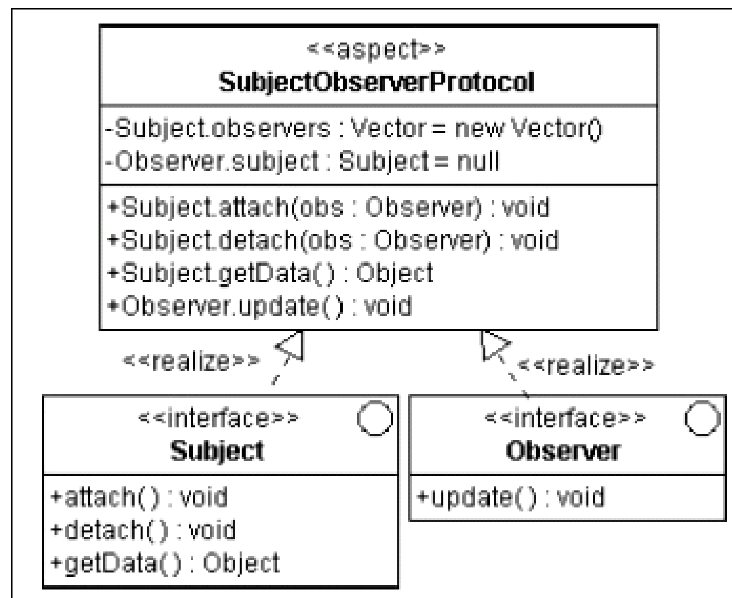

Figura 6.1 – Exemplo da notação de aspecto proposto por SUZUKI E YAMAMOTO (1999, pág.4).

Segundo, o uso do relacionamento *realização* para modelar a relação entre um aspecto e a sua classe base não é apropriado. Por definição, realização é o relacionamento entre classificadores, onde um dos elementos especifica um contrato cuja implementação (e execução) é garantida pelo outro elemento. Entretanto, na orientação a aspectos, não há relação de especificação/implementação entre a classe e o aspecto, ambos os papéis são executados pelo aspecto.

6.2 Abordagem de Herrero et al. para Representação de Aspectos na UML

A proposta de HERRERO (2000) é procurar separar o projeto dos comportamentos básicos dos objetos dos aspectos não funcionais em diferentes entidades de projeto. Estas entidades são relacionadas uma com as outras através de simples associações. Estes relacionamentos são fornecidos com um mapeamento de expressões indicando qual elemento na entidade

representando as classes bases corresponde com quais elementos na entidade de projeto representando os aspectos. Esta abordagem tem problemas também.

Na UML associações são utilizadas para expressar um relacionamento entre elementos, no qual há vínculos entre os elementos, envolvendo conexões entre as suas instâncias. De modo simplificado, representa relações do tipo "é parte de" ou "tem um". Entretanto não há relação desta natureza entre o aspecto e a classe base. Os adendos de um aspecto são introduzidos na classe base e, por isso, a associação não representa de modo adequado os efeitos sistêmicos decorrentes da introdução do adendo.

Outro problema na abordagem de Herrero, os pontos de atuação são expressos através de mapeamento de expressões, que é vinculado à associação. No AspectJ o ponto de atuação é uma propriedade do aspecto. A vinculação do ponto de atuação à associação impede o reúso através da herança de aspectos.

6.3 Proposta de Martin Griss para Linhas de Produtos Orientada a Aspectos

Martin constatou que os sistemas de comércio eletrônico são altamente dinâmicos e requerem mais flexibilidade e menor tempo para comercialização do que tradicionalmente é disponibilizado pelas linhas de produto e pelo desenvolvimento baseado em componentes (GRISS-2, 2000) . Ele propõe a integração de tecnologias para linhas de produto para criar a base de uma abordagem sistêmica.

A idéia principal por trás da proposta é implementar diretamente uma funcionalidade como um fragmento de código (um aspecto), e então usar algum mecanismo ou ferramentas para combinar estes fragmentos em métodos, classes ou componentes completos. O objetivo não é gerenciar diretamente os componentes correspondentes, mas apenas gerenciar as funcionalidades e correspondentes aspectos. Quando necessário, uma nova versão do componente completo é gerado.

O problema da proposta desta proposta consistiu em se restringir a tratar elementos ligados à implementação dos sistemas, ignorando elementos de análise ou modelagem. Ao tratar em nível de implementação, deixou-se em um nível mais baixo de abstração.

6.4 Proposta de Spincyzk e Beuche para Linhas de Produto Orientada a Aspectos

A proposta de SPINCYZK & BEUCHE (2004) parte do pressuposto que as linhas de produto reduzem o custo total para desenvolvimento de software, mas agregam uma complexidade extra. Por conta disto, os desenvolvedores só teriam sucesso caso tivessem a ferramenta adequada.

Os autores propõem um *plugin* para o eclipse, o *pure:variant.* A ferramenta eclipse foi escolhida por ser robusta, flexível e difundida. O *plugin* proposto cobre todas as fases do ciclo de desenvolvimento de uma linha de produto de software. É possível descrever um problema e o domínio da solução de uma linha de produto, famílias de software e artefatos variáveis e altamente flexíveis. O núcleo do *plugin* é composto por vários modelos que são usados para representar o domínio do problema de uma família, o domínio da solução e os requisitos específicos de cada membro da família. A figura 6.2 ilustra de forma simplificada o processo de derivação para variação do *pure:variant*.

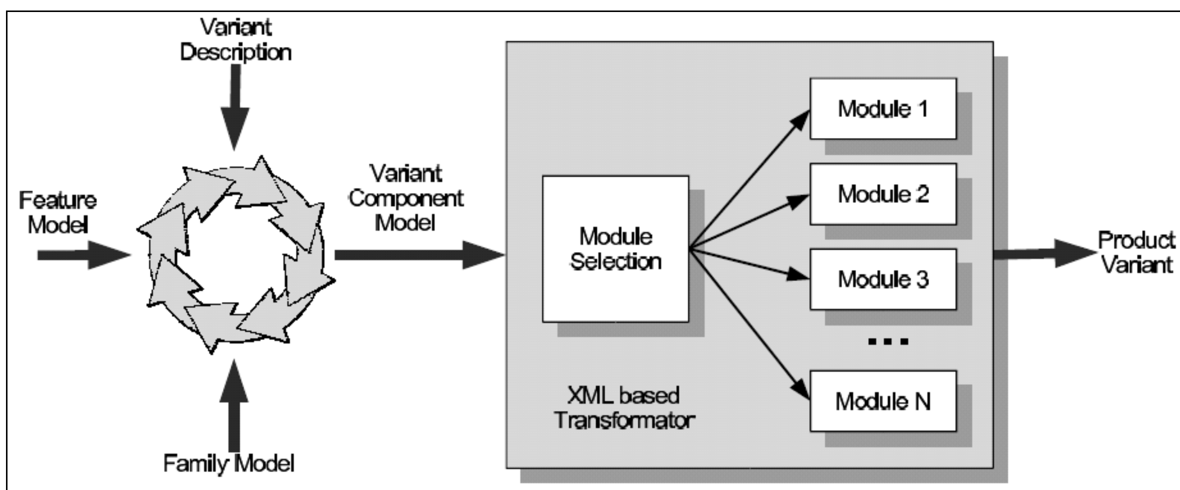

Figura 6.2 – Simplificação do modelo e processo de derivação (SPINCYZK & BEUCHE, 2004).

6.5 Proposta de Camargo, Ramos e Masiero para Implementação de Variabilidades em Frameworks *Orientados a Aspectos.*

No processo de desenvolvimento de *frameworks* há uma etapa destinada à implementação das variabilidades. Em Camargo et al (2004), é discutido o uso do paradigma orientado a aspectos e AspectJ para implementar os ganchos de variabilidade através dos pontos de atuação. O referido artigo procura apresentar algumas diretrizes que podem ser usadas para o projeto e implementação.

Esta abordagem, como o próprio artigo observa, tem três problemas discutidos: (i) a abrangência dos pontos de atuação do tipo gancho, aumentando as chances de inserção de erros; (ii) a complexidade dos pontos de atuação do tipo gancho, ocasionado por instruções de restrição de fluxo de controle ou do escopo de chamada; e (iii) dificuldade de evolução do *framework*, às vezes, sendo limitada pela incapacidade da linguagem AspectJ de derivar Aspectos concretos.

Considerações

Nos trabalhos relacionados podem ser agrupados em duas classes: que objetivam estender a UML de modo a fornecer suporte aos conceitos da orientação a aspectos e que procuram aplicar a orientação a aspectos a criação de linhas de produto e *frameworks.* De modo geral, eles são eficientes ao lidar com as questões levantadas, entretanto, nenhum destina-se especificamente à modelagem da variabilidade, presente nos *frameworks* e linhas de produto, através de um paradigma baseado em composição, presente na orientação a

aspectos. De fato, percebe-se a falta de um elo para unir as propostas apresentadas, por isso o presente trabalho diferencia-se das abordagens por propor o elo.

A criação de novos estereótipos para UML de modo a criar uma extensão justifica-se pela especificidade da proposta e ausência de elementos capazes de representá-los semanticamente de maneira satisfatória.

