Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture

Milton Cesar Fraga de Sousa

Trabalho Final de Mestrado Profissional

UNICAMP
BIBLIOTECA CENTRAL
DESENVOLVIMENTO DE COLEÇÕE

Instituto de Computação
Universidade Estadual de Campinas

Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture

Milton Cesar Fraga de Sousa
19 de abril de 2004

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
  Instituto de Computação - UNICAMP
- Prof. Dr. Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP
- Profa. Dra. Maria Beatriz Felgar de Toledo
  Instituto de Computação - UNICAMP
- Prof. Dra. Ariadne Maria Brito Rizzoni Carvalho (Suplente)
  Instituto de Computação - UNICAMP

UNICAMP
BIBLIOTECA CENTRAL
DESENVOLVIMENTO DE COLEÇÕES

NIDADE BC
CHAMADA
T/UNICAMP
So85p
EX
OMBO BC/ 61330
ROC. 16/361/05
C ☐ D ☑
REÇO 11,00
ATA 04-1-05
CPD
Bib Id 334841

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Sousa, Milton Cesar Fraga de

So85u Um processo de desenvolvimento baseado em componentes adaptado ao Model Driven Architecture / Milton Cesar Fraga de Sousa - - Campinas, [S.P. :s.n.], 2004.

Orientadora : Cecília Mary Fischer Rubira

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Software - Desenvolvimento. 3. Software - Arquitetura. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigida e defendida por Milton Cesar Fraga de Sousa e pela Banca Examinadora.

Campinas, 19 de abril de 2004

Profa. Dra. Cecília Mary Fischer Rubira
(Orientadora)

Trabalho final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 19 de abril de 2004, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Ivan Luiz Marques Ricarte
FEEC – UNICAMP

Profª. Drª. Maria Beatriz Felgar de Toledo
IC - UNICAMP

Profª. Drª. Cecília Mary Fischer Rubira
IC - UNICAMP

Resumo

O desenvolvimento de software baseado em componentes tem sido amplamente utilizado na construção de sistemas de grande porte. Nestes sistemas, intrinsecamente complexos, a adoção de um processo de desenvolvimento sistemático é muito importante. Outros fatores relevantes que podem ser considerados são: (1) a arquitetura de software, principal responsável pelo atendimento de requisitos não-funcionais, como tolerância a falhas e distribuição, (2) a evolução do sistema em face das mudanças tecnológicas, ponto central da abordagem *Model Driven Architecture* (MDA) proposta pelo consórcio *Object Management Group* (OMG) e (3) a distância semântica entre as abstrações da descrição arquitetural e as construções disponíveis nas plataformas alvo.

Este trabalho apresenta um processo de desenvolvimento de software baseado em componentes adaptado para (1) incorporar a abordagem MDA, (2) tratar explicitamente os requisitos não-funcionais através da arquitetura de software, e (3) reduzir a distância semântica entre as abstrações da descrição arquitetural através da utilização de modelos de estruturação de componentes independentes de plataformas como, por exemplo, J2EE e .NET.

A praticabilidade do processo adaptado baseado na abordagem MDA foi evidenciada no estudo de caso de um Sistema de Contingências Tributárias. Este estudo permitiu (1) demonstrar a utilização do processo proposto, (2) introduzir os conceitos da abordagem MDA, (3) tratar os requisitos não-funcionais através do refinamento da arquitetura de software, e (4) criar mapeamentos para as plataformas J2EE e .NET.

Abstract

Component-based software development has been widely used in the construction of large scale systems. In these systems, which are intrinsically complex, the adoption of a systematic development process is very important. Other relevant factors that should be considered in order to achieve a successful development of complex component-based system are: (1) its software architecture, main responsible for fulfilling the non-functional requirements, e.g. fault tolerance and distribution, (2) the system's evolution in face of technological changes, the central point of Model Driven Architecture (MDA) approach proposed by Object Management Group (OMG) consortium and (3) the semantic distance between the abstraction of the architectural description and the available constructions in the target platforms.

This work presents a component-based software development process adapted (1) to incorporate the MDA approach, (2) to address explicitly the non-functional requirements by means of the software architecture, and (3) to reduce the semantic distance between the abstraction's architectural description through the use of platform independent (e.g. J2EE and .NET) component structuring models.

The feasibility of the modified process based on the MDA approach was evidenced in the case study of a Tax Contingencies System. This study allowed us (1) to demonstrate the use of the proposed process, (2) to introduce the concepts of MDA approach, (3) to address the non-functional requirements through the refinement of the software architecture, and (4) to create mappings to the J2EE and .NET platforms.

Agradecimentos

Primeiramente agradeço a Deus por permitir que eu chegasse até aqui, superando todos os obstáculos.

Em segundo lugar gostaria de agradecer à minha Mãe pelo amor incondicional durante todos esses anos.

Um agradecimento todo especial à minha esposa Marilene, que, não só suportou a falta de atenção em detrimento do Mestrado, como me incentivou incansavelmente, espero poder retribuir.

Agradeço à minha orientadora, Cecília Rubira, pela amizade e paciência, por acreditar no meu esforço e trabalho, e também por me transmitir sua experiência e seus conhecimentos.

Agradeço ao Paulo Astério pela dedicação manifestada através das críticas e sugestões que muito contribuíram para o resultado final.

Agradeço à General Motors do Brasil, na pessoa do Sr. Carlos Campelo, por permitir a utilização do Sistema de Contingências Tributárias como estudo de caso, colaborando dessa forma para o enriquecimento deste trabalho.

Conteúdo

Resumo 5
Abstract 6
Conteúdo 8
Lista de Tabelas 11
Lista de Figuras 12
1 Introdução 15
1.1 O Problema 18
1.2 A Solução Proposta 18
1.3 Trabalhos Relacionados 19
1.4 Organização deste Documento 20
2 Fundamentos Teóricos 23
2.1 Desenvolvimento Baseado em Componentes 23
2.1.1 J2EE 25
2.1.2 .NET 27
2.2 Arquitetura de Software 28
2.2.1 Visões Arquiteturais 29
2.3 Processo de Desenvolvimento de Software Baseado em Componentes 30
2.4 O Processo UML Components 33
2.4.1 A Separação em Camadas 34
2.4.2 Os Fluxos de Trabalho 35
2.4.3 O Fluxo de Trabalho Definição de Requisitos 36
2.4.4 O Fluxo de Trabalho Especificação 38
2.4.5 Os Estereótipos do Processo UML Components 40
2.4.6 A Estrutura de Pastas para Armazenamento dos Modelos 41
2.5 A Abordagem Model Driven Architecture (MDA) 42
2.5.1 Os Modelos na Abordagem MDA 43
2.5.2 Especificações de Suporte para a Abordagem MDA 44
2.5.3 Extensões 45
2.5.4 Mapeamentos 48
2.5.5 Independência de Plataforma 49
2.6 O Modelo COSMOS 49
3 Um Processo Adaptado ao Model Driven Architecture 55

3.1 Adaptações Efetuadas no Processo UML Components 56
3.1.1 Estágio Definição de Requisitos Não-Funcionais 60
3.1.2 Estágio Refinamento da Arquitetura de Software 61
3.1.3 Estágio Mapeamento dos Componentes na Arquitetura de Software 64
3.1.4 Estágio Refinamento do PIM 65
3.1.5 Estágio Mapeamento do PIM para PSM 68
3.1.6 Estágio Mapeamento do PSM para Implementação 71
3.1.7 Estágio Preenchimento do Código Fonte 72
3.2 Mapeamentos PIM para PSM J2EE 72
3.2.1 Modelo de Componente COSMOS para J2EE 73
3.2.2 Modelo de Conector COSMOS para J2EE 77
3.2.3 Perfil para Mapeamento de Componentes dos Modelos PIM para PSM J2EE 81
3.2.4 Perfil para Mapeamento de Conectores dos Modelos PIM para PSM J2EE 84
3.2.5 Perfil para Mapeamento dos Demais Elementos dos Modelos PIM para PSM J2EE 86
3.3 Mapeamentos PIM para PSM .NET 86
3.3.1 Modelo de Componente COSMOS para .NET 87
3.3.2 Modelo de Conector COSMOS para .NET 90
3.3.3 Perfil para Mapeamento de Componentes dos Modelos PIM para PSM .NET 92
3.3.4 Perfil para Mapeamento de Conectores dos Modelos PIM para PSM .NET 96
3.3.5 Perfil para Mapeamento dos Demais Elementos dos Modelos PIM para PSM .NET 97
3.4 Mapeamento do PSM para Implementação 97
3.4.1 Mapeamento do PSM J2EE para Implementação 98
3.4.2 Mapeamento do PSM .NET para Implementação 98
4 Estudo de Caso: Um Sistema de Contingências Tributárias 99
4.1 Descrição do Sistema de Contingências Tributárias 99
4.2 Fluxo de Trabalho Definição de Requisitos 100
4.2.1 Estágio Processo do Negócio 100
4.2.2 Estágio Previsão do Sistema 101
4.2.3 Estágio Modelo Conceitual do Negócio 101
4.2.4 Estágio Modelo de Casos de Uso 103
4.2.5 Estágio Definição de Requisitos Não-Funcionais 107
4.3 Fluxo de Trabalho Especificação 108

4.3.1 Estágio Identificação de Componentes 108
4.3.2 Estágio Interação de Componentes 111
4.3.3 Estágio Especificação de Componentes 113
4.3.4 Estágio Refinamento da Arquitetura de Software 113
4.3.5 Estágio Mapeamento dos Componentes na Arquitetura de Software 118
4.3.6 Estágio Refinamento do PIM 122
4.4 Fluxo de Trabalho Provisionamento 123
4.4.1 Estágio Mapeamento do PIM para PSM 123
4.4.2 Estágio Mapeamento do PSM para Implementação 123
4.4.3 Estágio Preenchimento do Código Fonte 123
4.5 Fluxos de Trabalho Montagem, Testes e Implementação 124
4.6 Resultados Obtidos 124
5 Considerações Finais 127
5.1 Conclusões 128
5.2 Contribuições 130
5.3 Trabalhos Futuros 131
6 Referências Bibliográficas 133

Lista de Tabelas

Tabela 1 – Estereótipos definidos pelo processo UML Components 41
Tabela 2 – Estereótipos acrescentados ao processo UML Components 67
Tabela 3 – Estereótipos para definição do perfil UML para .NET 93
Tabela 4 – Tabela de verificação de criação/destruição 104
Tabela 5 – Tabela de verificação de atualização de associação 105
Tabela 6 – Tabela de verificação de atualização de associação 108

Lista de Figuras

Figura 1 - Processo típico [KLEPPE+03] 31
Figura 2 – Arquitetura em camadas usada pelo processo UML Components [CHEESMAN+01] 34
Figura 3 - Os fluxos de trabalho no processo UML Components [CHEESMAN+01] 35
Figura 4 - Os estágios do fluxo de trabalho Definição de Requisitos no processo UML Components 37
Figura 5 - Os estágios do fluxo de trabalho Especificação no processo UML Components 38
Figura 6 – Diagramas de modelagem de componentes no processo UML Components [CHEESMAN+01] 41
Figura 7 - Fragmento dos estereótipos e valores etiquetados definidos no modelo EJB [UML4EJB] 46
Figura 8 – Exemplo de extensão de um fragmento do metamodelo UML 47
Figura 9 – Modelo COSMOS - Solução para estruturação de componentes [SILVA03] 49
Figura 10 – Modelo COSMOS - Solução para o projeto dos componentes [SILVA03] 51
Figura 11 – Modelo COSMOS - Representação de componentes [SILVA03] 51
Figura 12 – Modelo COSMOS – Estrutura interna do componente [SILVA03] 52
Figura 13 – Modelo COSMOS – Componentes e conectores [SILVA03] 53
Figura 14 – Adaptações efetuados no processo UML Components 56
Figura 15 - Os estágios do fluxo de trabalho Definição de Requisitos no processo adaptado 57
Figura 16 - Os estágios do fluxo de trabalho Especificação no processo adaptado 58
Figura 17 - Os estágios do fluxo de trabalho Provisionamento no processo adaptado 59
Figura 18 – Diagramas de modelagem de componentes no processo adaptado 60
Figura 19 – Atividades do estágio Definição dos Requisitos não-Funcionais 61
Figura 20 – Atividades do estágio Definição da Arquitetura de Software 63
Figura 21 – Atividades do estágio Mapeamento dos Componentes na Arquitetura de Software 64
Figura 22 – Atividades do estágio Refinamento do PIM 66
Figura 23 – Atividades do estágio Mapeamento do PIM para PSM 68

Figura 24 – Atividades do estágio Mapeamento do PIM para PSM 71
Figura 25 – Exemplo de componente PIM 81
Figura 26 –Exemplo de interface PSM J2EE 81
Figura 27 – Exemplo de componente PSM J2EE 82
Figura 28 – Exemplo de interface e componente PSM .NET 94
Figura 29 - Diagrama de atividades do processo de negócios 100
Figura 30 - Modelo Conceitual do Negócio 102
Figura 31 - Diagrama de correspondência caso de uso e interface de componente de serviço de sistema 108
Figura 32 - Modelo de Tipos do Negócio 109
Figura 33 - Diagrama de responsabilidade de interface 110
Figura 34 - Especificação inicial da arquitetura de componentes 110
Figura 35 - Diagrama de colaboração 111
Figura 36 – Tipos de dados estruturados 112
Figura 37 - Especificação de interface de componente de negócio 113
Figura 38 – Visão lógica da arquitetura de software do sistema de contingências tributárias 113
Figura 39 – Visão de processos da arquitetura de software do sistema de contingências tributárias 115
Figura 40 – Interface do componente da camada de acesso 118
Figura 41 – Interface de entidade básica 119
Figura 42 – Interface do componente de persistência 119
Figura 43 – Interface de tipo de dados 120
Figura 44 - Diagrama de especificação de componentes – parte 1 120
Figura 45 - Diagrama de especificação de componentes – parte 2 121
Figura 46 - Diagrama de especificação de componentes – parte 3 122

Capítulo 1

Introdução

O quanto antes você começar, mais tempo demorará [BROOKS95]. Esta frase, aplicada ao desenvolvimento de software, embora pareça paradoxal, sinaliza a necessidade de planejar com antecedência os passos a serem tomados evitando descobrir, depois de pronto, ou no meio do caminho, que determinada decisão não era a mais acertada. Esta necessidade de planejamento prévio surge da complexidade inerente ao desenvolvimento de software, enfatizando a importância deste período. Este tempo dispendido inicialmente deve ser direcionado através de um processo. Este processo normalmente é dividido em: (1) processo gerencial, que programa as atividades, planeja entregas, aloca recursos, e monitora o andamento, e (2) processo de desenvolvimento, que cria o software a partir dos requisitos [CHEESMAN+01]. Este trabalho trata do processo de desenvolvimento, que é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). Ele é composto de boas práticas de engenharia de software que conduzem o desenvolvimento, reduzindo os riscos e aumentando a confiabilidade [JACOBSON+99].

Com a finalidade de diminuir o custo de desenvolvimento, minimizar a complexidade e melhorar a qualidade dos sistemas de software, no final dos anos 60 surgiu a idéia de reutilizar partes de sistemas já existentes [MCLLROY69]. Reutilização diz respeito à extensão com que um programa (ou partes de um programa) pode ser reutilizado em outras aplicações [MCCALL+77]. O conceito de modularização, decomposição do sistema em partes independentes (módulos) [GOGUEN86], foi o alicerce para o surgimento de uma metodologia de desenvolvimento de software, centrada na composição de componentes de software - unidades de software desenvolvidas e testadas separadamente e que podem ser integradas com outras para construir algo com maior funcionalidade [SZYPERSKI97] - chamada desenvolvimento baseado em componentes.

Para prover suporte ao desenvolvimento baseado em componentes surgiram algumas propostas de processo de desenvolvimento baseado em componentes como, por exemplo, *Catalysis* [DSOUZA+99], *Andersen Consulting CBD* [NING98] e *UML Components*

[CHEESMAN+01]. Desses destacamos o processo *UML Components* que utiliza a linguagem UML[1] [UML03]. O processo *UML Components* é dividido inicialmente em fluxos de trabalho, e cada fluxo de trabalho pode ser subdividido em estágios. O processo *UML Components* tem se mostrado eficaz na especificação de sistemas de software baseados em componentes, em virtude de sua simplicidade e facilidade de uso.

Juntamente com o processo de desenvolvimento pelo menos outras questões têm se mostrado relevantes no desenvolvimento de sistemas de software: (1) a evolução do software em função de mudanças tecnológicas, (2) o aumento da importância da arquitetura de software e (3) a distância entre as abstrações de uma descrição arquitetural e as construções disponíveis nas plataformas alvo.

A primeira questão, da evolução do software em função de mudanças tecnológicas, motivou a recente proposta do consórcio *Object Management Group* (OMG), de uma nova abordagem que separa a especificação da funcionalidade de um sistema da especificação da implementação desta funcionalidade em uma plataforma tecnológica específica, chamada de *Model Driven Architecture* (MDA) [MDA01]. Na abordagem MDA o termo plataforma é utilizado para referir-se a detalhes tecnológicos e de engenharia que são irrelevantes à funcionalidade fundamental de um componente de software [MDA01]. A abordagem MDA define que os modelos utilizados para representar um sistema devem utilizar a linguagem UML [UML03]. A abordagem MDA define duas categorias de modelos: modelos independentes de plataforma (PIM[2]) e modelos dependentes de plataforma (PSM[3]). O PIM provê especificação formal da estrutura e função do sistema que abstrai detalhes técnicos. O PSM é um refinamento do PIM que acrescenta os detalhes técnicos de uma determinada plataforma. Outro conceito importante na abordagem MDA é a idéia de mapeamento. Um mapeamento é um conjunto de regras e técnicas usadas para modificar um modelo no sentido de obter-se outro modelo [MDA01]. A manutenção de software contabiliza mais esforços que qualquer outra atividade de engenharia de software [PRESSMAN01], portanto, facilitar a evolução com relação às mudanças

---

[1] do inglês Unified Modeling Language

[2] do inglês Platform Independent Model

[3] do inglês Platform Specific Model

tecnológicas é particularmente importante atualmente na medida em que essas mudanças tecnológicas são mais freqüentes. A abordagem MDA, contudo, não prescreve um processo de desenvolvimento de software, apenas estabelece diretrizes que permitem o desenvolvimento de sistemas consoantes com a abordagem recomendada.

A segunda questão trata da arquitetura de software que pode ser definida como a estrutura ou estruturas de um sistema que englobam elementos de software, as propriedades externamente visíveis destes elementos, e as relações entre eles [BASS+03]. Sua importância tem crescido na medida em que: (1) os requisitos não-funcionais, aqueles que especificam características relativas à qualidade dos serviços como, por exemplo, desempenho e disponibilidade, ganham cada vez mais importância, e freqüentemente estes estão relacionados com alguma propriedade arquitetural [SOMMERVILLE01], (2) sistemas computacionais modernos são baseados na integração de numerosos componentes de software existentes, desenvolvidos por fontes independentes [BROWN+98], e (3) reutilização de software é melhor alcançada dentro de um contexto arquitetural [BASS+03].

Com relação à terceira questão, embora grandes empresas, como Microsoft e Sun, ofereçam plataformas para desenvolvimento baseado em componentes, com modelos como EJB [EJB+01] e .NET [DOTNET], existe uma distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas linguagens de programação, utilizadas para implementar o sistema de software [SILVA03] o que torna difícil materializar os componentes especificados, de acordo com determinado processo de desenvolvimento em uma determinada arquitetura. Visando diminuir tal distância foi proposto por Silva [SILVA03] um modelo para estruturação de componentes de software para sistemas orientados a objetos, chamado de COSMOS[4]. O modelo COSMOS define elementos que traduzem os elementos arquiteturais (componentes, interfaces e conectores), participantes da arquitetura de um sistema de software, que podem ser implementados em linguagens de programação ou mapeados para plataformas de componentes como J2EE e .NET.

---

[4] do inglês Component Structuring Model for Object-oriented Systems

1.1 O Problema

Alguns aspectos mencionados anteriormente não são tratados pelo processo *UML Components* [CHEESMAN+01], como: (1) evolução do software em função das mudanças tecnológicas, (2) tratamento dos requisitos não-funcionais, e (3) a distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas linguagens de programação, utilizadas para implementar o sistema de software [SILVA03].

Na questão da evolução do software em função das mudanças tecnológicas, embora o processo *UML Components* não esteja intrinsecamente vinculado a uma determinada plataforma, ele oferece pouco suporte para o mapeamento das especificações geradas ao longo do processo, o que torna difícil alterar as especificações, que normalmente são desenvolvidas focadas em uma determinada plataforma.

Na questão do tratamento dos requisitos não-funcionais, o processo *UML Components* estabelece como premissa uma arquitetura em quatro camadas: (1) de apresentação, (2) de diálogo, (3) de sistema e (4) de negócio. Embora esta estrutura atenda satisfatoriamente os requisitos funcionais, aqueles que representam os comportamentos que sistema deve apresentar diante de certas ações de seus usuários, ela não trata totalmente necessidades decorrentes dos requisitos não-funcionais, como segurança, distribuição e tolerância a falhas.

Na questão da distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas linguagens de programação, utilizadas para implementar o sistema de software o processo *UML Components* não introduz nenhum modelo que reduza tal distância, apenas elenca as questões que devem ser tratadas pelo desenvolvedor para produzir um software consoante com as especificações. Por outro lado o modelo COSMOS não especifica mapeamentos para plataformas específicas como J2EE ou .NET.

1.2 A Solução Proposta

A solução proposta é a adaptação de um processo de desenvolvimento de software baseado em componentes para incorporar as diretrizes da abordagem MDA proposta pelo consórcio OMG.

Essa adaptação inclui o tratamento explícito dos requisitos não-funcionais através do refinamento da arquitetura de software, e do uso de um modelo de estruturação de componentes

independente de plataforma, como por exemplo J2EE e .NET. O uso desse modelo independente de plataforma permite um mapeamento entre as abstrações de uma descrição arquitetural (baseada em componentes e conectores) para construções disponíveis nas plataformas específicas de componentes.

As diretrizes da abordagem MDA são incorporadas através: (1) da separação explícita entre os modelos independentes de plataforma e os modelos dependentes de plataforma, (2) e das regras de mapeamento entre ambos. A separação explícita é feita através da separação dos modelos gerados que foram nomeados como modelos independentes de plataforma (PIM) e modelos dependentes de plataforma (PSM). As regras de mapeamentos entre os modelos independentes e dependentes de plataforma são formalizadas através da especificação do modelo COSMOS para J2EE e .NET.

O tratamento dos requisitos não-funcionais é feito através: (1) da definição dos requisitos não-funcionais do sistema, e (2) do refinamento da arquitetura em camadas proposta pelo processo *UML Components*.

A incorporação do modelo de estruturação de componentes COSMOS é feita através da criação de modelos derivados para J2EE [EJB+01] e .NET [DOTNET], plataformas escolhidas em função da grande utilização no mercado, e de regras de mapeamento que permitem gerar tais modelos a partir dos modelos independentes de plataforma (PIM), conforme prescrito pela abordagem MDA.

1.3 Trabalhos Relacionados

A abordagem KobrA [COLIN+01] utiliza uma representação de componentes baseada em modelos UML e na abordagem de linha de produtos. A abordagem de linha de produtos objetiva a criação de componentes genéricos de software que são reutilizáveis para uma família de produtos alvo. A abordagem KobrA utiliza um método, chamado PuLSE[5], que prescreve um ciclo de vida com as seguintes fases: inicialização, construção da infra-estrutura de linha de produtos, uso e evolução. A semelhança da abordagem KobrA com o processo adaptado, é que ambos apoiam a abordagem MDA e o desenvolvimento baseado em componentes. Em comparação com o processo adaptado proposto a abordagem KobrA é mais extensa e mais

[5] do inglês Product Line Software Engineering

complexa por incorporar outros aspectos do desenvolvimento, como por exemplo, a infra-estrutura de linha de produtos.

O processo *Executable UML* baseia-se na construção de um conjunto de modelos precisos e testáveis do sistema a ser desenvolvido [MELLOR+01]. O processo *Executable UML* cria modelos que são classificados da seguinte forma: (1) dados, representados através de classes, atributos, associações e restrições, utilizando diagramas de classe da linguagem UML, (2) controles, representados através de estados, eventos, transições e procedimentos, utilizando diagramas de estado da linguagem UML e (3) algoritmos, representados através de ações, utilizando semântica de ação da linguagem UML. Este método assemelha-se ao processo adaptado por ser baseado na abordagem MDA, no entanto não é um método específico para desenvolvimento baseado em componentes, e, portanto, não se concentra na especificação de interfaces e componentes. A sua principal vantagem é enfatizar a especificação dos algoritmos através da utilização intensiva de semântica de ação da linguagem UML.

1.4 Organização deste Documento

Além deste capítulo introdutório este documento divide-se em quatro outros capítulos, da seguinte forma:

- **Capítulo 2 - Fundamentos Teóricos**: Como base para este trabalho o segundo capítulo apresenta fundamentos teóricos das áreas que o compõem, são elas: Desenvolvimento Baseado em Componentes, Arquitetura de Software, Processo de Desenvolvimento de Software Baseado em Componentes, o Processo *UML Components*, a Abordagem MDA, e o Modelo de Estruturação de Componentes COSMOS.
- **Capítulo 3 - Um Processo Adaptado ao Model Driven Architecture**: Neste capítulo são apresentadas as adaptações efetuadas no processo *UML Components*, bem como os Mapeamentos criados a partir do modelo COSMOS.
- **Capítulo 4 - Estudo de Caso: Um Sistema de Contingências Tributárias**: Neste capítulo é apresentado o resultado da aplicação do processo adaptado apresentado no capítulo 3 a um Sistema de Contingências Tributárias.
- **Capítulo 5 - Considerações** Finais: Este capítulo sintetiza as conclusões e

Capítulo 2

Fundamentos Teóricos

Este capítulo apresenta a terminologia e define os conceitos utilizados neste trabalho. Na seção 2.1 são apresentadas os princípios de desenvolvimento baseado em componentes utilizados neste trabalho, tais como componentes, interfaces providas e requeridas e conectores. Na seção 2.2 são apresentados os conceitos relacionados à arquitetura de software utilizados neste trabalho tais como propriedades arquiteturais e visões do sistema, bem como uma breve descrição das plataformas J2EE e .NET utilizadas no estudo de caso do capítulo 4. Na seção 2.3 são apresentados conceitos relativos a processos de desenvolvimento de software baseado em componentes. Na seção 2.4 é feita uma descrição do processo de desenvolvimento *UML Components*, que é o processo tomado como base no trabalho apresentado. Na seção 2.5 são apresentados os principais conceitos relacionados à abordagem *Model Driven Architecture,* aplicada ao processo *UML Components* no capítulo 3. Na seção 2.6 é apresentado um modelo para estruturação de componentes, chamado COSMOS, que visa diminuir a distância entre uma arquitetura de software baseada em componentes e sua implementação, utilizado como base para a criação dos mapeamentos (seção 2.5.4) entre os modelos independentes de plataforma e os modelos dependentes de plataforma (seção 2.5.1), prescritos na abordagem MDA.

2.1 Desenvolvimento Baseado em Componentes

Desenvolvimento baseado em componentes (DBC) enfatiza o projeto e a construção de sistemas baseados em computador usando componentes de software reutilizáveis [PRESSMAN01].

O modelo de desenvolvimento baseado em componentes lida com reutilização, e reutilização provê aos engenheiros de software vários benefícios mensuráveis. Baseado em estudos de reutilização, QSM Associates, Inc. relatou que a utilização de componentes implicou uma redução de 70% no tempo do ciclo de desenvolvimento; e uma redução de 84% no custo do projeto [YOURDON94].

O termo componente pode ser definido em diferentes níveis, a seguir são apresentadas algumas possibilidades [BROWN+96]:

- Componente: uma parte não trivial, quase independente, e substituível de um sistema que provê uma clara função em um contexto de uma arquitetura bem definida.
- Componente de software em tempo de execução: um pacote dinâmico ligável de um ou mais programas gerenciados como uma unidade e acessados através de interfaces documentadas que podem ser descobertas em tempo de execução.
- Componente de software: uma unidade de composição com dependências somente especificadas e explicitadas contratualmente.
- Componente de negócio: uma implementação de software de um conceito ou processo de negócio autônomo.

Os componentes incorporam alguns princípios básicos de orientação a objetos como (1) unificação de dados e funções, (2) encapsulamento, e (3) identidade. A unificação de dados e funções reforça a coesão, pois coloca os dados e as funções que os processam em uma mesma unidade. O encapsulamento reduz os problemas de acoplamento, uma vez que não expõe a forma como um objeto armazena seus atributos ou implementa suas operações. A identidade é a propriedade pela qualquer cada objeto pode ser unicamente identificado. Os componentes estendem estes princípios através da adoção de uma interface.

Uma interface é uma especificação do comportamento, através da qual é possível acessar um determinado serviço [PORTER92].

As interfaces de um componente podem ser classificadas em: (1) interfaces providas e (2) interfaces requeridas. Uma interface provida é uma interface representando os serviços providos por um componente para o mundo externo [NING99]. Uma interface requerida é uma interface representando os serviços que devem ser recebidos pelo componente para permitir que o mesmo execute as suas próprias operações [NING99].

Outro ponto importante é que o DBC reforça a idéia de projeto baseado em contrato. A idéia de projeto baseado em contrato foi bastante desenvolvida em orientação a objetos por Meyer [MEYER97] e aplicada aos componentes através do Catalysis [DSOUZA+99]. No projeto baseado em contrato podem-se distinguir dois tipos de contratos:

- Contratos de uso: contrato entre a interface do componente e seus clientes. Nesses contratos além da completa definição de interfaces (atributos, assinaturas das operações), devem ser definidas pré- e pós-condições para as operações, e, se necessário, invariantes

para os componentes.

- Contratos de realização: contrato entre a especificação do componente e sua implementação. Nesses contratos devem ser explicitadas regras e restrições quanto à implementação de determinadas funcionalidades, incluindo neste quesito a adoção de padrões de projeto.

O DBC incorpora a filosofia *compre, não faça* sustentada por Brooks [BROOKS87]. Na mesma maneira que as antigas sub-rotinas liberavam o programador de pensar nos detalhes, o DBC transfere a ênfase da programação do software para a composição de sistemas de software. O foco da implementação dá lugar à integração.

Em função do foco na integração um elemento bastante utilizado na construção de sistemas baseados em componentes é o conector. Conectores são abstrações das inter-relações dos componentes [TAI98]. No nível de especificação, um conector associa as interfaces providas e requeridas de um componente através de uma especificação abstrata de um estilo de interação entre componentes [NING99]. Em tempo de execução uma instância de um conector pode ser um pacote independente ou ser pode ser empacotado como parte de um código *proxy/stub* dentro da instância do componente.

Diversas plataformas têm sido propostas na literatura para o desenvolvimento de sistemas modernos, com arquiteturas em camadas e baseadas em componentes. Dentre elas podemos destacar duas bastante utilizadas pelos desenvolvedores de software, J2EE e .NET, e que por este motivo foram escolhidas para implementação do estudo de caso descrito no capítulo 4. Estas duas plataformas são descritas a seguir.

2.1.1 J2EE

J2EE é abreviação da especificação de plataforma *Java 2 Enterprise Edition* da Sun. J2EE provê uma descrição padronizada de como programas distribuídos orientados a objetos escritos na linguagem Java devem ser projetados e desenvolvidos e como os vários componentes Java podem se comunicar e interagir [BASS+03].

EJB[6], uma porção importante da especificação J2EE, e a que será mais utilizada neste trabalho, descreve um modelo de programação baseado em componente para o lado do servidor

[6] do inglês Enterprise JavaBeans

que objetiva:

- Prover uma arquitetura baseada em componentes para construção de aplicações de negócio distribuídas orientadas a objetos. EJB torna possível a construção de aplicações distribuídas pela combinação de componentes desenvolvidos com ferramentas de diferentes fabricantes.
- Tornar mais fácil escrever aplicações. Desenvolvedores de aplicação não tem que lidar com detalhes de baixo nível de transação, gerenciamento de estado, multi-tarefa, e alocação de recursos.

Mais especificamente, a arquitetura EJB faz o seguinte:

- Trata o desenvolvimento, implementação, e aspectos de tempo de execução do ciclo de vida de uma aplicação empresarial.
- Define o contrato que habilita ferramentas de vários fabricantes desenvolverem e implantarem componentes que podem operar em conjunto em tempo de execução.
- Opera em conjunto com outras interfaces de programa de aplicação[7] Java.

Além da especificação EJB o J2EE inclui várias outras facilidades, dentre as quais destacamos:

- *API JDBC* para acesso a banco de dados;
- Tecnologia *CORBA* para interação com recursos empresariais existentes;
- *Java Servlets API*, *JavaServer Pages* e tecnologia *XML*[8] para desenvolvimento para Internet;
- *Web Services* baseados em *SOAP*[9], *HTTP*[10] e *XML*.

