FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LÚCIO FELIPPE DE MELLO NETO
MARCELO JOSÉ DE MORAIS FILHO

ESTUDO E APLICABILIDADE DO DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTE

MARÍLIA
2005

LÚCIO FELIPPE DE MELLO NETO
MARCELO JOSÉ DE MORAIS FILHO

ESTUDO E APLICABILIDADE DO DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTE

Monografia apresentada ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", Mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora:
Profª. Drª. Maria Istela Cagnin

MARÍLIA
2005

LÚCIO FELIPPE DE MELLO NETO
RA. N.º 296211
MARCELO JOSÉ DE MORAIS FILHO
RA. N.º 301450

ESTUDO E APLICABILIDADE DO DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTE

BANCA EXAMINADORA DA MONOGRAFIA
PARA OBTENÇÃO DO GRAU DE BACHAREL
EM CIÊNCIA DA COMPUTAÇÃO.

CONCEITO FINAL: _____(_____)

ORIENTADORA: _____
Profª. Drª. Maria Istela Cagnin

1º EXAMINADOR: _____
Profª. Drª. Kalinka R. L. J. Castelo Branco

2º EXAMINADOR: _____
Prof. Dr. Valter Vieira de Camargo

Marília, 24 de Novembro de 2005.

AGRADECIMENTOS

Agradecemos primeiramente a Deus pelas nossas vidas e pela capacidade que nos concedeu para a realização deste trabalho.

Agradecemos aos nossos pais Lúcio Felippe de Mello Júnior e Marília Isaura Telles de Mello, Marcelo José de Morais e Rute Machioni pela compreensão e auxílio em todas as etapas de nossas vidas.

Agradecemos aos nossos irmãos André Telles de Mello e Márcio Rogério de Morais pela cooperação nas horas que necessitávamos utilizar o computador e pelo apoio nos motivando a seguir em frente nos momentos mais difíceis.

Agradecemos a compreensão e auxílio da Gabriela da Rocha Barbosa.

Agradecemos a professora Janaína Cintra Abib que nos orientou no início deste trabalho e a todos os professores da graduação pelos ensinamentos relevantes à nossa formação acadêmica.

Agradecemos especialmente à nossa orientadora e amiga Maria Istela Cagnin por nos orientar com muita paciência, dedicação e compreensão. Agradecemos também por todas as sugestões e questionamentos pertinentes que auxiliaram o andamento deste trabalho.

Agradecemos também a todos os amigos e pessoas que de alguma forma contribuíram para a realização deste trabalho.

MELLO NETO, Lúcio Felippe de; MORAIS FILHO, Marcelo José de. **Estudo e Aplicabilidade do Desenvolvimento de Software Baseado em Componente.** 2005. 136 f. Trabalho de Conclusão de Curso – Centro Universitário Eurípides Soares da Rocha, Marília, 2005.

RESUMO

Este trabalho propõe o estudo dos conceitos relacionados ao desenvolvimento baseado em componentes (DBC), bem como a construção de uma aplicação baseada em componentes EJB (*Enterprise JavaBeans*), a fim de se realizar uma comparação entre implementações de componentes de software com e sem o uso do padrão de projeto *Session Façade*. Como procedimento metodológico foi necessário desenvolver a pesquisa em fases. Numa primeira fase levantou-se a teoria pertinente relacionada a componentes de software e a criação de componentes na linguagem de programação Java. A segunda fase consistiu da escolha, estudo e configuração das ferramentas utilizadas, que são: Eclipse 3.0.2 (ferramenta para o desenvolvimento dos componentes EJB e da aplicação do cliente), JBOSS 4.0.1sp1 (servidor de aplicação em que os componentes são executados) e banco de dados MySQL 5.0. A terceira fase tratou da criação dos componentes de acesso ao banco de dados e a construção das aplicações cliente. A primeira e a terceira fases foram apoiadas pelo método de desenvolvimento baseado em componentes denominado *UML Components*. A aplicação que foi desenvolvida, com e sem o uso de um padrão de projeto, consiste do cadastro de cliente, funcionário e o controle das vendas efetuadas em uma empresa comercial. Como principais resultados tem-se: desenvolvimento da aplicação três camadas com componentes EJB e melhor desempenho da aplicação que utiliza o padrão de projeto *Session Façade*. Pôde-se concluir que é importante a utilização de padrões de projeto e que o tempo gasto na criação do componente pode ser recompensado com a posterior reutilização do mesmo.

Palavras-chave: DBC, componente de software, *UML Components*, EJB, padrões de projeto.

MELLO NETO, Lúcio Felippe de; MORAIS FILHO, Marcelo José de. **Estudo e Aplicabilidade do Desenvolvimento de Software Baseado em Componente.** 2005. 136 f. Trabalho de Conclusão de Curso – Centro Universitário Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

This work proposes the study of concepts related to component based development (CBD), as well as the construction of an application based on EJB (Enterprise JavaBeans) components, in order to do a comparison between implementations of software components with and without the use of Session Façade design pattern. As a methodological procedure it was necessary to develop the research in phases. In the first phase a pertinent theory related to the components of software and the creation of components in the programming language Java was risen. The second phase consisted of the choice, study and configuration of the tools used, which are: Eclipse 3,0,2 (tool for the development of EJB components and the client application), JBOSS 4.0.1sp1 (Application Server where the components are ran) and MySQL 5.0 database. The third phase dealt with the creation of the component access to the database and the construction of the clients applications. The first and third phases had been supported by the method of development based on components called UML Components. The application that was developed, with and without the use of a design pattern, consists on the client and employee recordes as well as the sales performed in a commercial bussines. Main results were: development of the three-layer application with EJB components and better performance of the application that uses the Session Façade design pattern. It was concluded that the use of design patterns is important and that the time spent in the creation of the component can be made up by its further re-use.

Keywords: CBD, software component, *UML Components*, EJB, design patterns.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Progamming Interface*

BMP – *Bean-Managed Persistence*

CBSE – C*omponent-Based Software Engineering*

CMP – *Container-Managed Persistence*

CORBA – *Commo Object Request Broker Architecture*

COTS – *Commercial off the Shelf*

DBC – Desenvolvimento Baseado em Componentes

DER – Diagrama Entidade-Relacionamento

DNS – *Internet Domain Name System*

ED – Engenharia de Domínio

EJB – *Enterprise JavaBeans*

FODA – *Feature Oriented Domain Analysis*

FORM – *Feature-Oriented Reuse Method*

IP *Internet Protocol*

JDBC – *Java DataBase Connectivity*

JMS – *Java Message Service*

JNDI – *Java Naming and Directory Interface*

J2EE – *Java 2 Enterprise Edition*

OO – Orientação a Objeto

RMI – *Remote Method Invocation*

RMI-IIOP – *Remote Method Invocation over Internet Inter-ORB Protocol*

RPC (*Remote Procedure Call*)

RUP – *Rational Unified Process*

SEI – *Software Engineering Institute*

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

XML – *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto

Atualmente, o desenvolvimento de curto prazo é uma tendência que muitas empresas estão adotando para construir seus aplicativos. Com o objetivo de conseguir redução nos custos e obter a aplicação final com menor tempo de desenvolvimento, grandes aplicações estão sendo criadas a partir de partes menores, previamente testadas, chamadas de componentes.

Os componentes são elementos de software reutilizáveis que fornecem uma funcionalidade específica dentro de um contexto. Dessa forma, são independentes dos aplicativos em que estão inseridos, pois são feitos para atender uma função específica e não para atender um único aplicativo.

Para desenvolver uma aplicação com apoio de componentes, é necessário que os componentes se comuniquem por meio de métodos especificados em suas interfaces (BASS *et al*., 2001; BACHMANN *et al*., 2000; SOMMERVILLE, 2003; D'SOUZA e WILLS, 1999; PRESSMAN, 2002). Os componentes devem seguir certas regras, definidas por um modelo de componentes, que proporciona a criação uniforme dos componentes.

A tecnologia EJB (*Enterprise JavaBeans*) é um modelo de componente que fornece uma padronização para a criação de componentes na linguagem Java, possibilitando a comunicação e a interoperabilidade dos mesmos. É uma arquitetura para o desenvolvimento de aplicações baseadas em componentes (DEMICHIEL, 2003). Os componentes EJB, criados sob esta tecnologia, possibilitam a construção de aplicações robustas e distribuídas (ROMAN; SRIGANESH; BROSE, 2005).

Para permitir o desenvolvimento de aplicações com a tecnologia de componentes há a Engenharia de Software Baseada em Componentes (*compnent-based software engineeiring,* CSBE) (PRESSMAN, 2002) que permite o desenvolvimento com a integração de componentes reutilizáveis. Essa abordagem tenta maximizar o reúso de softwares já existentes, com objetivo de obter eficiência no processo de desenvolvimento e qualidade no produto final. Essas vantagens são adquiridas a partir do Processo de Desenvolvimento Baseado em Componentes, que é composto pela Engenharia de Domínio e pelo Desenvolvimento Baseado em Componentes.

A literatura de Engenharia de Software Baseada em Componentes dispõe de vários métodos para a documentação de aplicações desenvolvidas com a integração de componentes de software reutilizáveis tais como: *Feature Oriented Domain Analysis* (FODA) (KANG 1990 apud TRIGAUX e HEYMANS, 2003), *Form-Oriented Reuse Method* (FORM) (KANG 1999 apud TRIGAUX e HEYMANS, 2003), *Rational Unified Process* (RUP) (KRUCHTEN, 2000 apud BARROCA; GIMENES e HUZITA, 2005), *Catalysis* (D´SOUZA e WILLS, 1998 apud WERNER e BRAGA, 2005) e UML *Components* (CHEESMAN e DANIELS, 2001 apud BARROCA; GIMENES e HUZITA, 2005). Dentre eles, o UML *Components*, utilizado neste trabalho, que apóia o Desenvolvimento Baseado em Componentes estruturando os sistemas em quatro camadas distintas: **interfaces com o usuário**, **comunicação com o usuário**, **serviços e sistemas** e **serviços de negócios**.

1.2 Objetivos

Este trabalho tem como objetivo estudar os conceitos relacionados ao desenvolvimento baseado em componentes e a elaboração dos componentes de software. Além disso, visa também o desenvolvimento de uma aplicação baseada em componentes de

software utilizando a linguagem de programação Java (componentes EJB), com e sem o uso de padrões de projeto (GAMMA *et al*., 1995) para realizar uma comparação entre as implementações a fim de verificar as principais diferenças.

1.3 Estrutura da Monografia

O primeiro capítulo da monografia fornece ao leitor uma compreensão dos principais aspectos relacionados aos componentes de software. Abrange definições, características e benefícios dos componentes que embasam esta monografia.

No segundo capítulo é apresentada a teoria acerca do desenvolvimento de software baseado em componente, descrevendo resumidamente os principais métodos encontrados na literatura.

Na terceira parte deste trabalho encontra-se uma descrição sobre a tecnologia EJB utilizada para a construção dos componentes. São descritas as características relevantes que a especificação EJB fornece para a construção dos componentes de software na linguagem Java.

No quarto capítulo é apresentada a documentação da aplicação construída utilizando o método de desenvolvimento baseado em componentes UML *Components*.

O quinto capítulo versa sobre a construção da aplicação abrangendo a descrição da arquitetura criada com e sem a utilização do padrão de projeto *Session Façade*. E, por último, há uma discussão sobre as características e diferenças das implementações realizadas.

Finalmente, no sexto e último capítulo, são discutidas as conclusões deste trabalho e os possíveis trabalhos futuros.

2 COMPONENTES DE SOFTWARE

2.1 Considerações Iniciais

Este capítulo apresenta parte da revisão bibliográfica, abordando conceitos importantes referentes aos componentes de software que embasam o desenvolvimento do trabalho. Na Seção 2.2 é apresentada a definição de componente, enquanto que na Seção 2.3 abordam-se diferentes definições sobre os componentes de software, suas características e vantagens. Na Seção 2.4 são apresentados os aspectos relevantes sobre as interfaces dos componentes de software. Na Seção 2.5 são relatadas as características de modelos e *frameworks* de componentes. Na Seção 2.6 são apresentadas características relevantes que o componente deve possuir para que o mesmo seja reutilizado. Por último, na Seção 2.7, encontram-se as considerações finais deste capítulo.

2.2 Definição de Componente

Atualmente, o processo de desenvolvimento de curto prazo é uma tendência que, cada vez mais, está se instalando na construção de aplicativos. Com o objetivo de conseguir redução nos custos e obter a aplicação final com menor tempo de desenvolvimento, os aplicativos grandes e complexos estão sendo construídos por meio de uma série de partes menores, conhecidas como componentes, os quais podem ser desenvolvidos em estruturas de tempo mais realistas (MORISSEAU-LEROY; SOLOMON; BASU, 2001).

Os aplicativos são montados com partes menores, previamente testadas e confiáveis, que agilizam seu processo de desenvolvimento facilitando o trabalho das equipes de desenvolvimento e obtendo redução de custos.

2.3 Componente de Software

Um componente “é uma unidade de software independente em nível de aplicativo, desenvolvida para um propósito específico e não para um aplicativo específico” (MORISSEAU-LEROY; SOLOMON; BASU, 2001).

Neste contexto é válido reiterar que componentes são elementos de software reutilizáveis que fornecem uma funcionalidade específica dentro de um contexto. Dessa forma, os componentes são independentes dos aplicativos em que estão inseridos, pois são feitos para atender uma função específica e não para atender um único aplicativo.

Brown e Wallnau (1996 apud PRESSMAN 2002, p. 705) apresentam as seguintes definições sobre componentes:

- Componente – uma parte não-trivial de um sistema, praticamente independente e substituível, que preenche uma função clara no contexto de uma arquitetura bem-definida.
- Componente de software em execução – um pacote dinamicamente constituído de um ou mais programas, geridos como uma unidade, ao qual se tem acesso através das interfaces documentadas, que podem ser identificadas durante a execução.
- Componente de software – uma unidade com dependências de contexto apenas explícitas e contratualmente especificadas.
- Componente de negócio – implementação, em software, de um conceito de negócio ou processo de negócio autônomo.

De acordo com Bachmann *et al*. (2000) existem na literatura várias definições para componentes. Em sua analogia apresenta que, metodologistas de software igualam componentes com unidades de projeto e gerenciamento de configuração; arquitetos de

software igualam componentes com planos de abstração; defensores de software reutilizáveis igualam componentes a qualquer coisa que possa ser reutilizada; profissionais liberais que utilizam componentes pronto para uso *(commercial off-the-shelf* – COTS) igualam componentes de software a produtos COTS.

O termo COTS, segundo Sommerville (2003), significa produtos de prateleira, e pode, em princípio, se aplicar a qualquer componente oferecido por um terceiro, ou seja, um fabricante. Contudo, ele é normalmente mais utilizado para se referir a produtos de software de sistema.

Para Sommerville (2003) os componentes são mais abstratos do que as classes de objetos e podem ser considerados provedores de serviços *stand-alone*. Quando um sistema precisa de algum serviço, ele chama um componente para fornecer esse serviço, sem se preocupar de onde esse componente está sendo executado ou com a linguagem de programação utilizada para desenvolver tal componente.

Sommerville (2003) apresenta o seguinte exemplo:

> "Um componente muito simples pode ser uma função matemática individual que calcula a raiz quadrada de um número. Quando um programa requer um cálculo de raiz quadrada, chama o componente que fornece esse cálculo. No outro extremo da escala, um sistema que precisa realizar algum cálculo aritmético pode chamar um componente de planilha de cálculo, que fornece um serviço de cálculo" (SOMMERVILLE, 2003, p. 263).

Normalmente, os componentes são objetos comerciais que têm comportamentos predefinidos e reutilizáveis. Os detalhes da implementação ficam ocultos nas interfaces, que isolam e encapsulam um conjunto de funcionalidades.

Bass *et al*. (2001) traz uma definição de componente de software como uma implementação, em software, de alguma funcionalidade. É reutilizado em diferentes aplicações, acessado por meio de interfaces e pode ser vendido como um produto comercial.

2.3.1 Diferença entre Componente de Software e Objeto

É importante ressaltar a diferença entre o termo componente e o termo objeto, embora na literatura exista o uso dos termos indistintamente.