CAPÍTULO 7 - Extensão para UML para Modelagem da Variabilidade Transversal em Componentes

O primeiro capítulo do presente trabalho apresentou o reúso como uma forma de resolver alguns dos problemas da engenharia de software, e tanto na obra de PIETRO-DIAZ (1993 e 2001) quanto de MARTIN GRISS (2000-1, 2000-2) ou tantos outros autores, esta preocupação ecoa. Entretanto, na definição do reúso está presente a relação entre um novo projeto e os demais projetos executados, pois o reúso préssupõe a existência de experiências para serem reutilizadas.

Entretanto, iniciar um novo projeto primando pelo reúso sem uma infra-estrutura apropriada pode levar ao fracasso ou apenas à sua aplicação oportunista, vinculado a situações ou à habilidade do engenheiro de software. Atuando de forma agravante, as funcionalidades (*features*) muitas vezes impedem a aplicação de maneira adequada, uma vez que as diferenças de alguns requisitos mascaram a semelhança dos demais. As abordagens que fazem uso de componentes delimitam uma estrutura compartilhada pelos projetos e permitem variações como já observado no presente trabalho.

A variabilidade em componentes é uma característica desejável no desenvolvimento, pois permite reduzir o número de componentes necessários para representar as mesmas situações. Em termos da modelagem, a capacidade de expressão da variabilidade deve ser favorecida, visto que o trabalho efetuado pode ser re-aplicado a outros projetos.

Através da aplicação da orientação a aspectos, é possível ter o melhor dois mundos. Como será apresentado, pode-se perceber os componentes como específicos para atender os mais diversos requisitos transversais, mas na hora de implementar, o núcleo permanece inalterado e faz-se a composição necessária com o aspecto ideal. O objetivo por trás deste capítulo será apresentar uma forma de modelar as variações, especificamente as ditas transversais, através da orientação a aspectos e dos elementos previstos pela UML.

7.1 Variabilidade em componentes

A variabilidade pode originar-se nas funções, nos dados, no fluxo de controle, na tecnologia ou no ambiente. Independente da sua origem, pode-se agrupá-la de acordo como ela influencia o sistema. Há dois grupos: (i) variabilidade funcional e (ii) variabilidade transversal.

A variabilidade funcional refere-se aos elementos ligados às funções, dados e fluxo de controle, influenciando diretamente sobre a parcela do código destinada à lógica de negócio. Na orientação a objetos, pode-se representar estes através da abstração de forma bastante eficiente.

A variabilidade transversal, ou sistêmica, refere-se à tecnologia, ao ambiente e em parte ao fluxo de controle, incidindo sobre elementos que dão suporte à solução e, normalmente, estão espalhados por vários elementos funcionais. Através da programação orientada a aspectos, é possível isolá-los em aspectos, permitindo que sejam reutilizados em mais de um componente. Para a presente abordagem, a característica funcional isolada será em um núcleo do componente, não sendo objetivo tratar a variabilidade aplicada a ele, mas apenas a aplicada às características transversais.

7.2 Motivação para Uso da Orientação a Aspectos na Modelagem Variabilidade Transversal

O uso da orientação a aspectos contribuirá na modelagem de duas formas:

- Ao permitir a decomposição de componentes em núcleos funcionais (direcionados apenas para a atividade necessária) e interesses sistêmicos (transversais), possibilitando a distinção de elementos destinados ao suporte sistêmico (implementados através de aspectos) dos elementos destinados a implementar questões funcionais (representados através de classes e componentes). No momento de (re)compor o sistema, possibilita-se ao engenheiro do software selecionar as características sistêmicas e funcionais desejáveis;
- Ao retirar as questões sistêmicas dos componentes, separando em um elemento reutilizável (o aspecto) permite-se à redução do número de componentes necessários para representar as mesmas situações.

7.3 Mecanismos de Extensibilidade da UML

A UML é a notação gráfica que define a semântica do meta-modelo de objetos, a notação para especificação e comunicação da estrutura, comportamento, estado e interação dos objetos. O seu organismo mantenedor é a OMG, que a definiu como uma arquitetura em quatro camadas: modelo do usuário (mais superior), modelo, perfis, meta-modelo e meta-meta-modelo.

A UML disponibiliza mecanismos para extensão da sua semântica, permitindo que seja utilizada para fins os quais não tenham sido previamente previsto no momento de sua criação. Ao criar uma arquitetura em camadas permite criar perfis capazes de lidar com qualquer sistema a ser modelado através dos mecanismos de extensibilidade.

Os mecanismos disponibilizados pela UML para suportar adaptações para novas tecnologias de desenvolvimento de software são os estereótipos (*stereotypes*), valores etiquetados (*tagged values*) e restrições (*constraints*). Um estereótipo permite a criação de novos blocos de construção, derivados dos existentes. Um valor etiquetado estende as propriedades de um bloco de construção. As restrições estendem a semântica de um bloco (BOOCH et al 2000, pág 434). A presente proposta irá propor estereótipos de classes, componentes, pacotes, relacionamentos e interfaces, que serão apresentados em tópico próximo.

7.4 Modelagem Estrutural e Comportamental

Na modelagem de soluções, devem ser consideradas tanto as características comportamentais quanto as funcionais do software, entretanto no que tange à variabilidade transversal, a influência no comportamento da aplicação deve ser minimizado direcionando a preocupação para questões funcionais do problema e não transversais. É importante observar que a variabilidade influencia de forma direta a estrutura dos componentes assim como o comportamento. Entretanto quando se restringe a modelagem das características transversais, em sistemas no qual a preocupação principal são as características funcionais, a influência da variabilidade sobre o comportamento torna-se minimizada e podendo ser eficientemente modelada através dos elementos já disponíveis na UML.

7.5 A Proposta para Extensão da UML

A modelagem da variabilidade é o processo no qual as possíveis variações são explicitadas e projetadas de modo a garantir o atendimento de requisitos em soluções residentes em um mesmo domínio. Este procedimento pode ser aplicado a qualquer conjunto de requisitos. Entretanto, nesta pesquisa será observado apenas o uso em características ditas transversais

(sistêmicas). A modelagem transversal permite criar componentes capazes de suprir necessidades normalmente engessadas pela codificação espalhada de interesses.

A presente proposta distingue quatro modelos, ou perspectivas, envolvidas no processo da modelagem da variabilidade transversal de componentes:

- modelo funcional-transversal: no qual é apresentado, de modo sedimentado, as características funcionais e transversais;
- modelo componente-variações: onde são detalhadas as variações para um determinado componente;
- modelo de variação: destinado a mostrar como um componente variante será implementado; e
- modelo de aplicação: onde é apresentado o conjunto de variações selecionadas para cada componente no contexto de uma aplicação completa.

Para cada uma das perspectivas por trás do processo da modelagem da variabilidade transversal será adotado um conjunto de estereótipos que podem ser agrupados em diagramas distintos ou representados em um único dependendo das características que precisam ser apresentadas.

7.5.1 *O Modelo Funcional-Transversal*

O modelo funcional-transversal apresenta a distinção entre os componentes funcionais e transversais em uma estrutura (*framework*) no qual esses serão inseridos. O objetivo é deixar claro quais aspectos transversais estarão sendo tratados de modo isolado dos componentes funcionais. Os componentes perceberão uns aos outros como elementos capazes de cumprir uma tarefa, ou um conjunto delas, independente do modo como estas foram implementadas.

Para representar a relação entre os elementos funcionais e transversais presentes no primeiro modelo, são utilizados elementos previstos na UML como pacotes, componentes e associações e, de forma adicional, são propostos dois estereótipos.

O primeiro aplicado sobre Componente, nomeado <<funcionalidade Transversal>>. Sua funcionalidade é diferenciar de componentes comuns da UML, indicando que o componente marcado destina-se às características transversais. Este estereótipo pode ser observado na figura 7.1.

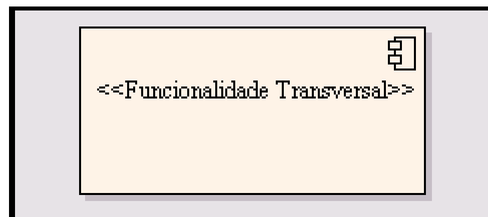

Figura 7.1 – Estereótipo Funcionalidade Transversal.

O segundo elemento adicionado altera a semântica da ligação de associação para explicitar um novo tipo de relação chamada <<composição>>. Ela representa uma relação que não é determinada na modelagem, mas em algum momento posterior, podendo até mesmo ser definido no momento da execução. Este estereótipo pode ser observado na figura 7.2.

Figura 7.2 – Estereótipo Composição.

7.5.2 *O Modelo Componente-Variações*

O modelo componente-variações apresenta a relação entre um componente e as possíveis variações para uma funcionalidade disponíveis para um mesmo contrato com a estrutura. Sua finalidade é expor os elementos disponíveis para efetuar determinada tarefa, ou conjunto delas, sem, contudo, explicitar como estes componentes são implementados,

permitindo ao engenheiro de software selecionar as melhores características disponíveis para uma solução.

Com o objetivo de representar as variações de um componente, são propostos dois novos estereótipos. O primeiro é derivado do pacote com a finalidade de armazenar as variações de uma funcionalidade, nomeado de <<Funcionalidade>>. Presente na figura 7.3.

Figura 7.3 – Estereótipo Funcionalidade

O segundo, <<Variação>>, destinado a representar a funcionalidade propriamente dita, onde é enunciada qual variação que ele representa de forma clara e explícita e a sua implementação. Como restrição ao uso deste estereótipo, ele deve se utilizado sempre inserido em um pacote <<Funcionalidade>>, junto com as demais variações. Ele pode ser observado na figura 7.4.