Como um todo, J2EE também descreve vários serviços, incluindo nomeação, transação, ciclo de vida de componente, e persistência, e como estes serviços devem ser providos e acessados. Finalmente, J2EE descreve como diferentes fabricantes precisam prover serviços de infra-estrutura para construtores de aplicações de tal forma que, enquanto a conformidade com o

---

[7] do inglês Application Program Interface, ou API

[8] do inglês eXtensible Markup Language

[9] do inglês Simple Object Access Protocol

[10] do inglês HyperText Transfer Protocol

padrão seja mantida, a aplicação resultante seja portável para qualquer plataforma J2EE.

2.1.2 .NET

Microsoft .NET é um conjunto de especificações e ferramentas para desenvolvimento de software, que consiste de:

- *.NET Framework*, usado para construir e rodar todos os tipos de software, incluindo aplicações baseadas em Internet, aplicações para clientes inteligentes, e *Web Services XML*, componentes que facilitam a integração através do compartilhamento de dados e funcionalidades através de uma rede utilizando protocolos padronizados independentes de plataforma como *XML*, *SOAP*, e *HTTP*.
- Ferramentas de desenvolvimento, como *Microsoft Visual Studio .NET 2003* que provê um ambiente de desenvolvimento integrado.
- Um conjunto de servidores, incluindo *Microsoft Windows Server 2003*, *Microsoft SQL Server*, e *Microsoft BizTalk Server*, que integra, roda, opera, e gerencia *Web Services* e aplicações baseadas em Internet.
- Software de cliente, como *Windows XP*, *Windows CE*, e *Microsoft Office XP*.

No que diz respeito ao desenvolvimento de aplicações o *.NET Framework* é o ponto central. O seu funcionamento é semelhante ao de uma máquina virtual, mas até o presente momento existe apenas para Windows. O *.NET Framework* é composto de uma linguagem comum de tempo de execução[11] e um conjunto de bibliotecas de classes.

A linguagem comum de tempo de execução é responsável por serviços de tempo de execução como integração da linguagem, reforço de segurança e gerenciamento de memória, processos, e tarefas. Adicionalmente, a linguagem comum de tempo de execução tem um papel em tempo de desenvolvimento no que se refere a aspectos como gerenciamento de ciclo de vida, nomeação de tipos fortes, manipulação de exceções, e ligação dinâmica.

As bibliotecas de classes podem ser agrupadas conforme segue:

- Bibliotecas de classes básicas provêem funcionalidades padrão como entrada e saída, manipulação de *strings*, gerenciamento de segurança, comunicação através de redes,

[11] do inglês Common Language Runtime

gerenciamento de *threads*, gerenciamento de textos, e desenho de interface de usuário.

- Bibliotecas de classes *ADO.NET* disponibilizam funcionalidades para interagir com dados acessados na forma de XML através de interfaces *OLEDB*, *ODBC*, *Oracle*, e *SQL Server*.
- Bibliotecas de classes XML habilitam manipulação, procura, e conversão de arquivos XML.
- Bibliotecas de classes *ASP.NET* habilitam o desenvolvimento de aplicações baseadas em Internet e *Web Services*.
- Bibliotecas de classes *Windows Forms* habilitam o desenvolvimento de aplicações clientes baseadas em *Windows*.

2.2 Arquitetura de Software

Arquitetura de software emergiu como uma importante sub-disciplina de engenharia de software, particularmente na área de desenvolvimento de grandes sistemas [BACHMANN+01]. A arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas de um sistema que englobam elementos de software, as propriedades externamente visíveis destes elementos, e as relações entre eles [BASS+03].

Outras definições podem ser encontradas em vários documentos da literatura, como por exemplo Perry [PERRY92], Garlan [GARLAN93], Hayes-Roth [HAYES-ROTH94], Gaycek [GAYCEK95] e Soni [SONI95], mas, na essência de todas as discussões, existe o foco sobre os aspectos estruturais do sistema. Aspectos estruturais incluem organização bruta e estrutura de controle global, protocolos para comunicação, sincronização, e acesso de dados; designação de funcionalidade aos elementos de projeto; distribuição física; composição de elementos de projeto; escalonamento e desempenho; e seleção através de alternativas de projeto [GARLAN93].

Uma determinada arquitetura de software apresenta algumas propriedades, chamadas de propriedades arquiteturais. Uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional [SOMMERVILLE01] e é também chamada de atributo de qualidade [BASS+03]. Alguns exemplos de propriedades arquiteturais são:

- *Disponibilidade*: refere-se a falhas no sistema e conseqüências associadas.
- *Modificabilidade*: diz respeito ao custo de mudança e depende fortemente de como o sistema é modularizado, pois reflete as estratégias de encapsulamento.

- *Desempenho*: diz respeito ao tempo de resposta.
- *Segurança*: diz respeito à habilidade do sistema resistir ao uso não autorizado, enquanto ainda provendo seus serviços aos usuários legítimos.
- *Testabilidade*: diz respeito à facilidade com que o software pode ser testado.
- *Adaptabilidade*: diz respeito à facilidade de modificar ou substituir um componente que participa de uma composição de software.
- *Reutilização*: característica que define quão genérico e independente da aplicação é um componente, para que possa ser reutilizado em diversas aplicações.
- *Usabilidade*: diz respeito à facilidade com que o usuário cumpre uma determinada tarefa e o tipo de ajuda ao usuário que o sistema provê.

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [MONROE+97, SHAW+96]. Um estilo arquitetural é uma descrição de elementos e tipos de relações juntamente com um conjunto de restrições de como eles podem ser utilizados [BASS+03].

2.2.1 Visões Arquiteturais

Uma vez definida a arquitetura de software a ser utilizada um importante passo é documentá-la de tal forma que as pessoas envolvidas no projeto possam entendê-la com clareza. Esta tarefa é particularmente difícil, pois a arquitetura de software é uma entidade complexa que não pode ser descrita de uma maneira unidimensional simples [BACHMANN+01]. Em função disso, para descrever uma arquitetura de software normalmente são utilizadas diversas representações da mesma, estas representações são normalmente chamadas de visões. Uma visão é uma representação de um conjunto coerente de elementos arquiteturais, conforme escritos e lidos pelos responsáveis do sistema. Ela consiste de uma representação de um conjunto de elementos e das relações entre eles. Uma estrutura é um conjunto de elementos em si mesma, na forma como existem em um software ou hardware [BASS+03].

As visões relevantes são aquelas que auxiliam nos propósitos da documentação da arquitetura. Por exemplo, uma visão das camadas será útil para efeito de portabilidade.

Cada visão pode ser definida através de diferentes diagramas. Cada diagrama apresenta

uma representação diversa, no entanto, os elementos descritos a seguir, devem sempre estar presentes, conforme sugerido por Bass, Clements e Kazman [BASS+03]:

- Apresentação primária da visão: normalmente um diagrama, mas também pode ser um texto descrevendo a apresentação primária.
- Catálogo dos elementos: catálogo dos elementos constantes na apresentação primária.
- *Rationale*: critérios utilizados na adoção da arquitetura.
- Análise dos resultados: análise dos resultados esperados através da adoção da arquitetura.
- Suposições: Premissas assumidas para adoção da arquitetura.

Um conjunto de visões bastante conhecido é o proposto por Kruchten [KRUCHTEN95], que é composto por cinco visões, a saber:

1. Visão lógica: descreve o modelo de objetos do projeto, quando é utilizado um método orientado a objetos.
2. Visão de processo: descreve os aspectos de concorrência e sincronização do projeto.
3. Visão física: descreve o mapeamento do software para o hardware e reflete os aspectos de distribuição.
4. Visão de desenvolvimento: descreve a organização estática do software no seu ambiente de desenvolvimento.
5. Visão de cenários: ilustra alguns casos de uso selecionados, organizados a partir das quatro visões anteriores.

2.3 Processo de Desenvolvimento de Software Baseado em Componentes

Todos os projetos de desenvolvimento de software seguem simultaneamente dois processos distintos. O processo gerencial programa as atividades, planeja entregas, aloca recursos, e monitora o andamento. O processo de desenvolvimento cria o software a partir dos requisitos [CHEESMAN+01].

A literatura de processos de software contém exemplos que incluem um dos processos, ou ambos. Por exemplo, *Dynamic Systems Development Method* (DSDM) é um processo de gerenciamento, *Catalysis* é basicamente um processo de desenvolvimento [DSOUZA+99], e o *Rational Unified Process* (RUP) [JACOBSON+99] cobre ambos. Este trabalho aborda apenas o

processo de desenvolvimento, doravante chamado apenas de processo.

Um processo é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). Ele é composto de boas práticas de engenharia de software que conduzem o desenvolvimento, reduzindo os riscos e aumentando a confiabilidade [JACOBSON+99].

O objetivo de um processo é desenvolver um software capaz de atender os requisitos de um determinado sistema. Estes requisitos são normalmente classificados em:

- Requisitos funcionais, que representam os comportamentos que um programa ou sistema deve apresentar diante de certas ações de seus usuários, são normalmente modelados através de casos de uso.
- Requisitos não-funcionais, que especificam características relativas à qualidade dos serviços, como segurança, tempo de resposta e disponibilidade.

Um processo típico [KLEPPE+03] (Figura 1) inclui uma série de fases, descritas a seguir:

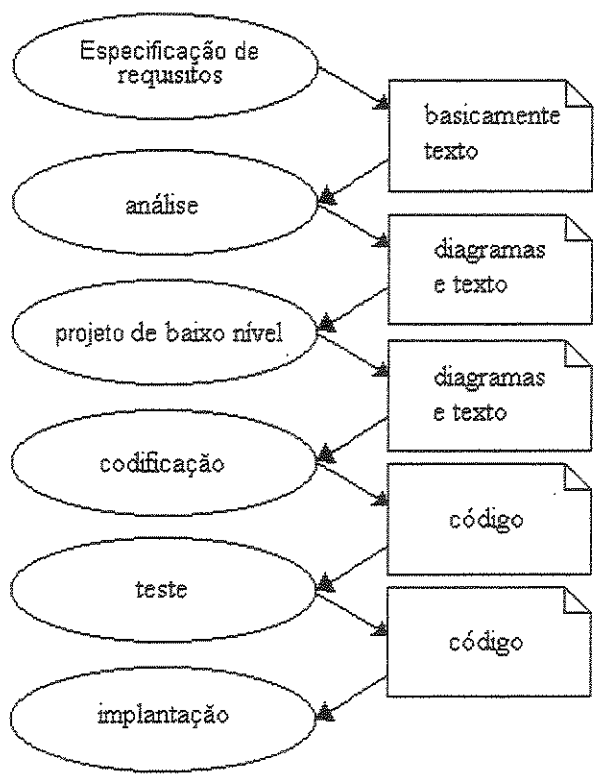

Figura 1 - Processo típico [KLEPPE+03]

- Especificação de requisitos: A fase de conceitualização e identificação dos requisitos tenta identificar os principais conceitos inerentes aos sistemas, bem como, os requisitos que o sistema deverá atender.
- Análise: A fase de análise e descrição funcional produz modelos capazes de traduzir os conceitos e realizar os requisitos funcionais identificados na fase anterior.
- Projeto de baixo nível: O projeto traduz os modelos produzidos na fase imediatamente anterior em modelos para uma determinada arquitetura e resolve questões relativas aos requisitos não-funcionais.
- Codificação: A fase de codificação parte dos projetos funcionais e não-funcionais e implementa em uma determinada linguagem na forma de código fonte.
- Teste: Durante a fase de testes é verificado se o sistema confeccionado atende os requisitos.
- Implantação: Durante a fase de implantação o sistema é instalado e configurado no ambiente no qual irá operar.

Vários modelos de desenvolvimento foram propostos como forma de resolver o problema do desenvolvimento de software como [PRESSMAN01]: (1) modelo seqüencial linear, (2) modelo de prototipagem, (3) modelo de desenvolvimento rápido de aplicações (RAD[12]), (4) modelos evolucionários, (5) modelos de métodos formais, (6) modelos de desenvolvimento baseados em componentes, entre outros.

Durante algum tempo, o modelo de orientação a objetos foi considerado uma forma poderosa de resolver a crise de software através de sua alta reutilização e manutenibilidade [KWON+00]. O modelo de orientação a objetos enfatiza a criação de classes que encapsulam os dados e os algoritmos usados para manipular os dados [PRESSMAN01]. No entanto, o modelo de orientação a objetos apresenta algumas fraquezas no sentido que nem sempre produz software reutilizável, não é adequada para um projeto de grande porte, e não permite o completo encapsulamento de classes em função da herança de subclasses [KWON+00].

Em seu lugar surgiu o modelo de desenvolvimento baseado em componentes, que consiste

[12] do inglês Rapid Application Development

de produção, seleção, avaliação e integração de componentes [KWON+00].

O modelo de desenvolvimento baseado em componentes incorpora muitas das características do modelo espiral. É evolucionário por natureza [NIERSTRASZ+92], demandando uma abordagem interativa para a criação de software. Contudo, o modelo de desenvolvimento baseado em componentes compõe aplicações a partir de componentes de software pré-empacotados [PRESSMAN01].

Diferentemente do modelo de desenvolvimento orientado a objetos cujo foco é a identificação de classes, o paradigma de desenvolvimento baseado em componentes tem o foco na identificação de interfaces e de contratos de uso e de realização, decorrente das características dos componentes, mencionadas na seção 2.1.

Somando-se às características mais comuns dos processos de desenvolvimento de software convencionais, um processo de desenvolvimento de software baseado em componentes apresenta algumas características particulares. Essas particularidades podem ser observadas no acréscimo de estágios técnicos ao processo convencional ou em uma ênfase maior em algumas práticas já realizadas nesses processos. Um processo de desenvolvimento de software baseado em componentes geralmente inclui a definição de estratégias para [SILVA03]:

- Separação de contextos a partir do modelo de domínios;
- Particionamento do sistema em unidades independentes (componentes);
- Identificação do comportamento interno dos componentes;
- Identificação das interfaces dos componentes;
- Definição de um *kit* de arquitetura, que inclua princípios e elementos que facilitem a conexão de componentes;
- Manutenção de um repositório de componentes.

2.4 O Processo UML Components

O processo *UML Components* [CHEESMAN+01], tomado como base para a processo adaptado apresentado no capítulo 3, é um processo de desenvolvimento baseado em componentes (seção 2.3) e é descrito nas seções seguintes.

2.4.1 A Separação em Camadas

O processo *UML Components* propõe que o sistema seja dividido em quatro camadas (Figura 2), descritas a seguir:

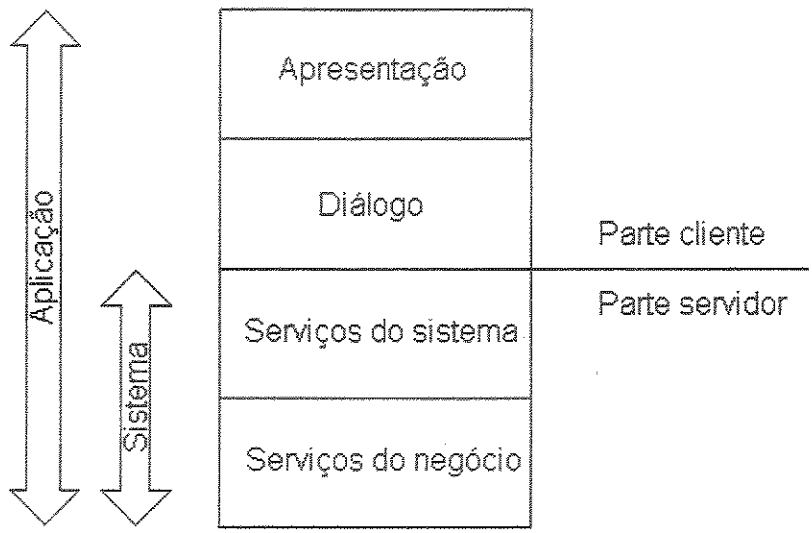

Figura 2 – Arquitetura em camadas usada pelo processo UML Components [CHEESMAN+01]

- *Camada de apresentação* – cria o que o usuário vê, manipula a lógica de interação com o usuário.
- *Camada de diálogo* – faz a manipulação dos diálogos, armazena informações não persistentes de sessão.
- *Camada de serviços do sistema* – executa as operações através de transações e estabelece que os componentes correspondem a sistemas de negócios. Não existem diálogos ou estados relacionados ao cliente.
- *Camada de serviços do negócio* – estabelece que os componentes correspondem a grupos ou tipos estáveis do negócio e, normalmente, tem bancos de dados associados.

As duas últimas camadas compõem o sistema, que juntamente com as outras duas constituem a aplicação.

A nomenclatura utilizada (serviços de negócio, por exemplo) deixa claro que o processo é recomendado para software de negócios, e não para outros tipos como software de tempo-real ou software de sistemas (como compiladores e editores). O processamento de informações de negócios é a maior área de aplicação de software. 'Sistemas' discretos (por exemplo, folha de

pagamento, contas a pagar/receber, controle de estoque) tornaram-se sistemas de informação gerencial que acessam um ou vários grandes bancos de dados contendo informações de negócios [PRESSMAN01].

2.4.2 Os Fluxos de Trabalho

O processo *UML Components* é dividido em vários fluxos de trabalho (Figura 3), descritos a seguir:

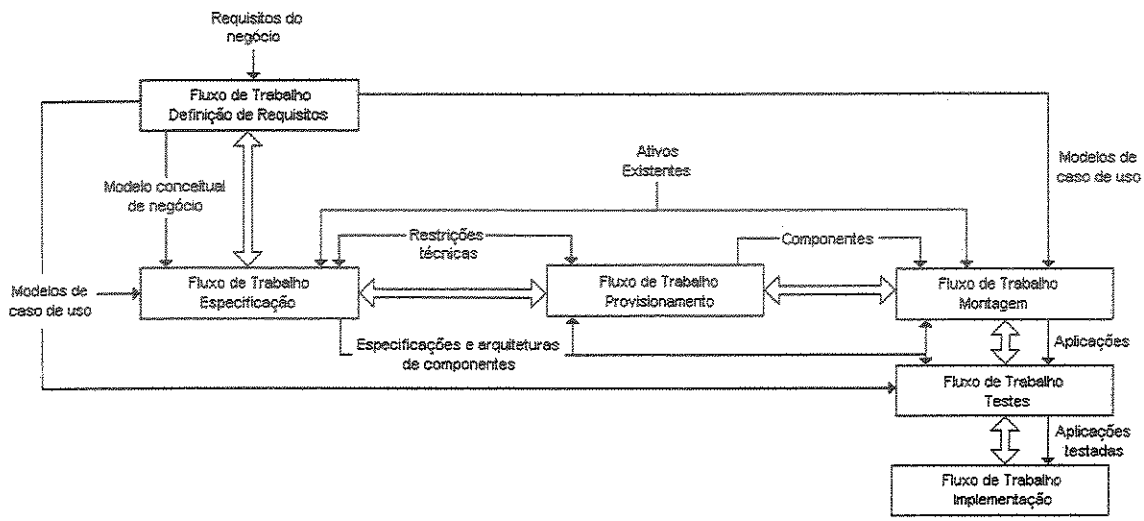

Figura 3 - Os fluxos de trabalho no processo UML Components [CHEESMAN+01]

- Fluxo de trabalho *Definição de Requisitos*: Levantamento e definição de requisitos do sistema, utilizando-se basicamente os Modelos de Caso de Uso e Modelo Conceitual do Negócio.
- Fluxo de trabalho *Especificação*: Subdivide-se em (1) estágio *Identificação de Componentes*, (2) estágio *Interação de Componentes* e (3) estágio *Especificação de Componentes*. Cria as especificações de componentes, de interfaces e de arquitetura de componentes.
- Fluxo de trabalho *Provisionamento*: Implementação das especificações utilizando uma linguagem de programação.
- Fluxo de trabalho *Montagem*: Montagem da aplicação em um ambiente de testes.
- Fluxo de trabalho *Testes*: Testes para verificar se o produto final atende os requisitos

especificados.

- Fluxo de trabalho *Implementação*: Montagem da aplicação no ambiente de produção.

As informações são carregadas entre os fluxos de trabalho através de artefatos. Os artefatos gerados durante o processo são:

- Modelo Conceitual do Negócio: modelo conceitual do domínio do negócio que precisa ser entendido e acordado.
- Modelos de Caso de Uso: modelo que especifica certos aspectos dos requisitos funcionais de um sistema.
- Especificações das Interfaces: um conjunto de especificações individuais de interface, onde cada especificação é um contrato com um cliente de um objeto componente.
- Especificações dos componentes: um conjunto de especificações individuais de componentes, onde cada especificação é definida em termos das especificações das interfaces e restrições.
- Especificação da arquitetura dos componentes: descreve como as especificações dos componentes se encaixam em uma determinada configuração.

Os fluxos de trabalho *Provisionamento*, *Montagem*, *Testes* e *Implementação* (Figura 3) não apresentam subdivisões em estágios. Os fluxos de trabalho *Definição de Requisitos* e *Especificação*, nos quais o processo foca maior atenção, são descritos nas seções 2.4.3 e 2.4.4 respectivamente.

2.4.3 O Fluxo de Trabalho Definição de Requisitos

O fluxo de trabalho *Definição de Requisitos* (Figura 3) é subdividido em estágios (Figura 4), descritos a seguir:

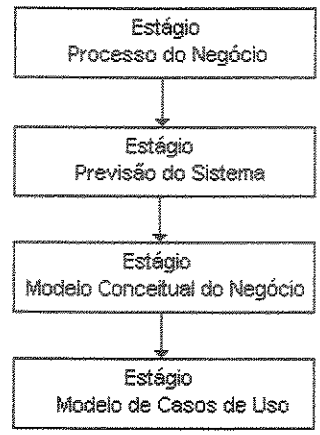

Figura 4 - Os estágios do fluxo de trabalho Definição de Requisitos no processo UML Components

- Estágio *Processo do Negócio*: tem por objetivo produzir um diagrama de atividade UML que, embora não tenha a pretensão de ser uma definição de requisitos, procura ilustrar, sem entrar em detalhes, qual o processo, ou processos, tratado pelo sistema. Na verdade este diagrama não define nem mesmo quais atividades serão ou não automatizadas, mostrando todo o processo no qual o sistema deve ser inserido.
- Estágio *Previsão do Sistema*: busca tornar claras que funções são de responsabilidade do software. Isto é normalmente chamada de definição das fronteiras do software. Para definir esta fronteira é necessário decidir, do ponto de vista dos usuários, como o sistema funciona e o que a criação do software significa. Isto é chamado de previsão do sistema e é uma área pouco explorada. É necessário fazer, pelo menos, uma descrição do funcionamento do sistema, ou até mesmo utilizar outras técnicas como *storyboarding*, para esclarecer aos usuários do futuro sistema exatamente como eles serão afetados pela introdução do sistema.
- Estágio *Modelo Conceitual do Negócio*: objetiva a construção de um mapa mental que relaciona os termos importantes, identificados na descrição do fluxo do processo, para os quais é necessário haver um claro entendimento. Este mapa é chamado Modelo Conceitual do Negócio e pode ser traduzido na forma de um diagrama de classes UML. O Modelo Conceitual do Negócio é um modelo conceitual. Não é um modelo de software, mas um modelo de informações que existem no domínio do problema. O principal propósito do Modelo Conceitual do Negócio é capturar conceitos e identificar

relacionamentos. Modelos Conceituais do Negócio tipicamente capturam classes conceituais e suas associações. Aos papéis das associações é facultativo especificar multiplicidades. O modelo pode conter atributos, caso sejam significantes, mas eles não precisam ser tipados, e operações não serão usadas. Uma vez que a ênfase do modelo é capturar o conhecimento do domínio, e não sintetizá-lo ou normalizá-lo, raramente serão usadas generalizações neste modelo. Da mesma forma, relacionamentos de dependência tipicamente não serão usados.

- Estágio *Modelo de Casos de Uso*: busca descrever as interações usuário-sistema que ocorram na fronteira do sistema. O Modelo de Casos de Uso é uma projeção dos requisitos funcionais do sistema, expressos em termos de interações que devem ocorrer através das fronteiras do sistema. Os participantes em um caso de uso são os atores e o sistema. Um ator é uma entidade que interage com o sistema, tipicamente uma pessoa desempenhando um papel. É possível para um outro sistema ser um ator para o sistema atual, mas neste caso os detalhes deste outro sistema devem ser ignorados. Um ator é sempre identificado como o ator que inicia o caso de uso; os outros atores, se existirem, são usados pelo sistema para atingir o objetivo do iniciador. Embora trate dos requisitos funcionais, pode ser acrescentada uma seção de qualidade de serviço para cada caso de uso declarando as expectativas, especialmente na área de segurança e desempenho.

2.4.4 O Fluxo de Trabalho Especificação

O fluxo de trabalho *Especificação* (Figura 3) é subdividido em estágios (Figura 5), descritos a seguir:

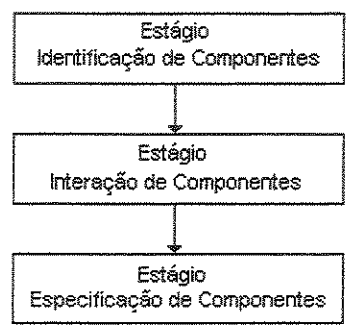

Figura 5 - Os estágios do fluxo de trabalho Especificação no processo UML Components

- Estágio *Identificação de Componentes*: parte do fluxo de trabalho *Definição de Requisitos*, tomando o Modelo Conceitual do Negócio e o Modelo de Casos de Uso como entradas. O objetivo do estágio *Identificação de Componentes* é criar um conjunto inicial de interfaces e especificações de componentes, que servirá como ponto de partida para os estágios seguintes, pelos quais ela será refinada e preenchida. Este estágio também produz um importante artefato interno de especificação, o Modelo de Tipos do Negócio, que é usado posteriormente para desenvolver modelos de informação de interface. O modelo de Tipos do Negócio é um artefato interno deste fluxo de trabalho, cujo propósito é formalizar o Modelo Conceitual do Negócio para definir o conhecimento do sistema para o mundo externo, e é representado por um diagrama de classes UML. A ênfase neste estágio é na descoberta - que informações precisam ser gerenciadas, que interfaces precisam ser gerenciadas, que componentes são necessários para prover determinada funcionalidade, e como eles se encaixaram num todo. Esta abordagem decorre do fato de que a aplicação em discussão está dividida em camadas (seção 2.4.1). Esta separação permeia o fluxo de trabalho *Especificação*. Neste estágio o objetivo é identificar as interfaces do sistema e os componentes do sistema na camada de serviços de sistema, e interfaces de negócio e componentes de negócio na camada de serviços de negócio. Ao final deste estágio são geradas as especificações iniciais das interfaces e da arquitetura de componentes.
- Estágio *Interação de Componentes*: parte do Modelo de Tipos do Negócio e nas especificações iniciais das interfaces, gerados no estágio imediatamente anterior, para definir como os componentes das diferentes camadas trabalham em conjunto para prover uma determinada funcionalidade. A Interação de Componentes é modelada, basicamente, através de diagramas de colaboração.
- Estágio *Especificação de Componentes*: concentra-se na construção dos contratos (seção 2.1). Neste estágio são definidas: (1) as pré- e pós-condições para as operações, (2) as invariantes para os componentes, (3) restrições nas interações dos componentes, e (4) restrições nas interações das interfaces.

2.4.5 Os Estereótipos do Processo UML Components

O processo *UML Components* define estereótipos adicionais para atribuir uma nova semântica a alguns elementos particulares da linguagem UML [UML03], são eles:

Estereótipo	Construção UML	Conceito associado
<<comp spec>>	*class*	Estereótipo indicativo que a classe representa um componente.
<<interface type>>	*type* (*class type*)	Estereótipo indicativo que a interface é uma interface de componente.
<<offers>>	*dependency*	Estereótipo indicativo que uma classe *<<comp spec>>* oferece uma interface *<<interface type>>*. Somente pode ser aplicada entre estes dois elementos.
<<concept>> (opcional)	*class*	Estereótipo indicativo que a classe representa uma classe conceitual.
<<type>>	*type* (*class type*)	Estereótipo indicativo que a classe representa um tipo do negócio.
<<core>>	*type* (*class type*)	Estereótipo indicativo que a classe representa um tipo central do negócio. Este o estereótipo *<<type>>*.
<<DataType>>	*type* (*class type*)	Estereótipo indicativo que a classe representa um tipo estruturado de dados.
<<info type>>	*type* (*class type*)	Estereótipo indicativo que a classe representa um tipo de informação.
<<att>>	*operation*	Estereótipo indicativo que a operação é um atributo

		parametrizado.
<<transaction>>	*operation*	Estereótipo indicativo que a operação requer uma nova transação.

Tabela 1 – Estereótipos definidos pelo processo UML Components

2.4.6 A Estrutura de Pastas para Armazenamento dos Modelos

Adicionalmente o processo *UML Components* define uma estrutura de pacotes ou pastas (Figura 6) onde são armazenados os artefatos gerados durante o processo. Cada artefato gerado, por exemplo, o Diagrama de Modelo Conceitual do Negócio que é representado através de um diagrama de classe, deve ser criado dentro do pacote ou pasta Modelo Conceitual do Negócio, e assim por diante. A implementação desta estrutura pode ser feita através de uma ferramenta UML (que permita a criação de pacotes) ou do sistema de arquivos (através da utilização de pastas).

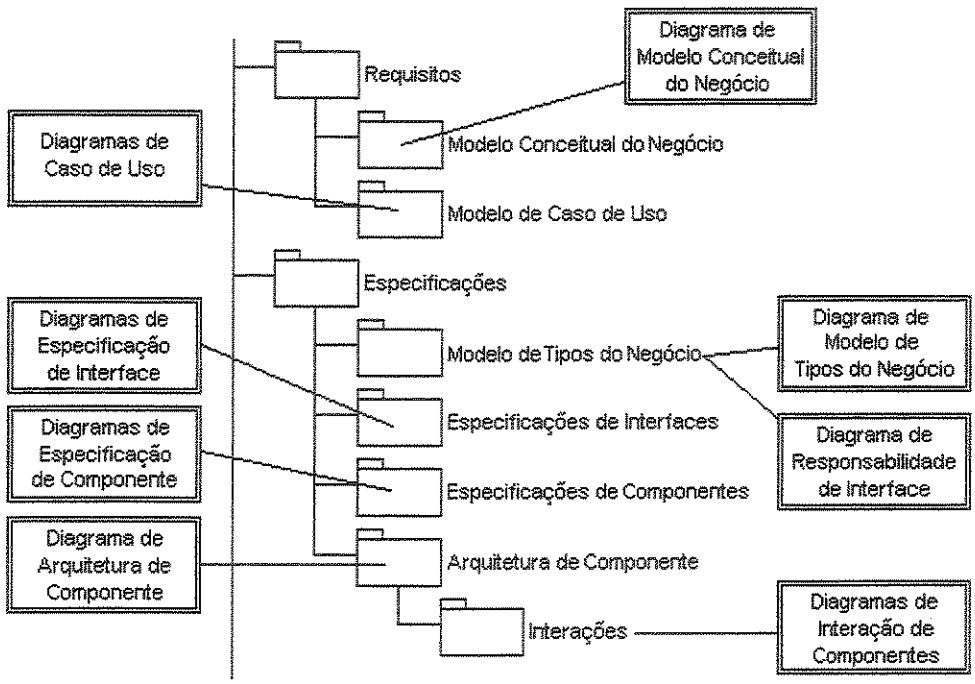

Figura 6 – Diagramas de modelagem de componentes no processo UML Components [CHEESMAN+01]

2.5 A Abordagem Model Driven Architecture (MDA)

O *Model Driven Architecture* (MDA) é uma abordagem proposta do consórcio *Object Management Group* (OMG), responsável por especificações consagradas como, por exemplo, UML e CORBA. O MDA define uma abordagem para especificação de sistemas de informação que separa a especificação da funcionalidade do sistema da especificação da implementação desta funcionalidade em uma plataforma tecnológica específica. Com este objetivo, a abordagem MDA define um conjunto de linhas mestre para estruturar especificações expressadas como modelos [MDA01].

A abordagem MDA e as especificações de apoio permitem que um mesmo modelo, especificando funcionalidades do sistema, seja implementado em múltiplas plataformas através de mapeamentos auxiliares padronizados, ou através de mapeamentos pontuais para plataformas específicas, e permite que diferentes aplicações sejam integradas através do relacionamento explícito dos seus modelos, permitindo integração e interoperabilidade e auxiliando a evolução dos sistemas à medida que as plataformas tecnológicas mudem.

A partir destas colocações é possível destacar alguns aspectos fundamentais da abordagem MDA, a saber:

- Separação da especificação da funcionalidade básica do sistema, e da especificação desta funcionalidade em uma determinada plataforma.
- Possibilidade de implementar um mesmo modelo em múltiplas plataformas através de mapeamentos.
- Auxiliar a evolução dos sistemas à medida que as plataformas tecnológicas mudam: Este aspecto decorre dos dois primeiros e é a motivação fundamental da abordagem MDA, oferecer suporte às mudanças tecnológicas.

Nas seções seguintes são definidos alguns conceitos relativos à abordagem MDA que serão importantes para o processo adaptado apresentado no capítulo 3, são eles: (1) modelos (seção 2.5.1), (2) especificações de suporte(seção 2.5.2), (3) extensões(seção 2.5.3), (4) mapeamentos (seção 2.5.4), e (5) independência de plataforma (seção 2.5.5).