As definições trazidas por Morisseau-Leroy; Solomon; Basu (2001, p. 5) apresentam um objeto como uma instância de uma classe que é criada durante a execução. Já um componente pode ser uma classe, mas normalmente é uma coleção de classes e interfaces. Os autores enfatizam que, durante a execução, um componente se torna vivo quando suas classes são instanciadas, portanto, um componente se torna uma teia de objetos.

Nessa mesma linha de pensamento, D'Souza e Wills (1998, p. 390) também diferenciam componentes de objetos. Para os autores componentes são artefatos de software e representam o trabalho dos desenvolvedores e suas ferramentas. Enquanto objetos são instâncias identificáveis criadas por códigos em execução que fazem parte de algum componente. Portanto, de acordo com os autores, um objeto não é um componente.

2.3.2 Vantagens dos Componentes de Software

Existem algumas vantagens na utilização de componentes de software. Sob a ótica de Morisseau-Leroy; Solomon; Basu (2001) as vantagens são múltiplas e se constituem em:

- Independente – Um componente é mais genérico e não depende de um aplicativo específico.
- Reusabilidade – Um componente pode ser reutilizado em diversas aplicações.
- Personalização – Existe a possibilidade de personalização dos componentes para que atendam a requisitos específicos.

- Montagem – Diversos componentes podem ser montados para a criação de aplicações.
- Fácil atualização – os componentes podem ser substituídos individualmente por outros componentes, não afetando as demais partes do conjunto.
- Distribuído – os componentes podem ser distribuídos na rede de uma empresa com o uso de padrões de sistemas de computação distribuída, como por exemplo, o *Enterprise JavaBeans* que será abordado no capítulo 4.

2.4 Interface

Segundo Gimenes e Huzita (2005), os componentes possuem pontos de interconexão chamados de interfaces que agrupam um conjunto de serviços relacionados. Elas garantem o encapsulamento dos dados e processos de um componente.

As interfaces são o meio pelo qual os componentes se conectam com outros componentes e habilitam a comunicação entre um cliente e o componente.

Quando um componente é visualizado como um provedor de serviços, Sommerville (2003) aponta que existem duas características importantes de um componente reutilizável. São elas:

1 – o componente é uma entidade executável independente; o código fonte não está disponível, de modo que o componente não é compilado com outros componentes do sistema.

2 – os componentes publicam sua interface e todas as interações são feitas por meio dessa interface; a interface do componente é expressa em termos de operações parametrizadas e seu estado nunca é exposto.

Pode-se concluir que, um cliente ou um componente que queira se comunicar ou utilizar um outro componente, só conseguirá obter sucesso através das interfaces especificadas e não terá acesso direto a este componente.

Os componentes são definidos por suas interfaces e nos casos mais genéricos, podem ser imaginados como duas interfaces relacionadas. Na Figura 2.1 é mostrada a representação gráfica das interfaces de um componente:

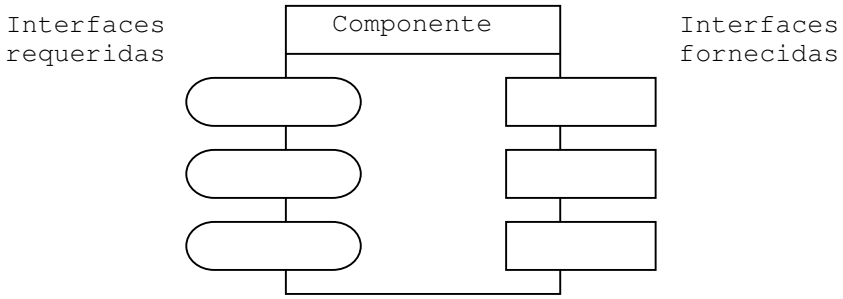

Figura 2.1 – Interfaces de componente (adaptado de Sommerville, 2003).

As interfaces denominadas fornecidas (*provided interfaces*) definem os serviços fornecidos pelo componente. Os serviços fornecidos são os serviços que o componente pode fornecer para o ambiente, tanto para um cliente utilizar como para um outro componente utilizar.

As interfaces denominadas requeridas (*required interfaces*) especificam quais serviços devem estar disponíveis a partir do sistema que está utilizando o componente. Estes serviços disponíveis são os serviços que o componente necessita para funcionar, ou seja, o sistema em que o componente está inserido deve fornecer tais serviços para o componente.

Gimenes e Huzita (2005) relatam que não há a necessidade da existência de interfaces requeridas caso o componente seja autocontido.

2.4.1 Níveis de Abstração de Componentes

Segundo Sommerville (2003) os componentes podem existir em diferentes níveis de abstração, desde uma simples sub-rotina de biblioteca até uma aplicação completa como o Excel, da Microsoft.

Meyer (1999 apud SOMMERVILLE, 2003) aponta diferentes níveis de abstração. São eles:

- Abstração funcional – o componente implementa uma única função, como uma interface matemática. Essencialmente a interface fornecida é a própria função;
- Agrupamentos casuais – o componente é uma coleção de entidades inadequadamente relacionadas, que podem ser declarações de dados, funções, entre outras. A interface fornecida consiste em nomes de todas as entidades do agrupamento.
- Abstrações de dados – O componente representa uma abstração de dados ou classe em uma linguagem orientada a objetos. A interface fornecida consiste em operações para criar, modificar e acessar a abstração de dados;
- Abstrações em *clusters* – o componente é um grupo de classes relacionadas que trabalham em conjunto, chamadas, às vezes, de *framework*. A interface fornecida é a composição de todas as interfaces fornecidas dos objetos que constituem o *framework*. Maiores informações sobre *framework* na Seção 2.5.
- Abstração de sistema – O componente é um sistema inteiramente autocontido. Reutilizar abstrações de nível de sistema é, às vezes, chamado de reúso de produtos COTS. A interface fornecida é chamada de API

(A*pplication Programming Interface* – interface de programação de aplicações) que é definida para permitir que os programas acessem os comandos e as operações do sistema.

2.5 Modelos e *Frameworks*

De acordo com Bachmann *et al*. (2000) um modelo de componente especifica as convenções e os padrões impostos aos desenvolvedores de componentes. Um componente deve estar de acordo com o modelo de componentes e isto é uma das características em que o componente se difere de outros pacotes de software.

Bachmann *et al*. (2000) relata que com estas convenções e padrões espera-se obter as seguintes características:

- Composição uniforme – dois componentes podem interagir, se e somente se, eles compartilharem suposições consistentes sobre o que cada componente requer e fornece para o outro. Os componentes têm que saber qual é a função que o outro componente executa. Existem também outras suposições que podem ser padronizadas, como por exemplo, qual protocolo de comunicação é usado, como os componentes são localizados, como o fluxo de controle é sincronizado, como os dados são codificados, e assim por diante.
- Atributos de qualidade apropriados – é estabelecido uma padronização dos tipos de componentes utilizados em um sistema e seus padrões de interação, garantindo que um sistema composto por componentes de terceiros irá possuir os atributos de qualidade desejados.
- Distribuição de componentes e aplicações – é estabelecida uma pré-condição para a composição, em que há a necessidade de saber quais componentes

> podem ser distribuídos a partir do ambiente do desenvolvedor para o ambiente de composição, e que aplicações construídas com componentes podem ser distribuídas a partir do ambiente de composição para o ambiente do cliente.

Verifica-se que as regras estabelecidas pelo modelo de componentes devem ser seguidas pelos componentes para se obter uma padronização e facilidade para a composição.

Para Weinreich e Sametinger (2001 apud ROSSI, 2004) um modelo de componentes define um conjunto de padrões para implementação, nomeação, interoperabilidade, customização, evolução e disposição de componentes. Para os autores, um modelo de componentes também define os padrões para o modelo de implementação dos componentes associados e o conjunto de entidades de software executável, que é preciso para fornecer suporte à execução dos componentes que estão de acordo com o modelo.

De acordo com Bachmann *et al*. (2000) um *framework* é um conjunto de serviços implementados para apoiar o modelo de componentes, ou seja, é uma infra-estrutura de sustentação para o modelo de componentes.

Este mesmo autor faz uma comparação em que o *framework* está para os objetos assim como um sistema operacional está para os processos. Assim, o *framework* pode ser visto como um mini-sistema operacional que atua diretamente sobre os componentes, possuindo a capacidade de fornecer meios para a comunicação dos componentes; gerenciar os recursos compartilhados pelos componentes; iniciar, suspender ou terminar a execução dos componentes.

Taligent (1997 apud CAGNIN, 2005) considera que um *framework* é um conjunto de blocos de construção de software pré-fabricados que programadores podem usar, estender ou ajustar para encontrar soluções específicas de computação. Este mesmo autor relata que, com

o *framework*, não há a necessidade dos programadores começarem do nada todas as vezes que iniciarem a construção de uma aplicação.

Para Sommerville (2003) um *framework* é uma estrutura genérica na qual pode ser ampliada com o intuito de se obter uma aplicação mais específica ou de se criar um subsistema. Ampliar o *framework* pode acontecer com a introdução de classes concretas que herdam operações de classes abstratas de um *framework*.

Pode-se concluir que um *framework* de componentes é um conjunto de classes relacionadas e de suas interfaces resultando em uma estrutura genérica que fornece suporte ao modelo de componentes (Figura 2.2). Ele fornece ainda a característica de reúso possibilitando que o desenvolvedor já tenha uma base para começar o desenvolvimento baseado em componente e que não gaste tempo começando a partir do zero.

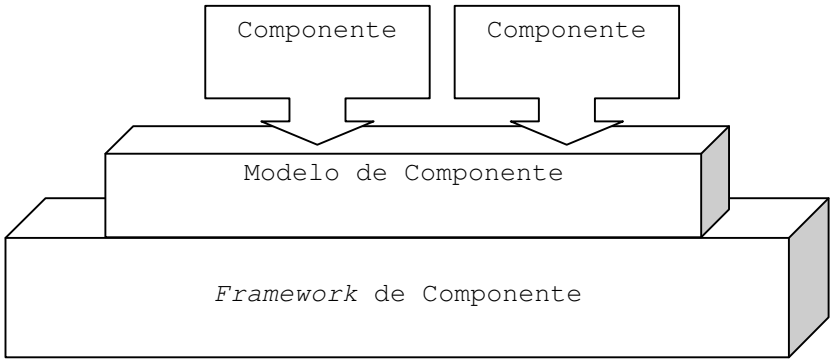

Figura 2.2 – *Framework* de Componente.

2.6 Características dos Componentes para Reúso

Uma característica dos componentes, já vista anteriormente, é a sua capacidade de ser reutilizável, ou seja, os componentes são artefatos de software que podem ser utilizados em uma variedade de aplicações. Mas para que isso ocorra, é importante que o componente

seja criado com algumas características que o levem à possibilidade de reúso. Sommerville (2003) apresenta algumas dessas características.

Sobre abstrações de domínio, o autor enfatiza que o componente deve refletir abstrações[1] estáveis de domínio, as quais se modificam aos poucos.

Outra característica importante é que o componente deve ocultar como o seu estado é representado e fornecer meios para que o estado seja alterado. Por exemplo, para um componente que representa uma conta bancária, deveriam existir as operações de atualizar o saldo, consultar o saldo, entre outras.

Sobre a independência do componente, Sommerville (2003) reforça que o ideal seria que o componente não necessitasse de nenhum outro componente para poder funcionar, mas isto só acontece com componentes mais simples. Os componentes mais complexos geralmente têm relações e dependências com outros componentes, sendo assim, deve-se minimizar ao máximo estas dependências.

As exceções[2], segundo Sommerville (2003), devem ser parte da interface dos componentes, e estes não devem manipular as próprias exceções. Já que os componentes podem funcionar em diferentes aplicações e cada aplicação pode ter seu próprio mecanismo de tratamento de exceções, torna-se necessário que o componente defina as exceções que podem aparecer, como também, publicá-las na interface. Por exemplo, para um componente que implemente uma estrutura de pilha, devem ser publicadas as exceções de *underflow*, que ocorre quando a pilha está vazia ao tentar retirar um elemento, e *overflow*, quando a pilha está cheia ao tentar adicionar um elemento.

Nota-se a importância das interfaces para os componentes reutilizáveis, pois são elas que encapsulam o funcionamento do componente como também é por meio delas que são feitas as interações entre os componentes. Para um componente se tornar reutilizável a

[1] Entende-se por abstrações de domínio as representações de dados de um determinado contexto.
[2] Segundo Deitel e Deitel (2003), exceção é a indicação de que ocorreu um problema durante a execução de um programa.

interface deve ser genérica, ou seja, não deve ser muito específica a ponto de impossibilitar sua reutilização.

2.7 Considerações Finais

Neste capítulo foram abordados alguns conceitos sobre componentes de software, interfaces, modelo e *frameworks* de componentes. Foram apresentadas as vantagens que os componentes oferecem na construção de aplicações, bem como as características dos componentes de software para reúso.

Uma característica importante no contexto dos componentes é a existência da interface. Ela encapsula toda a programação do componente e fornece as assinaturas dos métodos necessários para que os clientes acessem este componente e que este possa ser reutilizado.

Notou-se a importância do modelo de componentes que fornece uma padronização para a criação de componentes, possibilitando a comunicação e a interoperabilidade dos mesmos. Ressalta-se também a importância dos *frameworks*, mais especificamente *frameworks* de componentes, que fornecem uma base genérica para que o desenvolvedor não comece seu trabalho a partir do zero, aumentando a produtividade das equipes de desenvolvimento de software.

3 DESENVOLVIMENTO BASEADO EM COMPONENTES

3.1 Considerações Iniciais

Este capítulo trata sobre o Desenvolvimento de Softwares Baseado em Componentes. Na Seção 3.2 será descrita a Engenharia de Software Baseada em Componentes e na Seção 3.3, a Reutilização de Software, abrangendo conceitos relevantes disponíveis na literatura.

O Processo de Desenvolvimento de Software Baseado em Componentes será apresentado na Seção 3.4. Na Seção 3.5 será descrita a atividade ED - Engenharia de Domínio e a DBC – Desenvolvimento Baseado em Componentes na Seção 3.6, atividades que ocorrem em paralelo para o desenvolvimento de uma aplicação utilizando DBC.

Por último, na Seção 3.7 será visto alguns dos principais Métodos de ED e de DBC citados na literatura.

3.2 Engenharia de Software Baseada em Componentes

A Engenharia de Software Baseada em Componentes (*component-based software engineering*, CBSE) é um método para a criação de softwares com a composição de componentes previamente programados. Essa abordagem tenta maximizar o reúso de softwares já existentes. Segundo Pressman (2002), CBSE é o "processo que enfatiza o projeto e a construção de sistemas baseados em computador usando "componentes" de software reutilizáveis". Os componentes podem ser produzidos pela própria equipe de desenvolvimento ou adquiridos de terceiros (os chamados COTS – *Commercial off the Shelf*).

Clements (apud PRESSMAN, 2002) descreve a CBSE do seguinte modo:

> "CBSE está mudando o modo pelo qual grandes sistemas de software são desenvolvidos. CBSE incorpora a filosofia "comprar, em vez de construir" abraçada por Fred Brooks e outros. Do mesmo modo que as primeiras subrotinas liberaram o programador de pensar sobre detalhes, CBSE desloca a ênfase da programação de software para a composição de sistemas de software. A implementação deu lugar à integração como foco. Na sua fundamentação está a pressuposição de que há muitos pontos em comum, em vários sistemas grandes de software, para justificar o desenvolvimento de componentes reusáveis para explorar e satisfazer esses pontos em comum."

O processo de desenvolvimento de software baseado em componentes exige uma abordagem de reúso mais organizada, atualmente. Pois, segundo Pressman (2002), "sistemas complexos e de alta qualidade baseados em computador precisam ser construídos em períodos de tempo muito curtos".

3.3 Reutilização de Software

A reutilização de software tem como objetivo a eficiência no processo de desenvolvimento. Os projetistas podem adquirir diversas vantagens por reutilizarem software existente, como: a qualidade do produto que está em desenvolvimento e a redução do trabalho, que resultam no aumento da produtividade.

Sommerville (2004) ressalta que as unidades de software que são reutilizadas podem ser de tamanhos radicalmente diferentes:

- Reúso de sistemas de aplicações: todo o sistema de aplicações pode ser reutilizado pela sua incorporação, sem mudança, em outros sistemas (p. ex:, o reúso de produtos COTS).
- Reúso de componentes: os componentes de uma aplicação variados em tamanho e função podem se reutilizados. Por exemplo, um sistema de gerenciamento de banco de dados em desenvolvimento pode incorporar um

sistema de combinação de padrões, desenvolvido como parte de um sistema de processamento de texto.