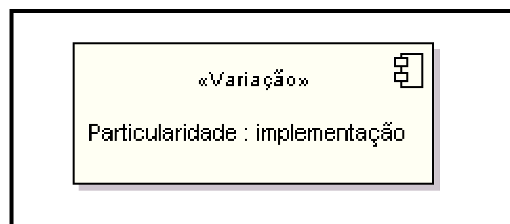

Figura 7.4 – Estereótipo Variação

7.5.3 *O Modelo de variação*

O modelo de variação apresenta o modo como uma variação de um componente será estruturada, de modo a cumprir a tarefa a qual ele se propõe. Este modelo pode disponibilizar informações ao o mecanismo de composição de componentes, para que possa ser automatizado este processo. Nele são apresentadas as relações entre os elementos que farão parte da variação de modo a formar um componente completo. Dois estereótipos novos são propostos para modelar a composição dos componentes com os aspectos transversais previsto pela terceira perspectiva: Variação Componente e Aspecto.

O <<Variação Componente>> é destinado à organização, serve como limites daquele, que nas abordagens convencionais, é o componente convencional. Desta forma, tudo o que estiver inserido nele, de alguma forma contribui para a sua formação ou compreensão, e a sua composição gera o componente <<Variante>>, estereótipo a ser apresentado no próximo tópico. Este estereótipo pode ser observado na figura 7.5.

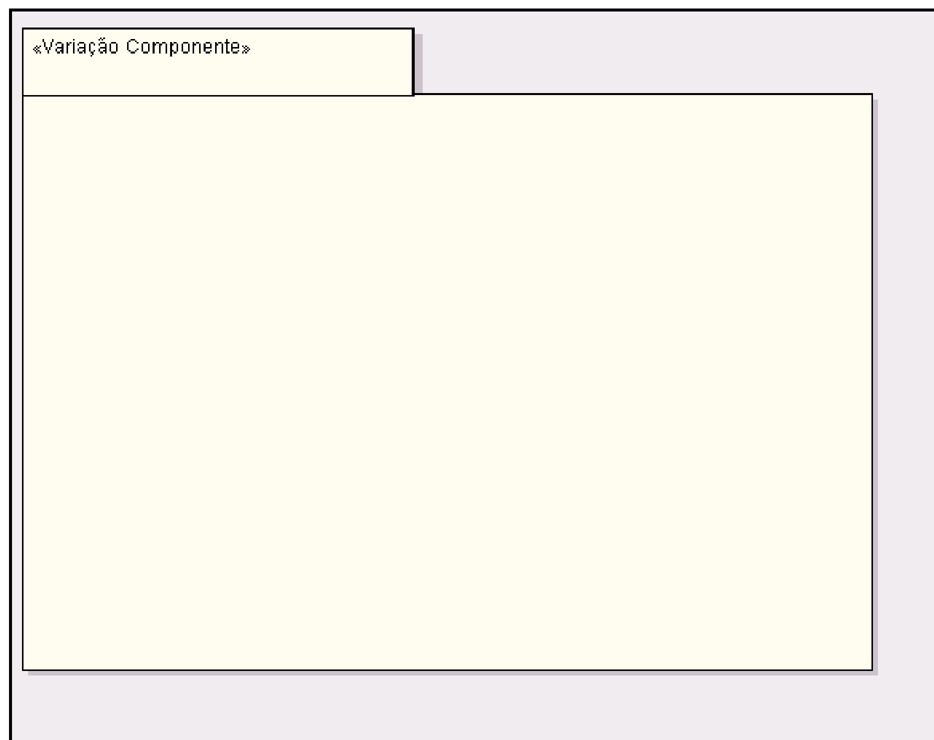

Figura 7.5 – Estereótipo Variação Componente.

O <<Aspecto>> é um estereótipo originado da classe convencional. Sua proposta deriva de SUZUKI & YAMAMOTO (1999). Há uma relação direta entre a <<Variação>> e o <<Aspecto>>, onde cada variação dá origem a um aspecto. Há duas caixas com informações relevantes nele, na do meio são listados e detalhados os pontos de atuação os quais o aspecto será composto. O detalhamento do ponto de atuação pode ser um nome de um método, ou uma listagem de métodos ou qualquer meio de delinear um local de atuação no componente funcional. Na caixa inferior, são especificadas as operações a serem executadas nos pontos de atuação durante a composição do componente. É importante notar que o nome do adendo e o nome do ponto de atuação precisam ser o mesmo, definindo uma relação "um para um". Este estereótipo pode ser observado na figura 7.6. Na codificação do adendo são expressos os comandos que serão mesclados ao componente durante a composição do componente. Vale observar que nas implementações de linguagens orientadas a aspectos como AspectJ, a composição pode ser efetuada sobre o código já compilado do componente, não sendo necessários os arquivos fontes.

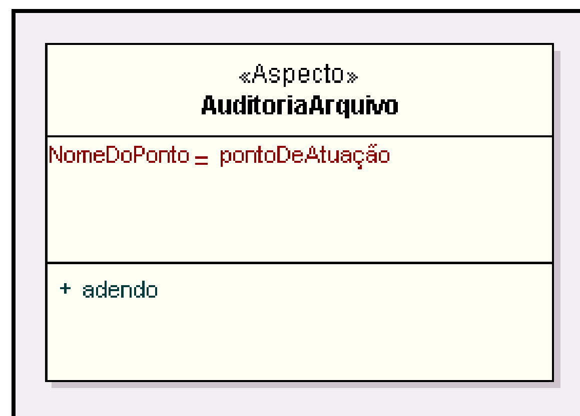

Figura 7.6 – Estereótipo Aspecto

7.5.4 *O Modelo de Aplicação*

O último modelo é destinado a expor, para uma determinada solução, quais foram as características transversais selecionadas para serem compostas com um determinado componente funcional, explicitando qual é a variação sendo aplicada.

Para esta perspectiva apenas um estereótipo, derivado do componente, é adicionado: o <<Variante>>. Este se destina a apresentar qual conjunto de funcionalidades transversais variantes foram selecionadas para um determinado componente. Neste componente há uma caixa inferior onde são detalhadas as funcionalidades, como pode ser observado na figura 7.7.

Figura 7.7 – Estereótipo Variante.

Considerações

A presente proposta consegue reduzir a quantidade de componentes a serem modelados ao possibilitar a criação de dois conjuntos distintos de componentes: funcionais e transversais, algo que até o advento da orientação a aspectos seria impossível.

Ao admitir a modelagem separadamente, permite-se reutilizar estruturas de projeto destinadas a suporte em outros componentes, assim como se permite que um mesmo componente funcional apresente, de acordo com a composição, características transversais distintas, reduzindo a necessidade de replicação, ou omissão no modelo. Uma conseqüência direta é:

- redução do tempo de desenvolvimento;
- melhoria dos componentes, através de sucessivos refinamentos replicáveis de maneira transparente aos demais componentes;
- redução da complexidade de administração dos componentes nos repositórios de componentes, pela redução do número de componentes necessários.

CAPÍTULO 8 - Estudo de Caso

O estudo de caso apresentado a seguir ilustrará a aplicação da apresente proposta. Será modelado um de sistema de pedido de compras. Neste sistema um cliente inicia uma ordem de compra, selecionando produtos: ao término da seleção, é gerada uma ordem de compra. A ordem de compra é efetivada, e os produtos são remetidos após o cliente efetuar o pagamento.

As características funcionais deste exemplo são: as políticas implementadas para crédito ao cliente, cálculo de desconto, formas de pagamento disponíveis, etc. As implementações destes elementos podem ser efetuadas através de classes ou componentes. Considerando que será aplicado o modelo de desenvolvimento baseado em componentes, um modelo do estudo de caso pode ser observado na figura 8.1.

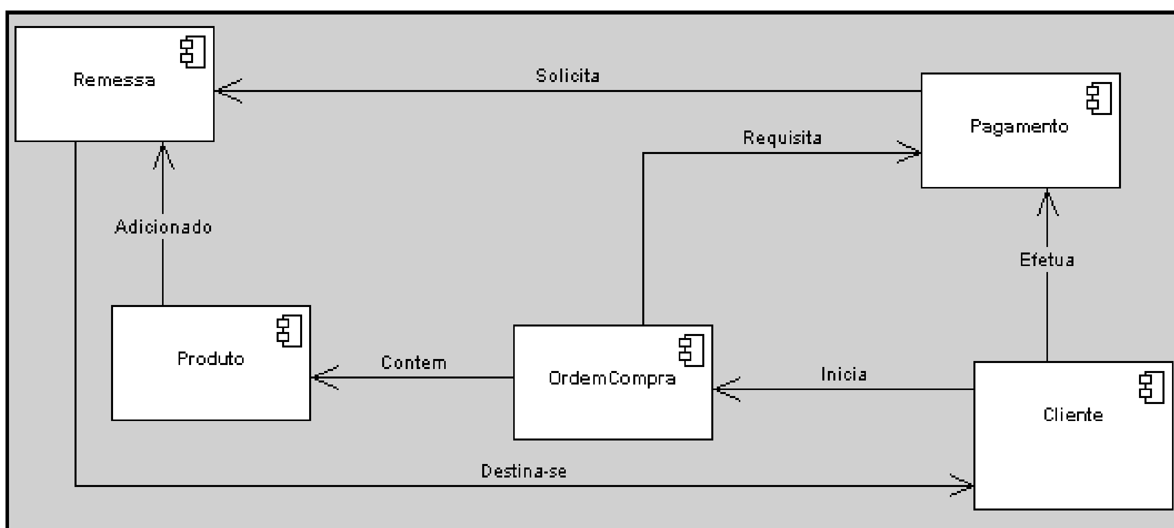

Figura 8.1 – Modelo do Estudo de caso.