2.5.1 Os Modelos na Abordagem MDA

A abordagem MDA separa certos modelos fundamentais para um sistema, e traz uma estrutura consistente para estes modelos. Os modelos de diferentes sistemas são estruturados explicitamente em Modelos Independentes de Plataforma (PIM[13]), e Modelos Dependentes de Plataforma (PSM[14]). A forma como uma determinada funcionalidade em um PIM é realizada é especificada de uma maneira específica para a plataforma no PSM, que é derivado do PIM através de algumas transformações.

No restante deste documento os modelos independentes de plataforma serão chamados simplesmente de PIM, e os dependentes de plataforma de PSM.

O PIM provê especificação formal da estrutura e função do sistema que abstrai detalhes técnicos. Uma Visão dos Componentes Independente da Plataforma descreve componentes computacionais e suas interações em uma maneira independente da plataforma. Estes componentes e interfaces, por sua vez, são uma maneira de realizar algum sistema de informação, ou aplicação, mais abstrato, que por si mesmos ajudam a realizar um modelo de negócio independente de detalhes computacionais. Os padrões da abordagem MDA são especificados em termos de um PIM e, normalmente, um ou mais PSM, todos utilizando a linguagem UML [UML03].

Adicionalmente, a abordagem MDA define relacionamentos consistentes entre estes modelos. Existem correspondências de refinamento entre o modelo de negócio, os componentes independentes de plataforma, e os componentes dependentes de plataforma. De forma similar, as interações entre dois sistemas distintos a serem integrados podem ser especificadas nos modelos da plataforma específica, nos modelos independentes da plataforma, ou até mesmo nos modelos de negócio.

A abstração da estrutura e comportamento de um sistema em um PIM ao invés de implementações específicas definidas nos PSM, apresenta três importantes benefícios:

- É mais fácil validar a correção do modelo sem levar em conta semânticas específicas da plataforma. Por exemplo, o PSM tem que usar os conceitos da plataforma de mecanismos

---

[13] do inglês Platform Independent Model

[14] do inglês Platform Specific Model

de exceção, tipos de parâmetro (incluindo regras específicas da plataforma sobre referências de objetos, tipos de valor, semânticas de chamadas por valor, etc) e construções de modelos de componentes; o PIM não precisa fazer estas distinções e pode ao invés disso usar um modelo simples e mais uniforme.

- É mais fácil produzir implementações em diferentes plataformas enquanto se está em conformidade com a mesma essencial e precisa estrutura e comportamento de um sistema.
- Integração e interoperabilidade entre sistemas podem ser definidas mais claramente em termos independentes de plataforma, e então mapeados abaixo em mecanismos específicos da plataforma.

2.5.2 Especificações de Suporte para a Abordagem MDA

Na abordagem MDA a construção de modelos e feita utilizando a linguagem UML [UML03] e tendo como suporte as especificações (1) MOF[15] [MOF02], (2) CWM[16] [CWM03], e (3) XMI[17] [XMI02].

A linguagem UML é utilizada para especificar, visualizar, construir, e documentar os artefatos de sistemas computacionais, bem como modelagem de negócios e outros sistemas não computacionais [UML03]. A linguagem UML é uma especificação do consórcio OMG bastante difundida no mercado e, normalmente, utilizada para modelagem orientada a objetos. Existe um grande número de ferramentas de diferentes fabricantes que implementam, nem sempre na sua totalidade, a linguagem UML.

Embora alguns autores identifiquem a especificação MOF como um subconjunto da linguagem UML [IYENGAR02], esta é uma simplificação excessiva, senão errônea. A especificação MOF é o principal fundamento da abordagem MDA porque até mesmo a linguagem UML é definida através da especificação MOF. A especificação MOF define uma forma universal de descrever diferentes tipos de construções usadas para modelagem. Com a especificação MOF é possível descrever uma base de dados através dos seus elementos (tabela, colunas, etc), a especificação CORBA através dos seus elementos (interfaces, valuetypes, etc) e

---

[15] do inglês *Meta-Object Facility*

[16] do inglês *Common Warehouse Metamodel*

[17] do inglês e*Xtensible markup language for Metadata Interchange*

até mesmo a linguagem UML através de seus elementos (classes, operações, atributos, etc). Os modelos criados que descrevem uma determinada especificação são chamados de metamodelos. Da mesma forma as demais especificações de referência, CWM e XMI, são descritas através da especificação MOF.

Outra especificação de referência da abordagem MDA, o CWM é uma especificação do consórcio OMG que provê um arcabouço para representação de metadados sobre fontes de dados, alvos de dados, transformações, e análise, e o processo e operações que criam e gerenciam armazéns de dados e provê informação sobre sua origem [CWM03].

Finalmente o XMI, é uma especificação do consórcio OMG baseada na especificação XML do consórcio W3C (*World Wide Web Consortium*). A especificação XMI permite o intercâmbio de qualquer tipo de metadado que pode ser expresso usando uma especificação MOF, incluindo tanto o modelo como o metamodelo [XMI02]. Este intercâmbio é feito através do formato XML, que é bastante difundido atualmente, em função da sua larga utilização na Internet como forma estruturada de transmissão de informação.

2.5.3 Extensões

Uma outra importante característica da linguagem UML, que é amplamente utilizada pela abordagem MDA, é a capacidade de extensão, que permite com que sejam atendidas necessidades específicas. As formas como a extensão pode ser implementada são (1) os perfis UML, e (2) a extensão da especificação MOF, descritas a seguir.

A linguagem UML prevê dois mecanismos de extensão próprios da linguagem, a saber: (1) estereótipos e (2) valores etiquetados[18]. Um estereótipo é um elemento de modelagem que define valores adicionais, restrições adicionais, e opcionalmente uma nova representação gráfica [UML03]. Definições de etiquetas[19] especificam novos tipos de propriedades que podem ser vinculadas aos elementos de modelagem. As reais propriedades de um determinado elemento de modelagem são especificadas através de valores etiquetados [UML03]. O conjunto de extensões, elaborado dessa forma, constitui um dialeto particular, e é chamado de perfil. Formalmente um perfil é um pacote estereotipado que contém elementos de modelagem adaptados para um

---

[18] do inglês *Tagged Values*

[19] do inglês *Tag Definition*

domínio ou propósito específico através da extensão do metamodelo através de estereótipos, valores etiquetados, e restrições [UML03].

A Figura 7 apresenta um exemplo de modelo formal de classes de um perfil para estender o estereótipo *Type* do metamodelo UML, através dos estereótipos *EJBRemoteInterface* e *EJBHomeInterface*, bem como o valor sinalizado *EJBTransAttribute* que estende a semântica de elemento *Operation* do metamodelo UML.

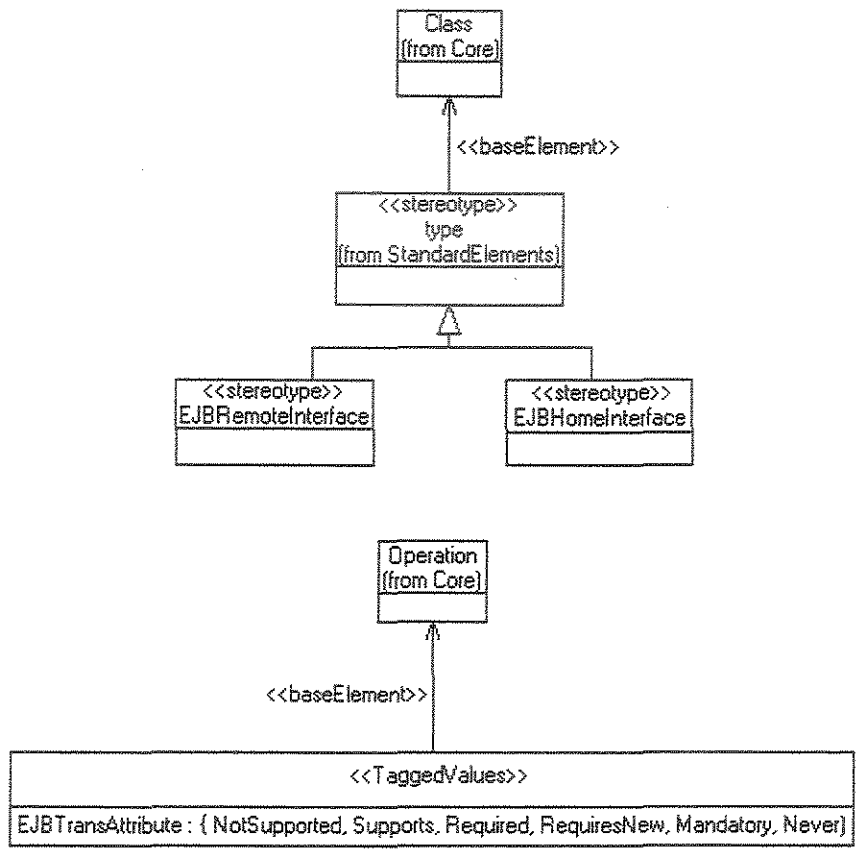

Figura 7 - Fragmento dos estereótipos e valores etiquetados definidos no modelo EJB [UML4EJB]

Além dos estereótipos e valores etiquetados, o perfil pode conter restrições que podem ser vinculadas aos elementos de construção para refinar a sua semântica.

Uma outra forma de extensão prevista na linguagem UML é através da extensão da especificação MOF (seção 2.5.2), conforme descrito a seguir (Figura 8).

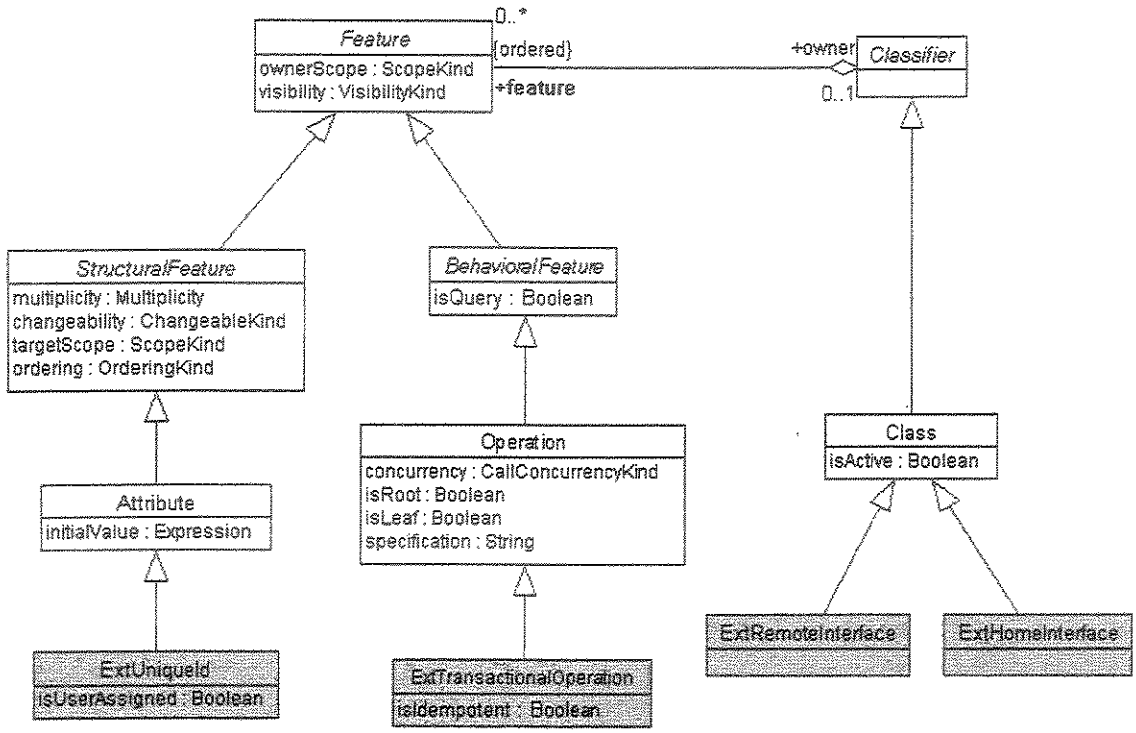

Figura 8 – Exemplo de extensão de um fragmento do metamodelo UML

O elemento *Class* da linguagem UML é especializado em *ExtRemoteInterface* e *ExtHomeInterface*, o elemento *Operation* é especializado em *ExtTransactionalOperation*, e o elemento *Attribute* é especializado em *ExtUniqueId*.

A primeira forma de extensão, através de perfil, é mais facilmente implementável, pois para tanto é possível utilizar qualquer ferramenta genérica com suporte para a linguagem UML, que são mais facilmente encontradas. Esta vantagem também pode ser vista sob o aspecto da representação gráfica. Os elementos criados através de extensão necessitam de uma forma distinta de visualização para poderem figurar em algum diagrama, caso contrário não poderiam ser visualmente distinguidos dos elementos dos quais foram estendidos. Por outro lado, a extensão através de estereótipos e valores etiquetados limita o uso do poder semântico de modelagem de classes orientadas a objetos que a especificação MOF oferece.

Existe, porém, uma ressalva no que diz respeito à escolha entre definir determinado detalhe como valor sinalizado ou estereótipo. Em princípio um estereótipo pode ser transformado

UNICAMP
BIBLIOTECA CENTRAL
DESENVOLVIMENTO DE COLEÇÕE

em um valor sinalizado e vice-versa. Na Figura 7 os estereótipos *EJBRemoteInterface* e *EJBHomeInterface* poderiam ter sido criados como definição de valor *location* e valores etiquetados *location:Remote* e *location:Home*. O efeito teria sido o mesmo. Poderiam também ter sido criados estereótipos *TransactionNotSupported*, *TransactionSupported*, etc, correspondentes aos valores etiquetados *EJBTransAttribute:NotSupported*, *EJBTransAttribute:Supports*, etc. No entanto, existem duas peculiaridades que restringem a utilização de uma e outra forma. A primeira é que o estereótipo somente pode ser utilizado uma vez em um determinado elemento. A segunda é que um valor sinalizado não pode conter elementos. Limitado a estas duas restrições a escolha é livre.

Existe, ainda, a possibilidade de combinar livremente as duas formas, utilizando, potencialmente, a total capacidade de cada uma.

2.5.4 Mapeamentos

Um dos aspectos fundamentais em toda a abordagem MDA é a noção de mapeamento. Um mapeamento é um conjunto de técnicas usadas para modificar um modelo no sentido de obter-se outro modelo [MDA01]. Através dos mapeamentos são estabelecidas as correspondências entre uma construção em um modelo e a construção correspondente em um outro modelo. Este é um importante aspecto no que diz respeito à rastreabilidade entre os modelos.

Para implementar um mapeamento, é necessário conhecer os metamodelos dos modelos de entrada e saída e suas regras de mapeamento.

Existem quatro maneiras de se fazer um mapeamento:

1. Estudar o modelo destino e fazer um refinamento um a um de cada elemento do modelo origem para o modelo destino.
2. Estudar o modelo destino e utilizar refinamentos padrões existentes para reduzir a carga de um refinamento um a um.
3. Aplicar um algoritmo no modelo origem e criar um esqueleto de um modelo destino a ser preenchido manualmente, talvez utilizando os refinamentos descritos em 2.
4. Aplicar um algoritmo para criar um modelo destino completo a partir de um modelo origem.

2.5.5 Independência de Plataforma

O termo plataforma é utilizado para referir-se aos detalhes tecnológicos e de engenharia que são irrelevantes para a funcionalidade fundamental de um componente de software [MDA01]. Embora não declarado explicitamente existe um outro conceito intermediário, incorporado no conceito *independente de plataforma* e que não está relacionado à funcionalidade básica do sistema, chamaremos isto de padrão arquitetural. Um padrão arquitetural é uma descrição de elementos e tipos de relações juntamente com um conjunto de restrições de como eles podem ser usados [BASS+03]. No entanto, o padrão arquitetural também não define uma tecnologia em particular. Pode-se, por exemplo, definir um padrão arquitetural para sistemas distribuídos, sem especificar que tais sistemas serão desenvolvidos utilizando determinada tecnologia. O conceito *independente de plataforma* da abordagem MDA representa, na verdade, independência dos seguintes fatores [FRANKEL03]:

- Tecnologia de formatação de informação, como *XML DTD* (*Document Type Definition*) ou esquema XML.
- Linguagens de terceira e quarta geração como Java, C# e Visual Basic.
- Componentes de *middleware* distribuído como J2EE, CORBA e .NET.
- *Middleware* de envio de mensagens, como WebSphere MQ Integrator (MQSeries) e MSMQ.

2.6 O Modelo COSMOS

O Modelo COSMOS (*Component Structuring Model for Object-oriented Systems*) [SILVA03] é um modelo de estruturação de componentes para sistemas orientados a objetos, que age principalmente na organização do sistema em termos de seus componentes, conectores e interações entre eles (Figura 9).

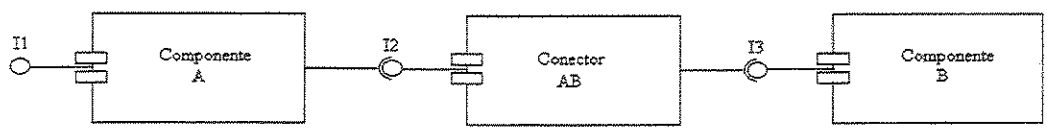

Figura 9 – Modelo COSMOS - Solução para estruturação de componentes [SILVA03]

O modelo COSMOS define elementos para a implementação de componentes e conectores, participantes da composição de software, que podem ser implementados em linguagens de programação.

O modelo preocupa-se principalmente em garantir as seguintes propriedades arquiteturais (seção 2.2): (1) reutilização, (2) adaptabilidade e (3) modificabilidade.

A reutilização dos componentes é promovida pelo baixo nível de acoplamento do sistema, uma vez que os componentes interagem somente através de conectores, e um componente não tem conhecimento dos demais componentes que participam da composição de software. A adaptabilidade, diz respeito à possibilidade de modificar ou substituir um componente que participa de uma composição de software, apenas modificando a implementação dos conectores, que podem, por exemplo, passar a referenciar um novo componente. O baixo acoplamento entre os componentes do sistema é obtido graças à declaração das dependências dos componentes em interfaces, chamadas de interfaces requeridas. Já a modificabilidade, está relacionada com o fato de que apenas a interface do componente é conhecida do usuário, possibilitando assim a modificação da implementação do componente sem afetar a sua utilização.

O modelo COSMOS também incorpora um conjunto de diretrizes de projeto que visam habilitar a implementação de sistemas baseados em componentes, mantendo a conformidade com a descrição da arquitetura do software, e construir componentes de software mais suscetíveis a mudanças, reutilizáveis e adaptáveis. As diretrizes de projeto são: (i) a materialização de elementos arquiteturais, que se preocupa com a materialização, em elementos do modelo de objetos, dos elementos arquiteturais, por exemplo, componentes e conectores; (ii) a inserção de requisitos não-funcionais do sistema nos conectores, para tornar mais simples o projeto dos componentes e promover a reutilização dos mesmos; (iii) a clara separação entre a especificação e a implementação do componente, de modo a garantir que apenas a especificação do componente seja pública; (iv) a declaração explícita dos serviços que o componente precisa para prover os seus próprios serviços; (v) as restrições quanto ao uso da herança na implementação do componente, e/ou entre classes de diferentes componentes, para facilitar a reutilização de código; (vi) e o baixo acoplamento entre as classes da implementação do componente para facilitar a evolução de sua implementação.

O modelo COSMOS baseia-se nestas diretrizes de projeto para compor as suas regras e definir

os seus elementos. Além da estruturação do sistema em termos de seus componentes, conectores e interfaces, o modelo COSMOS também se preocupa com o projeto dos componentes e conectores e com as interações entre eles. Assim, o modelo COSMOS define três modelos inter-relacionados (Figura 10), descritos a seguir:

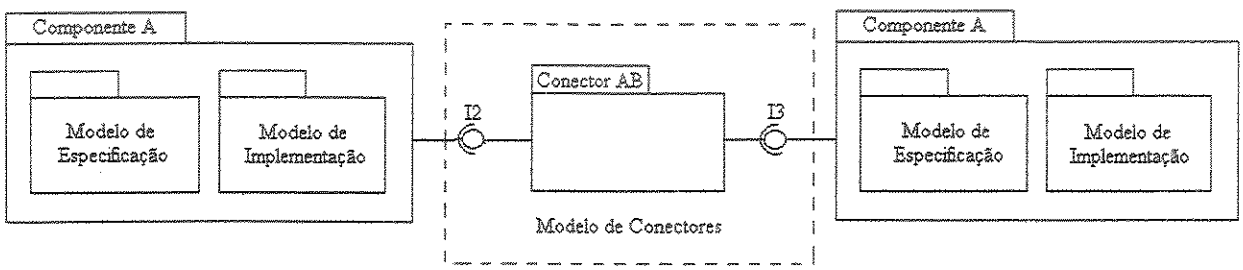

Figura 10 – Modelo COSMOS - Solução para o projeto dos componentes [SILVA03]

- Modelo de especificação: define a visão externa de um componente, ou seja, suas interfaces requeridas e providas, através das quais um usuário do componente pode resolver as suas dependências e acessar os seus serviços.
- Modelo de implementação: define como um componente deve ser implementado internamente.
- Modelo de conectores: define como se dão as interações entre os componentes, através de conectores.

No modelo COSMOS, um componente arquitetural é mapeado em um pacote contendo dois subpacotes (Figura 11), conforme descrito a seguir:

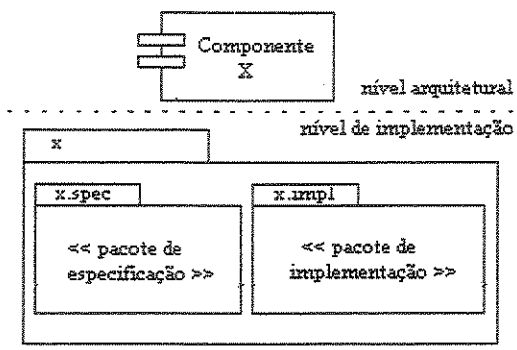

Figura 11 – Modelo COSMOS - Representação de componentes [SILVA03]

- Pacote de especificação: contém um conjunto de interfaces públicas que podem ser providas ou requeridas pelo componente. As interfaces são organizadas em dois subpacotes distintos, um para as interfaces providas (*spec.prov*) e outro para as interfaces requeridas (*spec.req*), e compreendem toda a informação necessária para integrar um componente numa configuração de software e acessar os seus serviços.
- Pacote de implementação: contém classes que implementam os serviços providos pelo componente e, opcionalmente, também pode conter interfaces auxiliares, internas ao pacote de implementação (Figura 12). Como regra geral, as classes de implementação e interfaces auxiliares têm visibilidade restrita ao pacote de implementação (*friendly classes* e *friendly interfaces*) e permanecem totalmente escondidas dos clientes do componente. A única exceção a essa regra é uma classe pública que implementa uma operação para instanciação do componente e deve ser usada pelos clientes do componente.

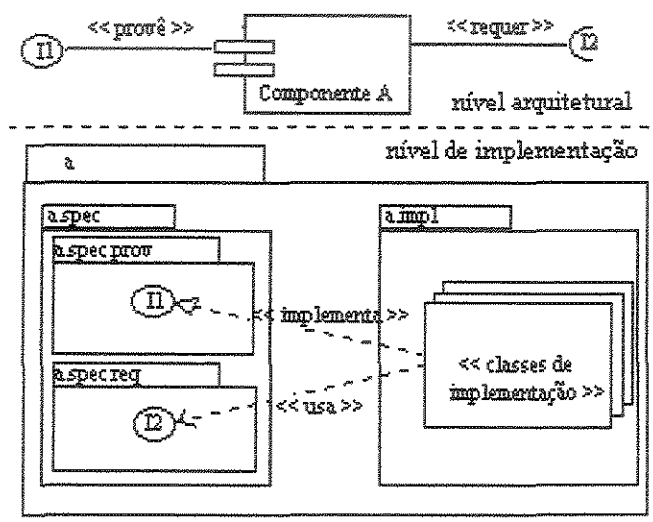

Figura 12 – Modelo COSMOS – Estrutura interna do componente [SILVA03]

Um conector arquitetural é mapeado em um pacote que contém classes que materializam as conexões entre os componentes, também chamadas de conexões de interface (Figura 13). Uma conexão de interface representa uma ligação entre uma interface requerida por um componente e uma ou mais interfaces providas por outros componentes [LUCKHAM+00]. A princípio, o mapeamento de um conector arquitetural, diferentemente do mapeamento feito para um componente arquitetural, não prevê

a existência de um pacote de especificação, ou seja, um conector não define novas interfaces públicas. Dessa forma, um conector apenas usa interfaces definidas pelos componentes com os quais interage, tendo sua estrutura interna similar ao pacote de implementação de um componente, com seus elementos inacessíveis externamente. No modelo proposto, um conector é também considerado como sendo um componente, porém, um componente mais simplificado, cuja existência se dá pela necessidade de conectar outros componentes.

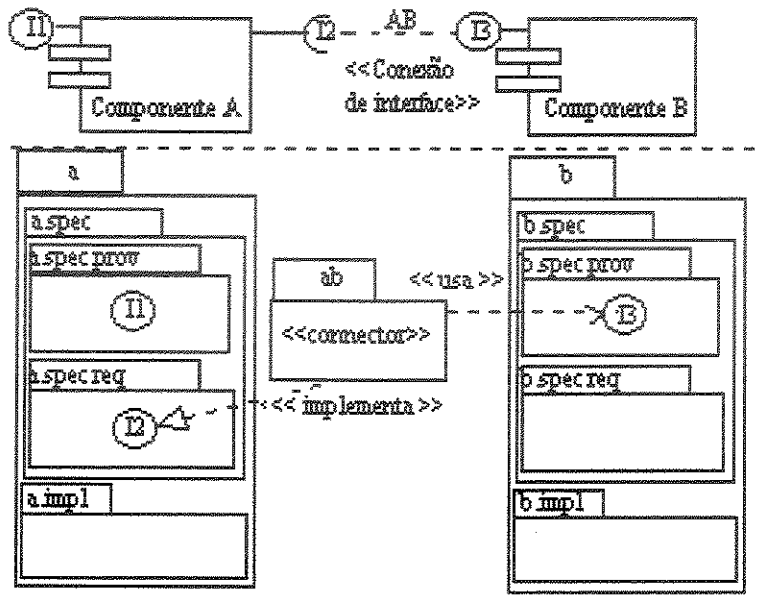

Figura 13 – Modelo COSMOS – Componentes e conectores [SILVA03]

Os pacotes de especificação e implementação que descrevem um componente arquitetural, e os conectores e conexões de interface responsáveis pelas inter-conexões entre componentes, compõem o modelo COSMOS.

Capítulo 3

Um Processo Adaptado ao Model Driven Architecture

Este trabalho propõe um processo de desenvolvimento de software baseado em componentes, adaptado a partir do *UML Components* (seção 2.4), para incorporar as diretrizes da abordagem MDA (seção 2.5) proposta pelo consórcio OMG.

Essa adaptação inclui o tratamento explícito dos requisitos não-funcionais através do refinamento da arquitetura de software, e do uso de um modelo de estruturação de componentes independente de plataforma, mais especificamente o COSMOS (seção 2.6). O uso desse modelo independente de plataforma permite um mapeamento entre as abstrações de uma descrição arquitetural (baseada em componentes e conectores) para construções disponíveis nas plataformas específicas de componentes.

A incorporação das diretrizes da abordagem MDA, particularmente o mapeamento (seção 2.5.4) entre os modelos independentes de plataforma e os modelos dependentes de plataforma (seção 2.5.1), são atingidos em grande parte através da utilização de um modelo de estruturação de componentes independente de plataforma. O tratamento dos requisitos não-funcionais é feito através da inclusão do estágio *Definição de Requisitos não-Funcionais* e do refinamento da arquitetura em camadas prevista no processo *UML Components* (Figura 2).

A seção 3.1 apresenta as adaptações efetuadas no processo *UML Components*, e as seções 3.2 e 3.3 apresentam os modelos criados a partir do modelo COSMOS, respectivamente para PSM J2EE e PSM .NET, bem como os mapeamentos definidos para gerar estes modelos a partir do PIM. A seção 4 apresenta as regras de mapeamento dos PSM J2EE e PSM .NET para implementação.

3.1 Adaptações Efetuadas no Processo UML Components

As adaptações efetuadas no processo *UML Components* estão ilustradas na Figura 14.

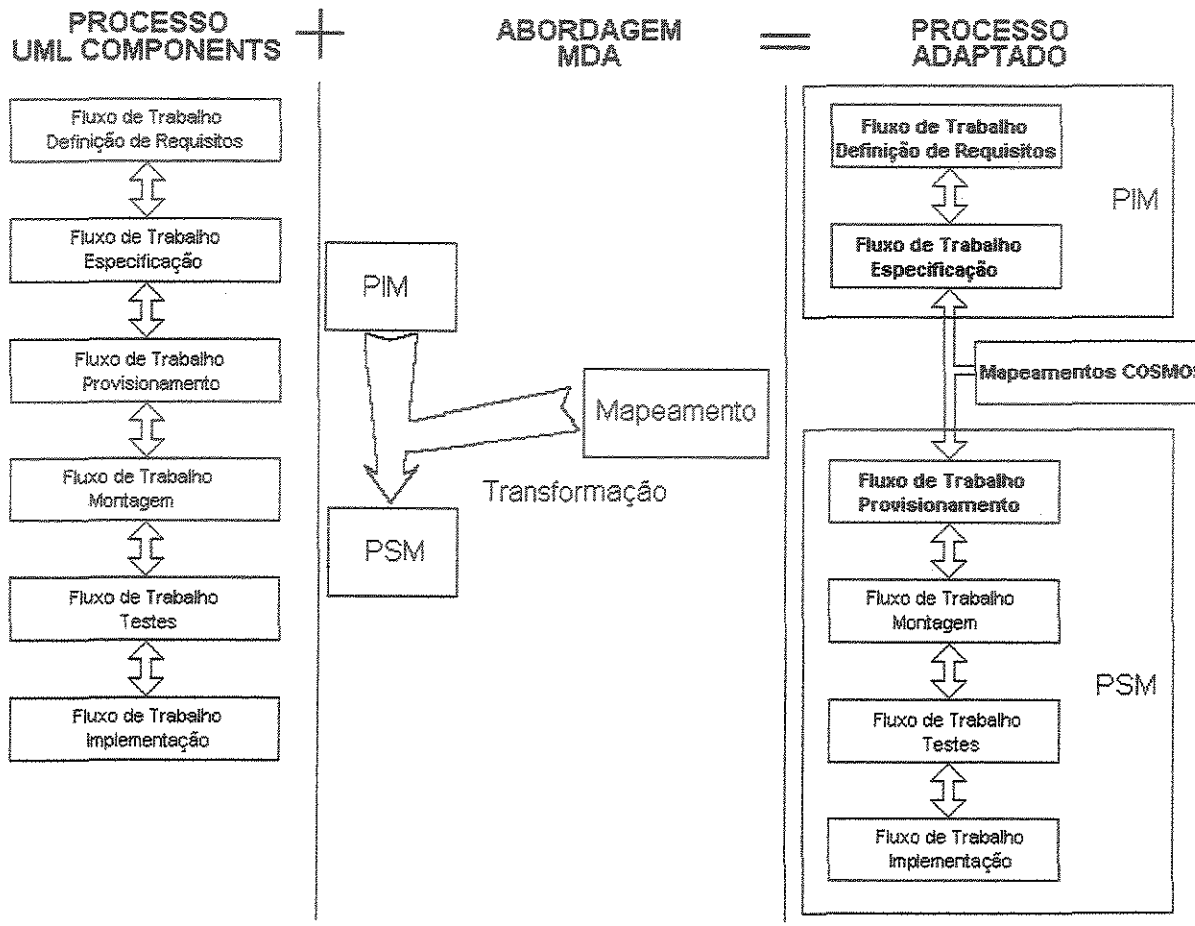

Figura 14 – Adaptações efetuados no processo UML Components

Os modelos criados no processo *UML Components* até o fluxo de trabalho *Especificação* não incorporam detalhes tecnológicos das plataformas alvo, portanto, seguindo a nomenclatura da abordagem MDA foram considerados modelos PIM. Da mesma forma, os modelos gerados a partir do fluxo de trabalho *Provisionamento* incorporam detalhes tecnológicos das plataformas alvo e, portanto, foram nomeados como modelos PSM.

A transição entre o fluxo de trabalho *Especificação* e o fluxo de trabalho *Provisionamento* passou a ser feita de forma sistemática através da utilização dos mapeamentos baseados no modelo de estruturação de componentes COSMOS, criados neste trabalho, conforme prescrito

pela abordagem MDA, o que ajudou a reduzir a distância semântica entre as descrições arquiteturais e as construções disponíveis nas plataformas tecnológicas. No processo *UML Components* esta transição não era feita de forma sistemática.

Não houve inclusão de novos fluxos de trabalho, no entanto foram criados novos estágios nos fluxos de trabalho que aparecem em negrito na Figura 14. As adaptações efetuadas dentro destes fluxos de trabalho são descritas a seguir:

- Fluxo de trabalho *Definição de Requisitos*: Assim como no processo *UML Components* (seção 2.4.3) este fluxo de trabalho, embora de extrema importância, não foi o foco de atenção. O processo limitou-se a criar os artefatos indispensáveis para o fluxo de trabalho *Especificação*. Foi criado, contudo, o estágio *Definição de Requisitos não-Funcionais* (Figura 15 em comparação com Figura 4, o estágio incluído aparece em negrito), que serve de insumo para o estágio *Refinamento da Arquitetura de Software* apresentado adiante.