- Reúso de funções: os componentes de software que implementam uma única função, como uma função matemática, podem ser reutilizados.

3.3.1 Conceitos e Definições

O conceito de reutilização de software não é novo. Desde que começaram a ser desenvolvidos os primeiros sistemas computacionais, havia um pensamento de reutilizar partes do código em outros sistemas de modo a reduzir a complexidade, os custos e aumentar a qualidade, uma vez que os artefatos reutilizados já foram testados (PRESSMAN, 2002).

A reutilização de software se baseia na programação modular onde se pode fazer o uso de procedimentos, funções, classes pré-existentes e componentes criados por outros, que servirão para que outros literalmente montem suas aplicações.

Na literatura sobre engenharia de software, existem muitas definições para o conceito de reutilização de software. Foram selecionadas aquelas que foram consideradas relevantes, para a apresentação nesta seção.

Basili; Caldiera; Cantone (1992 apud ROSSI, 2004) definem reutilização de software como sendo o uso de um mesmo objeto mais de uma vez, sendo que objeto, neste contexto, pode ser: programas, partes de programas, especificações, requisitos, arquiteturas e planos de teste.

Segundo Sommerville (2004), “o reúso sistemático requer um processo do projeto que considere como os projetos existentes podem ser reutilizados e que organize explicitamente o projeto em torno de componentes de software disponíveis”.

Segundo Lim (1998 apud ROSSI, 2004), a reutilização de software é o uso de bens existentes no desenvolvimento de outro software, com o objetivo de melhorar a produtividade, a qualidade e outros fatores, como por exemplo, a usabilidade. Neste contexto, bens ou componentes são os produtos ou subprodutos do processo de desenvolvimento de software, incluindo tanto elementos tangíveis (código, projeto, algoritmos, planos de teste, e documentação) e elementos intangíveis (conhecimento e metodologia).

3.4 Processo de Desenvolvimento de Software Baseado em Componentes

Segundo o SEI – *Software Engineering Institute*, processo de DBC envolve os passos técnicos para projeto e implementação de componentes de software, montagem de sistemas a partir de componentes de software previamente construídos e distribuição de sistemas para os seus ambientes destinos (ROSSI, 2004). Um outro conceito importante é a engenharia de software baseada em componentes, que em um nível de abstração mais elevado envolve as práticas necessárias para executar o DBC, de modo repetitivo, para construir sistemas que tenham suas propriedades conhecidas (BACHMAN, 2000 apud ROSSI, 2004).

Deste modo, esta abordagem tem como premissa fundamental a construção de um sistema de software a partir do máximo reúso de componentes de software já existentes. Isto contrasta com o processo de desenvolvimento tradicional, em que praticamente todos os elementos são construídos.

Rossi (2004) relata ainda que, como qualquer processo de desenvolvimento, o baseado em componentes envolve uma seqüência lógica de fases constituídas de atividades que transformam os artefatos de entrada em artefatos de saída para obter o sistema final. Entretanto, esta seqüência lógica do modelo de processos de desenvolvimento nem sempre é igual, mas segundo alguns autores da literatura de engenharia de software, possui um conjunto

de fases em comum (BROWN; SHORT, 1997; MCCLURE, 1997; POUR, 1998; PRESSMAN, 2000 apud ROSSI, 2004). Estas fases são:

1. Análise de Requisitos;
2. Aquisição do componente;
3. Entendimento do componente;
4. Adaptação do componente;
5. Composição do componente e;
6. Certificação do componente.

O processo de Engenharia de Software Baseado em Componentes identifica os candidatos a componente e a qualidade da interface de cada componente, adapta os componentes para remover discordâncias arquiteturais, monta componentes num estilo arquitetural selecionado e, modificando seus padrões repetidos com potencial de reúso atualiza os componentes, à medida que os requisitos do sistema se modificam (BRO, 1998 apud PRESSMAN, 2002).

Pressman (2002) relata que "o modelo de processo de software para a engenharia de software baseada em componentes enfatiza caminhos paralelos, nos quais a engenharia de domínio ocorre simultaneamente com o desenvolvimento baseado em componentes".

Na Figura 3.1 ilustra-se um modelo típico de processo que acomoda explicitamente a Engenharia de Software Baseada em Componentes.

A ED cria um modelo do domínio de aplicação, que é usado como base para analisar os requisitos do usuário no curso da engenharia de software. Uma arquitetura de software genérica fornece a entrada para o projeto da aplicação. Finalmente, depois de terem sido adquiridos, selecionados de bibliotecas existentes, ou construídos (como parte da ED), os componentes reutilizáveis são colocados à disposição dos engenheiros de software durante o desenvolvimento baseado em componentes.

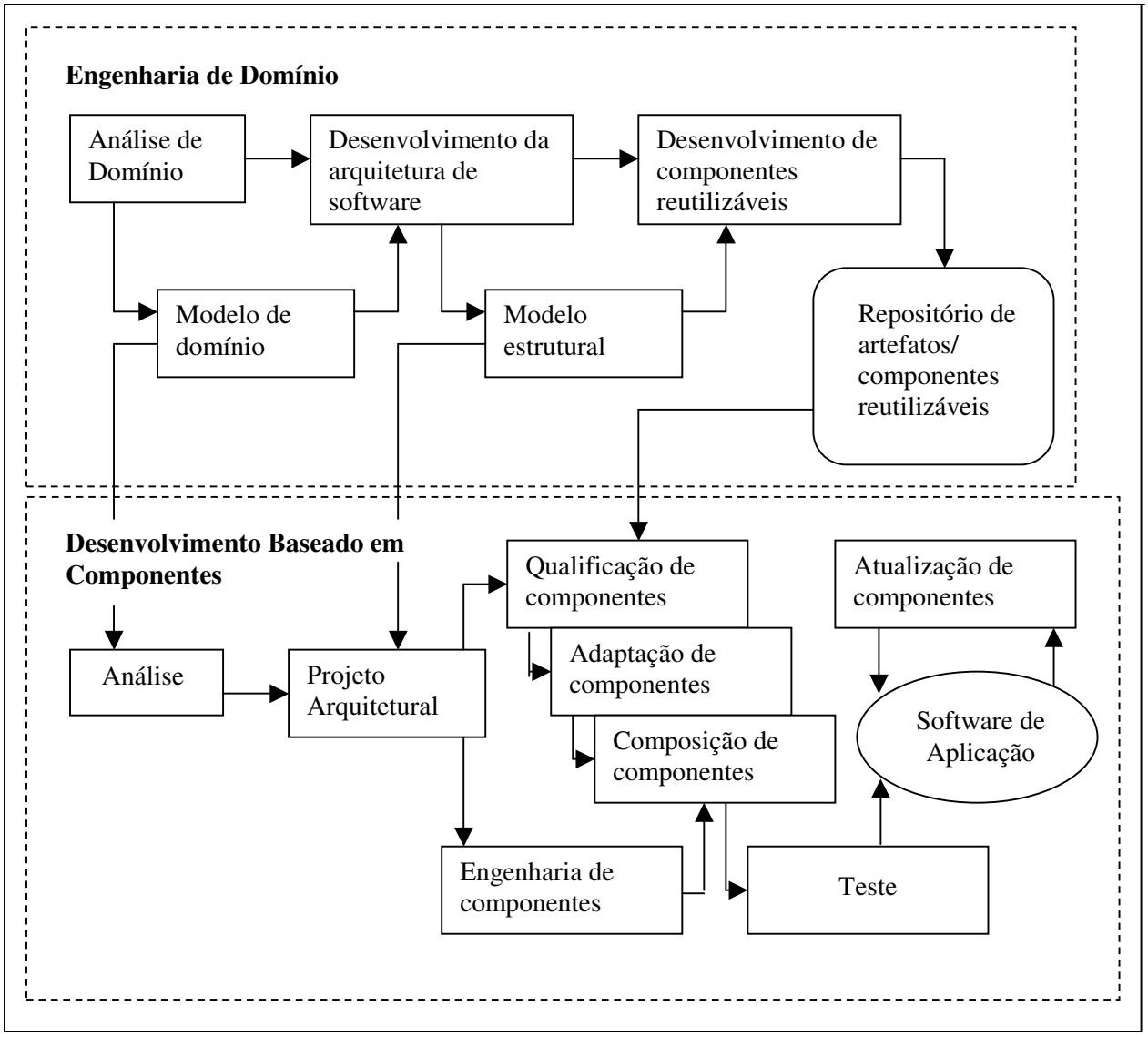

Figura 3.1 – Modelo de Processo que suporta Desenvolvimento de Software Baseado em Componentes (Pressman, 2002, p. 707).

O DBC, paralelamente, analisa os requisitos da aplicação baseando-se no modelo de domínio oferecido pela ED e constitui um projeto arquitetural que será preenchido pelos componentes já disponíveis.

As atividades de ED e DBC constituem o Processo de Desenvolvimento de Software Baseado em Componentes, mas serão descritas, respectivamente, na Seção 3.5 e Seção 3.6.

3.5 Engenharia de Domínio

O grau de reutilização de um componente está intrinsecamente ligado a um domínio específico de aplicações. Um componente será mais passível de reutilização dentro do domínio para o qual ele foi construído. Por isso, a busca por domínios se mostra interessante.

Tracs (1995 apud PRESSMAN, 2002) cita que "o reúso se tornará tão comum que seu termo irá desaparecer, não por eliminação, mas por integração do contexto da prática de engenharia de software". Portanto, à medida que o reúso tem uma maior ênfase, a engenharia de domínio tende a se tornar tão importante quanto a engenharia de software durante a próxima década.

Segundo Pressman (2002), "o objetivo da engenharia de domínio é identificar, construir, catalogar e disseminar um conjunto de componentes de software que tem aplicabilidade a software existente e futuro, num domínio particular de aplicação". Relata ainda que o objetivo é instituir mecanismos que permitam aos engenheiros de software compartilhar esses componentes, ou seja, reusá-los, enquanto estiverem trabalhando com sistemas novos ou com os já existentes. A ED inclui três principais atividades: análise, construção e disseminação.

3.5.1 O Processo de Análise de Domínio

Firesmith (1993 apud PRESSMAN, 2002) descreve a análise de domínio do seguinte modo: "a análise de domínio de software é a identificação, análise e especificação dos requisitos comuns de um domínio de aplicação específico, para reúso tipicamente em vários projetos dentro do domínio de aplicação".

A análise de domínio atua no desenvolvimento de descrições que generaliza uma família de aplicações com características comuns – um domínio de aplicações. A motivação para produzir uma especificação de um domínio de aplicação (modelo do domínio) é a meta de sua reutilização no desenvolvimento de aplicações para este domínio, com perspectiva de aumento de produtividade da atividade de desenvolvimento de software.

Pressman (2002) ressalta que "a análise de domínio é uma atividade contínua de engenharia de software, que não está ligada a um projeto de software específico". Ele compara, com o intuito de assemelhar, o papel de um engenheiro de domínio com o papel de um mestre ferramenteiro num ambiente de intensa manufatura, pois, o ferramenteiro projeta e constrói ferramentas que possam ser usadas por diversas pessoas que não necessariamente desenvolvem o mesmo trabalho, mas sim, um trabalho similar. Portanto, "o papel do analista de domínio é projetar e construir componentes reutilizáveis que possam ser usados por várias pessoas trabalhando em aplicações diferentes, mas não necessariamente as mesmas".

Pode-se dizer, então, que a análise de domínio é uma técnica essencial para a reutilização de software, uma vez que, através deste processo, é realizada a identificação e a criação de um conjunto de componentes reutilizáveis para ser utilizado no desenvolvimento de sistemas em dado domínio. Para isso, deve-se estabelecer um conjunto de componentes de software para reutilização, por meio da identificação e registro das características comuns e variáveis de sistemas em domínios, o que permite identificar os componentes de software candidatos a serem reutilizados.

Na Figura 3.2, mostram-se as entradas necessárias e saídas geradas pela fase de análise de domínio, segundo (MCCLURE, 1997 apud ROSSI, 2004). As entradas necessárias são as seguintes:

- Informação descrevendo os **sistemas existentes** em um domínio (por exemplo, código fonte, documentação de sistema, planos de testes, manual do usuário, modelo de projeto, versões do sistema);
- Perícia e conhecimento a partir de **especialistas no domínio**;
- Informação descrevendo sistemas futuros e planejados (por exemplo, **planejamento estratégico corporativo** e arquitetura de informação);
- Informação a respeito de **tendências tecnológicas**.

De acordo com McClure (1997 apud ROSSI, 2004) as saídas geradas pela análise de domínio são:

- **Modelo de domínio**: modelo de definição de um domínio que descreve os conceitos essenciais (por exemplo, funções, características, dados, classes de objetos, requisitos) e seus relacionamentos;

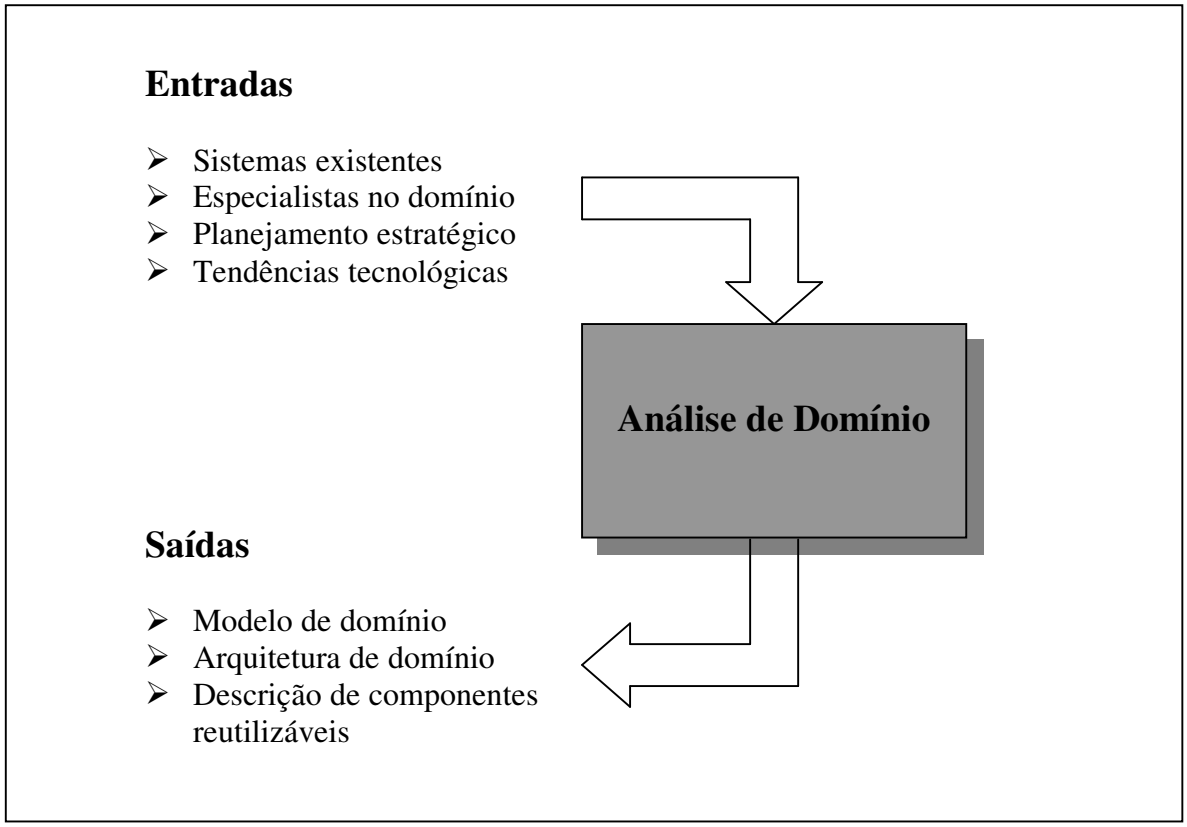

Figura 3.2 – Entradas e saídas da análise de domínio (Rossi, 2004, p. 73).

- **Arquitetura de domínio**: modelos de projeto em alto nível para construção de sistemas em um domínio a partir de componentes reutilizáveis;
- **Descrição dos componentes reutilizáveis**: documentação de uso de componentes para produção de sistemas e projeto de sistemas em um dado domínio.