Neste modelo existem características ignoradas, mas que são importantes. No desenvolvimento de produtos onde estas características podem ser variáveis, suprimi-las ou torná-las fixas no modelo ocasionam um problema.

No presente exemplo, estes pontos necessitam ter abordagens diferenciadas para funcionalidades como auditoria ou controle de acesso. Em abordagens convencionais só seria possível modelar estas variações ou de modo subjetivo (não deixando claro as variações disponíveis) ou de modo exaustivo (especificando exaustivamente cada variação), utilizando interfaces e criando componentes com cada característica desejada. A figura 8.2 modela de modo exaustivo o componente remessa.

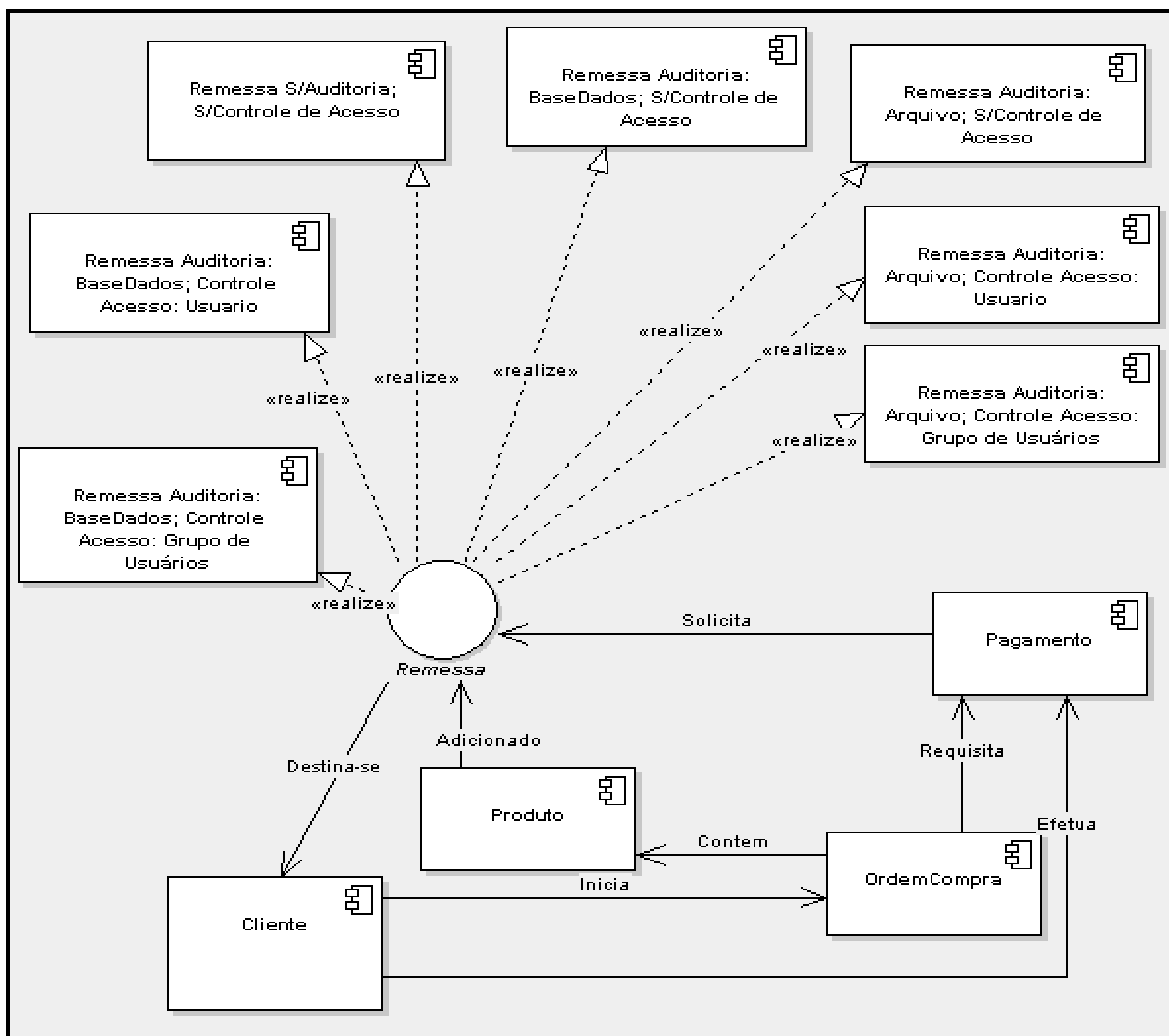

Figura 8.2 - Modelagem de Variabilidade de Remessa de Modo Exaustivo.

Esta situação acontece também para os demais componentes. Com a presente proposta, através do uso da orientação a aspectos, pretende-se minimizar o impacto deste problema, através da decomposição das funcionalidades.

8.1 Modelo Funcional-Transversal

O modelo funcional-transversal apresenta a relação de características transversais e sistêmicas. No estudo de caso proposto, há duas características transversais relevantes, controle de acesso e auditoria (figura 8.3).

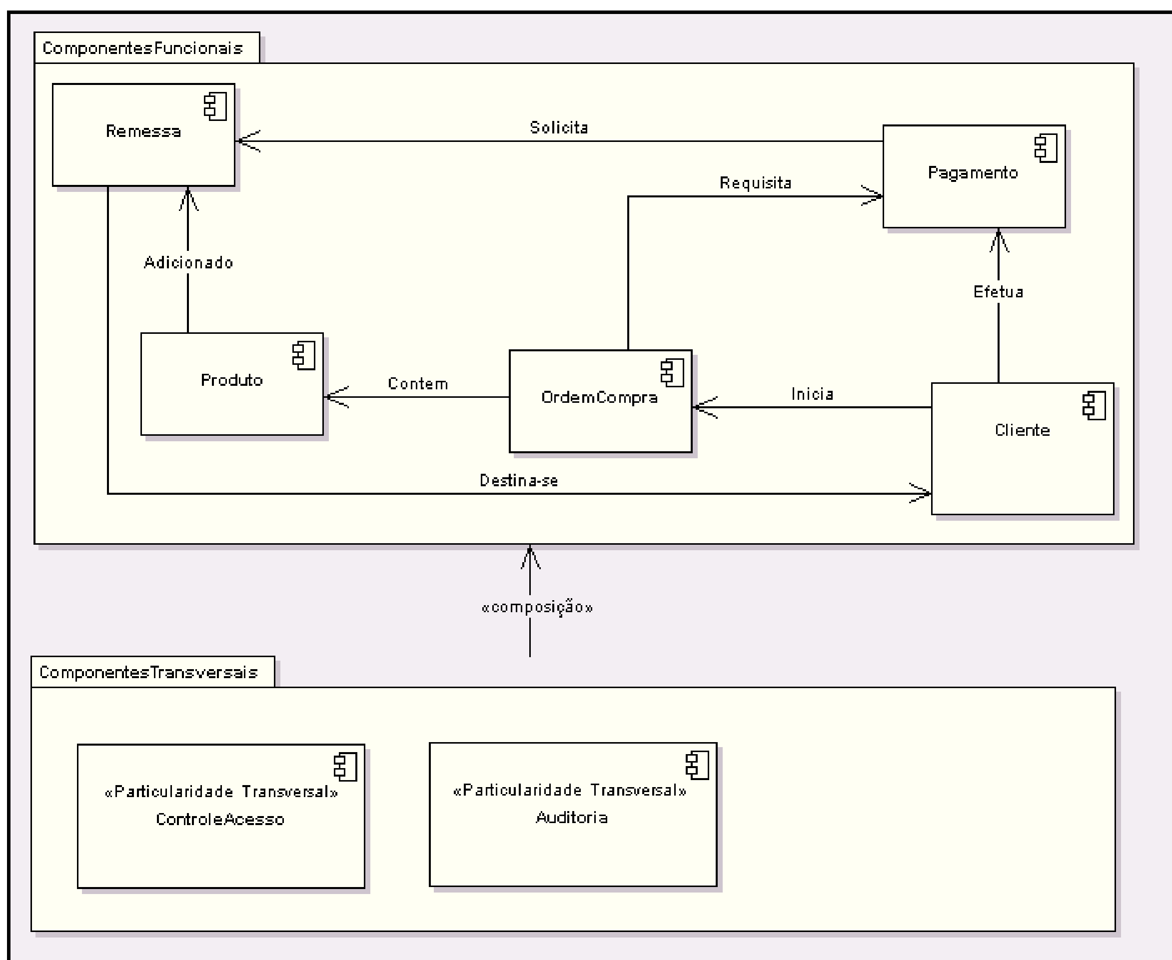

Figura 8.3 – Modelo Funcional-Transversal

Nos componentes funcionais, listados no pacote superior, não há qualquer preocupação com as características transversais. Elas estão listadas no pacote inferior. Em cada um dos componentes funcionais poderá atuar nenhum, um ou mais componentes transversais, entretanto, neste modelo, esta informação não está detalhada.

8.2 Modelo Componente-Variações

O Modelo de componente-variações apresenta a relação entre um componente funcional e as funcionalidades transversais. Para o componente remessa, podem ser aplicadas duas funcionalidades com três variações, e estas variações também podem ser aplicáveis a outros componentes (figura 8.4).