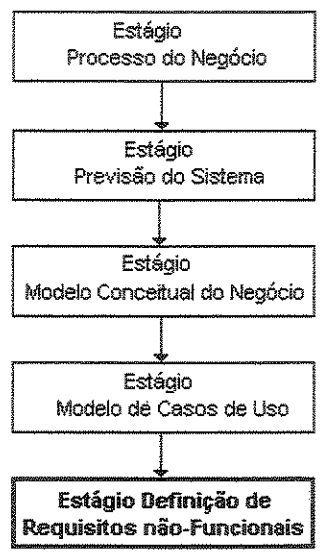

Figura 15 - Os estágios do fluxo de trabalho Definição de Requisitos no processo adaptado

- Fluxo de trabalho *Especificação*: Foram criados: (1) o estágio *Refinamento da Arquitetura de Software*, (2) o estágio *Mapeamento dos Componentes na Arquitetura de Software*, e (3) o estágio *Refinamento do PIM*, (Figura 16 em comparação com a Figura 5, os estágios incluídos aparecem em negrito). O estágio *Refinamento da Arquitetura de Software* parte da *Definição de Requisitos não-Funcionais*, gerada no fluxo de trabalho

anterior, o estágio *Mapeamento dos Componentes na Arquitetura de Software* parte da arquitetura de software e das especificações definidas, e o estágio *Refinamento do PIM* parte das especificações geradas até o estágio imediatamente anterior e do perfil para refinamento do PIM.

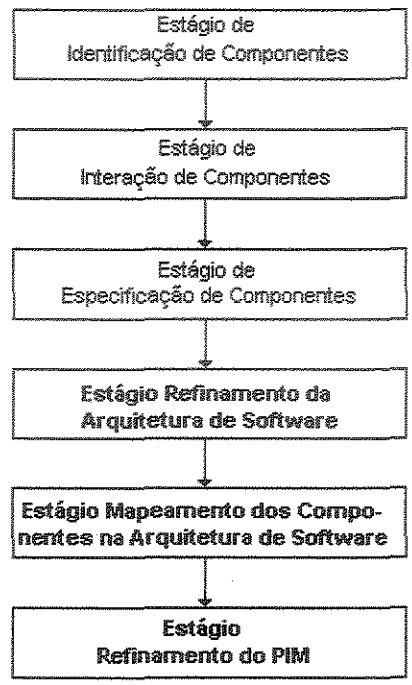

Figura 16 - Os estágios do fluxo de trabalho Especificação no processo adaptado

- Fluxo de trabalho *Provisionamento*: Foram criados: (1) o estágio *Mapeamento do PIM para PSM*, onde um perfil é aplicado ao PIM a fim de gerar o PSM, (2) o estágio *Mapeamento do PSM para implementação*, onde um perfil é aplicado ao PSM a fim de gerar os esqueletos dos códigos fontes, e (3) o estágio *Preenchimento do Código Fonte*, onde são completados os esqueletos dos códigos fontes gerados no estágio imediatamente anterior (Figura 17, os estágios aparecem em negrito, pois não existiam no processo original), diferentemente do processo *UML Components* no qual este fluxo de trabalho não era subdividido em estágios.

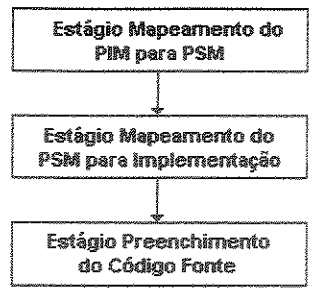

Figura 17 - Os estágios do fluxo de trabalho Provisionamento no processo adaptado

- Os fluxos de trabalho de *Montagem*, *Testes* e *Implementação* não sofreram adaptações.

Em resumo, as adaptações efetuadas, que são detalhadas nas seções subseqüentes, correspondem à inclusão dos seguintes estágios:

1. *Definição de Requisitos Não-Funcionais*, no fluxo de trabalho *Definição de Requisitos*.
2. *Refinamento da Arquitetura de Software*, no fluxo de trabalho *Especificação*.
3. *Mapeamento dos Componentes na Arquitetura de Software*, no fluxo de trabalho *Especificação*.
4. *Refinamento do PIM*, no fluxo de trabalho *Especificação*.
5. *Mapeamento do PIM para PSM*, no fluxo de trabalho *Provisionamento*.
6. *Mapeamento do PSM para Implementação*, no fluxo de trabalho *Provisionamento*.
7. *Preenchimento do Código Fonte*, no fluxo de trabalho *Provisionamento*.

A estrutura de pastas proposta no processo *UML Components* (Figura 6) também foi modificada (Figura 18, as alterações aparecem em negrito). A pasta *Especificações* da Figura 6 passou a ser chamada de *Especificações PIM* na Figura 18, e foi criada uma nova pasta, chamada de *Especificações PSM* na Figura 18 onde foram armazenados os modelos depois de efetuados os mapeamentos de PIM para PSM.

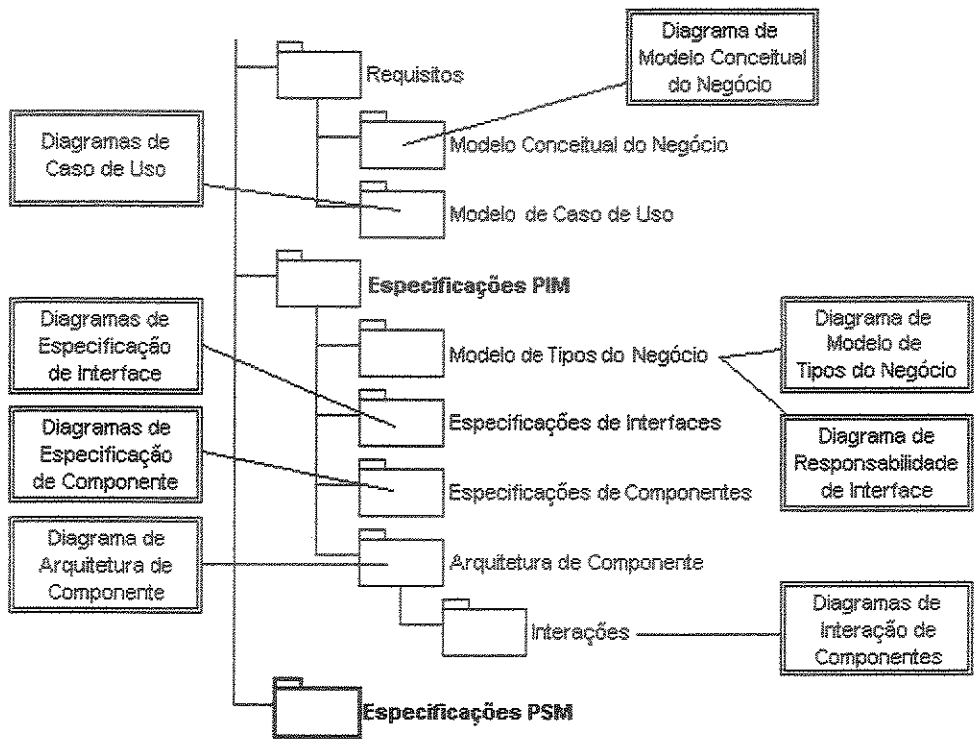

Figura 18 – Diagramas de modelagem de componentes no processo adaptado

3.1.1 Estágio Definição de Requisitos Não-Funcionais

A primeira adaptação efetuada no processo *UML Components* foi a inclusão do estágio *Definição de Requisitos Não-Funcionais* (Figura 15) no fluxo de trabalho *Definição de Requisitos* (Figura 14) que surgiu da necessidade de formalizar tais requisitos, que de acordo com o processo *UML Components* (seção 2.4.3), não são formalmente documentados. Os requisitos não-funcionais têm um papel cada vez mais importante no software como um todo. Como exemplo, podemos citar o aspecto de segurança em transações via Internet. A definição dos requisitos não-funcionais é um dos insumos para o estágio *Refinamento da Arquitetura de Software*.

Este estágio (Figura 19) apresenta apenas a atividade *Definição do requisito não-funcional*.

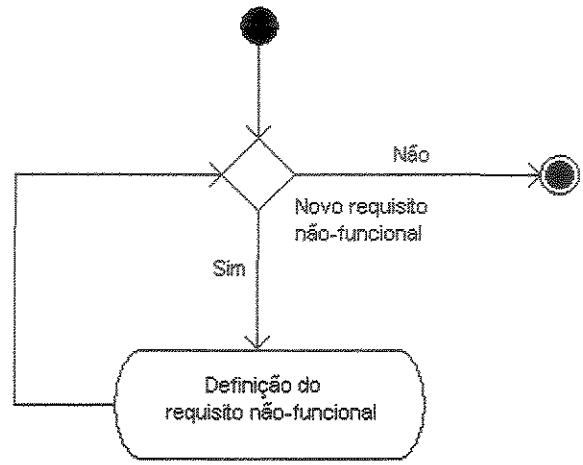

Figura 19 – Atividades do estágio Definição dos Requisitos não-Funcionais

Da mesma forma que se podem identificar requisitos relativos à qualidade de serviço como tempo de resposta e capacidade de atendimento para um caso de uso em particular, podem ser identificados requisitos não-funcionais para o sistema como um todo, ou uma parte dele, e que impactam na escolha da arquitetura de software.

Os requisitos não-funcionais devem ser definidos formalmente, bem como os critérios de avaliação dos mesmos, para que possam ser validados no fluxo de trabalho *Testes*. Alguns exemplos de requisitos não-funcionais desta categoria são escalabilidade, disponibilidade e desempenho. A definição de requisitos não-funcionais é textual, não havendo um diagrama UML correspondente. Para cada requisito não-funcional identificado devem ser definidos os seguintes aspectos:

- Nome do requisito não-funcional, que serve como identificação.
- Breve descrição do requisito não-funcional, para tornar claro o seu significado.
- Critério de avaliação do requisito não-funcional utilizado para validação.

3.1.2 Estágio Refinamento da Arquitetura de Software

A segunda adaptação foi a inclusão do estágio *Refinamento da Arquitetura de Software* (Figura 16) no fluxo de trabalho *Especificação* (Figura 14). O processo *UML Components* (seção 2.4.1) prevê a construção do software através de uma estrutura de camadas (Figura 2). O enfoque

do processo *UML Components* são as camadas que correspondem ao lado do servidor. Estas camadas são a de sistema e a de negócio, estritamente. De acordo com experiências realizadas verificamos a necessidade de, em alguns casos, refinar essa arquitetura de camadas. Portanto, o estágio *Refinamento da Arquitetura de Software* não faz mudanças estruturais radicais na maneira como o software é criado, em comparação com o processo *UML Components*, apenas permite dividir em mais camadas o software, para melhor tratar os requisitos não-funcionais. Além da possibilidade de separação em um número maior de camadas, o estágio *Refinamento da Arquitetura de Software* estabelece as atribuições de cada camada, o que norteia a criação de novos componentes, e cria uma especificação explícita da arquitetura de software que passa a integrar a especificação do sistema.

As camadas definidas no processo *UML Components* atendem satisfatoriamente os requisitos funcionais, no entanto, em função do atendimento dos requisitos não-funcionais do sistema pode haver necessidade de aumentar este número de camadas, por exemplo, para incluir uma camada de controle de acesso. Além desta, razões de ordem gerencial, como, por exemplo, granularidade dos componentes, podem demandar um número de camadas diferente do originalmente proposto. A arquitetura de software definida, assim como as demais especificações até este momento devem ser independentes de plataforma.

Os objetivos de criar este estágio no processo são:

- Tratar mais claramente o atendimento dos requisitos não-funcionais.
- Atender critérios internos de gerenciamento dos componentes.
- Documentar de forma explícita a arquitetura de software como parte integrante da especificação do sistema.
- Permitir um melhor mapeamento dos componentes nas diferentes camadas da arquitetura.

O principal ponto de partida para o refinamento da arquitetura é a Definição de Requisitos Não-Funcionais. Além da *Definição de Requisitos Não-Funcionais* outras informações relevantes podem ser obtidas nos casos de uso quando estes descreverem qualidade de serviço, como tempo de resposta, por exemplo.

As duas camadas, do lado do servidor, existentes até este ponto, a de sistema e a de negócio, podem ser ratificadas ou novas camadas podem ser criadas. Algumas motivações possíveis para criação de novas camadas são:

- Interpolação de camadas entre as existentes, visando atender requisitos não-funcionais, como, por exemplo, segurança.
- Divisão de camadas expondo elementos internos dos componentes, visando aumentar a granularidade.

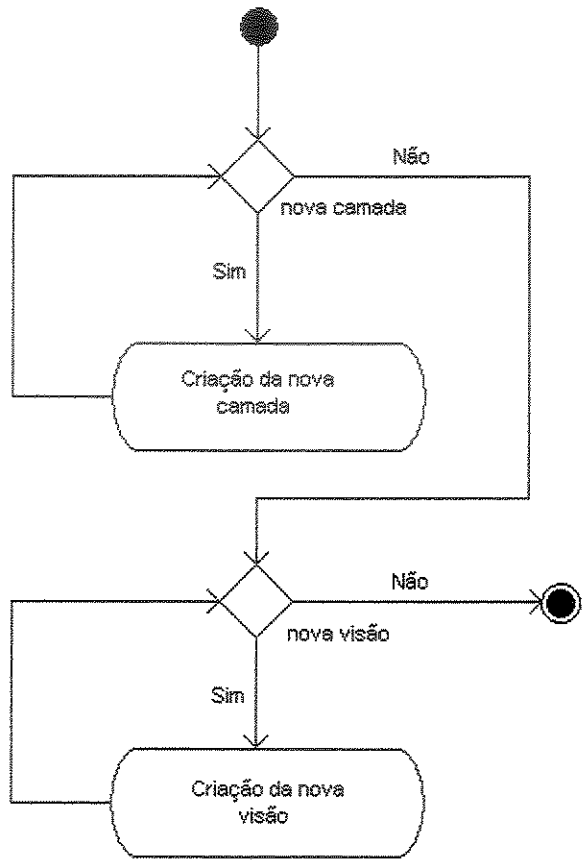

Figura 20 – Atividades do estágio Definição da Arquitetura de Software

Este estágio (Figura 20) apresenta as seguintes atividades: (1) criação de novas camadas, em função dos motivos expostos, se houver necessidade, e (2) criação de visões (seção 2.2.1) que documentam a arquitetura de software escolhida. A questão da criação das novas camadas já foi discutida anteriormente nesta mesma seção. A criação de visões segue as recomendações sugeridas por Bass, Clements e Kazman [BASS+03] e descritas na seção 2.2.1.

3.1.3 Estágio Mapeamento dos Componentes na Arquitetura de Software

A terceira adaptação foi a inclusão do estágio *Mapeamento dos Componentes na Arquitetura de Software* (Figura 16) no fluxo de trabalho *Especificação* (Figura 14). Neste estágio (Figura 21) são criados novos componentes, caso tenham sido criadas novas camadas (seção 3.1.2). A lógica para criação destes novos componentes é resultante da descrição de cada camada, e fica a critério do desenvolvedor responsável. No entanto, os novos componentes criados normalmente apresentam um dos seguintes comportamentos: (1) herdam, total ou parcialmente, as interfaces dos componentes das camadas de sistema ou de negócio, acrescentando, eventualmente operações relativas a requisitos não-funcionais; ou (2) representam uma fração das interfaces dos componentes das camadas de sistema ou de negócio. Em ambos os casos, as especificações originais das operações dos componentes devem ser herdadas, bastando especificar as operações das interfaces não atendidas pelas camadas de sistema ou negócio. A complementação das especificações das interfaces e dos componentes para os componentes ora criados deve ser feita utilizando mecanismo semelhante ao utilizado para descoberta das operações de sistema, quando a operação estiver relacionada com a camada de diálogo, ou semelhante ao utilizado para descoberta das operações de negócio, nas demais situações.

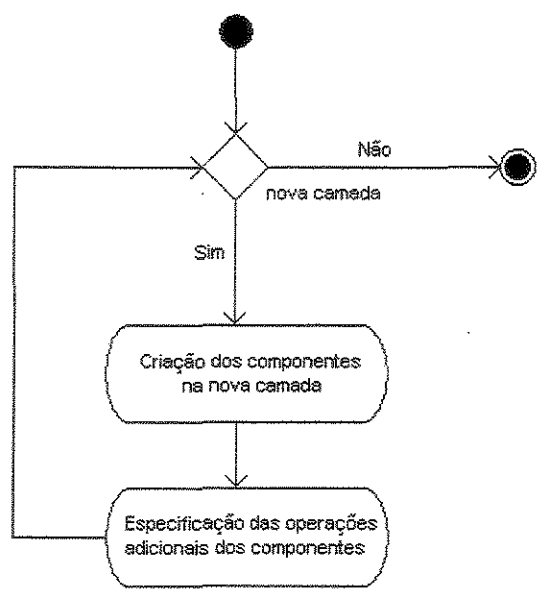

Figura 21 – Atividades do estágio Mapeamento dos Componentes na Arquitetura de Software

3.1.4 Estágio Refinamento do PIM

No processo *UML Components* os autores afirmam tentar *assegurar que as especificações de componentes criadas sejam, tanto quanto possível, independentes das idiossincrasias da tecnologia alvo* [CHEESMAN+01]. Em função disso, e do fato das especificações serem expressas utilizando UML, o resultado do fluxo de trabalho *Especificação*, no processo *UML Components*, é perfeitamente compatível com a abordagem MDA e conclui-se que o conjunto de especificações obtido pode ser considerado um PIM (seção 2.5.1), segundo a nomenclatura utilizada pela abordagem MDA. Em função disso, todos os modelos obtidos até este estágio são considerados PIM dentro da abordagem MDA.

Segundo a abordagem MDA o ideal é permitir gerar, tão automaticamente quanto possível, o PSM a partir do PIM, aumentando a rastreabilidade e facilitando a evolução do PSM. Para tanto é necessário refinar o PIM com detalhes que permitam um mapeamento mais preciso para o PSM. A abordagem MDA permite que tais detalhes sejam acrescentados ao PIM, desde que não incorporem as idiossincrasias da tecnologia alvo, garantindo, portanto, que o PIM permaneça independente de plataforma (seção 2.5.5). Este refinamento do PIM acrescenta detalhes através da utilização de um perfil (seção 2.5.3). A necessidade de se acrescentar detalhes no PIM decorre de dois fatores: (1) detalhes do padrão arquitetural que permitam distinguir, dentro de uma plataforma, diferentes construções para um determinado elemento, e (2) detalhes das plataformas alvo que necessitam ser abstraídos em características genéricas. Estes detalhes permitirão gerar automaticamente a implementação de determinada funcionalidade em qualquer plataforma alvo.

A quarta adaptação do processo decorre do exposto e refere-se à inclusão do estágio *Refinamento do PIM* (Figura 16) no fluxo de trabalho *Especificação* (Figura 14) onde primeiramente é feita a escolha do perfil adequado, que pode ser um perfil existente, um perfil existente modificado ou um novo perfil e, em seguida, o refinamento do PIM utilizando este perfil, acrescentando detalhes do padrão arquitetural e abstrações das plataformas alvo (Figura 22).

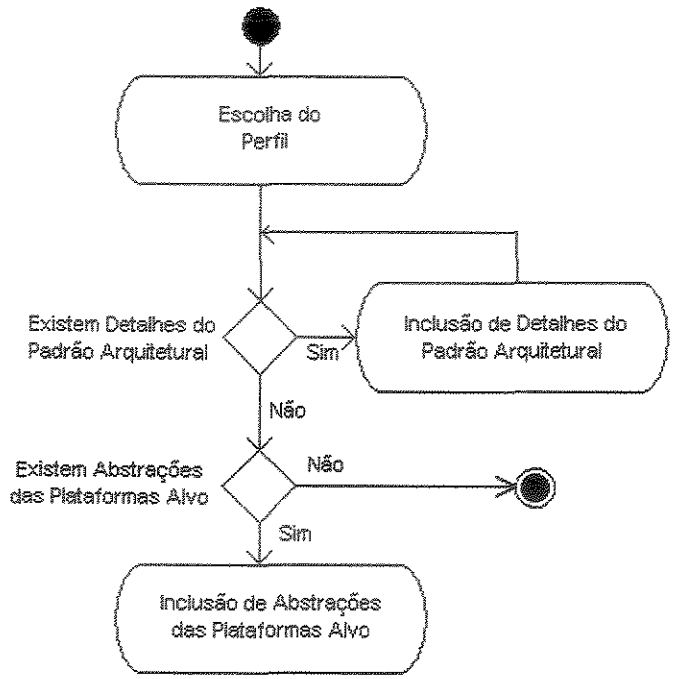

Figura 22 – Atividades do estágio Refinamento do PIM

No caso do processo *UML Components*, um elemento *<<comp spec>>* (Tabela 1) do PIM a ser implementado em uma plataforma distribuída pode precisar conter alguns detalhes que, independente da plataforma, alteram as características da geração do PSM, como, por exemplo, saber se uma determinada operação pode ou não ser incorporada em uma transação. Este detalhe pode, por exemplo, ser adicionado através de um valor sinalizado do tipo *transaction:true* no PIM e posteriormente ser mapeado nas plataformas alvo.

Existem outros detalhes que podem precisar ser incorporados no PIM no sentido de permitir um mapeamento mais preciso para os PSM e que não estão relacionados com o padrão arquitetural. Por exemplo, uma associação fim que especifica zero ou mais elementos (0..n) pode ser implementada de diferentes formas em qualquer plataforma alvo. Por exemplo, na linguagem Java ela pode ser implementada como *Set*, *SortedSet*, *List*, *Map*, *SortedMap*. Na linguagem C# ela pode ser implementada como *SortedList* ou *ArrayList*. E assim por diante. O que se deve fazer nesse caso é uma análise das necessidades e enriquecer o perfil com elementos abstratos que permitam distinguir, quando necessário, uma construção de outra. Nem sempre, porém, existe, em função das múltiplas possibilidades de implementação, a necessidade de detalhar o PIM.

Eventualmente pode-se adotar a estratégia de sempre gerar a mesma implementação para um elemento do PIM. Isto é possível desde que esta implementação única atenda a todas as necessidades.

Obviamente a solução mais simples é a adoção de algum perfil já pronto, como o *UML Profile for EDOC* [UML4EDOC], pois a preparação de um perfil é bastante trabalhosa. Contudo, pode não existir um perfil que atenda completamente as necessidades. Neste caso haverá a necessidade de (1) alterar algum perfil existente ou (2) criar um perfil totalmente novo. A técnica para criação ou extensão de um perfil está descrita na seção 2.5.3.

Nos mapeamentos utilizados no estudo de caso do PIM para PSM foi definido um perfil que inclui, além dos citados na Tabela 1, os seguintes estereótipos:

Estereótipo	Construção UML	Descrição
<<dlg comp spec>>	*class*	Estende a semântica do estereótipo *<<comp spec>>* (Tabela 1) do processo *UML Components* para indicar que o componente representa uma interface de diálogo com o usuário.
<<std comp spec>>	*class*	Estende a semântica do estereótipo *<<comp spec>>* (Tabela 1) do processo *UML Components* para indicar que o componente deverá ser gerado de acordo com o modelo de componentes COSMOS.

Tabela 2 – Estereótipos acrescentados ao processo UML Components

Idealmente a versão final do PIM deve conter informação suficiente para gerar o PSM de tal forma que nestes últimos não seja necessário fazer novos refinamentos. Portanto, caso haja necessidade de acrescentar alguma informação posteriormente no PSM, estas informações devem ser abstraídas e introduzidas no perfil utilizado, de tal forma que o PSM possa ser gerado automaticamente.

3.1.5 Estágio Mapeamento do PIM para PSM

A quinta adaptação do processo *UML Components* foi a inclusão do estágio *Mapeamento do PIM para PSM* (Figura 17) no fluxo de trabalho *Provisionamento* (Figura 14) onde são criados os modelos dependentes de plataforma (PSM), através de mapeamentos explícitos, a partir dos modelos independentes de plataforma (PIM), de acordo com a abordagem MDA.

Este estágio (Figura 23) inclui as atividades de: (1) Definição da plataforma alvo, (2) Escolha do Perfil e (3) Mapeamento do PIM para PSM. Os modelos obtidos até o estágio imediatamente anterior são independentes de plataforma. Neste estágio são aplicados os mapeamentos necessários para transformar estes modelos independentes de plataforma em modelos dependentes de plataforma. Ao definir as plataformas alvo é importante ter um bom grau de conhecimento, para saber se ela consegue atender as especificações geradas, sobretudo no que diz respeito à especificação da arquitetura de software. Da mesma forma, a escolha dos perfis para mapeamento exige um bom grau de conhecimento das plataformas alvo para tentar antecipar as necessidades de abstrações que devam ser acrescentadas no estágio *Refinamento do PIM* (seção 3.1.4).

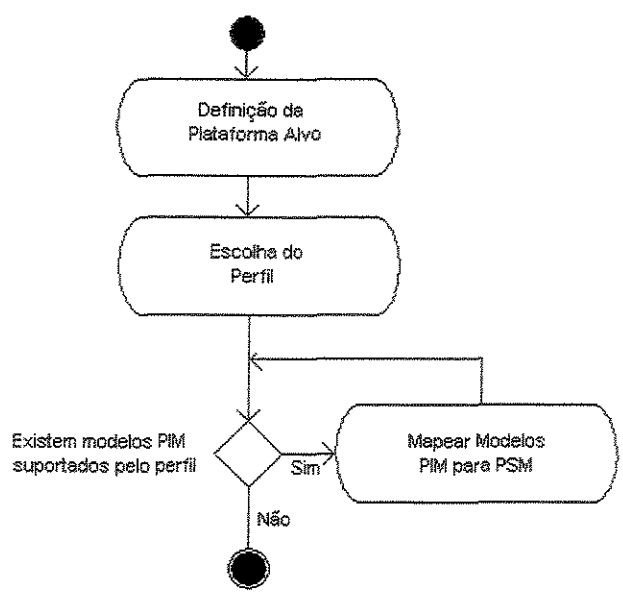

Figura 23 – Atividades do estágio Mapeamento do PIM para PSM

A idéia é permitir mapear automaticamente todos os elementos do PIM para PSM, eliminando a necessidade de refinamentos no PSM. Caso surja a necessidade de refinamentos no

PSM, tais necessidades devem ser abstraídas para o PIM, ou seja, transformadas em necessidades genéricas para quaisquer plataformas, e então gerado novamente o PSM.

Note-se que somente as informações contidas no perfil são mapeadas para o PSM. Portanto o perfil deve conter, além das extensões, regras para mapeamento entre os diversos modelos. Por exemplo, um elemento *Class* do PIM, deve ser mapeado como *Class* no PSM para J2EE. A não definição de regras para transformação do elemento implica a sua eliminação quando da transformação de um elemento de um modelo para outro.

O grau de rastreabilidade entre os modelos PIM e PSM é tanto maior quanto mais claras forem as regras de mapeamento, a tal ponto, que estes possam ser automatizados através de ferramentas com esta finalidade.

As regras de mapeamento devem ser capazes de traduzir os seguintes elementos dos modelos anteriores para PSM:

- Atributos do modelo de informações e seus tipos: os tipos do modelo de informações têm que ser mapeados para os tipos definidos na plataforma alvo. Além disso, é necessário verificar a existências invariantes associadas com o atributo, o que pode, dependendo da plataforma alvo, criar a necessidade de se definir o atributo como sendo uma Propriedade da Interface que tem um comportamento particular, sabidamente, a existência de uma operação de *Get()* e uma de *Set()*, utilizadas para comportar as restrições contidas na invariante.
- Assinaturas das operações: os tipos dos parâmetros têm de ser mapeados para os tipos definidos na plataforma alvo. Existe também a necessidade de mapear a direção dos parâmetros, ou seja, se os parâmetros são de entrada, saída, entrada e saída, ou retorno. Outro ponto que deve ser mapeado é a forma de passagem de parâmetro, ou seja, se os parâmetros serão passados por valor ou por referência.
- Criação dos objetos: É comum, entre as plataformas, a utilização do padrão *Factory* para criação de componentes, onde um objeto componente é usado para criar instâncias de outro componente. Cada plataforma define um objeto *Factory*. No modelo independente de plataforma a criação de tais classes, e respectivas operações de criação, não aparecem, podendo, eventualmente, aparecer métodos do tipo *create* ou algo parecido. Portanto, os métodos do tipo *create*, ou similar, devem ser mapeados para as estruturas de criação de

objetos particulares de cada plataforma, incorporando toda a parafernália necessária para tanto.

- Suporte a interfaces: As interfaces, conforme definidas no modelo independente de plataforma, não estão sujeitas às restrições relativas aos aspectos formais de cada uma das plataformas. Algumas plataformas permitem apenas uma única herança de interface, enquanto outras permitem herança múltipla de interfaces, por outro lado em algumas plataformas os componentes podem implementar várias interfaces. Existem, ainda, outros aspectos restritivos que dizem respeito a limitações impostas quando do registro da interface de um componente em uma determinada plataforma. Todas essas restrições devem ser mapeadas entre o PIM e o PSM de cada plataforma, de tal forma que a interface apresente na plataforma alvo a funcionalidade especificada de forma independente da plataforma.

Os PSM são gerados no pacote **Especificações PSM** (Figura 18), e correspondem à tradução, para a plataforma alvo, das especificações das interfaces, dos componentes, e da arquitetura de componentes.

À primeira vista, pode parecer desnecessário gerar o PSM, uma vez que a sua criação é automática, e a partir deste, são geradas as implementações, ou seja, trata-se apenas de um elemento de transição e, portanto, os dois mapeamentos podem ser seqüencialmente aplicados para gerar o código fonte a partir do PIM. No entanto, a criação destes é valiosa nos seguintes aspectos:

- O PSM baseado em UML ajuda na visualização da arquitetura específica da plataforma gerada a partir do PIM, incluindo, por exemplo, padrões de projeto da plataforma inexistentes no PIM, o que pode ser útil na depuração.
- O PSM baseado em UML, embora esteja em um mesmo nível de abstração que o código fonte na plataforma alvo, é semanticamente mais rico que este. As informações semânticas adicionais, como pré- e pós-condições, podem ajudar na depuração e testes, e para o implementador o diagrama expresso na tecnologia alvo é mais legível que o diagrama expresso na forma de PIM.

3.1.6 Estágio Mapeamento do PSM para Implementação

A sexta adaptação foi a inclusão do estágio *Mapeamento do PSM para Implementação* (Figura 17), ainda no fluxo de trabalho *Provisionamento* (Figura 14), onde são criados os esqueletos dos códigos fonte em cada uma das plataformas alvo a partir do PSM.

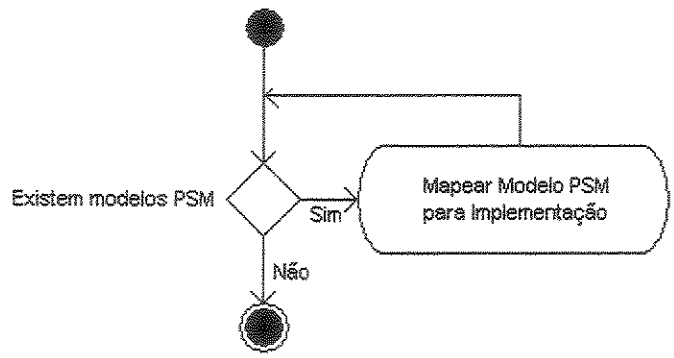

Figura 24 – Atividades do estágio Mapeamento do PIM para PSM

Conforme mencionado anteriormente (seção 2.5.4) existem quatro formas de mapeamento. No estágio *Mapeamento do PSM para Implementação* opta-se pela criação de um esqueleto (com o código fonte) a ser completado pelo programador. A geração do esqueleto é feita através do mesmo perfil utilizado para criar o PSM para cada plataforma, isto porque o PSM e o esqueleto do código fonte são modelos com um mesmo nível de abstração e o perfil, além de especificar como representar um determinado elemento do PIM no PSM (ambos modelos UML), também especifica como um elemento do PSM deve ser codificado no esqueleto a ser gerado. A geração de esqueleto é feita para cada um dos modelos PSM (Figura 24).