Segundo Pressman (2002) "o processo de análise de domínio pode ser caracterizado por uma série de atividades que começam com a identificação do domínio a ser investigado e terminam com a especificação dos objetos e classes que caracterizam o domínio". Os passos do processo foram definidos como:

1. **Definir o Domínio.** O analista deve identificar a área de negócio, ou seja, separar a categoria de produto de interesse. Em seguida, devem ser extraídos os itens, que podem, ou não, ser OO – Orientação a Objeto, dessa categoria de produto.
2. **Categorizar os Itens.** Organizar os itens em categorias de acordo com suas características. Quando adequado, são estabelecidas hierarquias de classificação.
3. **Coletar Amostras.** O analista deve atestar de que a atual aplicação possui itens que se encaixam nas categorias já definidas.
4. **Analisar as Aplicações.** Para realizar essa atividade, o analista deve seguir os seguintes passos (BERARD, 1993 apud PRESSMAN, 2002):
   - Identificar os objetos candidatos a reúso.
   - Apontar as características que identificaram o objeto para o reúso.
   - Determinar adaptações a objeto que possam também ser reusadas.
   - Estimar a porcentagem de aplicação do domínio que poderia fazer reúso do objeto.
   - Identificar os objetos por nome e usar técnicas de gerência de configuração para controlá-los.

5. **Desenvolver um Modelo.** Desenvolver o modelo de análise que servirá de base para o objeto e construção dos objetos do domínio.

Pressman (2002) ressalta que "é importante notar que a análise de domínio é aplicável a qualquer paradigma de engenharia de software e pode ser aplicada tanto para o desenvolvimento convencional quanto para o orientado a objetos".

O segundo passo da análise de domínio é dado por Prieto-Diaz (1987 apud PRESSMAN, 2002) que sugere oito passos para a identificação e categorização de componentes reutilizáveis:

1. Selecionar funções ou objetos específicos.
2. Abstrair funções ou objetos.
3. Definir uma taxonomia.
4. Identificar as características comuns.
5. Identificar as relações específicas.
6. Abstrair as relações.
7. Criar um modelo funcional.
8. Definir uma linguagem de domínio.

Os passos acima proporcionam um modelo útil para a análise de domínio, mas não indicam quais componentes de software são candidatos a reúso (PRESSMAN, 2002). Hutchinson e Hindley (1988 apud PRESSMAN, 2002) sugerem o seguinte conjunto de questões pragmáticas como guia para identificar componentes de software reutilizáveis:

- A funcionalidade do componente é necessária em implementações futuras?
- Quão comum é a função do componente dentro do domínio?
- Há duplicação da função do componente dentro do domínio?
- O componente é dependente de hardware?
- O hardware permanece imutável entre implementações?

- As especificidades do hardware podem ser removidas para outro componente?
- O projeto é suficientemente otimizado para a implementação seguinte?
- Podemos adicionar parâmetros a um componente não-reutilizável de modo que se torne reutilizável?
- O componente é reutilizável em várias implementações apenas com pequenas modificações?
- O reúso pela modificação é exeqüível?
- Um componente não-reutilizável pode ser decomposto para produzir componentes reutilizáveis?
- Quão válida é a decomposição de componente para reúso?

3.6 Desenvolvimento Baseado em Componentes

O Desenvolvimento Baseado em Componentes é uma atividade do Processo de Desevolvimento Baseado em Componentes em que preenche por componentes reutilizáveis e apropriados uma arquitetura estabelecida pela equipe de desenvolvimento.

"Quando um sistema precisa de algum serviço ele chama um componente para fornecer esse serviço, sem se preocupar a respeito de onde esse componente está sendo executado ou qual a linguagem de programação utilizada para desenvolver o componente" (SOMMERVILLE, 2003).

A Figura 3.3, abaixo, representa as etapas do DBC.

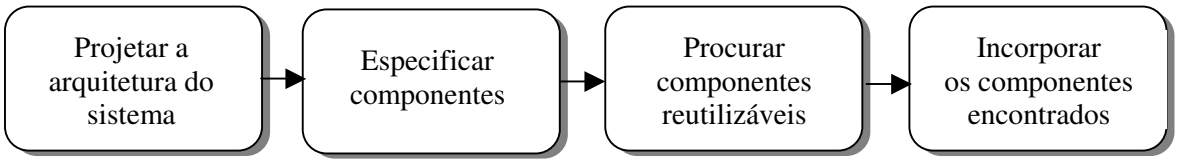

Figura 3.3 – Um processo de reúso "oportunista" (Sommerville, 2003, p. 264).

Os engenheiros de software estabelecem a arquitetura do projeto, que determinará o modelo de análise de componentes candidatos à composição da aplicação (criada durante a engenharia de domínio).

Segundo Pressman (2002), depois de estabelecida essa arquitetura, ela deve ser preenchida por componentes que:

1. estão disponíveis em bibliotecas de reúso e/ou
2. são trabalhados por engenharia para satisfazer as necessidades do cliente

Pressman (2002) explica que "quando os componentes reutilizáveis estão disponíveis para possível integração na arquitetura, eles precisam ser qualificados e adaptados. Quando novos componentes são necessários, precisam ser construídos". Esses componentes são integrados na arquitetura e testados.

Para que uma abordagem consistente de DBC tenha sucesso, é necessário que o engenheiro de software tenha conhecimento e entendimento dos conceitos que estão por trás das finalidades disponibilizadas pelos componentes e seus relacionamentos, um conhecimento concreto sobre componentes reutilizáveis e o uso de um modelo arquitetural que apresente a interação entre os componentes a serem utilizados em um dado contexto (WERNER E BRAGA, 2005).

3.6.1 Qualificação, Adaptação e Composição de Componentes

A seleção de componentes que são integrados nas aplicações em desenvolvimento é feita pela ED, como já foi abordado na Seção 3.6. Os componentes reutilizáveis que serão integrados na aplicação nem sempre garantem uma efetiva integração na arquitetura escolhida para o sistema, portanto, algumas atividades devem ser realizadas no DBC.

3.6.1.1 Qualificação de Componentes

A atividade "Qualificação de Componentes" visa a garantir que um componente realizará sua função proposta, podendo ser integrado efetivamente na arquitetura do sistema, e vai exibir as características de qualidade (p.ex., desempenho, confiabilidade e usabilidade) necessárias para a aplicação (PRESSMAN, 2002).

Segundo Brown e Short (1997 apud ROSSI, 2004), a qualificação do componente pode identificar interfaces que são possíveis fontes de conflito e sobreposição entre componentes. Para contornar este problema, o entendimento do componente deve ser apoiado pela sua documentação, que deve possuir a informação necessária para sua utilização de modo correto, evitando este tipo de conflito.

Para Brown (1996 apud PRESSMAN, 2002), uma relação de fatores devem ser consideradas durante esta fase, que são:

- Interface de programação da aplicação;
- Ferramentas de desenvolvimento e integração exigidas pelo componente;
- Requisitos de execução, incluindo uso de recursos (memória ou armazenamento), tempo ou velocidade e protocolo de rede;
- Exigências de serviço, incluindo interfaces do sistema operacional e apoio por outros componentes;
- Características de segurança, incluindo controles de acesso e protocolo de autenticação;
- Aquisição de projetos embutidos, incluindo o uso de algoritmos específicos numéricos ou não numéricos e;
- Manipulação de exceções.

Para componentes desenvolvidos internamente, ou seja, desenvolvido pela própria equipe de desenvolvimento da aplicação, cada um desses fatores é relativamente fácil de implementar. No entanto, é muito mais difícil avaliar, segundo a lista apresentada, o funcionamento interno de COTS ou componentes de terceiros, porque a única informação disponível pode ser a própria especificação da interface (PRESSMAN, 2002).

Pressman (2002) ressalta que "se boas práticas de engenharia de software foram aplicadas durante o desenvolvimento, as respostas às questões implicadas pela lista podem ser desenvolvidas."

Desde que o componente possua uma documentação adequada, as questões levantadas por essa lista podem ser extraídas a partir da sua documentação (ROSSI, 2004).

3.6.1.2 Adaptação de Componentes

Outra preocupação dos engenheiros de software é com a evolução e adaptação dos componentes para poderem ser reutilizados em diferentes contextos (WERNER E BRAGA, 2005). Mesmo depois de um componente ter sido qualificado para ser reutilizado na arquitetura de uma aplicação, ele pode conflitar em uma ou mais áreas.

A fase de adaptação de componentes ocorre entre o momento em que o engenheiro de software decide utilizar o componente e o momento em que o componente torna-se parte do produto (MILI; MILI; MILI, 1995 apud ROSSI, 2004).

Para assegurar que os conflitos gerados ao reutilizar um componente sejam minimizados, são realizados ajustes baseados em regras. Portanto, o processo de adaptação pode ser dividido em três sub-tarefas, que são (MILI; MILI; MILI, 1995 apud ROSSI, 2004):

- Seleção: caso o componente tenha uma parte variável ou alternativa de implementação explicitamente enumeradas, deve ser selecionada a que é apropriada para o problema em questão;
- Modificação: caso o componente ou qualquer uma de suas variações não for utilizado, deve ser modificado para ser reutilizado;
- Integração: analisa se o componente é compatível com o ambiente.

Além disso, para a atividade de adaptação de componentes uma técnica de adaptação chamada *empacotamento de componente* é freqüentemente usada (BROWN, 1996 apud PRESSMAN, 2002). Essa técnica classifica a atividade de acordo com o grau de acessibilidade da estrutura interna do componente e, segundo Pressman, pode ser de três diferentes abordagens:

- Empacotamento caixa-branca: quando a equipe de software tem pleno acesso ao código-fonte do componente, permitindo que seja modificado com o intuito de remover qualquer conflito.
- Empacotamento caixa-cinza: o código-fonte não pode ser modificado. A biblioteca de componentes fornece uma linguagem para extensão de componentes ou API – *Application Progamming Interface* que permite a remoção ou o mascaramento dos conflitos.
- Empacotamento caixa-preta: não existe linguagem de extensão ou API e o componente é fornecido apenas na forma executável binário. Desta forma, exige a introdução de pré e pós-processamento na interface do componente para remover ou mascarar os conflitos.

Segundo Pressman (2002), “a equipe de software deve determinar se o esforço necessário para empacotar adequadamente um componente é justificado ou se um

componente sob encomenda (projetado para eliminar os conflitos encontrados) deve, em vez disso, ser construído".

Szyperski (1998) e Sametinger (1997) (apud WERNER E BRAGA, 2005) argumentam que, adaptações e evoluções necessárias feitas apenas no nível de interface seriam ideais, portanto, um componente deve ser adaptado e baseado apenas na sua interface externa.

3.6.1.3 Composição de Componentes

A fase de composição de componentes se refere à combinação dos componentes de software que foram qualificados, adaptados e construídos para compor a arquitetura estabelecida para uma aplicação (PRESSMAN, 2002).

Desta forma, considerando que um sistema baseado em componentes é formado por componentes de software de origens diferentes (internas e externas), podem ser encontrados problemas gerados por estas incompatibilidades entre componentes, chamada de arquitetura mal combinada (GARLAN; ALLEN; OCKERBLOOM, 1995 apud ROSSI, 2004).

Portanto, segundo Pressman (2002), para unir os componentes em um sistema em desenvolvimento deve ser estabelecida uma infra-estrutura (usualmente uma biblioteca de componentes especializados), que fornecerá um modelo para a coordenação de componentes e serviços específicos, que permitem aos componentes se coordenarem entre si e realizarem tarefas comuns.

Existem muitos mecanismos para a criação de uma infra-estrutura. Adler (1995 apud PRESSMAN, 2002) apresenta para a criação de uma infra-estrutura efetiva um conjunto de quatro "ingredientes arquiteturais" que devem estar presentes para conseguir a composição do componente:

- **Modelo de intercâmbio de dados**. Refere-se à interação do usuário com a aplicação e transferência de dados (p. ex., arrastar e soltar, cortar e colar). Os mecanismos de intercâmbio de dados vão além da transferência de dados homem-software e componente-componente, permitindo também a transferência entre recursos do sistema (p. ex., arrastar um arquivo para o ícone de uma impressora, para saída).
- **Automação**. Refere-se à interação entre componentes reutilizáveis, ou seja, devem ser implementados diversas ferramentas, macros e *scripts* para que a interação entre os componentes reutilizáveis seja mais fácil.
- **Armazenamento estruturado**. Os dados heterogêneos (p. ex., dados gráficos, voz/vídeo, texto e dados numéricos) devem representar uma única estrutura de dados, ou seja, devem ser organizados e acessados como uma única estrutura de dados ao invés de como uma coleção de arquivos separados.
- **Modelo de objetos subjacente**. O modelo de objetos garante que objetos desenvolvidos em diferentes linguagens de programação, que residem em diferentes plataformas, possam ser interoperacionais, ou seja, o modelo de objetos define um padrão de interoperabilidade para que, através de uma rede, esses componentes possam se comunicar e interagir.

O impacto potencial de reúso e Engenharia de Software Baseada em Componentes é muito grande na indústria de software (PRESSMAN, 2002). Por isso, é muito importante fornecer uma infra-estrutura de comunicação e coordenação através de padrões de infra-estrutura como OMG/CORBA[3], Microsoft COM.[4] ou Sun JavaBean Components[5], que fornecem normas para software baseado em componentes.

[3] www.omg.org
[4] www.microsoft.com/COM
[5] http://java.sun.com/beans

3.7 Métodos de Engenharia de Domínio e Desenvolvimento Baseado em Componentes

Nesta Seção serão apresentados alguns métodos de ED e de DBC, disponibilizados na literatura e considerados relevantes. Para este trabalho não foi utilizado nenhum método de ED, pois foi escolhido para o DBC, o método UML *Components*, Seção 3.7.5, que não apresenta nenhum aspecto de ED.

3.7.1 *Feature Oriented Domain Analysis* (FODA)

O FODA - *Feature Oriented Domain Analysis* foi desenvolvido por Kang (KANG 1999 apud TRIGAUX e HEYMANS, 2003). É um método de análise de domínio, criado no SEI. A descrição completa do processo de análise de domínio o fez tornar-se o método mais popular no início dos anos de 1990.

Na literatura, segundo Werner e Braga (2005), os estudos apresentam o envolvimento do FODA principalmente na modelagem de estudos de caso na fase de análise, mas muitos autores descrevem esse método como um método para dar suporte à reutilização arquitetural.

No desenvolvimento baseado em componentes, a identificação e o entendimento do domínio de um componente são de muita importância. A abordagem adotada pelo FODA é, portanto, bastante interessante, pois auxilia o desenvolvedor nesse aspecto (WERNER E BRAGA, 2005). O método faz uma contribuição significativa aos desenvolvedores, pois o uso de diversas vistas complementares de um domínio o faz obter informação completa sobre esse domínio.

As autoras Werner e Braga (2005) ressaltam ainda que, com o FODA é possível modelar características adicionais ao domínio, como a modelagem de características operacionais disponíveis e as técnicas de implementação. Elas explicam que isso possibilita o desenvolvedor selecionar componentes que possuam finalidades mais adequadas às questões de implementação e ao ambiente operacional.

Segundo Trigaux e Heymans (2003), o método é composto por três fases principais:

- Análise do domínio: o âmbito do foco da análise de domínio é identificar os produtos que formam a linha da aplicação.
- Análise das características: aplica a análise de "atributos comuns e variações" para desenvolver uma lista de funções comuns e uma de variações do domínio.
- Modelo da característica: consiste na representação das características hierárquicas através de diagramas. Podem ser características alternativas, que são especializações de características mais gerais contendo características múltiplas da especialização, ou podem ser características opcionais, que modelam exigências que podem ser incluídas em um produto, mas podem ser saídas de outros.

FODA têm sua modelagem centralizada no domínio de aplicação do componente. Segundo Werner e Braga (2005), o método preocupa-se muito com a geração de código, mas não extrai devidamente todas as abstrações do domínio, pois somente dá ênfase em suas características funcionais. Isso é encarado pelas autoras como uma falha do método, pois além de o método ter sua modelagem centralizada e não ter um detalhamento eficiente em relação à criação de componentes reutilizáveis, as características ditas não codificadas, que facilitam o entendimento dos conceitos pelo reutilizador do componente, são descartadas.