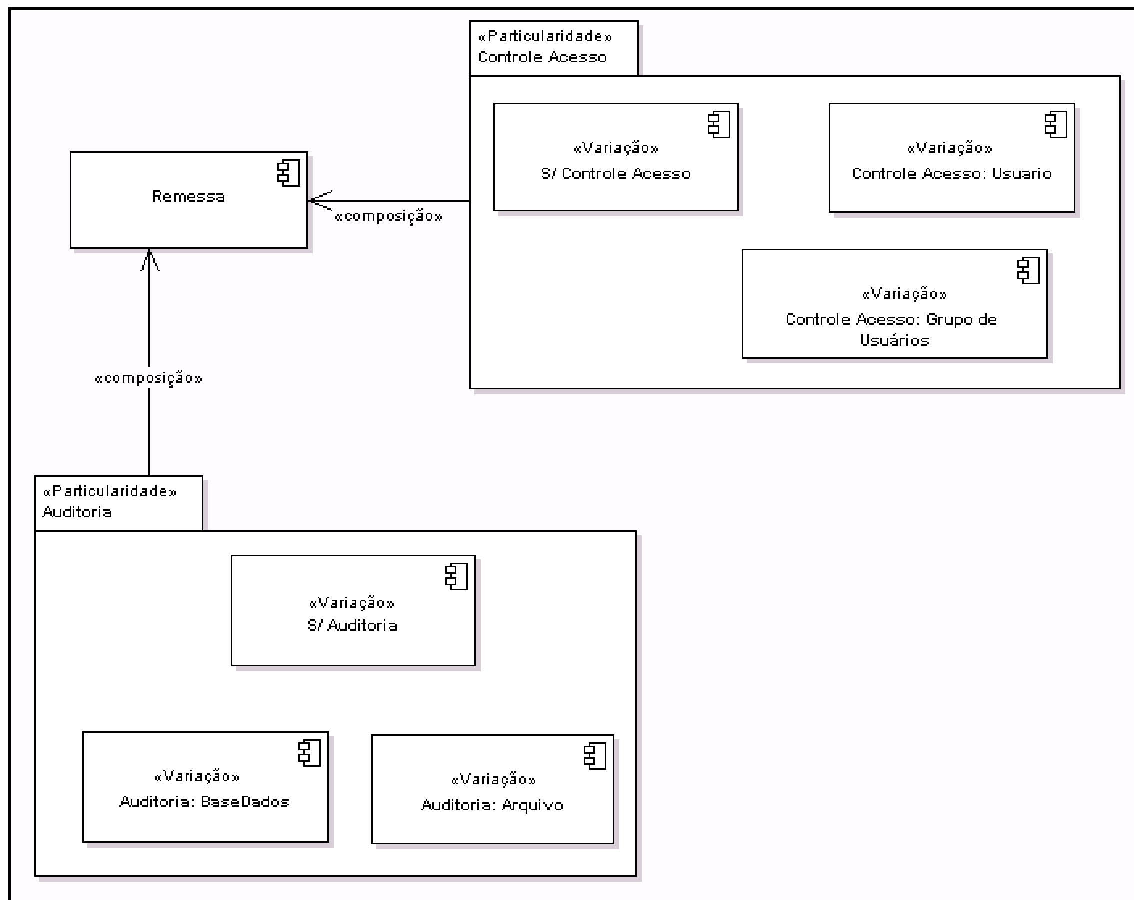

Figura 8.4 – Modelo de Componente-Variações

8.3 Modelo de Variação

O modelo de variação detalha a implementação de uma das possíveis variações de componentes. Neste diagrama fica modelado de forma clara como será a composição do sistema (figura 8.5). Para a funcionalidade Auditoria, foi selecionada a variação arquivo. Ela será mesclada ao código na execução da construtora com o adendo iniciar, na execução de todos os métodos públicos com método auditar e na execução do método destrutor através do método finalizar. O mesmo se aplica ao controle de usuário, porém de modo mais específico, onde foram selecionados métodos nomeados como iniciar e alterar.

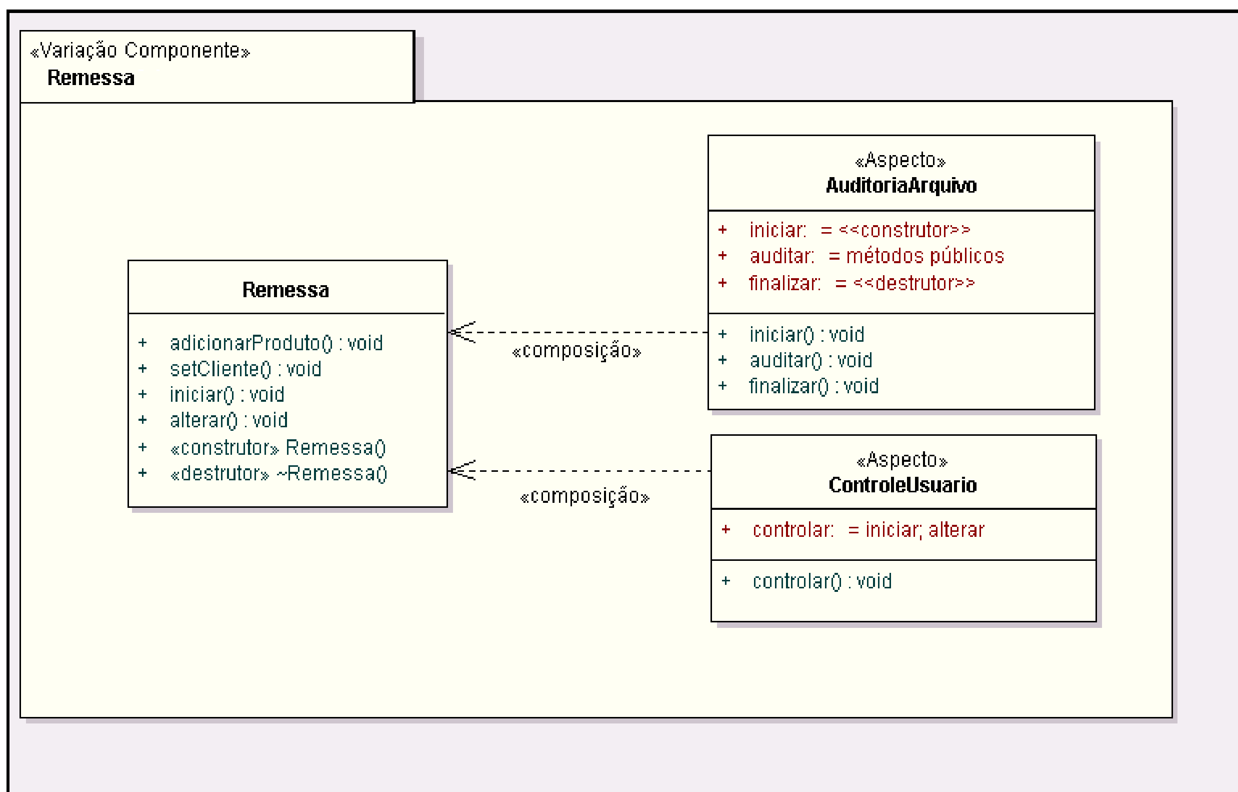

Figura 8.5 – Modelo de Variação

8.4 Modelo de Aplicação

O Diagrama de Aplicação modela o produto final da modelagem, aplicável a um contexto. Esta diagrama pode contribuir de forma documental, podendo relacionar um determinado produto a um grupo de clientes que o usam. Ele apresenta o conjunto de funcionalidades selecionado para componente Remessa (auditoria em arquivo e controle de acesso por usuário) (figura 8.6).

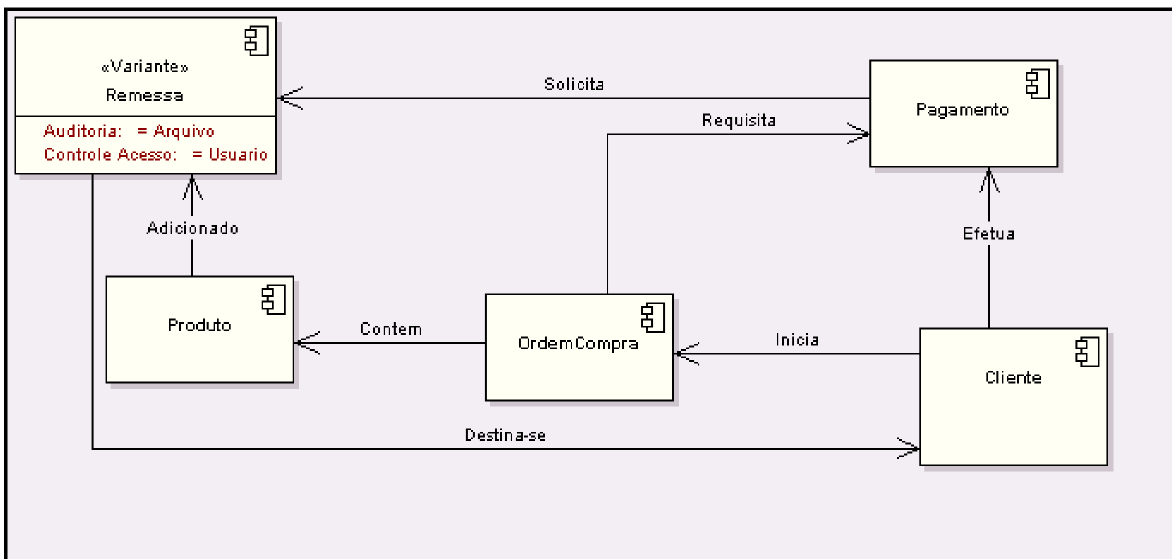

Figura 8.6 – Modelo de Aplicação

CAPÍTULO 9 - CONCLUSÃO

A indústria de software aplica a variabilidade funcional, ilustrada pelas personalizações de sistemas comercias, em abordagens que vão da empírica até a formalizada. Entretanto, a aplicação da variabilidade em características sistêmicas, pelas suas próprias características de paradigmas como o orientado a objetos ou procedural, é algo ainda difícil de ser alcançado. A dificuldade surge na incapacidade de modularizar os trechos de código referentes a estas características, impossibilitando o reúso ou o gerenciamento.