Com relação à geração do esqueleto com o código fonte cabem algumas considerações. Idealmente o modelo deve gerar todo o código necessário para sua execução. Embora essa idéia não seja utópica (ela existe, por exemplo, em um banco de dados que processa as informações a partir de um modelo de base de dados) a sua aplicabilidade torna-se bastante difícil quando da implementação de linguagem imperativa. Veja-se o exemplo de implementação de uma Classe simples contendo um atributo, uma operação *Get()*, e uma operação *Set()*. O modelo UML é capaz de representar a classe, os seus atributos e as suas operações. O modelo é capaz, até mesmo, de estabelecer invariantes para a classe, e pré- e pós-condições para as operações. No entanto, é extremamente difícil definir no modelo como, por exemplo, a operação *Get()* será

implementada. Talvez ela simplesmente retorne o valor do atributo. Talvez ela faça alguma operação aritmética simples com este atributo. Talvez este atributo sirva como parâmetro em um Web Service que retorne algum valor. Talvez o valor esteja armazenado em um banco de dados. E assim por diante. Mesmo se as invariantes, as pré- e pós-condições fossem integralmente traduzidas para a plataforma alvo seria necessário descrever a implementação desta operação, ou seja, modelar "como". As formas de resolver tal problema são: (1) incorporar o algoritmo utilizado na linguagem alvo, o que fica descartado, pois descaracteriza o PIM como modelo independente de plataforma, ou (2) descrever o algoritmo através de alguma linguagem de ação independente de plataforma, como, por exemplo, as semânticas de ação contidas na linguagem UML [UML03], mas a tradução automática desta em uma linguagem como Java ou C# está além do escopo deste trabalho. Já existem no mercado ferramentas que permitem a sincronização de modelos com o código fonte, neste trabalho, contudo, este mapeamento foi feito manualmente.

3.1.7 Estágio Preenchimento do Código Fonte

A sétima adaptação foi a criação, ou formalização, do estágio *Preenchimento do Código Fonte* (Figura 17) no fluxo de trabalho *Provisionamento* (Figura 14), onde é necessário preencher o esqueleto gerado a partir do mapeamento do PSM para implementação. Neste ponto a utilização dos PSM como material de apoio é de grande importância, pois permite uma visão do componente como um todo e não dos arquivos separadamente, bem como apresenta as restrições que devem ser mantidas na implementação do componente. Este estágio corresponde ao próprio fluxo de trabalho *Provisionamento* no processo *UML Components* (Figura 3), onde não havia subdivisão explícita em estágios.

3.2 Mapeamentos PIM para PSM J2EE

Os mapeamentos do PIM para PSM J2EE apresentados nesta seção são baseados no modelo COSMOS (seção 2.6) e na especificação EJB [EJB+01], e utilizados no estudo de caso apresentado no capítulo 4. As regras apresentadas a seguir foram definidas visando (1) manter as diretrizes do modelo COSMOS, (2) permitir um mapeamento posterior para uma unidade de distribuição autônoma, e, portanto, coerente com o conceito de componente como unidade substituível, e (3) expor a menor quantidade possível de classes e interfaces, simplificando a utilização do componente.

As seções 3.2.1 e 3.2.2 apresentam, respectivamente, o Modelo de Componente COSMOS para J2EE e o Modelo de Conector COSMOS para J2EE. Em seguida, nas Seções 3.2.3, 3.2.4 e 3.2.5 são apresentados os perfis utilizados, criados a partir destes modelos, para mapear o PIM para PSM J2EE.

3.2.1 Modelo de Componente COSMOS para J2EE

O modelo de componente COSMOS para J2EE é baseado no modelo COSMOS (seção 2.6) e na especificação EJB [EJB+01].

Atributos Utilizados

Alguns atributos dos componentes são utilizados para definição do modelo, a saber:

- *<component_name>*: nome do componente.
- *<interface_provided_name(n)>:* nome da enésima interface provida do componente.
- *<interface_required_name(n)>*: nome da enésima interface requerida do componente.

Grupos de Elementos

O modelo apresenta a estrutura interna do componente definida a partir do modelo COSMOS (Figura 12). No modelo cada componente é composto por um conjunto de elementos. Estes elementos são classificados em seis grupos distintos:

1. Especificação das interfaces providas: interfaces definidoras das funcionalidades do componente. Estas interfaces estão armazenadas no pacote *<component_name>.spec.prov*.
2. Especificação das interfaces requeridas (opcional): interfaces definidoras das funcionalidades requeridas pelo componente. Estas interfaces estão armazenadas no pacote *<component_name>.spec.req*.
3. *Factory* do componente: classe que retorna uma instância do gerenciador do componente, e que serve como ponto de entrada para utilização do componente. Esta classe está armazenada no pacote *<component_name>.impl*.
4. Implementação do componente: classes e interfaces utilizadas para gerenciamento no nível do componente. Estes elementos estão armazenados no pacote *<component_name>.impl*.
5. Implementação das interfaces providas: classes e interfaces que definem um *session bean*

para cada interface provida pelo componente e uma classe comum que implementa os métodos obrigatórios definidos na interface *javax.ejb.SessionBean*. Estes elementos são responsáveis pelo gerenciamento no nível da interface provida e estão armazenados no pacote *<component_name>.impl*.

6. Auxiliares (opcional): classes e interfaces auxiliares utilizadas pelas interfaces providas para implementar os métodos providos. Estes elementos, caso existam, devem ser armazenados no pacote *<component_name>.impl*. Do ponto de vista prático, todas as classes e interfaces não classificadas anteriormente são elementos auxiliares.

No modelo original (Figura 12) os grupos *Factory* do componente, Implementação do componente, Implementação das interfaces providas e Auxiliares aparecem como Classes de implementação, mas neste modelo foram divididos por motivos didáticos.

Especificação das Interfaces Providas

Os elementos da especificação das interfaces providas são:

- *I<interface_provided_name(n)>*: Uma especificação dos métodos providos para cada interface provida. Estas interfaces têm visibilidade pública.

Especificação das Interfaces Requeridas

Os elementos da especificação das interfaces requeridas são:

- *I<interface_required_name(n)>*: Uma especificação dos métodos providos para cada interface requerida. Estas interfaces têm visibilidade pública.

Factory do Componente

O *factory* do componente é formado por:

- *ComponentFactory*. Uma classe para cada componente, que implementa apenas o método *CreateInstance()* do tipo *static*, sem parâmetros e com retorno do tipo *IManager*. Esta classe tem visibilidade pública.

Implementação do Componente

Os elementos da implementação do componente são:

- *IManager*: Uma interface para cada componente, que define as operações de manipulação das interfaces providas pelo componente. Estas operações são *GetProvidedInterface()*, *GetProvidedInterfaces()*, *SetRequiredInterface()*, *GetRequiredInterface()*, *GetRequiredInterfaces()*, conforme o modelo COSMOS. Esta interface tem visibilidade

pública.

- *IComponentMgt*: Uma interface para cada componente, que estende a interface *IManager*, acrescentando o método *GetObjectFactory()*, responsável pela criação de instâncias para as interfaces internas utilizadas pelo componente. Como o método *GetObjectFactory()* não precisa e, preferencialmente, não deve ser visualizado externamente, este método é incorporado nesta interface, que tem visibilidade de pacote. Este método retorna uma referência para uma *factory* de criação dos objetos utilizados internamente pelo componente. Como esta interface é utilizada como parâmetro de cada *session bean* no método *ejbCreate()* é necessário que ela estenda a interface *Serializable*.
- *ComponentMgt*: Uma classe para cada componente, que implementa a interface *IComponentMgt*, e por extensão a interface *IManager*. Esta classe contém uma referência para uma *factory* que é retornada através do método *GetObjectFactory()* e também listas com os nomes e objetos correspondentes às interfaces providas e requeridas. Esta classe tem visibilidade de pacote.
- *IObjectFactory*: Uma interface para cada componente, que define os métodos para criação dos objetos utilizados internamente pelo componente. Esta interface tem visibilidade de pacote. Como esta interface é utilizada como parâmetro no método *GetObjectFactory()* da classe *ComponentMgt* ela precisa estender a interface *Serializable*.
- *ObjectFactory*: Uma classe para cada componente. Esta classe implementa os métodos definidos em *IObjectFactory*. Em particular os métodos para criação das interfaces providas, criam as interfaces locais (*home*) e, a partir destas, criam instâncias para as interfaces remotas, que correspondem às interfaces providas. Este trabalho normalmente realizado pelos clientes dos componentes EJB, fica encapsulado pela chamada do método *GetProvidedInterface()* correspondente. Esta classe tem visibilidade de pacote.

Implementação das Interfaces Providas

Os elementos da implementação das interfaces providas são:

- *I<interface_provided_name(n)>MgtHome*: Uma interface para cada interface provida. Esta interface estende a interface *javax.ejb.EJBHome* (chamada de interface local), obrigatória de acordo com a especificação EJB [EJB+01]. Esta interface define um método *create()* com parâmetro *IComponentMgt* e retorno

I<interface_provided_name(n)>MgtRemote. De acordo com a especificação EJB esta interface deve ser pública.

- *I<interface_provided_name(n)>MgtRemote*: Uma interface para cada interface provida. Esta interface (1) estende cada interface *I<interface_provided_name(n)>Mgt* definida no pacote *spec.prov*, e (2) estende a interface *javax.ejb.EJBObject* requerida pela especificação EJB [EJB+01]. Esta interface é chamada de interface remota. De acordo com a especificação EJB esta interface deve ser pública.
- *<interface_provided_name(n)>Mgt*: Uma classe para cada interface provida. Esta classe (1) implementa a interface *I<interface_provided_name(n)>MgtRemote* e (2) estende a classe *ObjectManager*, descrita adiante. O objetivo de estender a classe *ObjectManager* é fazer com que esta classe implemente somente os métodos providos pela interface *I<interface_provided_name(n)>Mgt*, ou seja, os métodos de negócio, promovendo uma separação de interesses que facilita a manutenção e evolução. De acordo com a especificação EJB esta classe deve ser pública.
- *ObjectManager*: Uma classe para cada componente. Esta classe implementa os métodos comuns obrigatórios para prover a interface *javax.ejb.SessionBean*, elimina o método *ejbCreate()* sem parâmetros, introduzindo o método *ejbCreate()* com o parâmetro *IComponentMgt*, ou seja, a referência da interface provida para o gerenciador do componente. Os métodos obrigatórios da especificação EJB [EJB+01] são: (1) *getSessionContext()*, (2) *setSessionContext()*, (3) *ejbCreate()*, (4) *ejbActivate()*, (5) *ejbPassivate()*, (6) *ejbRemove()*. O método *ejbCreate()* corresponde ao método *create()* definido na interface *I<interface_provided_name(n)>MgtHome*. Esta classe define, ainda, o método *GetManager()* que retorna a referência, passada como parâmetro no método *ejbCreate()*, e que permite acessar as interfaces requeridas, bem como a *factory* do componente. Esta classe tem visibilidade de pacote, atributo *abstract*, e é utilizada por todas as interfaces providas implementadas pelo componente.

Comentários

O modelo apresentado minimiza referências diretas entre classes, fazendo com que os elementos internos dos componentes utilizem interfaces e obtenham as instâncias destas através do gerenciador. Portanto, adota como padrão propagar o gerenciador através do construtor do

componente, passando *IComponentMgt* como parâmetro. Desta forma, o componente acessa os construtores e as interfaces requeridas através do gerenciador. Como a interface *IComponentMgt* não é pública, embora a classe *<interface_provided_name(n)>Mgt* e as interfaces *I<interface_provided_name(n)>MgtHome* e *I<interface_provided_name(n)>MgtRemote* sejam, não é possível para o cliente do componente acessar o componente EJB ao não ser através da chamada do método *GetProvidedInterface()*, pois a implementação do método *create()* da classe *ObjectManager* lança uma exceção caso o parâmetro seja passado como nulo.

3.2.2 Modelo de Conector COSMOS para J2EE

A estrutura do modelo de conector COSMOS para J2EE é próxima a do modelo de componentes. No entanto, enquanto um componente PSM surge a partir de um componente PIM, um conector PSM, que não existe no modelo PIM, surge a partir da interação entre dois ou mais componentes PIM. O componente especifica as interfaces providas enquanto as interfaces providas por um conector são especificadas no componente a quem o conector que se destina. Da mesma forma as interfaces requeridas por um conector não são especificadas em si, como nos componentes, mas em componentes PIM que provêem as interfaces.

O modelo de conector COSMOS para J2EE é baseado no modelo COSMOS (seção 2.6) e na especificação EJB [EJB+01].

Atributos Utilizados

Alguns atributos dos componentes são utilizados para definição do modelo, a saber:

- *<target_component_name>*: nome do componente cuja interface ou interfaces requeridas são supridas pelo conector.
- *<target_interface_required_name(n)>*: nome da enésima interface requerida pelo componente a quem o conector se destina.

Grupos de Elementos

O modelo apresenta a estrutura interna do conector definida a partir do modelo COSMOS. No modelo cada conector é composto por um conjunto de elementos. Estes elementos são classificados em quatro grupos distintos:

1. *Factory* do conector: classe que retorna uma instância do gerenciador do conector, e que serve como ponto de entrada para utilização do conector. Esta classe está armazenada no

pacote *<target_component_name>.impl*.

2. Implementação do conector: classes e interfaces utilizadas para gerenciamento no nível do conector. Estes elementos estão armazenados no pacote *<target_component_name>.impl*.
3. Implementação das interfaces providas: classes e interfaces que definem um session bean para cada interface provida pelo conector e uma classe comum que implementa os métodos obrigatórios definidos na interface *javax.ejb.SessionBean*. Estes elementos são responsáveis pelo gerenciamento no nível da interface provida e estão armazenados no pacote *<target_component_name>.impl*.
4. Auxiliares (opcional): classes e interfaces auxiliares utilizadas pelas interfaces providas para implementar os métodos providos. Estes elementos, caso existam, devem ser armazenados no pacote *<target_component_name>.impl*. Do ponto de vista prático, todas as classes e interfaces não classificadas anteriormente são elementos auxiliares.

Factory do Conector

O *factory* do conector é formado por:

- *ComponentFactory*. Uma classe para cada conector, que implementa apenas o método *CreateInstance()* do tipo *static*, sem parâmetros e com retorno do tipo *IManager*. Esta classe tem visibilidade pública.

Implementação do Conector

Os elementos da implementação do conector são:

- *IManager*: Uma interface para cada conector, que define as operações de manipulação das interfaces providas pelo conector. Estas operações são *GetProvidedInterface()*, *GetProvidedInterfaces()*, *SetRequiredInterface()*, *GetRequiredInterface()*, *GetRequiredInterfaces()*, conforme o modelo COSMOS. Esta interface tem visibilidade pública.
- *IComponentMgt*: Uma interface para cada conector, que estende a interface *IManager*, acrescentando o método *GetObjectFactory()*, responsável pela criação de instâncias para as interfaces internas utilizadas pelo conector. Como o método *GetObjectFactory()* não precisa e, preferencialmente, não deve ser visualizado externamente, este método é incorporado nesta interface, que tem visibilidade de pacote. Este método retorna uma referência para uma *factory* de criação dos objetos utilizados internamente pelo conector.

Como esta interface é utilizada como parâmetro de cada *session bean* no método *ejbCreate()* é necessário que ela estenda a interface *Serializable*.

- *ComponentMgt*: Uma classe para cada conector, que implementa a interface *IComponentMgt*, e por extensão a interface *IManager*. Esta classe contém uma referência para uma *factory* que é retornada através do método *GetObjectFactory()* e também listas com os nomes e objetos correspondentes às interfaces providas e requeridas. Esta classe tem visibilidade de pacote.
- *IObjectFactory*: Uma interface para cada conector, que define os métodos para criação dos objetos utilizados internamente pelo conector. Esta interface tem visibilidade de pacote. Como esta interface é utilizada como parâmetro no método *GetObjectFactory()* da classe *ComponentMgt* ela precisa estender a interface *Serializable*.
- *ObjectFactory*: Uma classe para cada conector. Esta classe implementa os métodos definidos em *IObjectFactory*. Em particular os métodos para criação das interfaces providas, criam as interfaces locais (home) e, a partir destas, criam instâncias para as interfaces remotas, que correspondem às interfaces providas. Este trabalho normalmente realizado pelos clientes dos componentes EJB, fica encapsulado pela chamada do método *GetProvidedInterface()* correspondente. Esta classe tem visibilidade de pacote.

Implementação das Interfaces Providas

Os elementos da implementação das interfaces providas são:

- *I<target_interface_required_name(n)>MgtHome*: Uma interface para cada interface provida. Esta interface estende a interface *javax.ejb.EJBHome* (chamada de interface local), obrigatória de acordo com a especificação EJB [EJB+01]. Esta interface define um método *create()* com parâmetro *IComponentMgt* e retorno *I<target_interface_required_name(n)>MgtRemote*. De acordo com a especificação EJB esta interface deve ser pública.
- *I<target_interface_required_name(n)>MgtRemote*: Uma interface para cada interface provida. Esta interface (1) estende cada interface *I<target_interface_required_name(n)>Mgt*, e (2) estende a interface *javax.ejb.EJBObject* requerida pela especificação EJB [EJB+01]. Esta interface é chamada de interface remota. De acordo com a especificação EJB esta interface deve ser pública.

- *<target_interface_required_name(n)>Mgt*: Uma classe para cada interface provida. Esta classe (1) implementa a interface *I<target_interface_required_name(n)>MgtRemote* e (2) estende a classe *ObjectManager*, descrita adiante. O objetivo de estender a classe *ObjectManager* é fazer com que esta classe implemente somente os métodos providos pela interface *I<target_interface_required_name(n)>Mgt*, ou seja, os métodos de negócio, promovendo uma separação de interesses que facilita a manutenção e evolução. De acordo com a especificação EJB esta classe deve ser pública.
- *ObjectManager*: Uma classe para cada componente. Esta classe implementa os métodos comuns obrigatórios para prover a interface *javax.ejb.SessionBean*, elimina o método *ejbCreate()* sem parâmetros, introduzindo o método *ejbCreate()* com o parâmetro *IComponentMgt*, ou seja, a referência da interface provida para o gerenciador do componente. Os métodos obrigatórios da especificação EJB [EJB+01] são: (1) *getSessionContext()*, (2) *setSessionContext()*, (3) *ejbCreate()*, (4) *ejbActivate()*, (5) *ejbPassivate()*, (6) *ejbRemove()*. O método *ejbCreate()* corresponde ao método *create()* definido na interface *I<target_interface_required_name(n)>MgtHome*. Esta classe define, ainda, o método *GetManager()* que retorna a referência, passada como parâmetro no método *ejbCreate()*, e que permite acessar as interfaces requeridas, bem como a *factory* do componente. Esta classe tem visibilidade de pacote, atributo *abstract*, e é utilizada por todas as interfaces providas implementadas pelo componente.

Comentários

O modelo apresentado minimiza referências diretas entre classes, fazendo com que os elementos internos dos conectores utilizem interfaces e obtenham as instâncias destas através do gerenciador. Portanto, adota como padrão propagar o gerenciador através do construtor do conector, passando *IComponentMgt* como parâmetro. Desta forma, o conector acessa os construtores e as interfaces requeridas através do gerenciador. Como a interface *IComponentMgt* não é pública, embora a classe *<interface_provided_name(n)>Mgt* e as interfaces *I<interface_provided_name(n)>MgtHome* e *I<interface_provided_name(n)>MgtRemote* sejam, não é possível para o cliente do conector acessar o componente EJB ao não ser através da chamada do método *GetProvidedInterface()*, pois a implementação do método *create()* da classe *ObjectManager* lança uma exceção caso o parâmetro seja passado como nulo.

3.2.3 Perfil para Mapeamento de Componentes dos Modelos PIM para PSM J2EE

O perfil para mapeamento de componentes dos modelos PIM para PSM J2EE inclui os estereótipos descritos na Tabela 1 ou na Tabela 2.

O perfil apresentado gera modelos PSM J2EE anotados com estereótipos contidos no perfil *UML Profile for EJB* [UML4EJB].

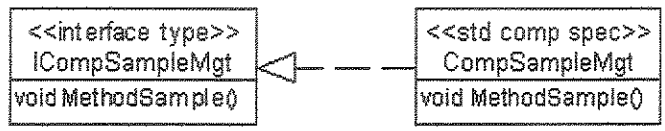

Figura 25 – Exemplo de componente PIM

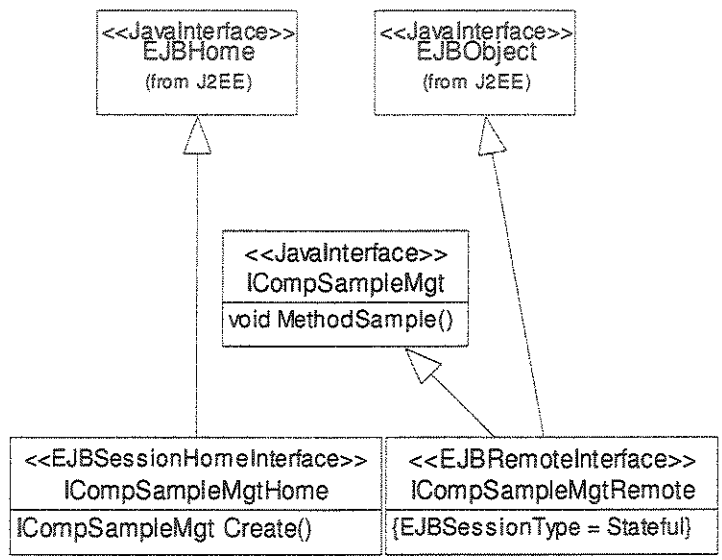

Figura 26 –Exemplo de interface PSM J2EE

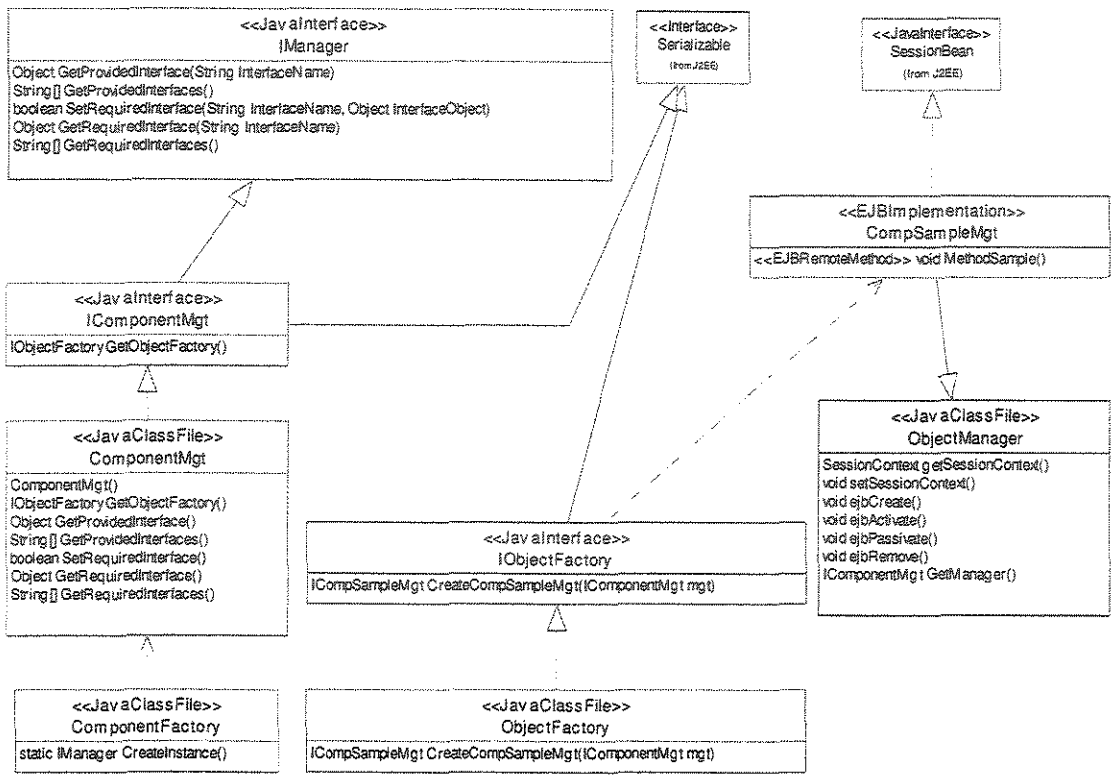

Figura 27 – Exemplo de componente PSM J2EE

O modelo PSM J2EE (Figuras 26 e 27), gerado a partir do modelo PIM (Figura 25), é um exemplo de aplicação das regras de mapeamento de interfaces e componentes. Por motivos didáticos a figura foi dividida em duas, mas na implementação as Figuras 26 e 27 representam um único componente, sendo que a classe *CompSampleMgt* da Figura 27, implementa a interface *ICompSampleMgtRemote* da Figura 26. As regras utilizadas nesta transformação são descritas a seguir:

1. Para cada classe do PIM com estereótipo *<<std comp spec>>* é criado um pacote *<component_name>* com estereótipo *<<EJB-JAR>>* e este é o pacote raiz para todos os elementos gerados a partir desta classe.
2. Para cada classe do PIM com estereótipo *<<std comp spec>>* é criado um componente no pacote *<component_name>* com estereótipo *<<EJBDescriptor>>*.
3. Para cada classe do PIM com estereótipo *<<std comp spec>>* são criados: (1) um pacote

<component_name>.spec.prov e (2) um pacote *<component_name>.impl.*

4. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta uma ou mais interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um pacote *<component_name>.spec.req.*
5. Para cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criada uma interface *I<interface_provided_name(n)>* com estereótipo *<<JavaInterface>>* no pacote *<component_name>.spec.prov.*
6. Para cada método de cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criado um método com assinatura equivalente na interface *I<interface_provided_name(n)>* do pacote *<component_name>.spec.prov.*
7. Para cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criada uma interface *I<interface_required_name(n)>* com estereótipo *<<JavaInterface>>* no pacote *<component_name>.spec.req.*
8. Para cada método de cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criado um método com assinatura equivalente na interface *I<interface_required_name(n)>* do pacote *<component_name>.spec.req.*
9. Para cada classe do PIM com estereótipo *<<std comp spec>>* são criados no pacote *<component_name>.impl* do PSM: (1) uma classe *ComponentFactory* com estereótipo *<<JavaClassFile>>*, (2) uma interface *IManager* com estereótipo *<<JavaInterface>>*, (3) uma interface *IComponentMgt* com estereótipo *<<JavaInterface>>*, (4) uma classe *ComponentMgt* com estereótipo *<<JavaClassFile>>*, (5) uma interface *IObjectFactory* com estereótipo *<<JavaInterface>>*, (6) uma classe *ObjectFactory* com estereótipo *<<JavaClassFile>>*, (7) uma classe *ObjectManager* com estereótipo *<<JavaClassFile>>*.
10. Para cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* são criados no pacote *<component_name>.impl*: (1) uma interface *I<interface_provided_name(n)>MgtHome* com estereótipo *<<EJBSessionHomeInterface>>* e valor sinalizado *EJBSessionType = Stateful*, (2) uma

interface *I<interface_provided_name(n)>MgtRemote* com estereótipo *<<EJBRemoteInterface>>*, (3) uma classe *<interface_provided_name(n)>Mgt* com estereótipo *<<EJBImplementation>>*.

11. Para cada método de cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criado um método com assinatura equivalente na classe *<interface_required_name(n)>Mgt* do pacote *<component_name>.impl* com estereótipo *<<EJBRemoteMethod>>*.
12. As conversões de tipo do PIM para o J2EE são feitas de acordo com a tabela de conversão de tipos do perfil *UML Profile for EJB* [UML4EJB].

Nas regras para mapeamento dos componentes cada classe *<<std comp spec>>* gera um *Session Bean*, sinalizado como *stateful*, ou seja, com informação de estado. Isto se deve ao fato de que as classes de implementação do *Session Bean*, ao herdarem de *ObjectManager*, precisam persistir *IComponentMgt* durante a sessão.

3.2.4 Perfil para Mapeamento de Conectores dos Modelos PIM para PSM J2EE

O perfil para mapeamento de conectores dos modelos PIM para PSM J2EE inclui os estereótipos descritos na Tabela 1 ou na Tabela 2.

O perfil apresentado gera modelos PSM J2EE anotados com estereótipos contidos no perfil *UML Profile for EJB* [UML4EJB].

As regras de mapeamento de conectores são:

1. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um pacote *<component_name>* com estereótipo *<<EJB-JAR>>* e este é o pacote raiz para todos os elementos gerados a partir desta classe.
2. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um componente no pacote *<component_name>* com estereótipo *<<EJBDescriptor>>*.
3. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado

um pacote *Connector<target_component_name>.impl.*

4. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, são criados no pacote *Connector<target_component_name>.impl*: (1) uma classe *ComponentFactory* com estereótipo *<<JavaClassFile >>*, (2) uma interface *IManager* com estereótipo *<<JavaInterface>>*, (3) uma interface *IComponentMgt* com estereótipo *<<JavaInterface>>*, (4) uma classe *ComponentMgt* com estereótipo *<<JavaClassFile>>*, (5) uma interface *IObjectFactory* com estereótipo *<<JavaInterface>>*, (6) uma classe *ObjectFactory* com estereótipo *<<JavaClassFile>>*, (7) uma classe *ObjectManager* com estereótipo *<<JavaClassFile>>*.
5. Para cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, são criados no pacote *Connector<target_component_name>.impl*: (1) uma interface *I<target_interface_required_name(n)>MgtHome* com estereótipo *<<EJBSessionHomeInterface>>* e valor sinalizado *EJBSessionType = Stateful*, (2) uma interface *I<target_interface_required_name(n)>MgtRemote* com estereótipo *<<EJBRemoteInterface>>*, (3) uma classe *<target_interface_required_name(n)>Mgt* com estereótipo *<<EJBImplementation>>*.
6. Para cada método de cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criado um método com assinatura equivalente na classe *<target_interface_required_name(n)>Mgt* do pacote *Connector<target_component_name>.impl* com estereótipo *<<EJBRemoteMethod>>*.
7. As conversões de tipo do PIM para o J2EE são feitas de acordo com a tabela de conversão de tipos do perfil *UML Profile for EJB* [UML4EJB].

Nas regras para mapeamento dos conectores cada classe *<<std comp spec>>* gera um *Session Bean*, sinalizado como *stateful*, ou seja, com informação de estado. Isto se deve ao fato de que as classes de implementação do *Session Bean*, ao herdarem de *ObjectManager*, precisam persistir *IComponentMgt* durante a sessão.

3.2.5 Perfil para Mapeamento dos Demais Elementos dos Modelos PIM para PSM J2EE

O perfil para mapeamento dos demais elementos dos modelos PIM para PSM J2EE inclui os estereótipos descritos na Tabela 1 ou na Tabela 2.

O perfil apresentado gera PSM J2EE anotados com estereótipos contidos no perfil *UML Profile for EJB* [UML4EJB].

As regras de mapeamento dos demais elementos são:

1. Para cada classe do PIM com estereótipo *<<comp spec>>* é criado um pacote com mesmo nome e com estereótipo *<<JavaArchiveFile>>*.
2. Para cada classe do PIM com estereótipo *<<comp spec>>* é criado no pacote com mesmo nome, uma classe com mesmo nome e com estereótipo *<<JavaClassFile>>*.
3. Para cada método de cada classe do PIM com estereótipo *<<comp spec>>* é criado no pacote com mesmo, na classe com mesmo nome, um método equivalente.
4. As conversões de tipo do PIM para o J2EE são feitas de acordo com a tabela de conversão de tipos do perfil *UML Profile for EJB* [UML4EJB].

3.3 Mapeamentos PIM para PSM .NET

Os mapeamentos do PIM para PSM .NET apresentados nesta seção são baseados no modelo COSMOS (seção 2.6) e na especificação .NET [DOTNET], e utilizados no estudo de caso apresentado no capítulo 4.. As regras apresentadas a seguir foram definidas visando (1) manter as diretrizes do modelo COSMOS, (2) permitir um mapeamento posterior para uma unidade de distribuição autônoma, e, portanto, coerente com o conceito de componente como unidade substituível, e (3) expor a menor quantidade possível de classes e interfaces, simplificando a utilização do componente.

As seções 3.3.1 e 3.3.2 apresentam, respectivamente, o Modelo de Componente COSMOS para .NET e o Modelo de Conector COSMOS para .NET. Em seguida, nas Seções 3.3.3, 3.3.4 e 3.3.5 são apresentados os perfis utilizados, criados a partir destes modelos, para mapear o PIM para PSM .NET.

3.3.1 Modelo de Componente COSMOS para .NET

O modelo de componente COSMOS para .NET é baseado no modelo COSMOS (seção 2.6) e na especificação .NET [DOTNET].

Atributos Utilizados

Alguns atributos dos componentes são utilizados na definição do modelo, a saber:

- *<component_name>*: nome do componente.
- *<interface_provided_name(n)>*: nome da enésima interface provida.
- *<interface_required_name(n)>*: nome da enésima interface requerida.

Grupos de elementos

O modelo apresenta a estrutura interna do componente definida a partir do modelo COSMOS (Figura 12). No modelo cada componente é composto por um conjunto de elementos. Estes elementos são classificados em seis grupos distintos:

1. Especificação das interfaces providas: interfaces definidoras das funcionalidades do componente. Estas interfaces estão armazenadas no pacote *<component_name>.spec.prov*.
2. Especificação das interfaces requeridas (opcional): interfaces definidoras das funcionalidades requeridas pelo componente. Estas interfaces estão armazenadas no pacote *<component_name>.spec.req*.
3. *Factory* do componente: classe que retorna uma instância do gerenciador do componente. Esta classe está armazenada no pacote *<component_name>.impl*.
4. Implementação do componente: classes e interfaces utilizadas para gerenciamento no nível do componente. Estes elementos estão armazenados no pacote *<component_name>.impl*.
5. Implementação das interfaces providas: classes que herdam da classe *ServicedComponent* (classe do *.NET Framework*, seção 2.1.2), obrigatória de acordo com a especificação .NET para utilização de recursos de componentes. Estas classes são responsáveis pelo gerenciamento no nível da interface provida e estão armazenadas no pacote *<component_name>.impl*.
6. Auxiliares (opcional): classes e interfaces auxiliares utilizadas pelas interfaces providas para implementar os métodos providos. Estes elementos, caso existam, devem ser

armazenados no pacote *<component_name>.impl*. Do ponto de vista prático, todas as classes e interfaces não classificadas anteriormente são elementos auxiliares.