3.7.2 *Form-Oriented Reuse Method* (FORM)

O método FORM – *Feature-Oriented Reuse Method* (KANG, TRIGAUX e HEYMANS, 2003) foi desenvolvido por Kyo C. Kang. É uma extensão do FODA. O FORM cobre a análise de domínio e o desenvolvimento dos recursos do núcleo, no que se diz respeito a componentes, pois foram adicionadas questões mais detalhadas sobre arquiteturas e implementações.

Comparado com o FODA, "o método ficou mais voltado para o detalhamento de como desenvolver componentes a partir das *features* identificadas" (WERNER e BRAGA, 2005). As autoras explicam que, para isso ser possível, Kyo teve de adicionar técnicas como *templates*, geração automática de código, parametrização, entre outros, para o desenvolvimento dos componentes.

Segundo Trigaux e Heymans (2003), o FORM começa com a análise de atributos comuns entre aplicações em um domínio particular. Durante a análise é construído um "modelo de características" que vai agrupar atributos comuns como AND/OR de forma gráfica. AND indica nós com características imperativas e OR indica os nós com características alternativas selecionáveis para várias aplicações.

O FORM herdou do FODA a falha no que se diz respeito à um melhor detalhamento dos conceitos de domínio, pois segundo Werner e Braga (20005), FORM não têm uma descrição mais detalhada de conceitos, apenas de funcionalidades. As autoras levantaram também como falha o aspecto de o método não possuir um suporte automatizado.

3.7.3 *Rational Unified Process* (RUP)

O RUP - *Rational Unified Process* (KRUCHTEN, 2000 apud BARROCA; GIMENES e HUZITA, 2005) "é um processo de engenharia de software desenvolvido pela Rational Software Corporation e tem como base as idéias de Jacobson" (NASCIMENTO, 2005). O método é uma instância do Processo Unificado.

O Processo Unificado é o processo de desenvolvimento de software baseado em componentes. Desde que os componentes possuam interfaces bem definidas, a aplicação pode ser construída através da incorporação dos mesmos. O método é interativo, orientado a objetos, controlado e suporta ferramentas, podendo ser aplicado a todo tipo de desenvolvimento de software.

Jacobson; Booch; Rumbaugh (1999 apud ROSSI, 2004) ressaltam que o Processo Unificado "resultou dos recursos dos métodos mais relevantes de análise e projeto orientados a objetos, e tem como característica, ser um processo muito amplo, que pode ser especializado para diferentes classes de sistema, organizações, projetos e áreas de aplicação".

As melhores práticas de processos de software incorporadas no RUP são (KRUCHTEN, 2000 apud ROSSI, 2004):

- Desenvolvimento iterativo;
- Gerência de requisitos;
- Utilização de arquitetura e componentes;
- Modelagem visual através da UML – *Unified Modeling Language*;
- Qualidade de processo e de produto;
- Gerência de configuração e versões;
- Casos de uso dirigindo muitos aspectos do desenvolvimento;

- Modelo de processo de desenvolvimento que pode ser adaptado e estendido para as características da organização;
- Necessidade de ferramentas de desenvolvimento de software para fornecer suporte ao processo.

O RUP não introduz nenhum conceito novo de modelagem, já que é completamente apoiado pela UML. Segundo Nascimento (2005), todo processo pode ser dividido em quatro fases: concepção, elaboração, construção e transição, com um número arbitrário de iterações. Cada fase agrupa diferentes tipos de atividades.

Dentro do método, o suporte ao DBC é encorajado, mas ainda é extremamente influenciado pela notação UML.

3.7.4 *Catalysis*

O *Catalysis* (D´SOUZA e WILLS, 1998 apud WERNER e BRAGA, 2005) foi desenvolvido na universidade de Brighton Inglaterra, por D'Souza e Wills (NASCIMENTO, 2005). Os descrevem que o *Catalysis* "é um método de desenvolvimento baseado em componentes que cobre todas as fases de um componente, desde a sua especificação até sua implementação".

De acordo com Werner e Braga (2005), todos os modelos são detalhados em sua descrição e a especificação da arquitetura da aplicação é vista com muita preocupação, pois na verdade, é a especificação da colaboração entre os componentes. As autoras ressaltam que essa descrição arquitetural não se prende a nenhum estilo arquitetural específico.

Este método é baseado num conjunto de princípios para o desenvolvimento de software. Dentre eles, pode-se destacar: **abstração**, **precisão**, **refinamento**, **componentes "*plug-in*"** e algumas **leis de reutilização** (NASCIMENTO, 2005).

O princípio da **abstração** guia o desenvolvedor na busca de aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do mesmo. O princípio da **precisão** visa descobrir erros e inconsistências na modelagem. **Refinamentos** sucessivos entre as transições de uma fase para outra auxiliam na obtenção de artefatos cada vez mais precisos e propensos à reutilização. O princípio **componentes "*plug-in*"** suporta a reutilização de componentes, com o intuito de construir outros. Por fim, a principal **lei da reutilização** do método *Catalysis* é não reutilizar código sem reutilizar os modelos de especificações desses códigos.

O processo de desenvolvimento de software utilizando *Catalysis* se divide nas seguintes fases (D´SOUZA e WILLS, 1998 apud ROSSI, 2005):

- Modelo de negócio: nesta fase deve-se entender as necessidades do cliente, capturar as regras de negócios, comportamento e restrições do problema a ser resolvido.
- Especificação do sistema: deve-se identificar as funções do sistema, estabelecer como será a interação do sistema com pessoas e sistemas e detalhar o modelo de negócios com a finalidade de construir interações mais precisas das ações executadas pelo sistema.
- Arquitetura: nesta fase é definida a arquitetura da aplicação para identificar os componentes necessários à implementação do sistema, e também descrever as colaborações e dependências entre os componentes.
- Projeto: são especificados os componentes e o contexto de utilização; nesta fase tem-se uma preocupação com o desacoplamento dos componentes.
- Implementação: nesta fase deve-se definir a implementação interna para os componentes e suas interações de forma que todos requisitos do componente sejam satisfeitos.

Segundo Werner e Braga (2005), o *Catalysis* é considerado como um dos mais completos métodos de DBC, pois o detalhamento do seu processo é bastante consistente.

3.7.5 UML *Components*

O método UML *Components* (CHEESMAN e DANIELS, 2001 apud WERNER E BRAGA, 2005), descrita em Teixeira (2003 apud WERNER E BRAGA, 2005) é uma extensão da UML e apóia o DBC. Segundo Nascimento (2005), UML *Components* é um processo para a especificação de software baseado em componentes, onde os sistemas são estruturados em quatro camadas distintas:

- Interfaces com o usuário;
- Comunicação com o usuário;
- Serviços e sistemas;
- Serviços de negócios.

Estas quatro camadas distintas representam a arquitetura da aplicação a ser desenvolvida (NASCIMENTO, 2005).

Segundo Werner e Braga (2005), a estratégia da abordagem UML *Components* é, a partir da análise de um problema, constituir a arquitetura com componentes. As autoras ressaltam ainda que os componentes devem adotar os princípios da orientação a objetos, ou seja, devem se comportar como unificação de dados e funções, encapsulamento e identidade, para se adequarem ao papel da interface e das especificações, independentes de tecnologia.

No capítilo 5 será dada maior ênfase no método, exemplificando o seu processo com uma aplicação destinada a qualquer loja comercial.

3.7.6 Metodologia para Utilização de Componentes de Software para Ambiente Cliente-Servidor

Esta seção apresenta a metodologia de desenvolvimento visando sistemas cliente-servidor, e é definida considerando os seguintes processos (TAKATA, 1999 apud ROSSI, 2004):

- Desenvolvimento de aplicações: relacionado com a engenharia de software;
- Desenvolvimento de componentes: relacionado com a engenharia de domínios.

Segundo Takata (1999 apud ROSSI, 2004), a existência de dois processos permitiu a segregação de funções em três grupos, que são os seguintes:

- Grupo de Análise e Projeto de Aplicação: composto por pessoas tanto do grupo de desenvolvimento de componentes como do grupo de desenvolvimento de aplicação, o que facilita a transmissão da especificação e dos requisitos entre os grupos, permitindo uma melhor utilização dos componentes existentes;
- Grupo de Desenvolvimento de Componentes: responsável pela engenharia de domínio e pela obtenção/manutenção dos componentes, que pode ser tanto por desenvolvimento interno ou externo;
- Grupo de Desenvolvimento de Aplicação: responsável pela construção de aplicações e também por fornecer consultoria na análise de domínio de negócio, já que possui uma experiência maior com o negócio do que o grupo de componentes.

Por meio da Figura 3.4, podem ser visualizadas as fases de cada um desses processos.

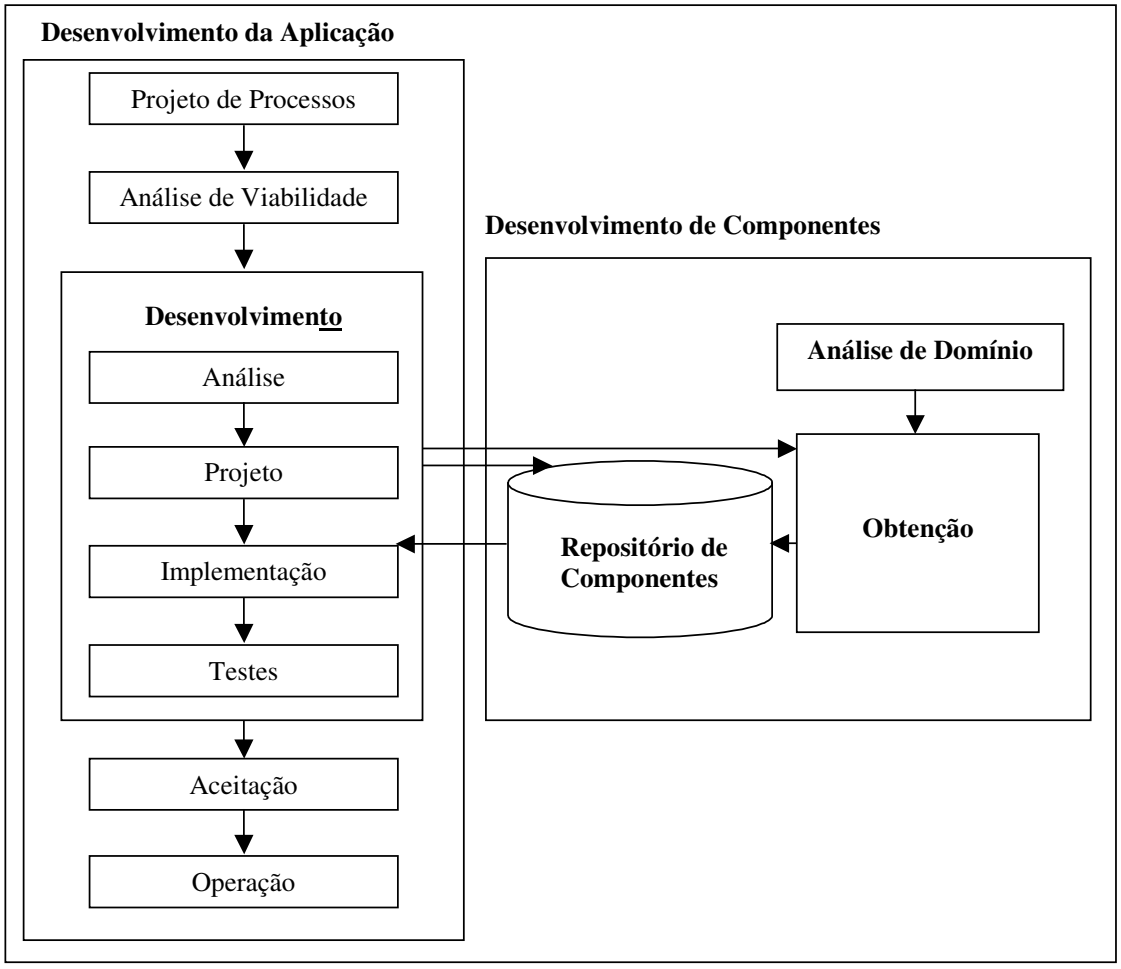

Figura 3.4 – Fases da metodologia proposta por Takata (1999, p. 104 apud Rossi, 2004).

3.8 Considerações Finais

Este capítulo apresentou os conceitos de Desenvolvimento de Software Baseado em Componentes, abrangendo desde os conceitos de reutilização até alguns métodos de desenvolvimento baseado em componentes.

Foram selecionados da literatura algumas definições e alguns conceitos considerados relevantes sobre o que diz respeito a reúso num contexto mais amplo, e não apenas restrito a componentes.

Foi abordado o Processo de Desenvolvimento de Software Baseado em Componentes, que tem como atividades paralelas a Engenharia de Domínio e o

Desenvolvimento Baseado em Componentes propriamente dito. Na atividade Engenharia de Domínio é descrito o Processo de Análise de Domínio e na atividade Desenvolvimento Baseado em Componentes, foram abordados as atividades Qualificação, Adaptação e Composição de Componentes, que devem ser realizadas para garantir uma efetiva integração dos componentes na arquitetura escolhida para o sistema.

Por fim, foram apresentados alguns métodos de Engenharia de Domínio e Desenvolvimento Baseado em Componentes encontrados na literatura e considerados relevantes.

No capítulo a seguir é apresentada a tecnologia EJB utilizada para a construção dos componentes. Esta tecnologia foi escolhida, pois possibilita a criação de componentes na linguagem Java que são executados no servidor. Ela propicia a criação de aplicações distribuídas e robustas e, atualmente, é muito utilizada.

No capítulo 5 será dada uma maior ênfase no método UML *Components* que será utilizado para apoiar a especificação, em UML, dos componentes desenvolvidos neste trabalho.

4 COMPONENTES EJB

4.1 Considerações Iniciais

Este capítulo apresenta a tecnologia EJB (*Enterprise JavaBeans*) utilizada para o desenvolvimento de aplicações baseadas em componentes de software. Na Seção 4.2 as características da arquitetura EJB são apresentadas para que se visualize como os componentes devem estar dispostos na arquitetura. Também são apresentados os benefícios dos componentes EJB e a diferença entre componentes EJB e *JavaBeans*. Na Seção 4.3 são apresentadas as seis funções estabelecidas pela especificação EJB para a construção de uma aplicação EJB. Na Seção 4.4 são citadas algumas das tecnologias existentes que se relacionam com o EJB. Na Seção 4.5 abordam-se os principais conceitos e características para a criação de componentes de software EJB. Na Seção 4.6 abordam-se os três tipos de componentes EJB existentes bem com suas principais características e funções. Na Seção 4.7 é apresentada a teoria sobre padrões de projeto bem como o padrão de projeto utilizado neste trabalho. Finalmente, na Seção 4.8 são apresentadas as considerações finais sobre este capítulo.

4.2 *Enterprise JavaBeans* (EJB)

A Sun Microsystems define a arquitetura EJB 2.1 como uma arquitetura para o desenvolvimento e distribuição de aplicações comerciais distribuídas baseadas em componentes (DEMICHIEL, 2003). Esta versão foi utilizada no desenvolvimento da aplicação deste trabalho.

A especificação EJB é um modelo que deve ser seguido no desenvolvimento de componentes EJB para que exista uma padronização na implementação destes componentes.

Os componentes EJB são componentes executados no servidor e servem para encapsular as regras de negócio, ou seja, as lógicas comerciais de uma aplicação. Para serem executados, os componentes EJB devem estar dentro do *container* EJB (repositório de componentes) que oferece vários serviços aos componentes e proporciona a comunicação destes com o ambiente. Os serviços oferecidos pelo *container* EJB são: segurança, gerenciamento de interações entre os componentes, conectividade com o banco de dados, flexibilidade e mobilidade. A existência de todos estes recursos proporciona ao desenvolvedor uma maior facilidade no processo de desenvolvimento do componente, pois ele se preocupa e se concentra apenas com a lógica comercial que o seu componente deve implementar.

Para que os componentes EJB sejam executados, eles devem estar em um *container* EJB, que por sua vez, necessita ser executado dentro de um servidor EJB. Na Figura 4.1 ilustra-se esta situação.