A orientação a aspectos permitiu uma nova forma de encarar e lidar com esta classe de problemas, fornecendo substratos que possibilitam criar mecanismos aplicáveis a ambientes produtivos. A capacidade de agrupar os códigos referentes a estas características transversais em unidades, possibilita a criação de meios para modelar, implementar e gerenciar variações. O uso da UML facilita a compreensão dos modelos, reduzindo o tempo de aprendizado, e o uso de estereótipos reduz a complexidade do modelo.

9.1 Aplicabilidade e Limitações da Proposta

A utilização da UML como fundamento para o presente trabalho amplia a aplicação do modelo, visto que a UML tornou-se uma linguagem padrão, em contrapartida, a orientação a aspectos, por ser um conceito relativamente pouco difundido, atua de modo contrário, mas não pode ser considerada uma grande barreira. Como ponto positivo, a aplicação do modelo não se restringe apenas às linhas de produto, *frameworks* ou desenvolvimento

baseado em componentes, podendo ser aplicado em outros meios de desenvolvimento onde se possa aplicar a orientação a aspectos e a abstração de componentização.

Uma limitação é a restrição ao tratar apenas das funcionalidades transversais. As funcionalidades funcionais poderiam ser tratadas de uma forma semelhante, entretanto haveria implicações na composição dos componentes.

9.2 Trabalhos Futuros

Como outros trabalhos futuros, listam-se:

- elaboração de um processo de análise para identificação das variações transversais;
- desenvolvimento de uma técnica de validação que possibilite garantir a qualidade das variações e da composição resultante;
- a criação de uma ferramenta que permita a utilização da presente proposta;
- detalhar sobre quais funcionalidades transversais poderão ter restrições para aplicação desta abordagem.

REFERÊNCIAS BIBLIOGRÁFICAS

ATKINSON, Colin ;KÜHNE, Thomas. Aspect-Oriented Development with Stratified Frameworks. IEEE Computer Society: IEEE SOFTWARE, Janeiro/Fevereiro de 2003. pág.: 81-89

BASILI, V. R.; ROMBACH, H. D., Support for Comprehensive Reuse. Software Engineering Journal, Vol. 6, Setembro de 1991, pág.: 301-318.

BASILI, V. R; CALDIERA,G. Identifying and Qualifying Reusable Software Components. IEEE Computer Society: IEEE Computer, Vol. 24, Issue 2, 1991, pág. 61-70.

BERGMANS, L.; AKSIT, M. Composing multiple concerns using composition filters, 2001. Disponível em: http://trese.cs.utwente.nl/composition filters/. Acessado em 12/01/2005.

BIDDLE, Robert; MARTIN, Angela; NOBLE, James. No name: just notes on software reuse. ACM SIGPLAN. Volume 38. ACM Press. NovaYork, USA. pág.: 76 a 96; Dezembro de 2003.

BOOCH, Grady. RUMBAUGH, James. JACOBSON, Ivar. UML: Guia do Usuário. Rio de Janeiro: Campus, 1999.

CAMARGO, Valter V.; RAMOS, Ricardo A.; MASIERO, Paulo César. Implementação de Variabilidade em *Frameworks* Orientado a Aspectos desenvolvidos em AspectJ. 2004.

CHAVEZ, Christina. GARCIA, Alessandro. LUCENA, Carlos. Desenvolvimento Orientado a Aspectos. Anais do XVII Simpósio Brasileiro de Engenharia de Software, Manaus – Amazonas. Universidade Federal do Amazonas. 2003.

CHAVEZ, Christina. LUCENA, Carlos. A Theory of Aspects for Aspect-Oriented Software Development. Anais do XVII Simpósio Brasileiro de Engenharia de Software. Manaus - Amazonas. Universidade Federal do Amazonas. 2003.

CLEMENTS, Paul C.; NORTHROP, Linda M. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003 ESC-TR-96-003. Fevereiro de 1996.

CRNKOVIC, Ivica; LARSSON, Magnus. Building Reliable Component-Based Software Systems. Artech House Computing. Boston. 2002

DUSINK, Liesbeth ; KATWIJK, Jan van. Reuse Dimensions.ACM: SSR. Seattle - EUA. 1995. pág.: 137-149

EIDSON, Thomas; DONGARRA, Jack; EIJKHOUT, Victor. Applying Aspect-Orient Programming Concepts to a Component-based Programming Model. International Parallel and Distributed Processing Symposium (IPDPS'03)

FILMAN, R; FRIEDMAN, D. Aspect-oriented programming is quantification nd obliviousness. Workshop em Advanced Separation of Concerns, OOPSLA 2000, Minneapolis - EUA, Outubro de 2000.

FRAKES, W., B.; FOX, C. J. Sixteen Questions about Software Reuse. Communications of the ACM, Junho de 1995.

FRAKES, W., B.; ISODA, S. Success Factors of Systematic Software Reuse. IEEE Computer Society: IEEE Software, Setembro de 1994.

FRAKES, William B.; BIGGERSTAFF, Ted J.; PRIETO-DIAZ, Ruben; MATSUMURA, Kazuo; SCHAEFER, Wilhelm. Software Reuse: Is It Delivering? International Conference

on Software Engineering, enviado para 13th international conference on Software engineering, Austin, Texas, EUA, IEEE Computer Society Press. pág 52-59. 1991

FRANCA, L. STAA, A. Geradores de Artefatos: Implementação e Instanciação de Frameworks. Anais do XV Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro – RJ. 2001.

FURLAN, José D. Modelagem de Objetos através da UML. São Paulo: Makron Books. 1998.

GIMENES, I. TRAVASSOS, G. O Enfoque de Linha de Produto para desenvolvimento de Software. Anais do XXII Congresso Brasileiro de Computação. Vol 2. Florianópolis - SC. 2002.

GRISS, Martin. Implementing Product-Line Features with Component Reuse – apresentado em: 6th International Conference on Software Reuse, Vienna, Áustria. junho de 2000.

GRISS, Martin. (2) Implementing Product-Line Features By Composing Component Aspects. Para apresentação em: First International Software Product Line Conference. Denver, CO. Agosto de 2000.

GURP; BOSCH; SVAHNBERG. Managing Variability in Software Product Lines. Disponível em http://citeseer.ist.psu.edu/568368.html. 2000. Acessado em dezembro de 2004.

GURP, J. van; BOSCH; J., SVAHNBERG, M. On the Notion of Variability in Software Product Lines. Segunda conferência Working IEEE / IFIP Conference on Software Architecture (WICSA), Pág 45-54. IEEE Computer Society, 2001. Disponível em http://citeseer.ist.psu.edu/vangurp01notion.html. Acessado em dezembro de 2004.

HERRERO, J.L.; SANCHEZ, F.; LUCIO, F.; TORRO, M. Introducing Separation of Aspects at Design Time. AOP Workshop em ECOOP'00. Cannes, France. 2000.

HOROWITZ, J. B.; MUNSON, E. An Expansive View of Reusable Software. IEEE Computer Society: IEEE Transactions on Software Engineering, 10(5):477487, Setembro de 1984.

KICZALES, G.; LAMPING,J. , Aspect-oriented programming, Para apresentação em: European Conference on Object-Oriented Programming (OOPSLA), Finlândia. Springer-Verlag, Junho 1997.

KICZALES,G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C. et al. Aspect-oriented programming. Para apresentação em: ECOOP'97, LNCS 1241. Springer, 1997. , pág: 220 a 242.

LIEBERHERR, K.; LORENZ, D.; MEZINI, M. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, março de 1999.

MCILORY, Douglas. Software Engineering, Relatório apresentado na conferência patrocinado por NATO Science Committee, Garmisch, Alemanha, Scientific Affairs Division, NATO, Brussels, 1969. Disponível em: http://cm.bell-labs.com/cm/cs/who/doug/components.txt. acessado em 10/12/2004. pág.: 138-155.

MEZINI, Mira ; OSTERMANN, Klaus. Variability Management with Feature Oriented Programming and Aspects. SIGSOFT'04/FSE12, outubro a novembro de 2004, Newport Beach, CA, EUA. Pág.:.127 - 136

MOHAGHEGHI, Parastoo; CONRADI, Reidar; KILLI, Ole M; SCHWARZ, Henrik. An Empirical Study of Software Reuse vs. Defect-Density and Stability. Para apresentação em: 26th International Conference on Software Engineering (ICSE'04). IEEE. 2004

MORISIO, M.; EZRAN, Tully, C. Success and Failure Factors in Software Reuse, IEEE Computer Society: IEEE Transactions on Software Engineering. Abril de 2002.