No modelo original (Figura 12) os grupos *Factory* do componente, Implementação do componente, Implementação das interfaces providas e Auxiliares aparecem como Classes de implementação, mas neste modelo foram divididos por motivos didáticos.

Especificação das Interfaces Providas

Os elementos da especificação das interfaces providas são:

- *I<interface_provided_name(n)>*: Uma especificação dos métodos providos para cada interface provida. Estas interfaces têm visibilidade pública.

Especificação das Interfaces Requeridas

Os elementos da especificação das interfaces requeridas são:

- *I<interface_required_name(n)>*: Uma especificação dos métodos providos para cada interface requerida. Estas interfaces têm visibilidade pública.

Factory do Componente

O *factory* do componente é formado por:

- *ComponentFactory*. Uma classe para cada componente, que implementa apenas o método *CreateInstance()* do tipo *static*, sem parâmetros e com retorno do tipo *IManager*. Esta classe tem visibilidade pública.

Implementação do Componente

Os elementos da implementação do componente são:

- *IManager*: Uma interface para cada componente, que define as operações de manipulação das interfaces providas pelo componente. Estas operações são *GetProvidedInterface()*, *GetProvidedInterfaces()*, *SetRequiredInterface()*, *GetRequiredInterface()*, *GetRequiredInterfaces()*, conforme o modelo COSMOS. Esta interface tem visibilidade pública.
- *IComponentMgt*: Uma interface para cada componente, que estende a interface *IManager*, acrescentando o método *GetObjectFactory()*, responsável pela criação de instâncias para as interfaces internas utilizadas pelo componente. Como o método *GetObjectFactory()* não precisa e, preferencialmente, não deve ser visualizado externamente, este método é incorporado nesta interface, que tem visibilidade de pacote. Este método retorna uma

referência para uma *factory* de criação dos objetos utilizados internamente pelo componente.

- *ComponentMgt*: Uma classe para cada componente, que implementa a interface *IComponentMgt*, e por extensão a interface *IManager*. Esta classe contém uma referência para uma *factory* que é retornada através do método *GetObjectFactory()* e também listas com os nomes e objetos correspondentes às interfaces providas e requeridas. Esta classe tem visibilidade de pacote.
- *IObjectFactory*: Uma interface para cada componente, que define os métodos para criação dos objetos utilizados internamente pelo componente. Esta interface tem visibilidade de pacote.
- *ObjectFactory*: Uma classe para cada componente. Esta classe implementa os métodos definidos em *IObjectFactory*. Esta classe tem visibilidade de pacote.

Implementação das Interfaces Providas

Os elementos da implementação das interfaces providas são:

- *<interface_provided_name(n)>Mgt*: Uma classe para cada interface provida. Esta classe (1) implementa a interface *I<interface_provided_name(n)>Mgt* e (2) estende a classe *ServicedComponent*, e (3) inclui um atributo do tipo *IComponentMgt*, e um conjunto de operações *Get()/Set()* para este atributo, que servem como referência da interface para o gerenciador do componente. Para evitar que o cliente do componente tenha acesso ao gerenciador do componente o conjunto de operações *Get()/Set()* é definido como *internal*.

Comentários

O modelo apresentado minimiza referências diretas entre classes, fazendo com que os elementos internos dos componentes utilizem interfaces e obtenham as instâncias destas através do gerenciador. Conforme mencionado anteriormente, os componentes na plataforma .NET devem herdar da classe *ServicedComponent*. No entanto, existe uma restrição de que uma classe que herde da classe *ServicedComponent* não pode possuir construtores parametrizados, portanto, é necessário criar um conjunto de operações *Get()/Set()* para acessar o gerenciador do componente. Desta forma, o componente acessa os construtores e as interfaces requeridas através do gerenciador.

3.3.2 Modelo de Conector COSMOS para .NET

A estrutura do modelo de conector COSMOS para .NET é próxima a do modelo de componentes. No entanto, enquanto um componente PSM surge a partir de um componente PIM, um conector PSM, que não existe no modelo PIM, surge a partir da interação entre dois ou mais componentes PIM. O componente especifica as interfaces providas enquanto as interfaces providas por um conector são especificadas no componente a quem o conector que se destina. Da mesma forma as interfaces requeridas por um conector não são especificadas em si, como nos componentes, mas em componentes PIM que provêem as interfaces.

O modelo de conector COSMOS para .NET é baseado no modelo COSMOS (seção 2.6) e na especificação .NET [DOTNET].

Atributos Utilizados

Alguns atributos dos componentes são utilizados na definição do modelo, a saber:

- *<target_component_name>*: nome do componente cuja interface requerida é suprida pelo conector;
- *<target_interface_required_name(n)>*: nome da enésima interface requerida pelo componente a quem o conector se destina.

Grupos de Elementos

O modelo apresenta a estrutura interna do conector definida a partir do modelo COSMOS. No modelo cada conector é composto por um conjunto de elementos. Estes elementos são classificados em quatro grupos distintos:

1. *Factory* do conector: classe que retorna uma instância do gerenciador do conector, e que serve como ponto de entrada para utilização do conector. Esta classe está armazenada em *Connector<target_component_name>.impl*.
2. Implementação do conector: classes e interfaces utilizadas para gerenciamento no nível do conector. Estes elementos estão armazenados em *Connector<target_component_name>.impl*.
3. Implementação das interfaces providas: classes que herdam da classe *ServicedComponent* (classe do *.NET Framework*, seção 2.1.2), obrigatória de acordo com a especificação .NET para utilização de recursos de componentes. Estas classes são responsáveis pelo gerenciamento no nível da interface provida e estão armazenados em

Connector<target_component_name>.impl.

4. Auxiliares (opcional): classes e interfaces auxiliares utilizadas pelas interfaces providas para implementar os métodos providos. Estes elementos, caso existam, devem ser armazenados em *Connector<component_name>.impl.* Do ponto de vista prático, todas as classes e interfaces não classificadas anteriormente são elementos auxiliares.

Factory do Conector

O *factory* do conector é formado por:

- *ComponentFactory*. Uma classe para cada componente, que implementa apenas o método *CreateInstance()* do tipo *static*, sem parâmetros e com retorno do tipo *IManager*. Esta classe tem visibilidade pública.

Implementação do Conector

Os elementos da implementação do conector são:

- *IManager*: Uma interface para cada conector, que define as operações de manipulação das interfaces providas pelo conector. Estas operações são *GetProvidedInterface()*, *GetProvidedInterfaces()*, *SetRequiredInterface()*, *GetRequiredInterface()*, *GetRequiredInterfaces()*, conforme o modelo COSMOS. Esta interface tem visibilidade pública.
- *IComponentMgt*: Uma interface para cada conector, que estende a interface *IManager*, acrescentando o método *GetObjectFactory()*, responsável pela criação de instâncias para as interfaces internas utilizadas pelo conector. Como o método *GetObjectFactory()* não precisa e, preferencialmente, não deve ser visualizado externamente, este método é incorporado nesta interface, que tem visibilidade de pacote. Este método retorna uma referência para uma *factory* de criação dos objetos utilizados internamente pelo componente.
- *ComponentMgt*: Uma classe para cada conector, que implementa a interface *IComponentMgt*, e por extensão a interface *IManager*. Esta classe contém uma referência para uma *factory* que é retornada através do método *GetObjectFactory()* e também listas com os nomes e objetos correspondentes às interfaces providas e requeridas. Esta classe tem visibilidade de pacote.
- *IObjectFactory*: Uma interface para cada conector, que define os métodos para criação

dos objetos utilizados internamente pelo conector. Esta interface tem visibilidade de pacote. Como esta interface é utilizada como parâmetro no método *GetObjectFactory()* da classe *ComponentMgt* ela precisa estender a interface *Serializable*.

- *ObjectFactory*: Uma classe para cada conector. Esta classe implementa os métodos definidos em *IObjectFactory*. Esta classe tem visibilidade de pacote.

Implementação das Interfaces Providas

Os elementos da implementação das interfaces providas são:

- *<target_interface_required_name(n)>Mgt*: Uma classe para cada interface provida. Esta classe (1) implementa a interface *I<target_interface_required_name(n)>Mgt* e (2) estende a classe *ServicedComponent*, e (3) inclui um atributo do tipo *IComponentMgt*, e um conjunto de operações *Get()/Set()* para este atributo, que servem como referência da interface para o gerenciador do conector. Para evitar que o cliente do componente tenha acesso ao gerenciador do conector o conjunto de operações *Get()/Set()* é definido como *internal*.

Comentários

O modelo apresentado minimiza referências diretas entre classes, fazendo com que os elementos internos dos conectores utilizem interfaces e obtenham as instâncias destas através do gerenciador. Conforme mencionado anteriormente, os componentes na plataforma .NET devem herdar da classe *ServicedComponent*. No entanto, existe uma restrição de que uma classe que herde da classe *ServicedComponent* não pode possuir construtores parametrizados, portanto, é necessário criar um conjunto de operações *Get()/Set()* para acessar o gerenciador do conector. Desta forma, o conector acessa os construtores e as interfaces requeridas através do gerenciador.

3.3.3 Perfil para Mapeamento de Componentes dos Modelos PIM para PSM .NET

O perfil para mapeamento de componentes dos modelos PIM para PSM .NET inclui os estereótipos descritos na Tabela 1 ou na Tabela 2.

Como não existe um perfil para mapeamento para C#, semelhante ao *UML Profile for EJB* [UML4EJB], são criados, de forma simplificada, alguns estereótipos adicionais para anotar os PSM gerados, conforme descrito na tabela 3.

Estereótipo	Aplica-se a	Definição
<<C#Interface>>	*class*	Indica que o elemento *class* representa uma interface C#.
<<C#Class>>	*class*	Indica que um elemento do tipo *class* representa uma classe C#.
<<C#Component>>	*class*	Indica que o elemento do tipo *class* representa uma classe C# que herda da classe *ServicedComponent*.
<<C#dll>>	*package*	Indica que o elemento do tipo *package* representa uma biblioteca dinâmica de ligação C#.

Tabela 3 – Estereótipos para definição do perfil UML para .NET

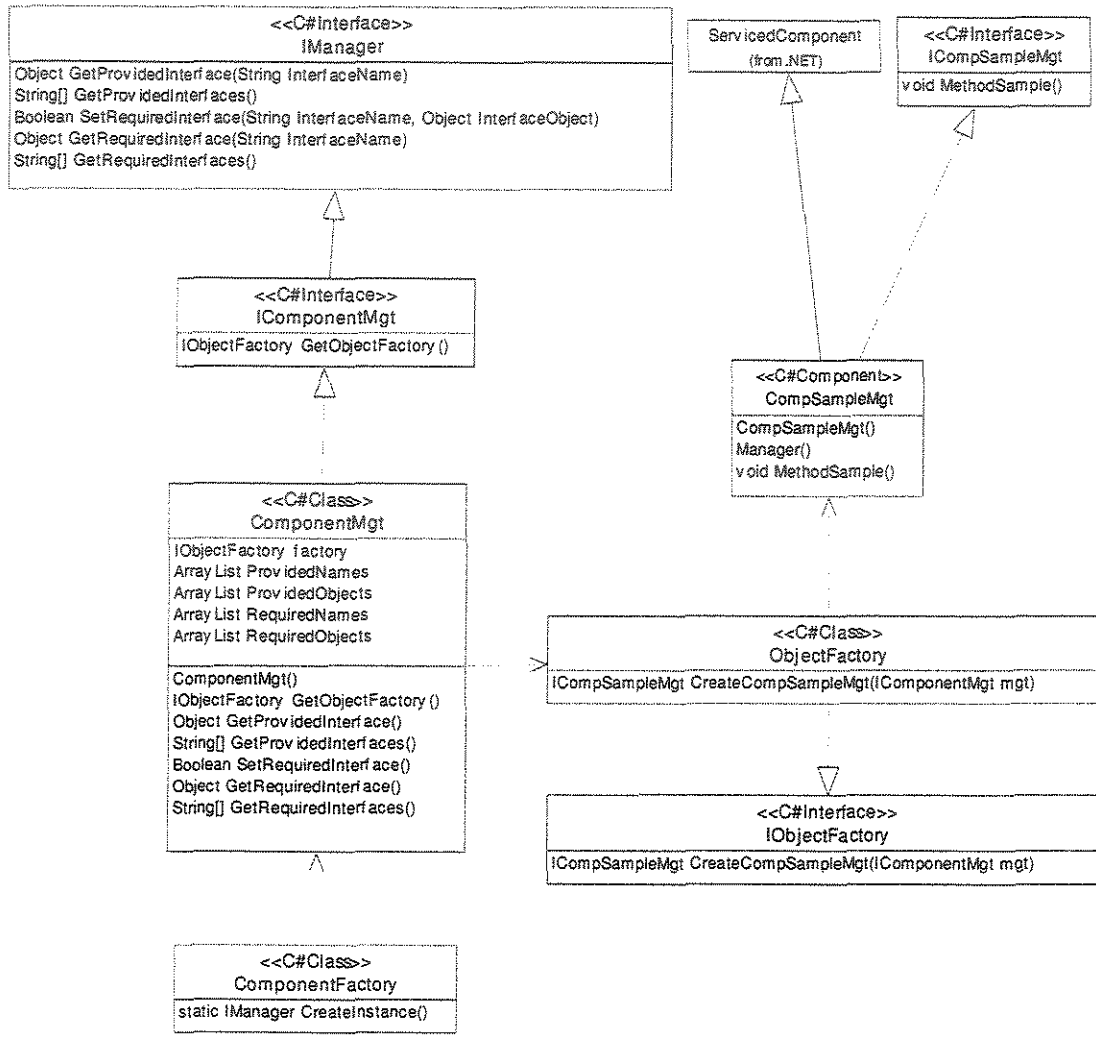

Figura 28 – Exemplo de interface e componente PSM .NET

O modelo PSM .NET (Figura 28), gerado a partir do modelo PIM (Figura 25), é um exemplo de aplicação das regras de mapeamento de componentes. A interface *ICompSampleMgt* da Figura 25 dá origem à interface *ICompSampleMgt* da Figura 28, a classe *CompSampleMgt* dá origem aos demais elementos. As regras utilizadas nestas transformações são descritas a seguir:

1. Para cada classe do PIM com estereótipo *<<std comp spec>>* é criado um pacote *<component_name>* com estereótipo *<<C#dll>>* e este é o pacote raiz para todos os

elementos gerados a partir desta classe.

2. Para cada classe do PIM com estereótipo *<<std comp spec>>* são criados: (1) um pacote *<component_name>.spec.prov*, (2) um pacote *<component_name>.impl*.
3. Para cada classe do PIM com estereótipo *<<std comp spec>>*, que apresenta uma ou mais interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um pacote *<component_name>.spec.req*.
4. Para cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criada uma interface *I<interface_provided_name(n)>* com estereótipo *<<C#Interface>>* no pacote *<component_name>.spec.prov*.
5. Para cada método de cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criado um método com assinatura equivalente na interface *I<interface_provided_name(n)>* do pacote *<component_name>.spec.prov*.
6. Para cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criada uma interface *I<interface_required_name(n)>* com estereótipo *<<C#Interface>>* no pacote *<component_name>.spec.req*.
7. Para cada método de cada interface requerida de cada classe do PIM com estereótipo *<<std comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criado um método com assinatura equivalente na interface *I<interface_required_name(n)>* do pacote *<component_name>.spec.req*.
8. Para cada classe do PIM com estereótipo *<<std comp spec>>* são criados no pacote *<component_name>.impl*: (1) uma classe *ComponentFactory* com estereótipo *<<C#Class>>*, (2) uma interface *IManager* com estereótipo *<<C#Interface>>*, (3) uma interface *IComponentMgt* com estereótipo *<<C#Interface>>*, (4) uma classe *ComponentMgt* com estereótipo *<<C#Class>>*, (5) uma interface *IObjectFactory* com estereótipo *<<C#Interface>>*, (6) uma classe *ObjectFactory* com estereótipo *<<C#Class>>*.
9. Para cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criada no pacote *<component_name>.impl* uma classe *<interface_provided_name(n)>Mgt* com estereótipo *<<C#Component>>*.

10. Para cada método de cada interface provida de cada classe do PIM com estereótipo *<<std comp spec>>* é criado um método com assinatura equivalente na classe *<interface_required_name(n)>Mgt* do pacote *<component_name>.impl.*
11. As conversões de tipo do PIM para o .NET devem ser feitas de acordo com uma tabela de conversão de tipos.

3.3.4 Perfil para Mapeamento de Conectores dos Modelos PIM para PSM .NET

O perfil para mapeamento de conectores dos modelos PIM para PSM .NET inclui os estereótipos descritos na Tabela 1 ou na Tabela 2 ou na Tabela 3.

As regras de mapeamento de conectores são:

1. Para cada classe do PIM com estereótipo *<<comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um pacote *Connector<target_component_name>* com estereótipo *<<C#dll>>* e este é o pacote raiz para todos os elementos gerados a partir desta classe.
2. Para cada classe do PIM com estereótipo *<<comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, é criado um pacote *Connector<target_component_name>.impl.*
3. Para cada classe do PIM com estereótipo *<<comp spec>>*, que apresenta interfaces requeridas oriundas de interface ligada a uma classe com o mesmo estereótipo, são criados no pacote *Connector<component_name>.impl*: (1) uma classe *ComponentFactory* com estereótipo *<<C#Class>>*, (2) uma interface *IManager* com estereótipo *<<C#Interface>>*, (3) uma interface *IComponentMgt* com estereótipo *<<C#Interface>>*, (4) uma classe *ComponentMgt* com estereótipo *<<C#Class>>*, (5) uma interface *IObjectFactory* com estereótipo *<<C#Interface>>*, (6) uma classe *ObjectFactory* com estereótipo *<<C#Class>>*.
4. Para cada interface requerida de cada classe do PIM com estereótipo *<<comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criada no pacote *Connector<target_component_name>.impl* uma classe

<target_interface_required_name(n)>Mgt com estereótipo *<<C#Class>>*.

5. Para cada método de cada interface requerida de cada classe do PIM com estereótipo *<<comp spec>>*, oriunda de interface ligada a uma classe com o mesmo estereótipo, é criado um método com assinatura equivalente na classe *<target_interface_required_name(n)>Mgt* do pacote *Connector<target_component_name>.impl*.
6. As conversões de tipo do PIM para o .NET devem ser feitas de acordo com uma tabela de conversão de tipos.

3.3.5 Perfil para Mapeamento dos Demais Elementos dos Modelos PIM para PSM .NET

O perfil para mapeamento dos demais elementos dos modelos PIM para PSM .NET inclui os estereótipos descritos na Tabela 1 ou na Tabela 2 ou na Tabela 3.

As regras de mapeamento dos demais elementos são:

1. Para cada classe do PIM com estereótipo *<<comp spec>>* é criado um pacote com mesmo nome.
2. Para cada classe do PIM com estereótipo *<<comp spec>>* é criado no pacote com mesmo nome, uma classe com mesmo nome e com estereótipo *<<C#Class>>*.
3. Para cada método de cada classe do PIM com estereótipo *<<comp spec>>* é criado no pacote com mesmo, na classe com mesmo nome, um método equivalente.
4. As conversões de tipo do PIM para o .NET devem ser feitas de acordo com uma tabela de conversão de tipos.

3.4 Mapeamento do PSM para Implementação

Os modelos PSM e a implementação apresentam um grau de abstração bastante próximo, conforme comentado na seção 3.1.5, portanto o mapeamento do PSM para implementação implica a criação no sistema de arquivos de artefatos correspondentes aos pacotes (diretórios), classes (arquivos) e interfaces (arquivos) que possam ser interpretados por uma ferramenta ou ambiente de desenvolvimento.

3.4.1 Mapeamento do PSM J2EE para Implementação

O mapeamento do PSM J2EE para implementação é feito através da aplicação do perfil *UML Profile for EJB* [UML4EJB].

3.4.2 Mapeamento do PSM .NET para Implementação

Como não existe um perfil semelhante ao *UML Profile for EJB* [UML4EJB] são definidas alguns regras simples para mapeamento do PSM .NET para código fonte .NET. São elas:

- Para cada pacote com estereótipo *<<C#dll>>* é gerada um projeto .NET, especificando uma biblioteca de classes.
- Para cada interface do PSM .NET com estereótipo *<<C#Interface>>* é criado um arquivo contendo uma interface C#.
- Para cada classe do PSM .NET com estereótipo *<<C#Interface>>* é criado um arquivo contendo uma classe C#.
- Para cada assinatura de método de cada interface do PSM .NET com estereótipo *<<C#Interface>>* é criada uma assinatura de método equivalente na interface C# correspondente.
- Para cada método de cada classe do PSM .NET com estereótipo *<<C#Class>>* é criado um método equivalente na classe C# correspondente.

Capítulo 4

Estudo de Caso: Um Sistema de Contingências Tributárias

Este capítulo apresenta um estudo de caso no qual foi utilizado o processo adaptado apresentado no capítulo 3. O estudo de caso escolhido é um Sistema de Contingências Tributárias que se enquadra na categoria de sistema de negócio [PRESSMAN01] e é baseado num sistema real. Existe uma versão anterior do mesmo sistema sendo utilizada por uma grande empresa. A versão anterior foi desenvolvida de acordo com o padrão arquitetural de cliente/servidor, utilizando um banco de dados centralizado. Nesta versão a camada do cliente foi desenvolvida utilizando a linguagem *Visual Basic 6.0* e a camada do servidor foi desenvolvida utilizando o banco de dados *Sybase 11.5*.

Na versão desenvolvida neste estudo de caso foram criados todos os modelos do sistema e implementado integralmente um caso de uso para cada uma das plataformas alvo escolhidos, J2EE e .NET, desde a camada de apresentação até a camada de persistência.

A seção 4.1 descreve o Sistema de Contingências Tributárias, escolhido como estudo de caso. Na seção 4.2 são descritos os estágios do fluxo de trabalho *Definição de Requisitos*. Na seção 4.3 são descritos os estágios do fluxo de trabalho *Especificação* 4.4. Na seção 4.5 são descritos os estágios do fluxo de trabalho *Montagem*. Finalmente na seção 4.6 são apresentados alguns resultados obtidos em função da aplicação do processo.

4.1 Descrição do Sistema de Contingências Tributárias

O sistema utilizado no estudo de caso é o Sistema de Contingências Tributárias. Contingência é o termo utilizado para referir-se aos valores relacionados com processos judiciais. O sistema em questão trata de processos judiciais da área tributária e das contingências a eles associados. Nos processos em geral, e nos tributários em particular, o controle dos valores envolvidos é particularmente importante para as empresas, pois repercutem diretamente nos demonstrativos financeiros, e na apuração do resultado. O objetivo do sistema é criar uma base de dados dos processos tributários existentes, incluindo o acompanhamento histórico do processo e

dos valores associados, e efetuando a atualização monetária e o cálculo de juros destes valores, bem como a demonstração destes valores através de relatórios. Dada a confidencialidade das informações a questão da segurança no acesso é particularmente importante. Foram feitas algumas simplificações para que este trabalho pudesse comportar o estudo de caso.

4.2 Fluxo de Trabalho Definição de Requisitos

4.2.1 Estágio Processo do Negócio

Para entender o processo associado ao sistema foi elaborado o diagrama de processo do negócio (Figura 29), conforme prescrito no processo *UML Components* (seção 2.4.3).

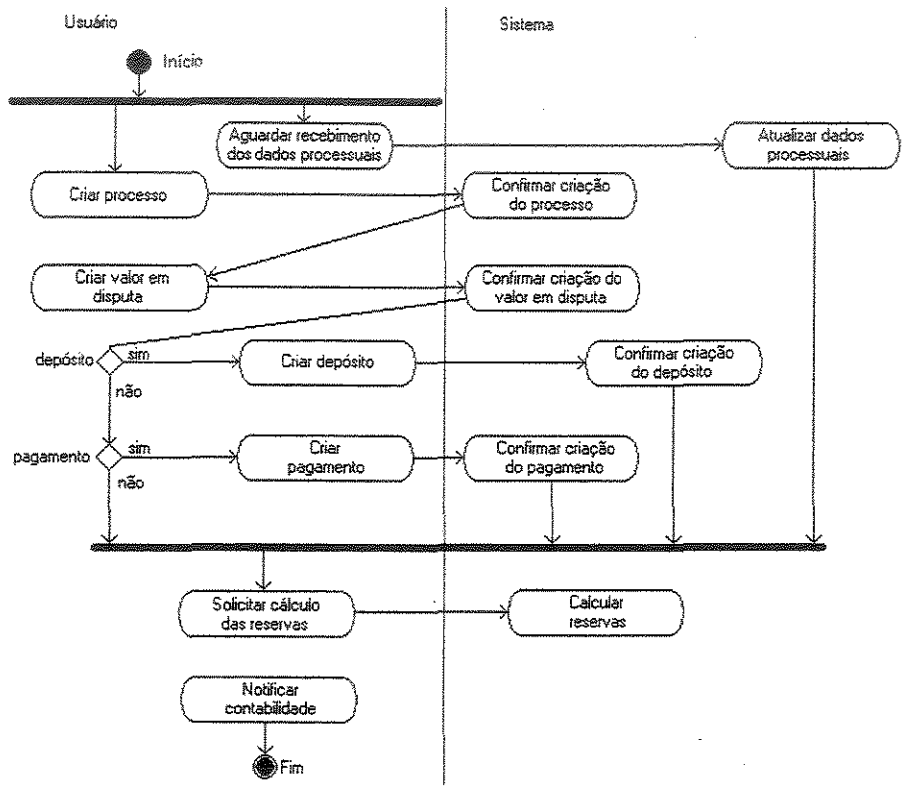

Figura 29 - Diagrama de atividades do processo de negócios

Cada vez que é aberto um novo processo, seja por iniciativa da empresa, ou por iniciativa de um oponente (União Federal, Estados ou Municípios) é cadastrado um novo processo no sistema. Juntamente com o processo são cadastrados os valores de causa, ou seja, o montante em

disputa. Caso os valores de causa, em função de decisão judicial, tenham sido depositados em juízo, esta informação deve ser cadastrada no sistema. Caso os valores de causa, em função de decisão judicial, tenham sido pagos, esta informação deve ser cadastrada no sistema. O sistema recebe, ainda, periodicamente dos escritórios responsáveis as informações do acompanhamento processual, como por exemplo os despachos, deferimentos, e outras medidas ou pareceres relacionados ao processo, que são atualizados na base de dados. Finalmente, com base nos dados processuais e nos valores de causa, o sistema calcula o valor das reservas contábeis, incorporando atualização monetária e juros, que devem ser lançados nos demonstrativos contábeis da empresa.

No sistema em estudo não existem interfaces com sistemas existentes.

4.2.2 Estágio Previsão do Sistema

A previsão do sistema, conforme prescrito no processo *UML Components* (seção 2.4.3), foi feita com base em entrevistas e foi transcrita a seguir:

Um sistema de controle de contingências tributárias é necessário para controlar os processos tributários, tanto no aspecto do acompanhamento processual, quanto no aspecto dos valores associados. O sistema deverá ser alimentado por informações processuais oriundas dos escritórios responsáveis pelos processos, por informações extraídas dos processos administrativos ou judiciais referentes a valores de causa, e pelas informações sobre pagamentos e depósitos ocorridos durante o processo. O sistema será responsável por armazenar e recuperar informações deste banco de dados de processos tributários, bem como calcular os valores a serem reportados à contabilidade como reserva legal contábil referente a cada processo.

4.2.3 Estágio Modelo Conceitual do Negócio

A partir de entrevistas com os usuários foram identificados os principais elementos que compõem o domínio da aplicação, bem como as relações entre estes elementos. Estes elementos e relações foram sintetizados no *Modelo Conceitual do Negócio* (Figura 30), conforme prescrito no processo *UML Components* (seção 2.4.3).

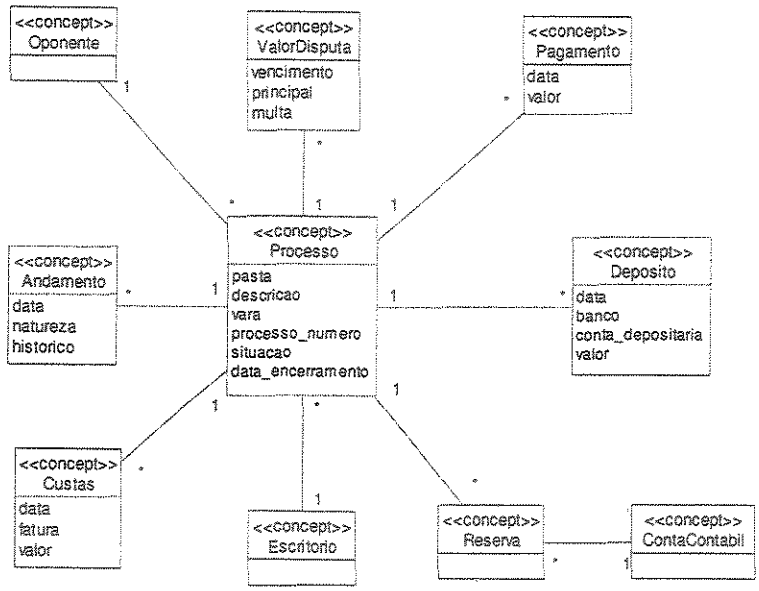

Figura 30 - Modelo Conceitual do Negócio

Complementarmente foram especificados todos os itens constantes no *Modelo Conceitual do Negócio*, de tal forma a criar um vocabulário do sistema, que possa ser entendido pelos usuários e pelos desenvolvedores. Este vocabulário é apresentado abaixo:

- Andamento: Evento registrado no decorrer no processo.
- Banco: Instituição bancária onde pode ser feito depósito.
- ContaContabil: Conta do plano de contas onde são registrados os valores de reservas relativos a um ou mais processos.
- conta_depositaria: Número da conta bancária onde foi efetuado o depósito.
- Custas: Pagamentos efetuados ao escritório responsável pelo caso em função das custas processuais.
- data_encerramento: Data de encerramento do processo.
- Data de início do processo: Data do primeiro andamento relativo ao processo.
- Deposito: Valores dos depósitos em juízo ocorridos durante um processo.
- Escritorio: Escritório advocatício responsável pelo processo.
- Fatura: Número da fatura emitida pelo escritório em decorrência das custas processuais.

- Natureza: Natureza da ação (por exemplo: mandado de segurança, liminar, agravo de instrumento, embargos de execução, etc).
- Oponente: Pessoa física ou jurídica que interpõe oposição em juízo (autor) ou contra a qual é interposta oposição em juízo (réu).
- Pagamento: Valores dos pagamentos ocorridos durante um processo. Os pagamentos correspondem aos desembolsos de caixa efetuados diretamente ao oponente, e são tratados de forma distinta dos depósitos em juízo, que são tratados no modelo como Deposito.
- Pasta: Número da pasta física onde estão arquivados os documentos relativos ao processo.
- Processo: Processo é um conjunto de peças que servem à instrução do juízo. Por processo aqui se entende a transcrição de algumas informações significativas deste conjunto que o identificam, classificam e descrevem.
- processo_numero: Número que identifica o processo junto à vara por onde o mesmo tramita.
- Reserva: Reserva é um valor calculado com base nos valores em disputa, pagamentos e depósitos e que é registrado na contabilidade como provisão.
- Situacao: Situação do processo, que pode ser *ativo* ou *inativo*.
- Uf: Unidade da federação.
- ValorDisputa: Valores em disputa registrados para um processo.
- Vara: Vara por onde tramita o processo.

4.2.4 Estágio Modelo de Casos de Uso

As informações a serem tratadas pelo sistema foram divididas em dois grupos distintos: (1) informações sobre o andamento do processo, e (2) informações sobre os valores associados. A partir destes grupos de informações foram definidos grupos distintos de usuários: (1) advogados, responsáveis pelo cadastramento e andamento do processo; (2) analistas financeiros, responsáveis pelo cadastramento e atualização dos valores. A cada grupo distinto de usuários corresponde um ator nos casos de uso.

Conforme prescrito no processo *UML Components* (seção 2.4), a partir do *Modelo Conceitual do Negócio* foram construídas a tabela de verificação de criação/destruição (Tabela 4)

e a tabela de verificação de atualização de associação (Tabela 5).