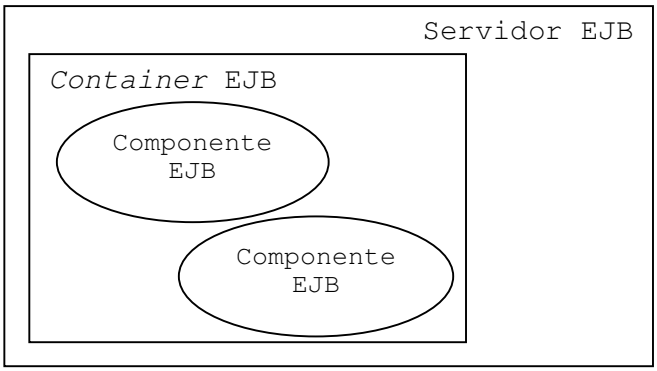

Figura 4.1 – Servidor e *Container* EJB (adaptado de Kurniawan, 2002).

4.2.1 Benefícios de EJB

O EJB proporciona vários benefícios a seus usuários principalmente pela existência do *container* EJB. Kurniawan (2002) aponta alguns benefícios na utilização de EJB:

- O desenvolvedor se concentra em criar o seu componente e aproveita os benefícios que o *container* EJB fornece, como por exemplo, o gerenciamento de transações.
- A especificação EJB assegura que os componentes fornecidos por diferentes desenvolvedores sejam reutilizáveis em outros sistemas facilitando o processo de desenvolvimento da aplicação.
- Existe uma distinção nas etapas de desenvolvimento, distribuição e administração de um aplicativo EJB que facilita e agiliza o processo de criação.
- A arquitetura EJB é compatível com outras APIs Java.

4.2.2 Diferenças entre EJB e JavaBeans

O EJB e o *JavaBeans* são modelos de componentes utilizados para a criação de componentes Java, porém, são tecnologias criadas para propósitos diferentes e não possuem nenhuma relação entre si.

JavaBeans são componentes Java reusáveis que possuem propriedades, eventos e métodos e que são facilmente integrados para criar aplicações freqüentemente visuais. Podem ser usados para montar componentes maiores ou construir aplicações inteiras (ROMAN; SRIGANESH; BROSE, 2005).

Os componentes Java *Swing*, utilizados para a criação de interfaces gráficas, como por exemplo, botões e caixas de texto, são componentes *JavaBeans*.

Por outro lado, os componentes EJB são criados com o intuito de serem distribuídos, não visuais e que são executados no servidor implementando as regras de negócio da aplicação.

Os componentes *JavaBeans* não necessitam de um ambiente especial para que sejam executados, enquanto que os componentes EJB precisam ser distribuídos em um *container* EJB, que por sua vez, está em um servidor EJB (Figura 4.1).

4.3 As Seis Funções EJB

Uma das vantagens que a tecnologia EJB oferece é a clara distinção de trabalho que existe no processo de desenvolvimento, distribuição e gerenciamento de um aplicativo EJB. Esta vantagem é alcançada devido às funções que a especificação EJB estabelece para o processo de desenvolvimento e distribuição dos componentes EJB. São seis funções estabelecidas: desenvolvedor *bean*, montador de aplicativo, distribuidor, administrador de sistema, provedor de *container* e provedor de servidor EJB (DEMICHIEL, 2003).

- Desenvolvedor *bean* – É o programador do componente EJB. É o responsável por criar as classes e interfaces do componente e que conhece a lógica comercial do aplicativo. A saída gerada pelo desenvolvedor é um arquivo `.jar` que contém um ou mais componentes EJB. Ele deve apenas se concentrar nas regras de negócio que o componente realiza, não havendo a necessidade de se preocupar com questões complexas como transações, concorrência e segurança. O desenvolvedor deixa estes problemas para que o *container* EJB resolva.
- Montador de Aplicativo – Para que um aplicativo EJB seja construído é necessário que os componentes sejam agrupados para a criação do mesmo, pois geralmente, aplicativos EJB são formados por mais de um componente EJB. Para tal, é importante a figura do montador de aplicativo que é o responsável por reunir os componentes, escritos pelos desenvolvedores *bean*,

para montar a aplicação. O montador de aplicativo deve ter bastante conhecimento em composição de aplicativos a partir de componentes EJB, mas por outro lado, não necessita ter nenhum conhecimento sobre a implementação dos componentes.

- Distribuidor – Depois do aplicativo EJB ser montado é necessário que ele seja distribuído no ambiente. O distribuidor é o responsável em distribuir os componentes em um ambiente operacional que possua um servidor EJB e um *container* EJB específico. Ele deve possuir um bom conhecimento do ambiente operacional porque ele tem a função de resolver as dependências externas dos componentes em relação ao ambiente. O distribuidor utiliza ferramentas oferecidas pelo provedor de *container* para realizar as tarefas de distribuição.
- Administrador de Sistema – Responsável em manter o aplicativo EJB funcionando a todo o momento. Utiliza ferramentas para supervisionar o sistema as quais podem ser oferecidas pelo servidor EJB e pelo *container* EJB.
- Provedor de *Container* – Fornece as ferramentas necessárias para a distribuição dos componentes EJB. Também é responsável por fornecer recursos para escrever um *container* EJB e que este esteja de acordo com a especificação EJB, ou seja, que o *container* seja escalável e seguro para ser integrado com um servidor EJB. Responsabiliza-se também em fornecer ferramentas para gerenciar e controlar o *container*, que por sua vez, gerencia e controla os componentes EJB nele inseridos.
- Provedor de Servidor EJB – O provedor fornece um servidor EJB que é responsável por hospedar o *container* EJB, ou seja, o servidor EJB é uma

estrutura na qual os *containers* EJB são executados. Muitos fornecedores oferecem o servidor e o *container* EJB em um mesmo pacote.

Dentro destas seis funções EJB pode existir, em alguns casos, a possibilidade em que uma mesma pessoa desenvolva mais de uma função EJB, como por exemplo, um programador que desenvolve os componentes EJB e também monta a aplicação.

4.4 Tecnologias Relacionadas ao EJB

A tecnologia EJB utiliza-se de outras tecnologias Java que fornecem suporte para a implementação de aplicativos EJB. Tais tecnologias oferecem benefícios que facilitam o desenvolvimento dos aplicativos EJB.

4.4.1 JNDI

A JNDI (*Java Naming and Directory Interface*) é uma API especificada pela tecnologia Java que fornece um serviço de nomeação e um serviço de diretório para aplicações escritas em linguagem Java.

Kurniawan (2002) define que um serviço de nomeação relaciona um objeto a um nome, como por exemplo, o *Internet Domain Name System* (DNS) que mapeia nomes de domínios a endereços IP (*Internet Protocol*). Este mesmo autor relata que um serviço de diretórios permite que objetos tenham atributos que descrevam tal objeto. Assim, um objeto pode ser localizado sem que se conheça seu nome.

A tecnologia EJB utiliza o serviço de nomeação JNDI para associar um nome a um componente EJB e, com isso, o cliente pode encontrar um componente EJB conhecendo

apenas o nome do mesmo. O cliente fornece o nome do componente e a JNDI se encarrega de encontrá-lo. Para utilizar a API JNDI em aplicações EJB, utiliza-se o pacote *javax.naming*.

4.4.2 JDBC

A tecnologia JDBC (*Java Database Connectivity*) é uma API Java definida pela Sun Microsystems com o intuito de padronizar a conectividade de banco de dados com aplicativos Java, possibilitando o acesso a qualquer banco de dados relacional com o uso de uma única API.

JDBC fornece suporte para a conexão e manipulação de dados em um banco de dados, o envio de comandos SQL e o processamento dos resultados.

Para que a conexão se concretize, é necessária a utilização de *drivers* JDBC que são programas geralmente fornecidos pelo fabricante do banco de dados e que funcionam como intermediários entre a API JDBC e o banco de dados do fabricante.

A tecnologia EJB pode utilizar a API JDBC para o acesso ao banco de dados utilizando os pacotes *java.sql* e *javax.sql*.

4.4.3 RMI-IIOP

RPC (*Remote Procedure Call*) é quando um processo em uma máquina chama um procedimento que está em uma outra máquina.

RMI (*Remote Method Invocation*) utiliza o conceito da RPC que propicia a invocação de método remoto em programas escritos em linguagem Java, ou seja, habilita que um programa possa invocar um método que esteja em outra máquina virtual Java como se esse método estivesse disponível na mesma máquina do programa que o chamou.

Para o ambiente EJB utiliza-se o RMI-IIOP (*Remote Method Invocation* sobre o *Internet Inter-ORB Protocol*) que é uma versão especial do RMI sendo complacente com CORBA[6] (*Common Object Request Broker Architecture*) (ROMAN; SRIGANESH; BROSE, 2005). A RMI-IIOP utiliza os pacotes *java.rmi* e *javax.rmi*.

Estes mesmos autores ressaltam três características que RMI-IIOP possui:

- *Marshalling* e *unmarshalling*: Na passagem de parâmetros, duas máquinas podem ter problemas na comunicação. Por exemplo, se as máquinas possuem sistemas de representação de dados diferentes, ou seja, uma mesma informação binária representa dados diferentes em cada máquina. Com isso, existe o empacotamento e o desempacotamento dos parâmetros para que eles possam ser utilizáveis em dois ambientes distintos.
- Passagem de Parâmetros: RMI-IIOP proporciona a passagem de parâmetro por valor e a passagem de parâmetro por referência.
- Instabilidade da Rede: Em aplicações de objetos distribuídos existem várias máquinas virtuais Java trabalhando e caso ocorra a queda de uma dessas máquinas, a aplicação não terminará. RMI-IIOP manipula os problemas ocorridos em máquinas virtuais Java, em um computador ou na rede.

4.5 Arquitetura dos Componentes EJB

No processo de desenvolvimento do componente EJB existe a necessidade da criação de diversos arquivos para que se consiga construir um componente por completo. Isso se deve ao fato de que um componente EJB é formado por um conjunto de interfaces, classes e outros arquivos complementares.

[6] Roman; Sriganesh; Brose (2005) definem CORBA como um padrão único para a escrita de sistemas de objetos distribuídos.

Kurniawan (2002) relata que os clientes, ou seja, os usuários dos componentes, não os acessam diretamente, isso é feito pelas interfaces *home* e *remote* que fazem a comunicação do componente EJB com o cliente. Assim, para se construir um componente EJB, deve-se ter no mínimo uma classe *bean* que implemente as regras de negócio e duas interfaces *home* e *remote*. Na Figura 4.2 ilustra-se esta situação.

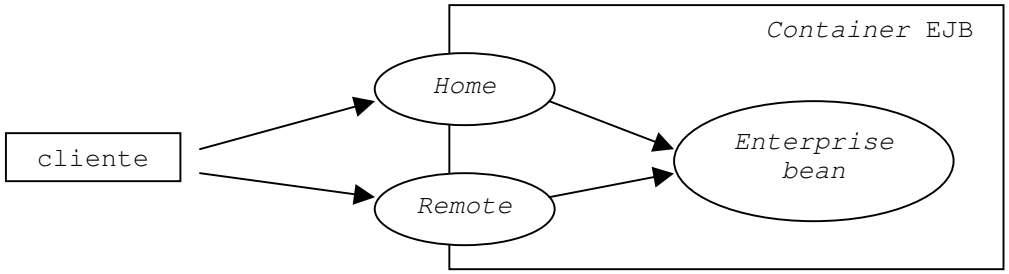

Figura 4.2 – Interfaces *home* e *remote* (adaptado de Kurniawan, 2002).

4.5.1 Interface *Home*

A interface *home* precisa estender a interface *javax.ejb.EJBHome*. DeMichiel (2003) relata que esta interface define métodos para criar, remover e localizar os componentes, sendo especificada pelo desenvolvedor *bean*.

O *container* gera automaticamente uma classe que implementa a interface *home*. Esta classe criada é o *home object,* que, por meio da interface *home*, tem a capacidade de saber como instanciar e destruir os EJB *objects*. É por meio do *home object* e do EJB *object* que o cliente se relaciona com o *bean*. Na Figura 4.3 ilustra-se a criação detalhada de um EJB *object*.

Pela interface *home*, o cliente solicita a criação de um novo EJB *Object* (Passo 1). *Home Object* cria um EJB *Object* (Passo 2) e, por último, é retornado ao cliente a referência deste EJB *Object* (Passo 3).

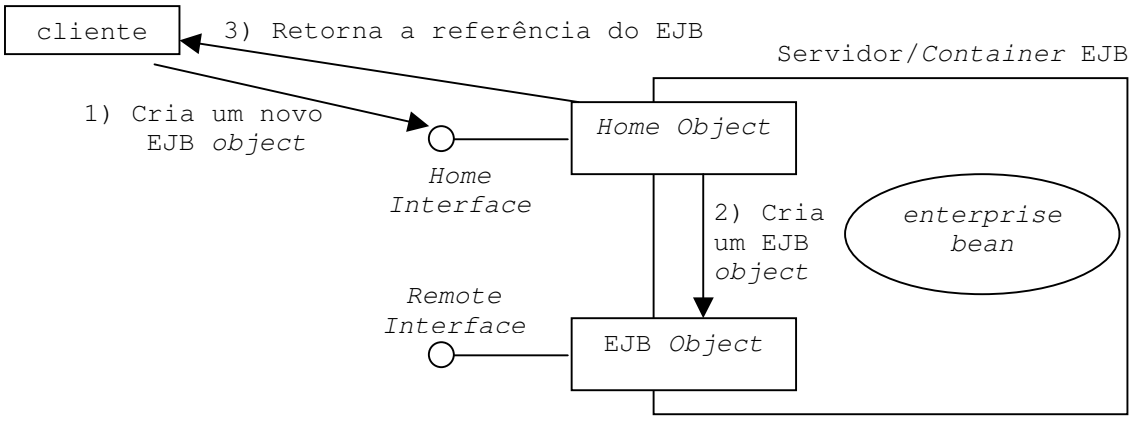

Figura 4.3 – Criação de um EJB *object* (adaptado de Roman; Sriganesh; Brose 2005).

Roman; Sriganesh; Brose (2005) relatam que a maioria dos *containers* possui uma relação 1:N entre *home objects* e as instâncias do *bean*, ou seja, todos os clientes utilizam a mesma instância do *home object* para criar os EJB *objects*.

4.5.2 Interface *Remote*

É uma interface escrita pelo desenvolvedor *bean* que possui todos os métodos de negócios que o cliente pode acessar e como ele pode acessar. Ela deve estender a interface *javax.ejb.EJBObject* e todos os seus métodos devem lançar uma exceção da classe *java.rmi.RemoteException*.

A interface *remote* declara os métodos que a classe *enterprise bean* implementa e disponibiliza para os seus clientes, com isso, o cliente não acessa os métodos do *bean* diretamente.

Após o cliente possuir uma referência de EJB *Object*, apresentado na Subseção 4.5.1, ele pode invocar os métodos definidos na interface *Remote*. A Figura 4.4 ilustra o que ocorre quando um método da interface *Remote* é invocado. Quando o cliente solicita um método do *bean* (Passo 1), o EJB *Object*, que implementa a interface *Remote* e sabe quais os métodos

estão disponíveis para o cliente, recebe a solicitação e a delega para o *bean* (Passo 2). Assim, o EJB *object* recebe a resposta do *bean* (Passo 3) e a retorna para o cliente (Passo 4).

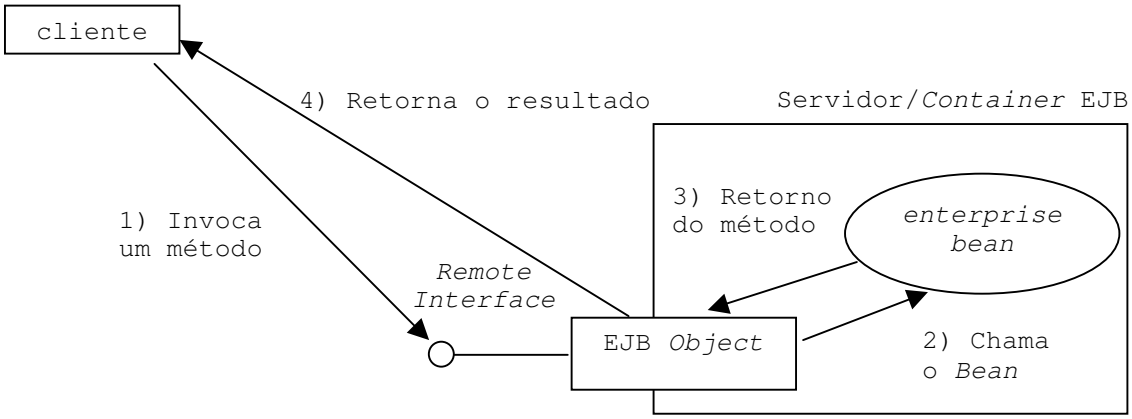

Figura 4.4 – EJB *Object* (adaptado de Roman; Sriganesh; Brose 2005).