PAGE-JONES, Meilir. Fundamentos do Desenho Orientado a Objeto com UML. São Paulo: Makron Books, 2001.

PARNAS; D. On the Criteria to Be Used in Decomposing Systems into Modules Communication of ACM, vol. 15, no. 12, 1972, pág.: 1053-1058.

PIETRO-DÍAZ, Rubens. Software Reuse: Issues and Experiences. American Programmer Vol.6, No.8, pag:10-18. 1993.

PIETRO-DÍAZ, Rubens. Reuse in Engineering vs. Reuse in Software: Are They Incompatible? Symposium on Software Reusability SSR'2001. Toronto, Canada 17 a 19 de maio de 2001.

PIVETA, EDUARDO. Um Modelo de Suporte à Programação Orientada a Aspectos. Florianópolis - SC, Junho de 2001. Dissertação submetida à Universidade Federal de Santa Catarina para obtenção de título de mestre. Programa de pós-graduação em Ciência da Computação.

PRESSMAN, R. S. Engenharia de Software. 5 ed. Rio de Janeiro, RJ: McGraw-Hill, 2002.

SCHILDT, Herbert. HOLMES, James. A Arte do Java. São Paulo, SP. Campus. 2002.

SCHWARTZBACH, Michael. Developments in Object-Oriented Type Systems. Tutorial apresentado no POPL'94 - 21s.t Annual ACM SIGACT SIGPLAN (Symposium on Principles of Prgramming Languages). 1994.

SINTES, Tony. Aprenda Programação Orientada a Objetos em 21 dias. São Paulo: Makron Books, 2002

SOARES, Sérgio. BORBA, Paulo. AspectJ: Programação orientada a aspectos em Java. Anais do XVII Simpósio Brasileiro de Engenharia de Software, Manaus – AM. Universidade Federal do Amazonas. 2003.

SPINCZYK, Olaf; BEUCHE, Danilo. Modeling and Building Software Product Lines with Eclipse. OOPSLA 2004. ACM., Vancouver, British Columbia, Canadá. Outubro de 2004.

STAA, Arndt von. Programação Modular - Desenvolvendo programas complexos de forma organizada e segura. Rio de Janeiro. Campus, 2000.

STEIN, D.; HANENBERG, St.; UNLAND, R. Designing AspectOriented Crosscutting in UML. Artigo apresentado na AOSD-UML Workshop em AOSD'02. http://citeseer.ist.psu.edu/stein02designing.html. Acessado em março de 2005.

SUZUKI, Junichi e YAMAMOTO, Yoshikazu. Extending UML with Aspects: Aspect Support in the Design Phase. ECOOP Workshops. pág 299-300. Disponível em: http://citeseer.nj.nec.com/suzuki99extending.html. 1999. Acessado em março de 2005.

TRACZ, W.; Software Reuse: Motivators and Inhibitors. 32nd International Conference, San Francisco, Fevereiro de 1987, Digest of Papers. IEEE Computer Society:IEEE Computer Society Press Washington DC, 1987. págs.: 358-363.

WEISS, David M. , Defining Families: The Commonality Analysis. Lucent Technologies Bell Laboratories. submitted to TSE. Disponível em: http://citeseer.ist.psu.edu/mark97defining.html. 1997. Acessado em março de 2005.

WERNER, C., BRAGA, R. Desenvolvimento Baseado em Componentes. Anais do XIV Simpósio Brasileiro de Engenharia de Software. João Pessoa, PB. 2000.

ANEXO A - UML 1.4 (*Unified Modeling Language*)

No período entre o final da década de 1980 e o início da década de 1990, existia uma infinidade de métodos para a modelagem sistemas orientado a objetos. Muitos dos quais não possuíam nenhuma linguagem de modelagem capaz de atender inteiramente às suas necessidades. Novos métodos foram surgindo aproveitando as experiências, ocasionando um certo clima de competição. Na metade da década 1990 este processo culminou com os principais criadores reunindo-se para criar uma linguagem unificada capaz de atender aos métodos de forma satisfatória.

As metas que delimitaram a unificação foram:

- Estabelecer uma ligação explícita entre conceitos e código executável;
- Levar em conta os fatores de escala que são inerentes aos sistemas complexos e críticos;
- Criação de uma linguagem de modelagem utilizável tanto por homens quanto por máquinas.

A.1. Definição

A UML é a uma linguagem-padrão gráfica para a elaboração da estrutura de projetos de software resultante da unificação dos métodos de Booch, Rumbaugh e Jacabson. Ela provê a base para um comum, estável e expressivo método de desenvolvimento orientado a objetos, sendo, conseqüentemente, o sucessor direto e mais compatível dos métodos que foram unificados (BOOCH et al, 2000, MATTOS,1999).

Linguagens de modelagens, como a UML, formadas por vocabulário e regras focados na representação conceitual e física de um sistema; é a notação utilizada por algum método.

Um método, diferentemente, é a utilização do vocabulário e regras, de acordo com um processo a ser seguido (FOWLER, 2000).

A capacidade de modelagem da UML abrange desde sistemas baseados Web até sistemas complexos de tempo real. Ela não se restringe a representar sistemas orientados a objetos bem estruturados, mas também a sistemas legados e/ou mal definidos. Sua aplicação também não está limitada a sistemas de software, podendo ser aplicada na representação de sistemas de sociais (como de saúde ou legal) ou projeto de hardware.

A.2. Elementos da UML

A UML utiliza como pilar conceitual três elementos: blocos de construção (vocabulário), regras que determinam como os blocos serão combinados (gramática), mecanismos comuns aplicados sobre UML (meta-linguagem).

A.3. Blocos de Construção

Os blocos de construção da UML podem ser agrupados em três conjuntos: itens, relacionamentos e diagramas.

Os itens são os blocos básicos para construção de modelos. Os itens estruturais representam os elementos fundamentais de um sistema, sendo análogo aos substantivos. A segunda classe de itens representa os comportamentos, destinada às partes dinâmicas dos modelos. São equivalentes aos verbos de um modelo. Há itens destinados a representar agrupamentos, são as partes organizacionais dos modelos. Por último, os itens anotacionais são as partes explicativas da UML (BOOCH et al, 2000, pág 23-27).

As Classes são itens estruturais e podem ser representadas por basicamente duas formas. A primeira, mais simplificada, composta por um retângulo e o nome da classe ao centro. O nome pode estar sozinho (nome simples) ou acompanhado da indicação do pacote que a referida faz parte. A outra forma, mais complexa, é formado pelo retângulo contendo o

nome mais dois compartimentos abaixo, um situado na posição mais central contendo os atributos e outro, mais abaixo, contendo os métodos. Exemplos da representação de classe podem ser observados na figura A.1.

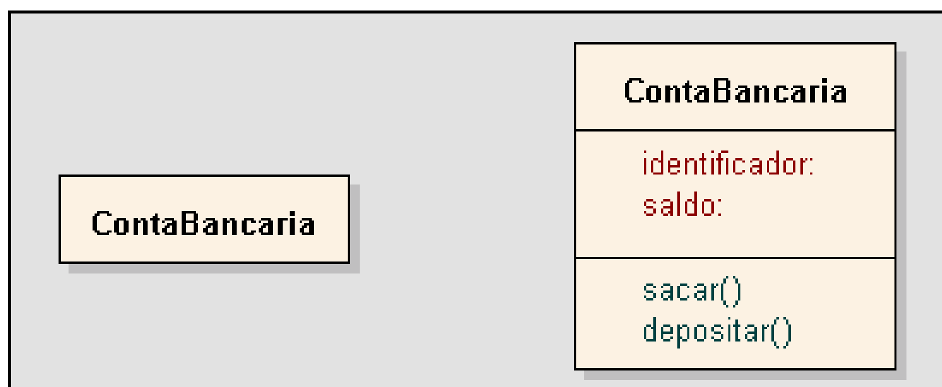

Figura A.1 – Exemplos de Classes

Objetos, ou instâncias, são representados de forma similar às classes, utilizando dois retângulos, um acima para o nome e outro abaixo para representar o estado, podendo este ser omitido. O nome é sublinhado, e pode ser indicada a classe que deu origem à instância (com o caminho) (BOOCH et al, 2000, pág 183). Exemplos de representação de objetos podem ser observados na figura A.2.

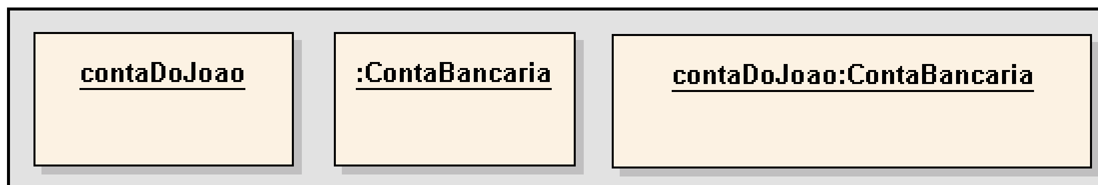

Figura A.2 – Exemplos de Objetos

Interface é um conjunto de operações que especificam serviços de uma classe com componente, descrevendo o comportamento externamente visível. Ela pode representar todo o comportamento de uma classe ou componente, com apenas parte. A interface é representada por um círculo e o respectivo nome (BOOCH et al, 2000, pág 18-19). Pode ser observado na figura A.3.