Classe	Comentário
Andamento	Andamentos são criados e removidos somente através da atualização dos dados processuais enviados pelo escritório, portanto, não serão necessários casos de uso para estes eventos.
ContaContabil	Contas contábeis são criadas e removidas, portanto, serão necessários casos de uso para estes eventos.
Custas	As custas são criadas e removidas somente através da atualização dos dados processuais enviados pelo escritório, portanto, não serão necessários casos de uso para estes eventos.
Deposito	Depósitos são criados e removidos, portanto, serão necessários casos de uso para estes eventos.
Escritorio	Escritórios são criados e removidos, portanto, serão necessários casos de uso para estes eventos.
Oponente	Oponentes podem ser criados e removidos, portanto, serão necessários casos de uso para estes eventos.
Processo	Processos podem ser criados, removidos ou desativados (processo encerrado), portanto, serão necessários casos de uso para estes eventos.
Pagamento	Pagamentos podem ser criados e removidos, portanto, serão necessários casos de uso para estes eventos.
Reserva	As reservas são calculadas automaticamente pelo sistema, portanto, não serão necessários casos de uso para estes eventos.
ValorDisputa	Valores em disputa são criados e removidos, portanto, serão necessários casos de uso para estes eventos.

Tabela 4 – Tabela de verificação de criação/destruição

Tabelas relacionadas	Comentário
Processo-Andamento	Somente será alterada através da atualização dos

	dados processuais enviados pelo escritório, portanto, não são necessários casos de uso para este evento.
Processo-Custas	Somente será alterada através da atualização dos dados processuais enviados pelo escritório, portanto, não são necessários casos de uso para este evento.
Processo-Deposito	Somente será alterada como parte da alteração do depósito, portanto, não são necessários casos de uso para este evento.
Processo-Escritorio	Somente será alterada como parte da alteração do Processo, portanto, não são necessários casos de uso para este evento.
Processo-Oponente	Somente será alterada como parte da alteração do Processo, portanto, não são necessários casos de uso para este evento.
Processo-Pagamento	Somente será alterada como parte da alteração do pagamento, portanto, não são necessários casos de uso para este evento.
Processo-Reserva	Somente será alterada como parte da alteração do processo, portanto, não são necessários casos de uso para este evento.
Processo-ValorDisputa	Somente será alterada como parte da alteração do valor em disputa, portanto, não são necessários casos de uso para este evento.
Reserva-ContaContabil	Somente será alterada como parte da alteração do processo, portanto, não são necessários casos de uso para este evento.

Tabela 5 – Tabela de verificação de atualização de associação

A partir da tabela de verificação de criação/destruição, da tabela de verificação de atualização de associação e do diagrama de atividade do processo foram definidos os casos de uso do sistema, conforme relação que segue:

- Criar / alterar / apagar / desativar processo.
- Criar / alterar / apagar valor em disputa.
- Criar / alterar / apagar depósito.
- Criar / alterar / apagar pagamento.
- Atualizar dados enviados pelo escritório.
- Calcular reservas.
- Criar / alterar / apagar conta contábil.
- Criar / alterar / apagar escritório.
- Criar / alterar / apagar oponente.

A seguir aparece, como ilustração, a descrição de um dos casos de uso do sistema.

Caso de uso 6.

Nome: **Calcular reservas**

Iniciador: **Financeiro**

Objetivo: **Calcular o valor das reservas e notificar a contabilidade.**

Cenário principal de sucesso

1. Financeiro solicita Calcular reservas.
2. Sistema solicita Data de contabilização (formato mm/aaaa).
3. Financeiro informa a data de contabilização.
4. Sistema calcula os valores das reservas de cada processo com base no último dia do mês/ano informado. Os valores calculados são armazenados, de tal forma que, mesmo que os valores em disputa, pagamentos, ou depósitos sejam alterados, os valores das reservas contabilizados permaneçam os mesmos.
5. Sistema imprime relatório, com quebra por número de conta contábil, contendo as

seguintes informações:

- Data digitada (cabeçalho);
- Número da conta contábil (cabeçalho);
- Pasta;
- Descrição do processo;
- Valor da reserva (com totalizador por conta contábil e geral).

4.2.5 Estágio Definição de Requisitos Não-Funcionais

Foram relacionados os requisitos não-funcionais que dizem respeito ao sistema como um todo, conforme prescrito no processo adaptado (seção 3.1.1), são eles:

Identificação	Descrição	Critérios
Controle de acesso	Habilidade do sistema em permitir que somente usuários autorizados tenham acesso ao sistema.	Somente os usuários cadastrados previamente poderão acessar o sistema, o que deve ser feito informando a identificação do usuário e a senha correspondente. Cada usuário somente poderá executar as funções permitidas para o nível de acesso a ele associado.
Portabilidade	Refere-se a facilidade com que é possível migrar o sistema para outros sistemas operacionais, bancos de dados, *middleware*, etc.	Mudanças de sistemas operacionais e bancos de dados devem poder ser implementadas com impacto reduzido no sistema.
Disponibilidade	Refere-se a falhas no sistema e conseqüências associadas.	O sistema deve estar disponível 24 horas por dia, 7 dias por semana.
Escalabilidade	Refere-se a capacidade do sistema de aumentar a sua capacidade de atendimento	O sistema deve ser capaz de atender um aumento de demanda de usuários, sem que seja necessário alterações no código fonte,

	sem grandes alterações no sistema.	somente através da reconfiguração da implantação.

Tabela 6 – Tabela de verificação de atualização de associação

4.3 Fluxo de Trabalho Especificação

4.3.1 Estágio Identificação de Componentes

O partir dos casos de uso foram identificados foram identificadas as interfaces do sistema e as operações destas interfaces, a partir de diagramas (por exemplo, Figura 31) que ilustram a correspondência entre o caso de uso e a interface do componente de serviço de sistema, conforme prescrito no processo *UML Components* (seção 2.4).

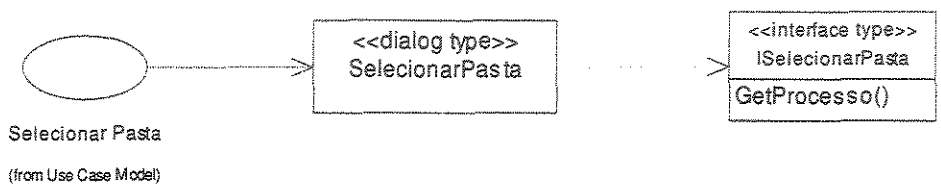

Figura 31 - Diagrama de correspondência caso de uso e interface de componente de serviço de sistema

De forma análoga foram analisados os demais casos de uso e identificadas as interfaces e operações correspondentes.

O Modelo Conceitual do Negócio foi refinado e gerou o Modelo de Tipos do Negócio. Este modelo apresenta um maior nível de detalhe, que pode ser percebido na definição dos atributos das entidades, bem como na inclusão de restrições, que aparecem no diagrama na forma de anotações (Figura 32).

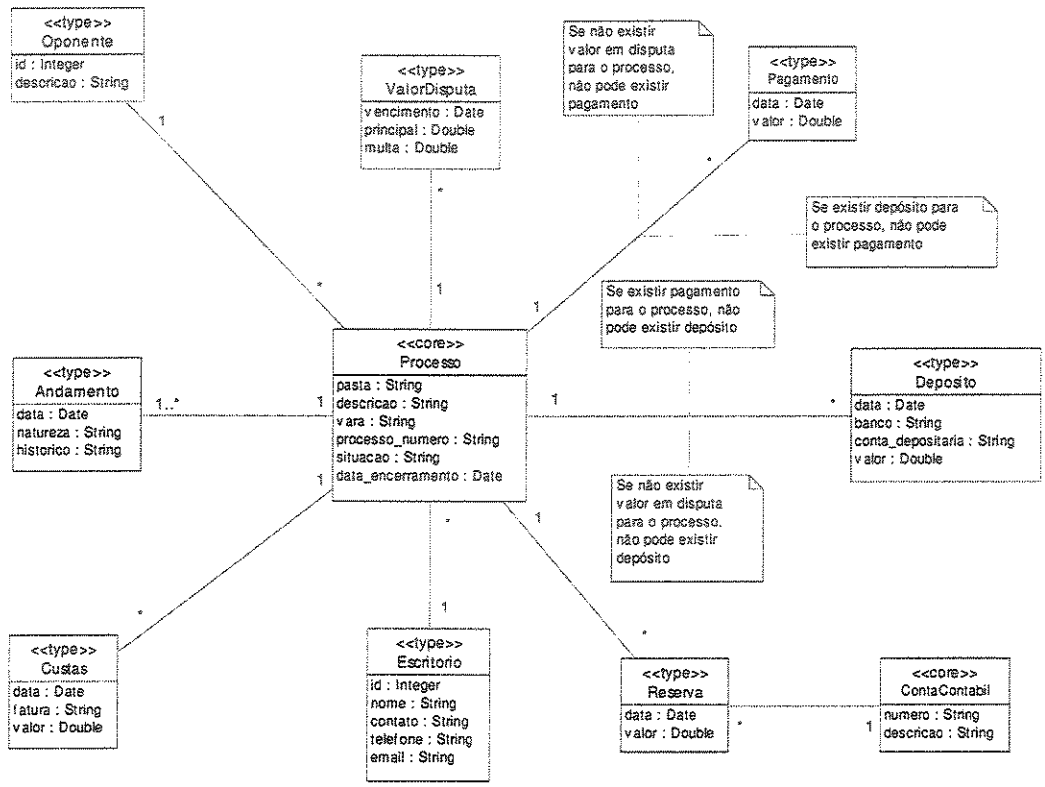

Figura 32 - Modelo de Tipos do Negócio

A partir do Modelo de Tipos do Negócio foram identificados os tipos centrais.

Os tipos identificados foram o Processo e a ContaContabil, pois ambos são entidades identificadoras do negócio e não possuem relação obrigatória com outras entidades, exceto as entidades categorizadoras.

A partir da identificação dos tipos centrais foi elaborado um diagrama de responsabilidade de interface que destaca os tipos centrais e ilustra a relação de dependência dos demais tipos em relação a estes (Figura 33).

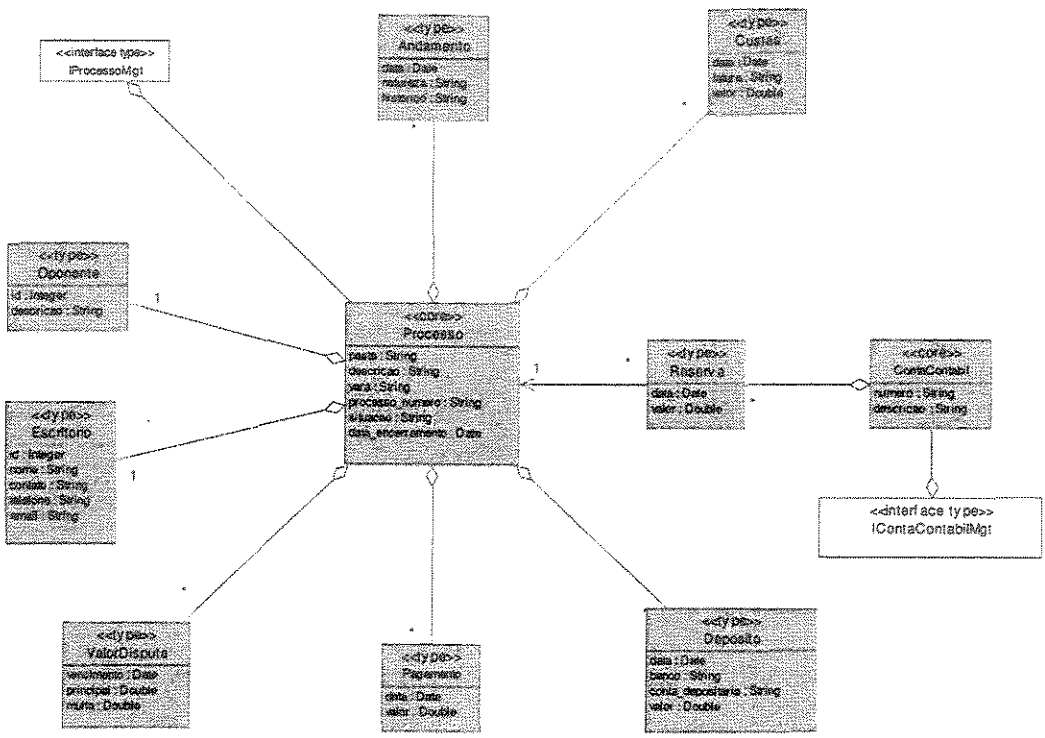

Figura 33 - Diagrama de responsabilidade de interface

A partir das interfaces do sistema e dos tipos centrais foi definida uma proposta inicial de arquitetura de componentes (Figura 34).

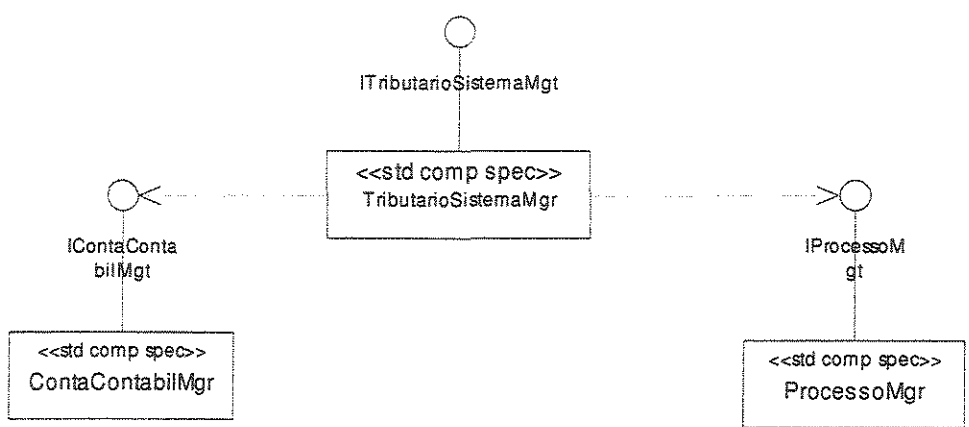

Figura 34 - Especificação inicial da arquitetura de componentes

As interfaces dos serviços do sistema foram unificadas em uma única interface. As razões para tanto foram: (1) os conceitos representados pelas diferentes interfaces têm o mesmo tempo de vida, em função da elevada sobreposição dos modelos de informação das interfaces; (2) como não se trata de um sistema muito grande decidiu-se tornar a interface do sistema como um único componente.

As interfaces dos serviços de negócio foram definidas a partir dos tipos centrais anteriormente identificados.

4.3.2 Estágio Interação de Componentes

A partir da interação entre os componentes de sistema e os componentes de negócio, representados na especificação inicial da arquitetura de componentes, foram identificadas as interações possíveis entre os mesmos e as operações necessárias nas interfaces dos componentes de negócio, conforme prescritos no processo *UML Components* (seção 2.4). Este processo foi realizado através da análise dos diagramas de colaboração, para cada um dos métodos dos componentes do sistema (Figura 35).

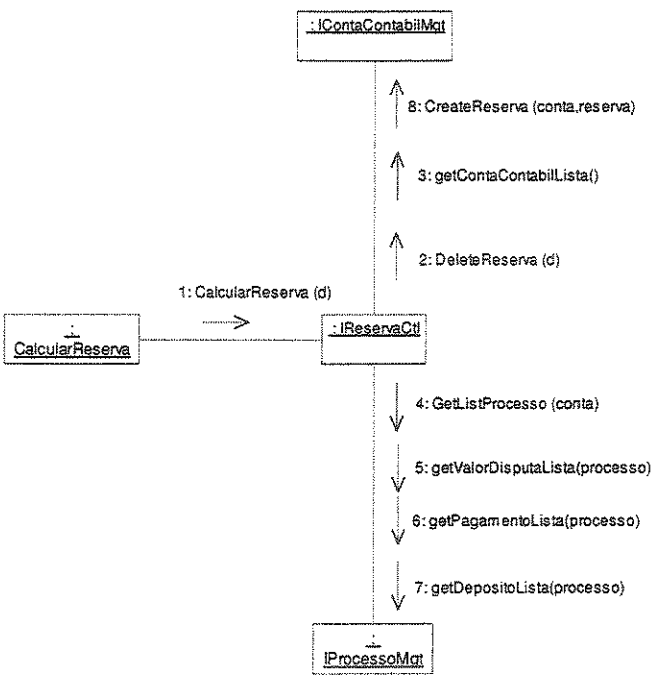

Figura 35 - Diagrama de colaboração

UNICAMP
BIBLIOTECA CENTRAL
DESENVOLVIMENTO DE COLEÇÕES

Em alguns casos as assinaturas dos métodos dos componentes do sistema são resolvidas pelo próprio componente, em outros casos as assinaturas são simplesmente repassadas para os componentes de negócio, que as resolve, e em outros casos os métodos precisam ser quebrados em duas ou mais chamadas para componentes de negócio.

Além das interfaces e operações foram definidos tipos de dados estruturados para facilitar a troca de informações através das chamadas dos métodos. Estes tipos de dados estruturados foram modelados como classes (Figura 36), identificadas através do estereótipo *<<DataType>>* (Tabela 2).

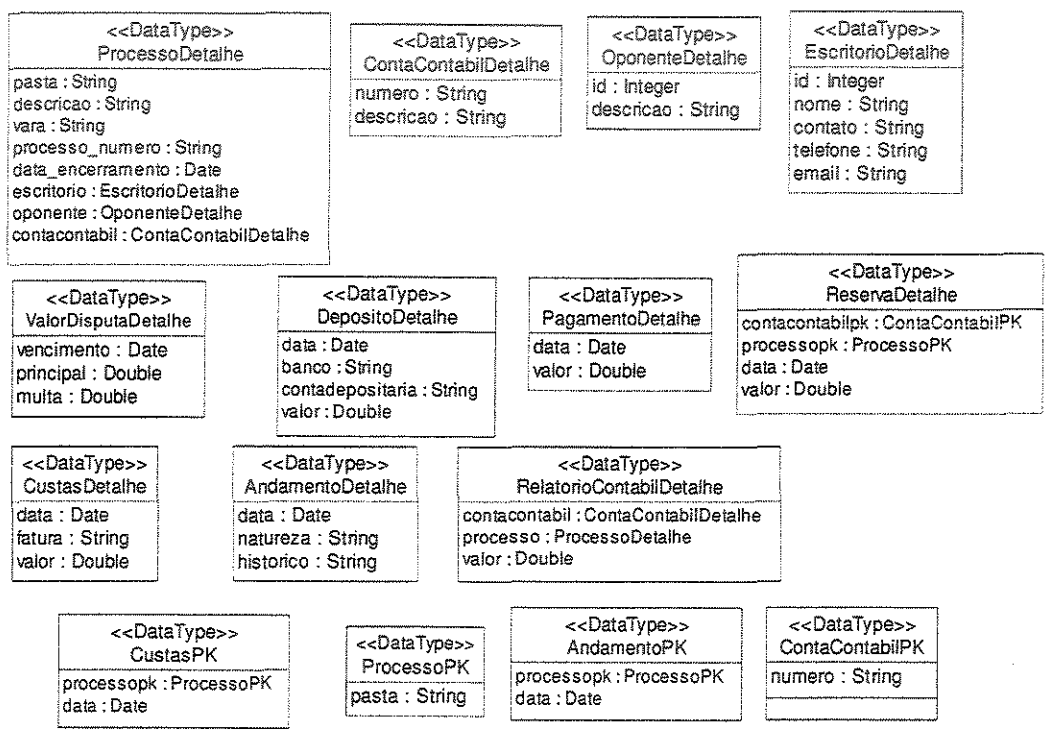

Figura 36 – Tipos de dados estruturados

Feito isso para todos os métodos foram obtidas as especificações das interfaces. A Figura 37 ilustra a especificação de uma interface de componente de negócio após a finalização deste processo.

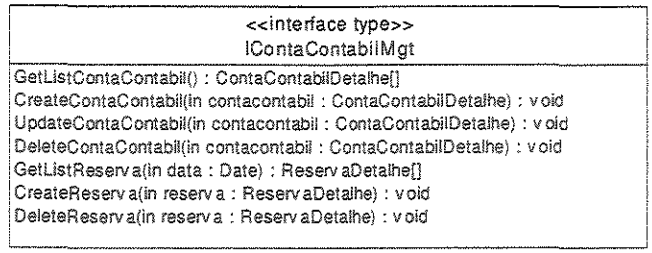

Figura 37 - Especificação de interface de componente de negócio

4.3.3 Estágio Especificação de Componentes

No estágio *Especificação de Componentes* foram definidas as pré- e pós-condições das operações das interfaces, bem como, definidas as invariantes associadas aos tipos, conforme prescrito no processo *UML Components* (seção 2.4).

4.3.4 Estágio Refinamento da Arquitetura de Software

O refinamento da arquitetura de software, assim como os modelos neste nível de especificação, são independentes de plataforma.

Em função dos requisitos não-funcionais foi definida uma arquitetura genérica que pode ser implementada em diferentes tecnologias. Esta arquitetura foi ilustrada através de duas visões, a lógica e a de processo, conforme prescrito no processo adaptado (seção 3.1.2).

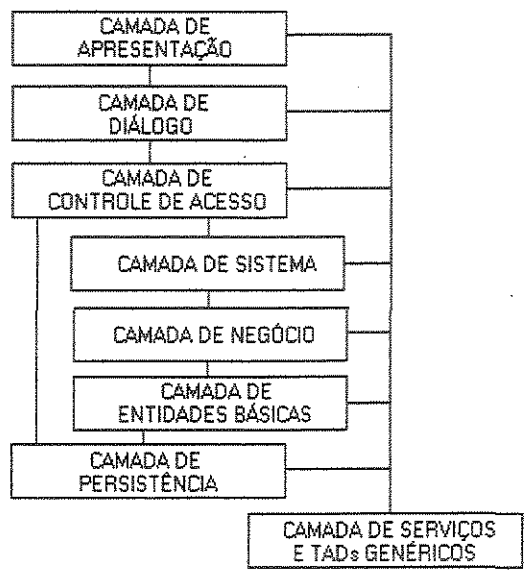

Figura 38 – Visão lógica da arquitetura de software do sistema de contingências tributárias

Na visão lógica (Figura 38) encontram-se representadas as seguintes camadas:

- Camada de apresentação: a camada de apresentação corresponde à interface visual com o usuário.
- Camada de diálogo: a camada de diálogo implementa os casos de uso definidos para a aplicação. Nesta camada não há preocupação com reutilização e novos casos de uso podem ser criados e modificados rapidamente
- Camada de controle de acesso: esta é uma camada intermediária entre a camada de diálogo e a camada de sistema, sua função é atender o requisito não-funcional de controle de acesso. Estes requisitos especificam que somente usuários cadastrados terão acesso ao sistema, e que o acesso às funções será dado em função do nível de acesso de cada usuário. O sistema é composto de quatro níveis de acesso. Esta camada fará a validação inicial do usuário (login) e a validação de execução de cada função. As informações de segurança (cadastro de usuários, níveis de acesso, restrições em função do nível de acesso, etc) ficarão armazenadas na camada de repositório. A criação desta camada reflete a importância do requisito não-funcional controle de acesso.
- Camada de sistema: a camada de sistema expõe uma interface para todas as funcionalidades da aplicação, servindo como um façade para as regras de negócio do sistema e as interfaces dos módulos de negócio.
- Camada de negócio: a camada de negócio expõe as interfaces dos componentes estáveis da aplicação e que poderão ser utilizados por outras aplicações, e que são responsáveis pelo gerenciamento das entidades básicas.
- Camada de entidades básicas: a camada de entidades básicas constitui-se de elementos que implementam tipos abstratos de dados para o sistema de controle tributário, como escritório, oponente, etc. Esta camada, de acordo com o processo *UML Components*, equivale aos sub-componentes da camada de negócio.
- Camada de persistência: a camada de persistência implementa um conjunto de serviços para persistência de objetos em um Sistema Gerenciados de Banco de Dados.
- Camada de Serviços e TADs genéricos: esta camada implementa funções e tipos abstratos de dados genéricos como por exemplo *Sort*, *Array*, etc.

A divisão do sistema em diversas camadas reflete os objetivos primários do desenvolvimento baseado em componentes: (1) adaptabilidade, ou seja, a capacidade do sistema de adequar-se às mudanças, e (2) separação de interesses, pois cada camada é responsável por um aspecto bem definido da aplicação. A organização em camadas, diferentemente de uma aplicação monolítica, permite que cada camada possa ser alterada com menor impacto nas demais, e que determinados aspectos sejam alterados independentemente, por exemplo, pode-se mudar o mecanismo de controle de acesso sem alteração das regras de negócio.

Os aspectos desfavoráveis desta arquitetura dizem respeito, a princípio, com a queda de eficiência, pois para a execução de determinada funcionalidade a aplicação deverá percorrer diversas camadas, e por mais eficiente que seja a comunicação entre as camadas, ela certamente implicará uma sobrecarga inexistente, por exemplo, em uma aplicação monolítica.

Na visão lógica apresentada cada camada pode invocar apenas elementos da própria camada ou de camadas que estejam abaixo dela, sem exceção. Um elemento da camada de negócio não pode, por exemplo, fazer uma chamada para um elemento da camada de sistema.

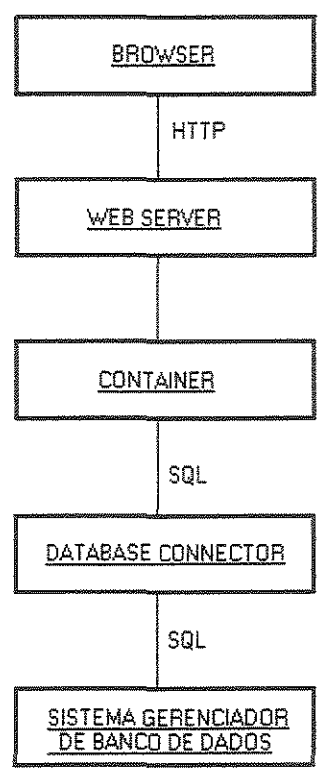

Figura 39 – Visão de processos da arquitetura de software do sistema de contingências tributárias

Na visão de processos (Figura 39) encontram-se os seguintes elementos:

- *Browser*: este processo terá como função mostrar as telas em formato HTML geradas pelo *Web Server* e permitir a entrada de dados, contendo, eventualmente, *Scripts* de consistência. Este processo é responsável pelo processamento dos elementos descritos na camada de apresentação da visão lógica.
- HTTP: o protocolo http[20] é o protocolo a ser utilizado para conexão da camada de apresentação com a camada de diálogo. A escolha deste protocolo deve-se à sua larga utilização no mercado e facilidade de uso.
- *Web Server*: o *Web Server* é um processo capaz de gerar páginas HTML dinâmicas a partir de requisições do browser. O *Web Server* deve, ainda ser capaz de acessar os elementos do *Container*. Este processo é responsável pelo processamento dos elementos descritos na camada de diálogo da visão lógica.
- *Container*: o *container* é um processo capaz de prover serviços de armazenamento e execução de componentes (por exemplo, nomeação, segurança, gerenciamento de ciclo de vida). Este processo é responsável pelo processamento dos elementos descritos nas camadas de (1) controle de acesso, (2) sistema, (3) negócio, (4) entidades básicas, (5) persistência, e (6) serviços e Tipos Abstrados de Dados (TAD) genéricos da visão lógica.
- SQL: a comunicação entre o *Container* e o *Database Connector* e entre o *Database Connector* e o Sistema Gerenciador de Banco de Dados será feita utilizando a especificação SQL[21] [SQL ANSI 92].
- *Database Connector*: o *DataBase Connector* é o processo a ser utilizado para conexão entre o *Container* e o Sistema Gerenciador de Banco de Dados, funcionando como um *Adapter*, e tem por finalidade estabelecer um formato padronizado de acesso ao Sistema Gerenciador de Banco de Dados.
- Sistema Gerenciador de Banco de Dados: o SGBD é responsável por armazenar o esquema do banco de dados da aplicação e prover mecanismos de acesso aos dados

[20] do inglês HyperText Transfer Protocol

[21] do inglês Structured Query Language

compatíveis com o *Database Connector* utilizado, e serviços de segurança (controle de acesso). Este processo é responsável pelo processamento dos elementos descritos na camada do repositório da visão lógica.

A distribuição da aplicação nestes cinco processos permite atingir os requisitos não-funcionais obtidos na definição de requisitos não-funcionais, a saber: (1) adaptabilidade, (2) portabilidade, (3) escalabilidade, e (4) segurança.

A portabilidade é alcançada de várias formas. Através da utilização de elementos arquiteturais sobrepostos ao sistema operacional (*Browser*, *Web Server*, *Container*, SGBD) que permite que a aplicação seja implementada em qualquer plataforma que disponibilize tais elementos. Através da utilização de um *browser*, o que implica que não é necessária qualquer configuração adicional (instalação) nos clientes da aplicação, e, portanto, o cliente pode executar a aplicação através de qualquer computador com *browser* instalado e acesso ao sistema.

A escalabilidade é obtida através da separação de camadas. Uma vez replicados, os elementos arquiteturais responsáveis por cada camada podem ser replicados ou delegar suas funções para atender um aumento da demanda. Por exemplo, o *Web Server* pode servir apenas como um computador para balanceamento de carga, e distribuir as reais atividades de geração dinâmica de páginas HTML para outros computadores que estejam a ele conectados, de forma transparente para o requisitante de tais páginas. A camada de banco de componentes pode prover serviços de pool de componentes para atender mais rapidamente a demanda por criação de componentes, evitando sobrecarga no processamento, também de forma transparente ao requisitante pela instanciação de um componente.

A segurança é obtida através dos serviços de segurança providos pelos processos *Web Server*, *Container*, e SGBD. Estes processos disponibilizam serviços de segurança para validação e autenticação de usuários, com restrições de acesso para consulta e alteração dos conteúdos por eles gerenciados.

Os aspectos desfavoráveis são basicamente os mesmos mencionados na visão lógica, ou seja, sobrecarga em função da existência de vários processos.

Na visão de processos apresentada cada camada pode invocar apenas elementos da própria camada ou da camada que esteja imediatamente abaixo dela, sem exceção. Um elemento da camada de diálogo não pode, por exemplo, fazer uma chamada para um elemento da camada

de banco de dados.

4.3.5 Estágio Mapeamento dos Componentes na Arquitetura de Software

Neste estágio foram criados novos componentes, conforme prescrito no processo adaptado (seção 3.1.3), em função das novas camadas criadas na seção imediatamente anterior.

A camada de apresentação foi mapeada como um conjunto de arquivos de apresentação em formato HTML. A camada de diálogo é basicamente composta de arquivos JSP (*Java Server Pages*) na implementação J2EE e arquivos ASPX (*Active Server Pages*) na implementação .NET. Mas eles não são o foco deste processo, eles compõem o lado do cliente (seção 2.4.1).

Na camada de controle de acesso foi criado um componente responsável pela validação dos usuários e das operações de cada usuário. Este componente estende a interface do componente de sistema, e inclui as operações de *login* e *logout* (Figura 40).

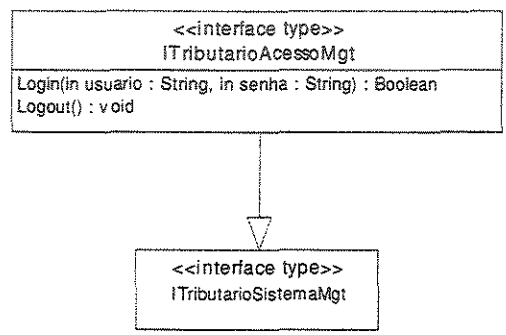

Figura 40 – Interface do componente da camada de acesso

A camada de entidades básicas será responsável pelo gerenciamento dos tipos do sistema, no nível de gerenciamento de persistência. Estes tipos que, de acordo com o processo *UML Components*, deveriam ser considerados como sub-componentes e incorporados os componentes da camada de negócio, foram externalizados e transformados em componentes, denominados entidades. Cada entidade, desta forma, será responsável por gerenciar a sua persistência, e não o componente de negócio, promovendo uma separação de interesses que visa facilitar a evolução do sistema. Cada tipo será mapeado para uma interface, e cada interface para um componente. Todas as operações de persistência que o componente de negócio tiver que realizar serão repassadas para esses componentes. A Figura 41 apresenta um exemplo de interface

correspondente a um tipo do sistema. O tipo de parâmetro *IOponenteDetalheMgt* é um tipo de dado do sistema e será explicado mais adiante.

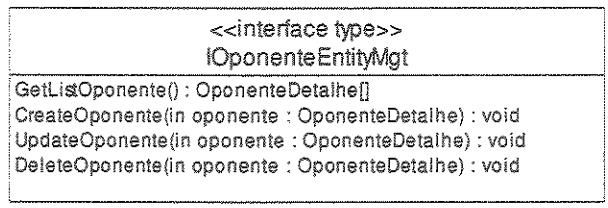

Figura 41 – Interface de entidade básica

Para atender a camada de persistência foi necessário criar um componente, responsável pela ligação das demais camadas com a camada de repositório. Como se adotou como padrão a utilização de um servidor de banco de dados que utilize a especificação SQL, a interface deste componente será compostas por operações para atendimento de comandos de ação (*INSERT*, *UPDATE*) e de consultas (*SELECT*) (Figura 42).

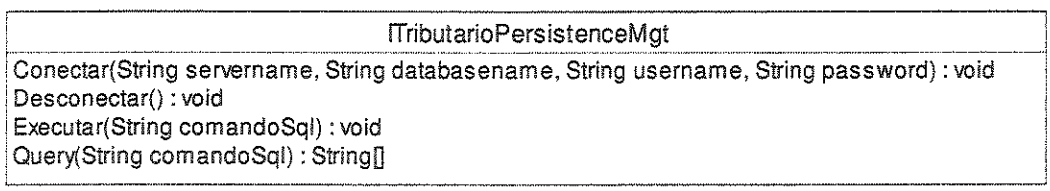

Figura 42 – Interface do componente de persistência

A camada de repositório é composta pelo servidor de banco de dados não sendo necessário criar componentes programáveis, apenas utilizar as interfaces existentes para acesso ao mesmo.