Verifica-se a importância da interface *remote*, pois ela expõe para os clientes apenas as assinaturas dos métodos que interessam ao cliente, com isso, a classe *enterprise bean* pode conter métodos não especificados na interface que somente ela pode acessar.

4.5.3 Interfaces Locais

A comunicação dos componentes EJB ocorre por meio de invocações de métodos remotos que estejam em outra máquina virtual Java. Mas esta mesma comunicação era utilizada entre os componentes executados no servidor e, portanto, na mesma máquina virtual Java, acarretando uma sobrecarga desnecessária. A partir da versão EJB 2.0 foram criadas as interfaces locais *LocalHome* e *Local* com o objetivo de diminuir esta sobrecarga na comunicação local entre cliente e componente ou entre os próprios componentes. Estas interfaces são opcionais e substituem as interfaces *Home* e *Remote*, respectivamente.

A interface *LocalHome* estende a interface *javax.ejb.EJBLocalHome* enquanto a interface *Local* estende a interface *javax.ejb.EJBLocalObject*.

4.5.4 Classe *Enterprise* B*ean*

A classe *enterprise bean* é a principal classe do componente. Esta classe possui a real codificação do componente e deve ter os métodos exatamente com a mesma assinatura dos métodos contidos na interface *Remote* ou dos métodos contidos na interface *Local*, caso o componente seja acessado localmente.

Roman; Sriganesh; Brose (2005) relatam que a classe *enterprise bean* é uma classe Java que possui os detalhes de implementação do componente, está em conformidade com uma interface bem definida e segue regras para que o *container* EJB possa executá-la.

Devido à existência de três tipos de EJB, a classe *enterprise bean* pode ser:

- um *bean* de sessão que abrange as lógicas comerciais e implementa a interface *javax.ejb.SessionBean* (Maiores informações são apresentadas na Subseção 4.6.1);
- um *bean* de entidade que abrange as lógicas relacionadas aos dados e implementa a interface *javax.ejb.EntityBean* (Maiores informações são apresentadas na Subseção 4.6.2);
- um *bean* de mensagem direcionada que abrange a lógica de envio de mensagens e implementa a interface *javax.ejb.MessageDrivenBeans* (Maiores informações são apresentadas na Subseção 4.6.3).

A classe *enterprise bean* possui métodos que são utilizados pelo *container* para interagir com os componentes e são definidos pelo tipo de interface que a classe *enterprise bean* herdará.

A especificação EJB define a interface base *javax.ejb.EnterpriseBean* e os três tipos de *beans* que devem estender esta interface. No Quadro 4.1 mostra-se esta hierarquia.

```
        - Interface Java.io.Serializable
                  - Interface javax.ejb.EnterpriseBean
                          - Interface javax.ejb.EntityBean
                          - Interface javax.ejb.MessageDrivenBean
                          - Interface javax.ejb.SessionBean
```

Quadro 4.1 – Hierarquia de interfaces.

A interface *javax.ejb.EnterpriseBean* estende a interface *java.io.Serializabe* que possibilita a serialização de objetos, ou seja, os objetos podem ser transformados em uma forma binária para serem transportados pela rede.

4.5.5 Descritor de Distribuição

O aplicativo EJB necessita ter um descritor de distribuição que fornece metainformações para o *container* EJB. O descritor de distribuição especifica várias informações acerca do *enterprise bean*, como por exemplo, o nome (Linha 4) e tipo do *enterprise bean* (Linha 3), o nome qualificado das interfaces *home* e *remote* (Linhas 5 e 6) e o nome da classe principal do *bean* (Linha 7). O descritor pode definir mais de um *enterprise beans* (KURNIAWAN, 2002). No Quadro 4.2 mostra-se um fragmento do código do descritor de distribuição utilizado neste trabalho.

```
1. . .
2<enterprise-beans>
3       <entity>
4               <ejb-name>Cliente</ejb-name>
5               <home>br.fundanet.lucio.oficina.ClienteHome</home>
6               <remote>br.fundanet.lucio.oficina.Cliente</remote>
7               <ejb-class>br.fundanet.lucio.oficina.ClienteBean</ejb-class>
8               . . .
9       </entity>
10</enterprise-beans>
```

Quadro 4.2 – Exemplo de Descritor de Distribuição.

O descritor é um arquivo XML (*Extensible Markup Language*) que oferece informações para que o *container* saiba como gerenciar o *bean* durante sua execução. Tais informações podem descrever requisitos em relação à transação, segurança e persistência. Este arquivo deve possuir o nome *ejb-jar.xml* e deve ser colocado em uma pasta de nome `META-INF` para que posteriormente seja empacotado junto com os arquivos `.class` que compõem os componentes.

4.5.6 Arquivo de Distribuição

Após a criação da interface *home*, da interface *remote*, da classe *enterprise bean* e do descritor de distribuição, é necessário que todos estes arquivos sejam empacotados em um arquivo `.jar` (*Java Archive*) que se assemelha aos arquivos `.zip`, ou seja, são arquivos compactados. Este arquivo empacota todos os arquivos necessários para a criação do componente e deve ser distribuído em um *container* EJB.

Quando os arquivos são distribuídos, Roman; Sriganesh; Brose (2005) relatam que as ferramentas que o *container* EJB possui são responsáveis por descompactar, ler e extrair as informações contidas no arquivo.

4.6 Tipos de EJB

Nesta Seção são descritos detalhadamente os três tipos de EJB: *beans* de sessão (*session beans*), *beans* de entidade (*entity beans*) e *beans* de mensagem direcionada (*message-driven beans*), citados na Subseção 4.5.4.

4.6.1 *Beans* de Sessão

O *bean* de sessão é um tipo de EJB que implementa as lógicas comerciais para um cliente. Ele é usado por um único cliente, ou seja, não é compartilhado entre vários clientes. Kurniawan (2002) relata que os *beans* de sessão recebem esta nomenclatura porque o *bean* fica ativo enquanto existe a sessão cliente que o utiliza.

O *container* EJB, responsável pelo gerenciamento dos componentes EJB, cria o *bean* de sessão para o cliente utilizá-lo e também se responsabiliza pela destruição do mesmo quando o cliente termina.

De acordo com Kurniawan (2002) os *beans* de sessão podem ser utilizados para implementar um cálculo, um método de criptografia, acessar o banco de dados, entre outros. Dentre as características mais relevantes do *bean* de sessão, relatadas na especificação EJB, tem-se:

- É geralmente temporário, ou seja, seu tempo de vida depende da duração da conexão do cliente.
- Se o *container* EJB pára, o *bean* de sessão é removido e o cliente precisa re-estabelecer um novo *bean* para continuar o seu trabalho.
- Não representa diretamente os dados de um banco de dados, mas pode ser usado para inserção, alteração e atualização dos dados.
- É executado com um único cliente.

Monson-Haefel (2004) relata que os *beans* de sessão freqüentemente controlam as interações entre os *beans* de entidade, descrevendo como eles trabalham juntos para realizarem uma tarefa específica.

Nota-se a vantagem de possuir a lógica comercial implementada em um *bean* de sessão, pois, além de permitir que diferentes tipos de clientes acessem o componente EJB,

este mesmo componente usufrui os serviços oferecidos pelo *container* EJB. O *container* pode executar vários *beans* de sessão concorrentemente.

Os *beans* de sessão se dividem em duas categorias: *beans* de sessão *stateless* e *beans* de sessão *stateful*.

Os *beans* de sessão *stateless* não possuem a capacidade de manter o estado de comunicação com o cliente, ou seja, o *bean* não possui dados relativos ao cliente durante a conversação.

Quando um método é invocado em um *bean* de sessão *stateless*, o *bean* executa o método e retorna o resultado, portanto, ele não tem o conhecimento da identidade do cliente, das requisições anteriores e nem das requisições seguintes.

Monson-Haefel (2004) exemplifica um *bean stateless* como um conjunto de procedimentos que recebem uma requisição com parâmetros e retornam um resultado.

Já os *beans* de sessão *stateful*, ao contrário dos *beans stateless*, possuem a capacidade de manter um estado de comunicação com o cliente. Assim, os *beans stateful* possuem dados relativos ao cliente ou informações que o cliente forneceu, mantendo-se uma relação na qual o *bean* fica dedicado a atender o seu cliente. Eles não são usados concorrentemente pelos clientes.

Uma aplicação conhecida que exemplifica o uso de *beans stateful*, citada por Pharoah e Arni (2001), é o carrinho de compras *on-line*, em que através de métodos remotos o cliente pode adicionar, retirar ou modificar os produtos do carrinho de compras. Nota-se a existência da interação entre o cliente e o *bean* até o momento da compra ser finalizada.

4.6.2 *Beans* de Entidade

Os *beans* de entidade são objetos de dados que representam os dados da aplicação. São objetos Java que armazenam as informações do banco de dados (ROMAN; SRIGANESH; BROSE, 2005).

Outra definição apresentada por Kurniawan (2002) define *bean* de entidade como um componente de dados que persiste os dados em uma armazenagem secundária, tal como um banco de dados. A entidade *bean* possui campos para armazenar dados e métodos que realizam operações nos campos.

Morisseau-Leroy; Solomon; Basu (2001) apresentam algumas características dos *beans* de entidade:

- Representam dados de dispositivos de armazenamento persistentes, como um banco de dados. Sobre essa característica Kurniawan (2002) argumenta que, em um aplicativo EJB, uma entidade *bean* representa um registro de uma tabela de banco de dados. Ao invés de manipular dados diretamente em um banco de dados, como se faz em alguns aplicativos não EJB, manipula-se um *bean* de entidade que representa um conjunto de dados.
- Os *beans* de entidade são transacionais, ou seja, eles permitem que desenvolvedores escrevam aplicativos que atualizam automaticamente dados residentes em um banco de dados. Kurniawan (2002) atenta para o fato de que o *container* EJB cuida da sincronização e de outras tarefas na manutenção de dados. Um *bean* de entidade vive tanto quanto os dados que ele representa, portanto uma entidade *bean* poderia durar meses ou até anos.
- A última característica é que os *beans* de entidade são persistentes e, portanto, sobrevivem à falhas do servidor EJB e não terminam quando os

clientes terminam ou se desconectam do banco de dados. Desde que os dados permaneçam no banco de dados, o *bean* de entidade existe.

Nota-se que um *bean* de entidade representa um registro de uma tabela, portanto, em sua classe principal, a classe *enterprise bean*, devem existir atributos que representem os campos de um registro. Portanto, se existir alguma alteração no valor do atributo, o *container* se encarrega de alterar este valor no campo correspondente da tabela evitando o acesso direto do cliente com o banco de dados da aplicação.

DeMichiel (2003) especifica dois tipos de entidades *beans* em relação à persistência dos dados: entidades *beans* com persistência gerenciada por *beans* (BMP) e entidades *beans* com persistência gerenciada por *container* (CMP).

Beans de entidade BMP ocorre quando o próprio *bean* gerencia a sua persistência. Pharoah e Arni (2001) relatam que o desenvolvedor *bean* deve implementar os métodos para o acesso ao banco de dados. São mais complexos para desenvolver e controlar devido ao fato do programador ter que inserir os códigos no *bean* de entidade para controlar a transação, porém eles proporcionam maior flexibilidade para se relacionar tabelas.

Beans de entidade CMP ocorre quando o *container* é quem gerencia a persistência dos dados, não sendo necessário que o desenvolvedor *bean* implemente os códigos de acesso ao banco. Kurniawan (2002) relata que os *beans* de entidade CMP confiam no *container* EJB para fazer as tarefas relativas ao banco de dados.

Pharoah e Arni (2001) relatam que o *container* é o responsável por ler os dados do banco e popular o *bean* de entidade.

É mais fácil de escrever e manter o *bean* de entidade CMP, entretanto, é necessário que se configure o descritor de distribuição, como, por exemplo, inserir os nomes das tabelas e do banco de dados.

4.6.3 *Bean* de Mensagem Direcionada

De acordo com Kurniawan (2002) um serviço de mensagem é um serviço que oferece comunicação entre aplicativos ou entre componentes de software.

A Sun Microsystems definiu a API JMS – *Java Message Service* – que é um serviço de mensagem Java e que permite aplicativos Java enviar e receber mensagens, estes aplicativos são os clientes JMS. Os clientes que enviam mensagens não necessitam saber nada sobre o destinatário e vice-versa.

DeMichiel (2003) define que o *bean* de mensagem direcionada é um consumidor de mensagem assíncrona, em que os clientes enviam as mensagens e não esperam qualquer retorno. São componentes que recebem mensagens dos clientes JMS, ou seja, a interação é feita através do envio de mensagens e não através da invocação de métodos.

Este tipo de EJB foi introduzido a partir da versão EJB 2.0.

4.7 Padrões de Projeto

Os padrões de projeto são utilizados para auxiliar o desenvolvimento de aplicações com o objetivo de orientá-lo a uma solução para melhorar o projeto do sistema. Os padrões de projeto não são apenas sugestões, e sim, padrões que já foram utilizados e testados em projetos anteriores.

De acordo com Bond *et al*. (2005) um padrão é “uma solução para um problema em um contexto”. É uma idéia sobre como resolver um problema encontrado no domínio de um projeto. Este mesmo autor relata que a principal característica dos padrões é que são soluções já comprovadas.

Segundo Gamma *et al*. (1995), padrões de projeto são descrições de objetos que se comunicam e de classes que são personalizadas para resolver um problema de projeto genérico em um contexto específico. Ele propõe vinte e três padrões de projeto que podem ser utilizados. Este mesmo autor relata que um padrão de projeto deve possuir quatro elementos essenciais:

- **Nome:** O padrão deve possuir um nome que indique a função que realiza. Descreve a essência do padrão.
- **Problema:** Descreve quando se deve aplicar o padrão e em que condições.
- **Solução:** A partir da descrição do problema, a solução diz como usar os elementos disponíveis para resolvê-lo.
- **Conseqüências:** São os resultados da utilização do padrão. São importantes para o entendimento do custo e benefício de sua utilização.

Metsker (2002) classifica os vinte e três padrões definidos por Gamma *et al*. (1995) em cinco categorias de acordo com a intenção de cada uma delas. As categorias e os padrões são:

- Fornecer uma interface: *Adapter, Façade, Composite, Bridge*.
- Atribuir responsabilidade: *Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight*.
- Realizar a construção de classes ou objetos: *Builder, Factory Method, Abstract Factory, Prototype, Memento*.
- Controlar formas de operação: *Template Method, State, Strategy, Command, Interpreter*.
- Implementar uma extensão para a aplicação: *Decorator, Iterator, Visitor*.

O padrão utilizado na aplicação desenvolvida, descrita no Capítulo 5, foi o *Façade* que tem como objetivo fornecer uma interface uniforme para um conjunto de interfaces de um

subsistema, ou seja, fornece uma interface única de nível mais elevado que deixa o subsistema mais fácil de usar (GAMMA *et al*., 1995), como apresentado na Figura 4.5 (b).

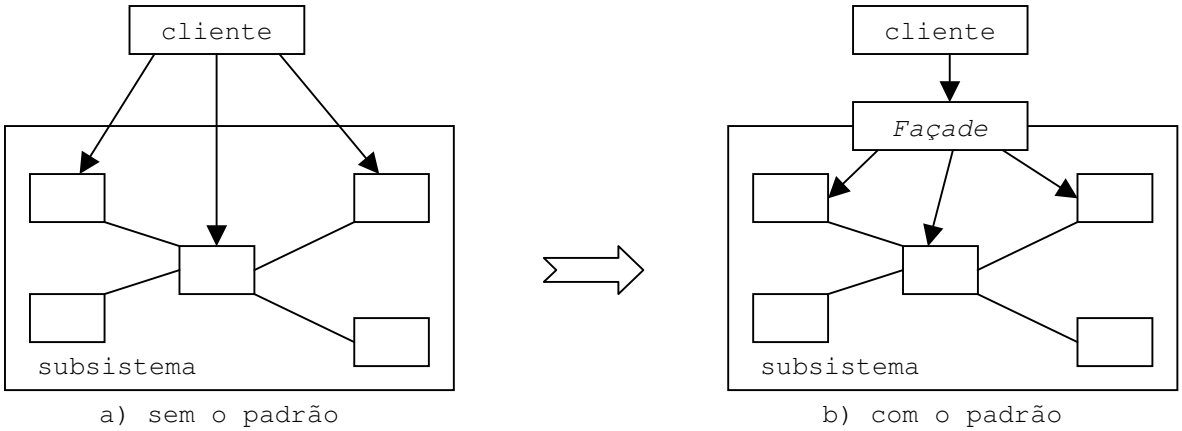

Figura 4.5 – Padrão *Façade* (Adaptado de Gamma *et al*., 1995).