Componentes são partes físicas e substituíveis de um sistema, que proporcionam a realização de um conjunto de interfaces. Eles, normalmente, representam o pacote físico de

elementos lógicos diferentes. Os componentes são representados como retângulo com abas, incluindo apenas o seu nome (BOOCH et al, 2000, pág 20). Pode ser observado na figura A.3.

Os pacotes destinam-se à organização dos modelos da UML, agrupando os elementos. Os pacotes são puramente conceituais, existindo apenas em tempo de desenvolvimento. Um exemplo de pacote pode ser observado na figura A.3.

A nota é a parte explicativa dos modelos da UML, é um símbolo gráfico para a representação de restrições ou de comentários anexados a um elemento ou a uma coleção de elementos. Seu símbolo está representado na figura A.3.

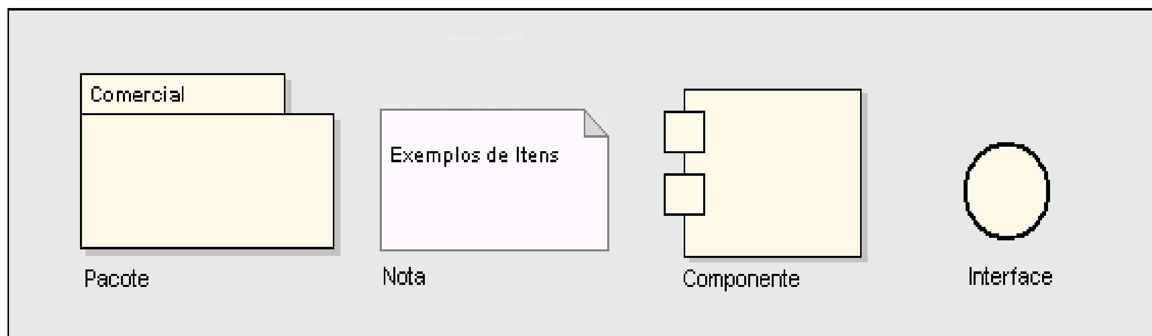

Figura A.3 – Itens da UML.

O segundo agrupamento de blocos de construção representa os vínculos entre os blocos de construção da UML (figura A.4). Podem ser de quatro tipos: dependência, associação, generalização e realização.

A dependência é a relação entre dois itens onde a alteração de um (o item independente) pode afetar a semântica do outro (do que depende). A associação representa a relação estrutural entre o todo e suas partes. A generalização permite informar os casos onde é aplicada a herança para demonstrar o vínculo entre um item base destinado ao caso genérico e o item derivado destinado ao caso específico. A realização é um relacionamento semântico entre classificadores, em que um classificador especifica um contrato que outro classificador garante executar.

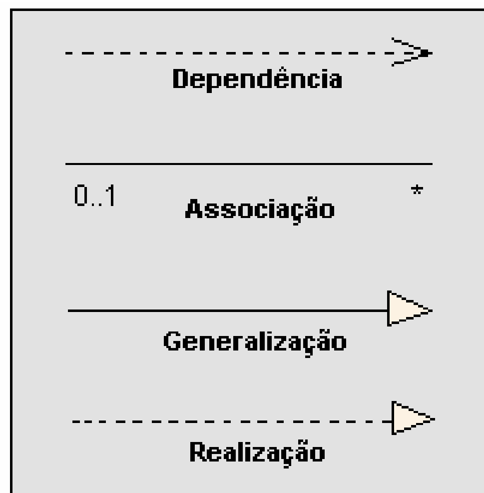

Figura A.4 – Tipos de relacionamento da UML

Os diagramas compõem o terceiro e último grupo de blocos de construção. Eles são a apresentação de vértices (itens) e arcos (relacionamentos) destinados à visualização de um sistema sob diferentes perspectivas. Eles são os meios utilizados para modelagem de sistemas na UML. Através deles são organizados os elementos relevantes, reunidos de modo a favorecer a compreensão. A UML prevê nove diagramas que procuram representar duas perspectivas: (i) comportamental e (ii) estrutural.

A perspectiva comportamental é utilizada visualizar, especificar, construir e documentar os aspectos dinâmicos de um sistema. Os aspectos dinâmicos de um sistema são aqueles os quais as suas partes sofrem alterações. Os cinco diagramas comportamentais são: de caso de uso, de seqüência, de colaboração, de gráfico de estados e de atividades.

A perspectiva estrutural procura representar o esqueleto de um sistema de modo estático. Dos nove diagramas, quatro são destinadas visualizar, especificar, construir e documentar os aspectos estáticos de um sistema. A relação dos Diagramas pode ser observada no quadro 5.1(BOOCH et al, 2000, pág 432-43).

Quadro A.1 – Diagramas da UML 1.4.

Diagrama	**Descrição**
Diagrama de classes	Apresenta um conjunto de classes, interfaces, colaborações e seus relacionamentos. É o diagrama encontrado com maior freqüência, representando uma visão estática da estrutura do sistema.
Diagrama de objetos	Mostra um conjunto de objetos e seus relacionamentos. Representa retratos estáticos de instâncias de itens encontrados em diagramas de classes, representando sob perspectiva de casos reais ou de protótipos.
Diagrama de componentes	Este diagrama exibir um conjunto de componentes e seus relacionamentos, apresentando as organizações e as dependências entre eles.
Diagrama de implantação	Um diagrama que mostra um conjunto de nós e seus relacionamentos. Abrange a visão estática do funcionamento de uma arquitetura.
Diagrama de caso de usos	Apresenta um conjunto de casos de uso, atores e seus relacionamentos.
Diagrama de seqüências	Mostra uma interação, dando ênfase à ordenação temporal das mensagens.
Diagrama de colaboração	Diagrama que apresenta uma interação, dando ênfase à organização estrutural de objetos que enviam e recebem mensagens.
Diagrama de gráfico de estados	Diagrama destinado a exibir uma máquina de estados, dando ênfase ao comportamento ordenado por eventos de um objeto.
Diagrama de atividades	Diagrama que mostra uma máquina de estados, dando ênfase ao fluxo de uma atividade para outra.

A.4. As Regras

A UML possui um conjunto determinado de regras que especificam o que deverá ser um modelo bem-formado. Os modelos bem formados são aqueles autoconsistentes semanticamente e em harmonia com todos os modelos a eles relacionados (BOOCH et al, 2000, pág 27)..

Há regras para: definição de nomes (como nomear elementos), escopo (contexto de um nome), visibilidade (permissão de acesso pelos outros), integridade (como os elementos se

relacionam) e execução. Elas destinam-se a incentivar reflexão sobre quais questões são relevantes na análise, no projeto e na implantação que levam os modelos a se tornarem bem formados ao longo do tempo.

A.5. Mecanismos Comuns

Na tentativa de tornar a UML mais simples quando aplicada, são previstos quatro mecanismos básicos, aplicados de maneira consistente na linguagem: (i) especificações, (ii) adornos, (iii) divisões comuns e (iv) mecanismos de extensões (BOOCH et al, 2000, pág 28 – 31).

Por trás de cada um das notações gráficas, existe uma especificação capaz de fornecer uma declaração textual da sintaxe e da semântica do respectivo bloco de construção, por exemplo, em uma classe são especificados atributos e métodos. Estas especificações fornecem um repertório semântico, contendo todas as partes de modelos de determinado sistema, cada parte relacionada às demais de forma consistente.

Os adornos representam de forma visual, única e direta, os aspectos mais importantes do elemento. No caso de um atributo de uma classe, a visibilidade pode ser apresentada através de um adorno onde o sinal ‘ – ’ simboliza a visibilidade privada, o sinal ‘+’ visibilidade pública e ‘#’ a visibilidade protegida.

Na modelagem de sistema costuma-se trabalhar com a divisão em duas parcelas de responsabilidades. Pode-se exemplificar esta divisão através da dualidade entre classe/objeto, interface/implemementação ou operações/métodos.

Os mecanismos de extensibilidade tornam a UML capaz de representar nuances que uma única linguagem fechada não seria capaz de representar. A extensibilidade é obtida através dos estereótipos, que permitem a criação de novos tipos de blocos de construção derivados de existentes, de valor que atribuído, estende as propriedades dos blocos de construção, e restrição, que ampliam as semânticas dos blocos de construção da UML.

O conjunto de itens estruturais disponibilizado pela UML permite modelar a grande maioria dos sistemas. Entretanto, às vezes, é preciso incluir novos itens, relacionados com o vocabulário do domínio que se está modelando. Para sanar esta necessidade, na UML há disponível o estereótipo. Os estereótipos são metatipos que criam o equivalente a uma nova classe no metamodelo da UML (BOOCH et al, 2000, pág 78-80). Eles acrescentam novos elementos ao vocabulário da UML.

Um estereótipo é representado pelo seu nome entre um duplo sinal de menor e um duplo sinal de maior. É possível acrescentar um ícone à direita do nome ou utilizar o próprio ícone como forma o estereótipo (op. cit.). Cada uma destas representações pode ser observada na figura A.5.

Figura A.5 – Formas de representar Estereótipos na UML.

Os valores atribuídos especificam pares de valores chaves de um elemento do modelo, indicando uma nova propriedade para um elemento existente da UML ou pode ser aplicado a estereótipos.

As restrições são um conjunto de regras estruturadas expressadas através da OCL ou de linguagem natural. Restrições é o meio pelo qual novas regras semânticas podem ser adicionadas a UML ou regras existentes podem ser modificadas.