Os tipos de dados definidos no Modelo de Tipos do Negócio foram agrupados em um componente responsável por tornar pública a definição dos tipos utilizados. Este componente global, chamado de *TributarioDataTypeMgr*, poderá, e deverá, ser acessado por todos os outros componentes, e os tipos nele contidos serão utilizados, quando aplicáveis, como parâmetros para troca de informações entre os demais componentes. Este componente, juntamente com as bibliotecas de classe das plataformas alvo, comporão a camada de serviços e tipos abstratos de dados genéricos. Uma das interfaces que compõem o componente *TributarioDataTypeMgr* é a

interface *IOponenteDetalheMgt* (Figura 43).

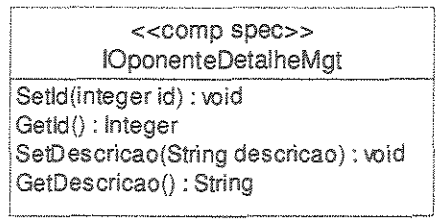

Figura 43 – Interface de tipo de dados

Estas interfaces foram definidas a partir das classes definidas com estereótipo *<<DataType>>*.

A especificação final dos componentes de acordo com a estrutura proposta está ilustrada nas Figuras 44, 45 e 46. A ligação entre a Figura 44 e 45 é feita através do componente *ContaContabilMgr*. A ligação entre a Figura 44 e 46 é feita através do componente *ProcessoMgr*.

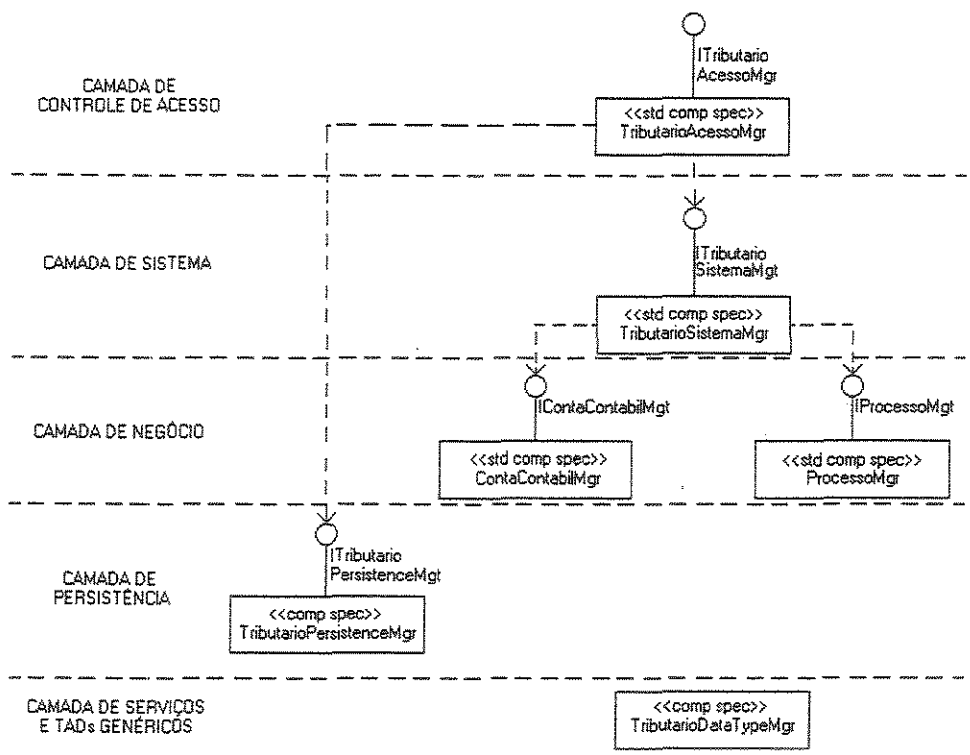

Figura 44 - Diagrama de especificação de componentes – parte 1

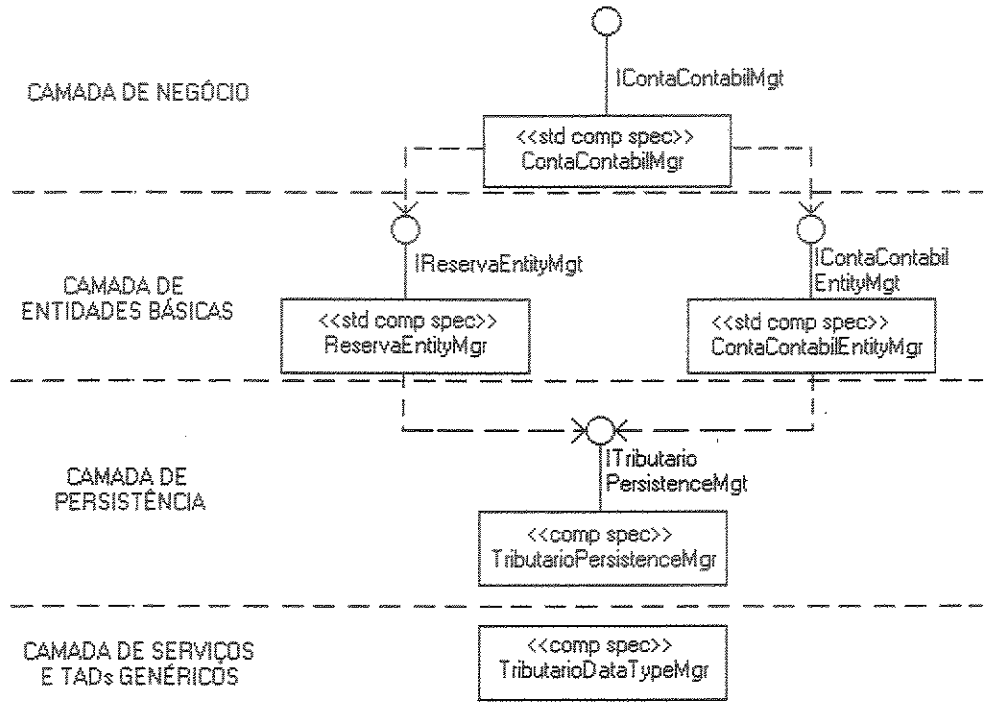

Figura 45 - Diagrama de especificação de componentes – parte 2

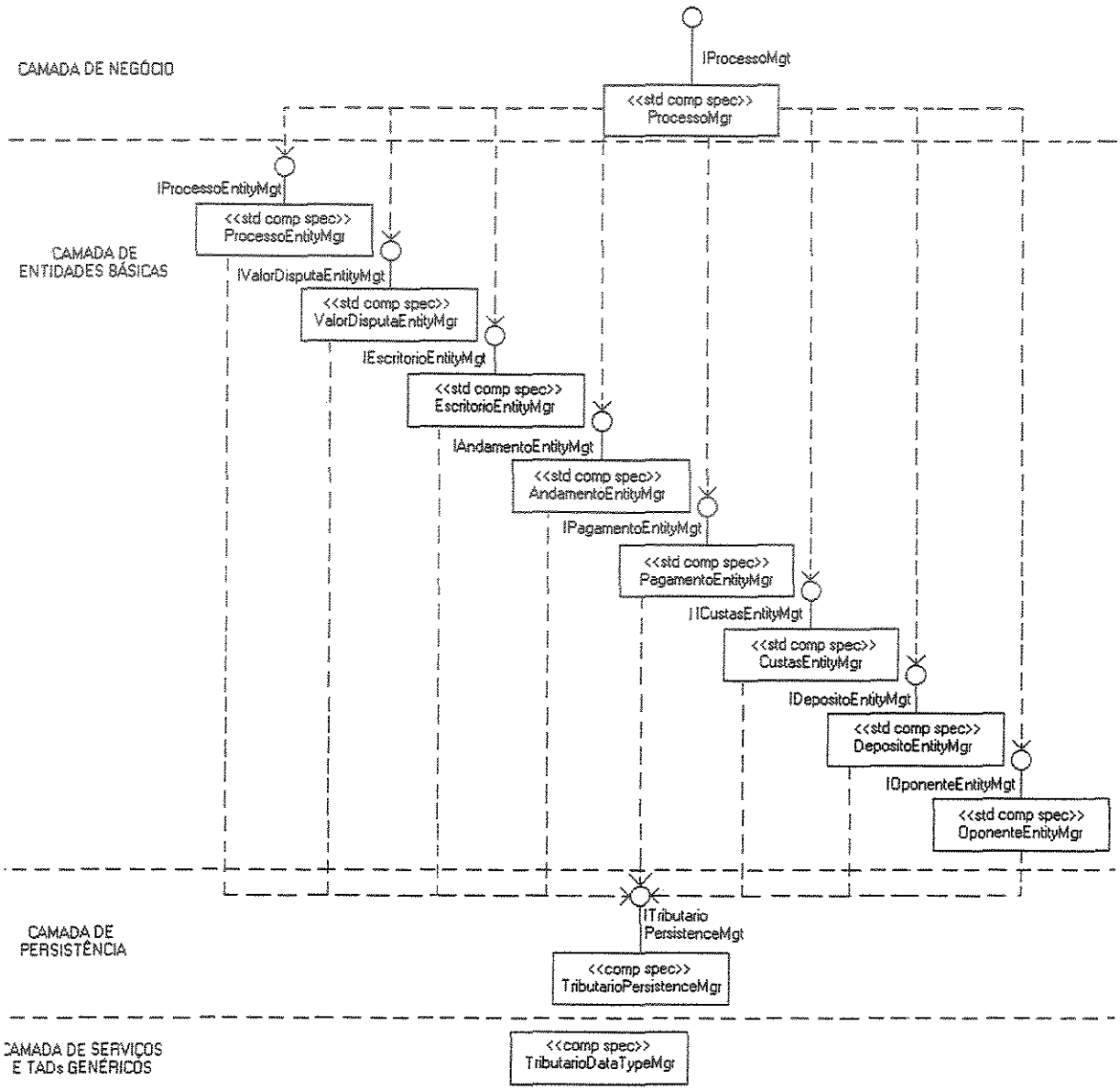

Figura 46 - Diagrama de especificação de componentes – parte 3

4.3.6 Estágio Refinamento do PIM

No estágio *Refinamento do PIM* foram refinadas as especificações das interfaces e dos componentes, conforme prescrito no processo adaptado (seção 3.1.4). Basicamente foram identificadas, através do estereótipo *<<std comp spec>>* (Tabela 2), as classes para as quais se

pretende gerar componentes de acordo com a modelo COSMOS (seção 2.6), e quais representam interfaces com o usuários, através do estereótipo *<<dlg comp spec>>* (Tabela 2).

4.4 Fluxo de Trabalho Provisionamento

4.4.1 Estágio Mapeamento do PIM para PSM

No estágio *Mapeamento do PIM para PSM* foram aplicadas as regras de mapeamento para J2EE (seção 3.2) e para .NET (seção 3.3), prescritas no processo adaptado (seção 3.1.5), gerando modelos representativos dos componentes e conectores nestas plataformas. Foram geradas representações para os seguintes elementos:

- Componentes de acordo com o modelo COSMOS: *TributarioAcessoMgr*, *TributarioSistemaMgr*, *ContaContabilMgr*, *ProcessoMgr*, *ReservaEntityMgr*, *ContaContabilEntityMgr*, *ProcessoEntityMgr*, *ValorDisputaEntityMgr*, *PagamentoEntityMgr*, *DepositoEntityMgr*, *AndamentoEntityMgr*, *CustasEntityMgr*, *EscritorioEntityMgr* e *OponenteEntityMgr*.
- Conectores de acordo com o modelo COSMOS: *ConectorTributarioAcessoMgr*, *ConectorTributarioSistemaMgr*, *ConectorContaContabilMgr* e *ConectorProcessoMgr*.
- Componentes na forma de bibliotecas de classes: *TributarioDataTypeMgr*, *TributarioPersistenceMgr*.

4.4.2 Estágio Mapeamento do PSM para Implementação

No estágio *Mapeamento do PSM para Implementação* foram aplicadas as regras de mapeamento para J2EE (seção 3.4.1) e .NET (seção 3.4.2), prescritas no processo adaptado (seção 3.1.6). Esse mapeamento foi feito manualmente e foram utilizadas, a partir deste ponto as seguintes ferramentas:

- Plataforma J2EE: *IBM Websphere Studio Application Developer* (WSAD) *5.1*.
- Plataforma .NET: *Visual Studio .NET Professional, version 2002*.

4.4.3 Estágio Preenchimento do Código Fonte

No estágio *Preenchimento do Código Fonte*, prescrito no processo adaptado (seção 3.1.7), foram preenchidos os esqueletos dos códigos fontes gerados para cada arquivo de cada

plataforma. O preenchimento foi efetuado respeitando as restrições impostas pelos PSM relacionados, e utilizando as ferramentas mencionadas na seção imediatamente anterior.

4.5 Fluxos de Trabalho Montagem, Testes e Implementação

As ferramentas utilizadas para preenchimento do código fonte nas plataformas alvo, *Microsoft Visual Studio .NET* na plataforma .NET, e WSAD 5.1 na plataforma J2EE, mais que simples editores de texto são *Ambientes Integrados de Desenvolvimento* (IDE[22]), que possuem recursos de edição de software, edição visual de telas, simulação de servidores WEB ou integração com Servidores WEB instalados local ou remotamente, e recursos de depuração local ou remoto, entre outras facilidades. Adicionalmente foi instalado o banco de dados MySql onde foram criadas as tabelas utilizadas para persistência das informações. A conexão com o banco de dados foi feita através de JDBC (*Java Database Connectivity*) no ambiente J2EE e ODBC (*Open Database Connectivity*) no ambiente .NET. Portanto, foram utilizados os recursos dos *IDEs* para efetuar a montagem e testes do aplicativo e a simulação da implementação do sistema no ambiente de trabalho.

4.6 Resultados Obtidos

As adaptações efetuadas no processo *UML Components* mostraram-se eficazes no desenvolvimento do sistema utilizado no estudo de caso, e alcançaram os seguintes objetivos: (1) demonstração dos princípios da abordagem MDA, (2) o tratamento dos requisitos não-funcionais através da arquitetura de software, (3) a utilização dos mapeamentos criados para .NET e J2EE, a partir do modelo COSMOS.

Os princípios da abordagem MDA foram demonstrados através da separação explícita entre a especificação do sistema e das diferentes especificações de implementação da mesma especificação original em diferentes plataformas tecnológicas, no caso J2EE e .NET.

O tratamento dos requisitos não-funcionais do sistema como, por exemplo, controle de acesso, foi obtido através da definição dos requisitos não-funcionais e do refinamento da arquitetura de camadas prevista no processo *UML Components*.

---

[22] do inglês Integrated Development Environment

A utilização dos mapeamentos criados para .NET e J2EE a partir do modelo COSMOS foram importantes tanto do ponto de vista de diminuir a distância semântica entre as abstrações da descrição arquitetural e as construções disponíveis nas plataformas utilizadas, como do ponto de vista de demonstrar os princípios da abordagem MDA, uma vez que serviu como base para definição de regras claras de mapeamento entre os modelos independentes de plataforma e os modelos dependentes de plataforma.

As principais dificuldades surgiram em função da abordagem MDA ser bastante recente e, pior que isso, ainda não estar completa. Esta afirmação é baseada em autores como Kleppe, Warner e Bast [KLEPPE+03] e Frankel [FRANKEL03] que são membros do consórcio OMG e escreveram livros sobre o assunto. Particularmente dois pontos dificultaram sobremaneira o desenvolvimento dos experimentos: (1) a inexistência de uma especificação de definição de regras de transformação; e (2) a indisponibilidade de ferramentas para utilização da abordagem MDA. Quanto à inexistência de uma especificação de definição de regras de transformação, segundo estes mesmos autores, o consórcio OMG está trabalhando atualmente em tal especificação e ela deve disponibilizada em breve. Quanto a indisponibilidade de ferramentas não significa que elas não existam, mas são poucas e caras (por exemplo, MIA-Software), e não existe versão acadêmica ou gratuita.

Capítulo 5

Considerações Finais

Apesar do processo *UML Components* (seção 2.4) mostrar-se eficaz, alguns aspectos não considerados são importantes no desenvolvimento de sistemas e merecem um tratamento sistemático. Dentre eles podemos citar: (1) o suporte à evolução tecnológica das plataformas nas quais os sistemas são implementados, (2) o tratamento dos requisitos não-funcionais, e (3) diminuir a distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas plataformas alvo. Este trabalho apresentou uma adaptação de um processo de desenvolvimento de software baseado em componentes utilizando a abordagem MDA buscando incorporar estes aspectos.

O suporte à evolução tecnológica das plataformas é particularmente importante em sistemas de grande porte cuja vida útil costuma estender-se por vários anos. Nestes sistemas deve-se procurar preservar o investimento feito na especificação inicial, reduzindo o retrabalho uma vez que haja alguma mudança tecnológica como, por exemplo, mudança de plataforma CORBA para J2EE ou mesmo a mudança para uma versão J2EE mais recente. Buscando tratar esta questão o consórcio OMG propôs uma abordagem chamada *Model Driven Architecture* (MDA) (seção 2.5) que busca separar a especificação da funcionalidade, da especificação dessa mesma funcionalidade em uma determinada plataforma. Neste trabalho foram incluídos estágios que permitiram incorporar a abordagem MDA (seções 3.1.4, 3.1.5, 3.1.6 e 3.1.7), em particular o conceito de mapeamento (seção 2.5.4) entre os modelos independentes de plataforma e dependentes de plataforma (seção 2.5.1), foi alcançado através da utilizações de modelos de estruturação de componentes baseados no modelo COSMOS (seção 2.6).

Em função do aumento da importância dos requisitos não-funcionais, como por exemplo, disponibilidade, controle de acesso e tolerância a falhas, e pelo fato de que estes normalmente estão relacionados à arquitetura de software, esta tem representado um papel cada vez mais importante na especificação dos sistemas. Neste trabalho também foram incluídos estágios (seções 3.1.1, 3.1.2 e 3.1.3) no processo *UML Components* que permitem tratar mais claramente os requisitos não-funcionais do sistema, através da definição formal destes requisitos não-

funcionais e da possibilidade de refinamento da arquitetura de software.

A utilização dos modelos de estruturação de componentes baseados no modelo COSMOS, além dos objetivos alcançados em relação à abordagem MDA, conforme mencionado anteriormente, permitiu diminuir distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas plataformas alvo, pois definiu elementos para a implementação dos componentes e conectores, participantes da composição de software, que puderam ser implementados nas plataformas escolhidas.

5.1 Conclusões

Um ponto nebuloso no início deste trabalho era saber como aplicar a abordagem MDA. Em setembro de 2003 participei de um congresso internacional (IFIP 3E Conference) no Guarujá e verifiquei que a dúvida era comum a outros participantes. Esta dúvida decorre da abordagem MDA não ser uma especificação formal com a qual pode ser feita uma validação do tipo atende ou não atende, ela apenas estabelece diretrizes. Para conseguir transpor este obstáculo no meu trabalho foi necessário fixar um contexto, conforme fizemos no trabalho apresentado: um processo de desenvolvimento de software baseado em componentes, que prescreve uma arquitetura de software baseada em camadas, um modelo de estruturação de componentes e duas plataformas alvo com suporte a componentes. A reunião destes elementos permitiu aplicar os conceitos prescritos pela abordagem MDA, como modelos dependentes e independentes de plataforma (seção 2.5.1) e o conceito de mapeamento (seção 2.5.4), no que foi de fundamental importância o modelo de estruturação de componentes COSMOS (seção 2.6). Portanto, tentamos depreender resultados gerais a partir de um contexto particular.

No documento inicial do MDA [MDA01] um modelo "independente de plataforma" era considerado uma especificação formal que abstraía os detalhes técnicos. Em 2003 o consórcio OMG publicou um documento entitulado *MDA Guide* [MDAGUIDE03] onde, o termo independência de plataforma é definido como uma qualidade e, portanto, uma questão de grau, ou seja, um modelo pode ser mais independente de plataforma ou menos independente de plataforma. Esta foi uma mudança importante e ratifica as nossas conclusões em função dos experimentos realizados: uma especificação apresenta um determinado grau de independência de plataforma, mas não se pode afirmar com certeza que uma determinada especificação é

totalmente independente de plataforma. Portanto, o MDA é uma abordagem válida no sentido de aumentar o nível de abstração e, conseqüentemente, de independência de plataforma, mas não de eliminar a dependência de plataforma.

Apesar da definição das plataformas alvo ter sido feita *a priori*, grande parte da especificação resultante do processo apresentado na seção 3 apresenta um elevado grau de independência de plataforma e pode ser implementada em alguma outra plataforma, desde que essa comporte a arquitetura de software definida no estágio *Refinamento da Arquitetura de Software* (seção 3.1.2).

Para atender a abordagem MDA é necessário um esforço adicional, ou seja, é mais trabalhoso do que definir uma plataforma e desenvolver os sistemas com base nesta plataforma. No entanto, para sistemas de grande escala que normalmente tem uma vida útil de vários anos, acreditamos que este sobrepreço seja vantajoso, uma vez que a especificação de sistemas para uma empresa de grande porte é um ativo extremamente valioso. A abordagem MDA é mais trabalhosa em função dos seguintes aspectos:

- Necessidade de criar mapeamentos (como neste trabalho em particular onde foi utilizada uma solução *ad hoc*) ou estudar os mapeamentos existentes para verificar se eles atendem as necessidades.
- Necessidade de refinar os modelos PIM com elementos que permitam um mapeamento mais preciso.

De qualquer forma este trabalho adicional pode ser minimizado em função das seguintes medidas:

- Reutilização de mapeamentos existentes.
- Criação de ferramentas de automação das transformações entre modelos.

Estes aspectos são particularmente importantes e vários fabricantes estão engajados nesta proposta, apesar de ainda não haver uma especificação de linguagem de transformação entre modelos, que foi uma outra dificuldade encontrada neste trabalho

Também se verificou que uma melhor qualidade de mapeamento entre modelos, em última análise de geração de código, decorre da limitação das escolhas do programador. Inevitavelmente automatizar os mapeamentos significa adotar soluções padronizadas, em detrimento da escolha de soluções pelo programador.

5.2 Contribuições

Como principais contribuições deste trabalho destacam-se:

1. O Processo de Desenvolvimento Baseado em Componentes Adaptado ao *Model Driven Architecture*: A principal contribuição deste trabalho foi a proposta de um processo de desenvolvimento de software, que adapta o processo *UML Components*, e permite especificar um sistema baseado em componentes seguindo a abordagem MDA e tratando explicitamente os requisitos não-funcionais através da arquitetura de software, conforme descrito no capítulo 3. Ao torná-lo compatível com a abordagem MDA este trabalho ilustrou a separação dos modelos independentes de plataforma, ou seja, aqueles que não incorporam detalhes técnicos das plataformas alvo, dos modelos dependentes de plataforma. Ao tratar os requisitos não-funcionais, cada vez mais importantes na implementação dos sistemas, este trabalho ressaltou a importância da arquitetura de software na especificação de sistemas de software.
2. Criação de mapeamentos para J2EE e .NET a partir do modelo COSMOS: A partir do modelo COSMOS (seção 2.6) foram criados mapeamentos para J2EE e .NET que: (1) reduziram a distância semântica entre as abstrações da descrição arquitetural e as construções disponíveis nas plataformas utilizadas no estudo de caso para implementar o sistema de software, (2) permitiram implementar os mecanismos de mapeamento entre os modelos independentes de plataforma e os modelos dependentes de plataforma prescritos na abordagem MDA, e (3) aumentaram a rastreabilidade entre os modelos. Estes mapeamentos estão descritos nas seções 3.2 e 3.3.
3. Exemplo de utilização do processo adaptado: A aplicação do processo adaptado foi exemplificada através do estudo de caso de um Sistema de Contingências Tributárias. Trata-se de um sistema real que possibilitou verificar a exeqüibilidade do processo e está descrito no capítulo 4.

Adicionalmente este trabalho resultou na publicação do seguinte artigo:

- *Component-Based Development using Model Driven Architecture*
  Autores: Milton C. F. Sousa, Paulo A. C. Guerra e Cecília M. F. Rubira
  Conferência: IFIP Conference on E-commerce, E-business, E-government

I3E 2003 – Guarujá – SP – Brasil, Setembro 2003.

5.3 Trabalhos Futuros

O ponto principal a ser focado para evolução do trabalho apresentado é a automação dos mapeamentos efetuados, tanto dos mapeamentos PIM para PSM, como dos mapeamentos PSM para código fonte. Já existem atualmente algumas ferramentas que permitem automatizar parcialmente estes mapeamentos, todavia, não existe, ainda, uma especificação para definição de regras de transformação, sendo esta a principal lacuna a ser preenchida na abordagem MDA. Mesmo que tal linguagem existisse, há que se definir até que ponto o PIM deve incorporar detalhes que permitam um mapeamento mais preciso a ponto de gerar totalmente o código fonte, através do uso intensivo da especificação OCL[23] e *Action Semantics*, mas certamente grandes avanços ainda poderão ser feitos nesta área.

Ainda com relação à automação dos mapeamentos, um outro ponto importante a ser estudado é a sincronização de modelos a partir de alterações efetuadas. Idealmente uma alteração no PIM deveria refletir imediatamente no código fonte, passando, obviamente, por uma nova geração dos PSM. Talvez o contrário, não seja possível, ou desejado, mas de qualquer forma seria interessante dispor de ferramentas que definissem e restringissem a sincronização de modelos como forma de manter atualizada a documentação do sistema, e evitar que, após a implementação inicial, as alterações passem a ser feitas somente no código fonte, tornando os modelos obsoletos.

Um outro aspecto pouco explorado no processo apresentado é o tratamento de exceções. A Metodologia para Definição do Comportamento Excepcional (MDCE) [FERREIRA01], que mantém a preocupação com as situações excepcionais e seus tratadores desde a definição de requisitos do sistema, se estendendo pelas atividades de projeto e implementação, é um exemplo de metodologia que poderia ser incorporada com o trabalho ora apresentado, resultando em uma metodologia para desenvolvimento de componentes, utilizando a abordagem MDA, e que levasse em consideração o comportamento normal e o comportamento excepcional.

---

[23] do inglês Object Constraint Language

Referências Bibliográficas

[BACHMANN+01] F. Bachmann et al, *Document software architectures: Organization of documentation package*. CMU/SEI-2001-TN-010, August 2001.

[BASS+03] Bass, L; Clements, P.; Kazman R., *Software Architecture in Practice*. Second Edition, Addison-Wesley, 2003.

[BROOKS95] F.P.Brooks Jr., *The Mythical Man-Month*. Addison-Wesley, Reading, MA, 1995.

[BROOKS87] Brooks, F. P. Jr. *No Silver Bullet: Essence and Accidents of Software Engineering*. Computer *20*, 4 (April 1987): 10-9.

[BROWN+96] Brown, A.W. e K.C. Wallnau, *Engineering of Component Based Systems*. IEEE Computer Society Press, 1996.

[CHEESMAN+01] Cheesman, J. e Daniels, J., *UML Components A Simple Process for Specifying Component-Based Software*. First Edition, 2000.

[COLIN+01] Colin, A, et.al., *Component-Based Product Line Engineering with UML*. First Edition, 2001.

[CWM03] *Common Warehouse Metamodel (CWM) Specification, Version 1.1*. Volume 1, OMG document formal, 2003.

[DOTNET] *Microsoft .NET* em http://www.imasters.com.br.

[DSOUZA+99] D'Souza, D.F., e A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[EJB+01] DeMichiel L, et.al., *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, 2001.

[FERREIRA01] Ferreira, G. R. M., *Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes*. Dissertação de Mestrado, IC/Unicamp, 2001.

[FRANKEL03] Frankel, D.S., *Model Driven Architecture - Applying MDA to Enterprise Computing*. First Edition, OMG Press, 2003.

[GARLAN93] Garlan, D. & Shaw, M. *An Introduction to Software Architecture - Advances in Software Engineering and Knowledge Engineering*. Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993.

[GAYCEK95] Gaycek, C.; Abd-Allah, A.; Clark, B.; & Boehm, B. *On the Definition of Software System Architecture*. Invited talk, First International Workshop on Architectures for Software Systems. Seattle, WA, April 1995.

[GOGUEN86] J. A. Goguen. *Reusing and Interconecting Software Components*. IEEE Computer, pages 16-27, February 1986.

[HAYES-ROTH94] Hayes-Roth. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA)* Program [online]. Available WWW <URL:http://www.sei.cmu.edu/arpa/dssa/dssa-adage/dssa.html>(1994).

[IYENGAR02] Iyengar, S., *Model Driven Architecture (MDA) meets Web Services*. http://www.omg.org/mda/presentations.htm

[JACOBSON+99] I. Jacobson, J. Rumbaugh e G. Booch, *The Unified Software Development Process*. Addison-Wesley, Reading -MA, 1999.

[KLEPPE+03] Kleppe, A; Warmer, J.; Bast, W, *MDA Explained the model driven architecture: practice and promise*. Addison-Wesley, 2003.

[KRUCHTEN95] P. Kruchten. *The 4+1 View Model of Architecture*. IEEE Software, 12(6), 1995.

[KWON+00] O. Kwon, S. Yoon, G. Shin, *Component-Based Development Enrionment: An Integrated Model of Object-Oriented Techniques and Other Technologies*. Computer & Software Technology Laboratory, Eletronics and Telecommunications Research Institute, Taejon, Korea. September, 2000.

[LUCKHAM+00] D. C. Luckham, J.Vera, and S. Meldal. *Key Concepts in Architecture Definition Languages, In Foundations of Component-Based Systems*. Cambridge University Press, 2000, pp. 23-25.

[MCCALL+77] McCall, J., P.Richards, and G. Walters, *Factors in Software Quality*. Three volumes, NTIS AD-A049-014, 015, 055, November 1977.

[MCLLROY69] M.D. Mcllroy. *Mass produced software components. In P. Naur and B. Randall, editors, Software Engineering: Report on a conference by the NATO Science Committe*, pages 138 - 150. NATO Scientific Affair Division, 1968.

[MDA01] *Model Driven Architecture (MDA)*. Document number ormsc, Architecture Board

ORMSC, 2001.

[MDAGUIDE03] *MDA Guide Version 1.0.* Document number omg/2003-05-01, , 2003.

[MELLOR+01] Mellor, S. J. e Balcer, M. J., *Executable UML: A Foundation for Model-Driven Architecture.* Addison Wesley, 2002.

[MEYER97] Meyer, B., *Object-Oriented Software Construction.* Second Edition, Prentice-Hall, 1997.

[MOF02] *Meta Object Facility (MOF) Specification, Version 1.4.* OMG document formal, 2002.

[MONROE+97] R .T Monroe, A. Kompanek, R. Melton and D. Garlan, David. *Architectural styles, design patterns, and objects.* IEEE Software, pages 43-52, January, 1997.

[NIERSTRASZ+92] Niertstrasz, O., S. Gibbs, and D. Tsichritzis, *Component-Oriented Software Development.* CACM, vol. 35, no. 9, September 1992, pp. 160-165.

[NING98] J. Q. Ning, CBSE Research at Andersen Consulting, Proceedings of the 1st International Workshop on Component-Based Software Engineering, April 1998.

[NING99] J. Ning, *A Component Model Proposal.* 1999 International Workshop on Component-Based Software Engineering, paper ,May 1999.

[PERRY92] Perry, D.E. & Wolf, A.L. *Foundations for the Study of Software Architecture.* Software Engineering Notes, ACM SIGSOFT 17, 4 (October 1992): 40-52.

[PORTER92] H. Porter, *Separating the Subtype Hierarchy from the Inheritance of Implementation.* Journal of Object-Oriented Programming, 4(9):20-29, February 1992.

[PRESSMAN01] Pressman, R.S., *Software Engineering, a practitioner's approach.* Fifth edition, McGrawHill, 2001.

[SHAW+96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SILVA03] Silva Jr., M.C., *COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos.* Dissertação de Mestrado, IC/Unicamp, 2003.

[SOMMERVILLE01] I. Sommerville *Software Engineering.* Addison-Wesley, 2001

[SONI95] Soni, D.; Nord, R.; & Hofmeister, C. *Software Architecture in Industrial Applications.* 196-210. Proceedings, 17th International Conference on Software Engineering.Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995.

[SZYPERSKI97] C. Szyperski. *Component Software - Beyond Object Oriented Programming.*

Addison-Wesley, 1997.

[TAI98] S. Tai, *A connector model for Object-Oriented Component Integration.* 1998 International Workshop on Component-Based Software Engineering.

[UML03] *OMG Unified Modeling Language Specification - Version 1.5.* OMG document formal, 2003.

[UML4EDOC] *UML Profile for Enterprise Distributed Object Computing Specification.* ptc/2001-12-04.

[XMI02] *OMG XML Metadata Interchange (XMI) Specification - Version 1.2.* OMG document formal, 2002.

[YOURDON94] Yourdon, E., *Application Development Strategies.* Vol.6, no. 12, 1994.