Na tecnologia J2EE existem diversos padrões de projeto úteis que podem ser utilizados com o intuito de fornecer melhor desempenho para a aplicação. Nesta tecnologia, o padrão *Façade* é chamado de *Session Façade*, pois é um *bean* de sessão que deve ser utilizado como ponto de entrada uniforme para o acesso ao subsistema.

O *Session Façade*, implementado com um *bean* de sessão, define um componente de mais alto nível que centraliza e contém as complexas interações entre os componentes de mais baixo nível, ou seja, o *bean* de sessão impede o acesso direto do cliente aos *beans* de entidade. Na Figura 4.6 ilustra-se esta situação.

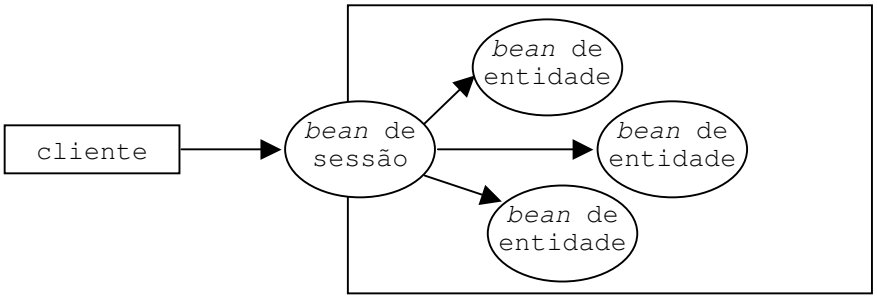

Figura 4.6 – *Session Façade*.

O padrão *Session Façade* foi escolhido por ser um padrão amplamente utilizado nas aplicações construídas com a tecnologia EJB e por se adequar de modo eficiente na relação entre os usuários da aplicação com os *beans* de entidade, trazendo simplicidade à aplicação.

4.8 Considerações Finais

Este capítulo apresentou a tecnologia EJB utilizada para a criação de componentes em linguagem Java, bem como as funções estabelecidas para o desenvolvimento e distribuição dos mesmos. Foram abordados os três tipos de *enterprise beans* definidos pela especificação EJB e os demais arquivos que fornecem as funcionalidades necessárias para o desenvolvimento de componentes. Essa tecnologia será utilizada para a criação da aplicação baseada em componentes apresentada no Capítulo 6, para verificar quais os benefícios trazidos pela arquitetura EJB com o uso do padrão *Session Façade* durante o desenvolvimento e a utilização da aplicação.

5 ESTUDO DE CASO: DOCUMENTAÇÃO DA APLICAÇÃO COM UML *COMPONENTS*

5.1 Considerações Iniciais

Este capítulo apresenta a documentação de uma aplicação, destinada a qualquer empreendimento comercial, segundo o processo do método UML *Components*. Essa aplicação comercial foi desenvolvida com base em uma oficina de chaves que fornece produtos e/ou presta serviços. Pelo fato do método ser uma adaptação do RUP, foi dado ênfase nas atividades de Especificação, Adequação e Codificação, que substituem diretamente suas atividades de Análise, Projeto e Implementação.

Embora a ED faça parte do contexto do que é relacionado ao desenvolvimento e elaboração de componentes de software e ao DBC, aspectos relacionados não foram tratados na aplicação deste trabalho, pois o método UML *Components* não os apresenta.

Na Seção 5.2 será apresentado o método UML *Components*. Na Seção 5.3 serão apresentados os estereótipos utilizados pelo método, já que é uma extensão da UML. Por fim, na Seção 5.4 serão apresentadas as atividades que compõem o processo de desenvolvimento do UML *Components*, dando ênfase somente nas atividades mencionadas acima. Dentre essas atividades, será detalhada a Especificação, através de diagramas referentes à aplicação de uma loja comercial.

.

5.2 UML *Components*

O UML *Components* (CHEESMAN e DANIELS, 2001 apud BARROCA; GIMENES; HUZITA, 2005) é um processo de desenvolvimento de software baseado em

componentes, que trata do problema de especificar e projetar sistemas baseados em componentes. Ele não é um processo de desenvolvimento completo, já que não inclui atividades relacionadas ao processo gerencial, e enfatiza as etapas de análise, projeto e, em menor grau, implementação (PAGANO, 2004).

Segundo Sousa (2004), o método UML *Components* é iterativo e define um conjunto de passos para que, a partir da especificação de um conjunto de casos de uso, seja construído um sistema de software baseado em componentes que simplifique o trabalho de lidar com mudanças (evolução/substituição de componentes).

O método propõe a utilização da UML para modelar todas as fases do desenvolvimento de sistemas baseados em componentes, contendo as atividades de definição de requisitos, identificação e descrição das interfaces entre componentes, especificação e modelagem, além de implementação e montagem. Além disso, foram criados novos diagramas para o DBC, visto que havia essa necessidade (WERNER e BRAGA, 2005). O Quadro 5.1 apresenta os diagramas utilizados pela abordagem UML *Components*.

UML *Components* também contextualiza os diagramas tradicionais de UML. Os diagramas apresentados abordam somente a fase de análise e especificação da arquitetura do software, pois a implementação dos componentes podem ser feitas com os diagramas originais da própria UML.

Um ponto interessante levantado por Werner e Braga (2005) é o fato de o UML *Components* não reforçar qualquer prática de reutilização, nem mesmo por sua metodologia ser muito utilizada pelos desenvolvedores que adotaram a prática do desenvolvimento baseado em componentes. Isso ocorre porque aspectos de ED não são apresentados, ou seja, para a reutilização de componentes, a metodologia deveria se preocupar e considerar de maneira adequada os aspectos de ED.

	Diagrama	***Descrição***
Análise	Diagrama Modelo do Conceito do Negócio (*Business Concept Model Diagram*)	Modelo conceitual das informações que existem no Domínio.
	Diagrama Tipo do Negócio (*Business Type Diagram*)	Diagrama com os tipos de negócios (e seus relacionamentos) que precisam ser mantidos pelo software.
	Diagrama de Casos de Uso (*Use Case Diagram*)	Diagrama de casos de uso do software.
Especificação da arquitetura	Diagrama de Especificação da Interface (*Interface Specification Diagram*)	Definição precisa sobre as ações de uma interface, possuindo um modelo de informação, a especificação das operações e as invariantes.
	Diagrama de Especificação do Componente (*Component Specification Diagram*)	Diagrama que descreve as interfaces fornecidas e exigidas de uma especificação de componente.
	Diagrama de Arquitetura do Componente (*Component Architecture Diagram*)	Diagrama da arquitetura de componentes.
	Diagrama de Responsabilidade da Interface (*Interface Responsibility Diagram*)	Diagrama que mostra o conhecimento de uma interface sobre os elementos de negócio.
	Diagrama de Interação do Componente (*Component Interaction Diagram*)	Diagrama de interação entre uma especificação de componente e suas interfaces em tempo de execução.

Quadro 5.1 – Diagramas utilizados pela abordagem UML *Components* (adaptada de Werner e Braga, 2005, p. 94).

O Quadro 5.2, abaixo, apresenta um resumo constituído das principais características do método.

Processo	O processo de DBC é detalhado, sendo uma modificação do RUP para atender aspectos específicos de DBC. Aspectos relacionados à ED não são apresentados.
Produto	Um conjunto de modelos UML modificados parra incluir características DBC. É clara a falta de um modelo que guie o desenvolvedor no processo de reutilização de componentes em uma aplicação.
Ferramental automatizado	Utilização de ferramental proposta pela Rational.

Quadro 5.2 – Principais características do método UML *Components* (adaptada de Werner e Braga, 2005, p. 95).

A linguagem UML já possui o conceito de "componente", mas de acordo com Cheesman e Daniels (2001 apud NASCIMENTO, 2005), o método UML *Components* é indispensável no que diz respeito à reutilização de componentes, pois enquanto a linguagem

assume que haverá uma implementação orientada a objeto, o método não chega a esse nível de detalhe deixando livre a forma como os componentes serão implementados. Outro fator que também distingue a linguagem do método é que UML é baseada em diagramas, enquanto que UML *Components* é baseado em técnicas.

O UML *Components* possui as seguintes características:

1. **Documentação**: o método UML *Components* dispõe de uma documentação vasta e de fácil acesso.
2. **Fácil integração com UML**: o método apresenta fácil integração com a UML devido à utilização de estereótipos. Além disso, várias ferramentas disponíveis no mercado que permitem modelagem em UML suportam esta funcionalidade.
3. **Processo flexível e simples**: o método somente adiciona o conceito de seis estereótipos e por isso torna-se simples para modelagem de componentes.
4. **Específico para DBC**: ao contrário do RUP, por exemplo, que é um processo genérico para desenvolvimento de software, o UML *Components* se apresenta voltado exclusivamente para o DBC.
5. **Plataforma aberta**: UML *Components* pode ser usado para especificar componentes em qualquer plataforma de desenvolvimento.
6. **Bem aceito na indústria e academia**: a idéia desse método já vem sendo bem usada tanto pela indústria de software quanto pela academia (NASCIMENTO, 2005).

O método propõe extensões da linguagem UML através de seu mecanismo de estereótipos para a especificação das interfaces de componentes, de componentes propriamente ditos e de suas implementações.

5.3 Os Estereótipos do UML *Components*

Existe um grande número de mecanismos diferentes de extensão para UML, mas, provavelmente, a mais utilizada, de acordo com Cheesman e Daniels (2001 apud NASCIMENTO, 2005) é utilizar estereótipos. Segundo Nascimento (2005), um estereótipo pode ser anexado a qualquer elemento UML.

Segundo Sousa (2004), "o processo UML *Components* define estereótipos adicionais para atribuir uma nova semântica a alguns elementos particulares da linguagem UML". Os estereótipos, apresentados nas próximas Seções, são encapsulados pelo símbolo "<<>>".

5.3.1 *Type*

O estereótipo <<type>> é um indicativo de que a classe é um tipo do negócio. É usado para representar uma classe no modelo de negócios de um componente em nível de especificação. O conceito de classe mencionado não é o mesmo que em uma linguagem de programação orientada a objetos como Java.

5.3.2 *Datatype*

O estereótipo <<datatype>> é um indicativo de que a classe representa um tipo estruturado de dados. Possui a mesma representação de *type*, mas é usado para classes que utilizam persistência.

5.3.3 *Interface type*

O estereótipo <<interface type>> é um indicativo de que a interface é uma interface de componente. Representa uma interface em nível de especificação. Por convenção, o nome de classes desse tipo são iniciadas com um "I". É diferente do estereótipo <<interface>>, já presente em UML, que é utilizado para modelagem orientada a objetos e deve ser evitada quando for feita a modelagem ou especificação de componentes.

5.3.4 *Component Specification*

O estereótipo <<comp spec>> indica que a classe é um componente. É usado para apresentar uma especificação de um componente. Este estereótipo, normalmente, está associado ao estereótipo <<interface type>> através de uma ligação de <<offers>>, apresentado na próxima Seção.

5.3.5 *Offers*

O estereótipo <<offers>> indica que uma classe <<comp spec>> oferece uma interface <<interface type>>. Somente pode ser aplicada entre estes dois elementos. Ele liga uma especificação de um componente à sua interface. Este estereótipo é análogo a <<realize>> de UML. Ou seja, ao invés de um componente implementar uma determinada interface, o conceito muda, de tal forma que, a especificação de um componente oferece uma interface.

5.3.6 *Core*

O estereótipo <<core>> é um indicativo de que a classe representa um tipo central do negócio. O "*core*" subentende um "*type*" sem ocorrência de dependências, sendo assim, o tipo "*core*" pode ser encontrado a partir de um refinamento do modelo de um componente, observando quais são os tipos que não apresentam dependência com outros componentes.

5.4 O Processo de UML *Components*

A abordagem de UML *Components* proposta por Chessman e Daniels (2001 apud BARROCA; GIMENES; HUZITA, 2005), é baseada no RUP, que utiliza a idéia de uma seqüência de atividades encadeadas em que o resultado de uma atividade serve como entrada para a próxima. Com esse conceito, Cheesman e Daniels (2001 apud NASCIMENTO, 2005) definem um *workflow* para o desenvolvimento de componentes, apresentado na Figura 5.1.

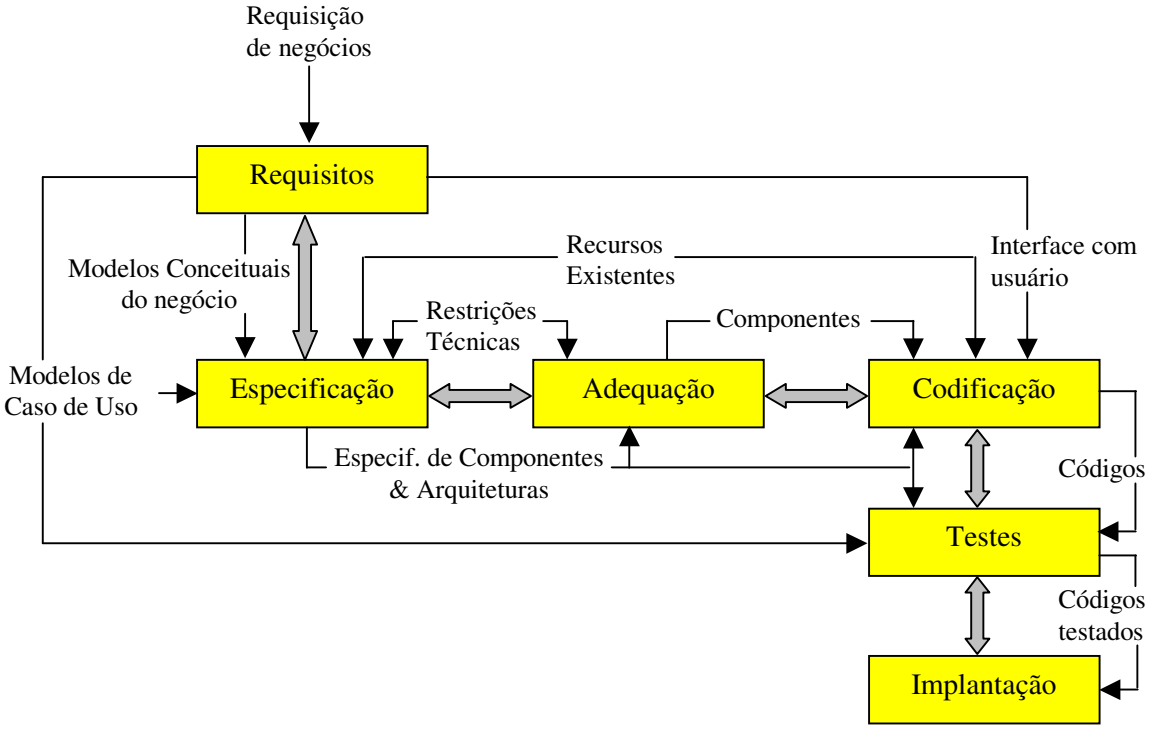

Figura 5.1 – *Workflow* para desenvolvimento de componentes (Cheesman e Daniels apud Nascimento, 2005, p. 31).

Segundo Sousa (2004); Nascimento (2005), as atividades Requisitos, Testes e Implantação correspondem diretamente às atividades encontradas no método RUP, inclusive possuem a mesma denominação. Portanto, será dada somente ênfase nas atividades de Especificação, Adequação e Codificação, que substituem diretamente as atividades de Análise, Projeto e Implementação do RUP.

As atividades do processo UML *Components* são representadas pela Figura 5.2. As fases do UML *Components* aplicados neste trabalho constituem apenas a atividade Especificação, que segundo Sousa (2004); Nascimento (2005), se divide em três fases: **Identificação do Componente, Interação do Componente** e **Especificação do Componente**.

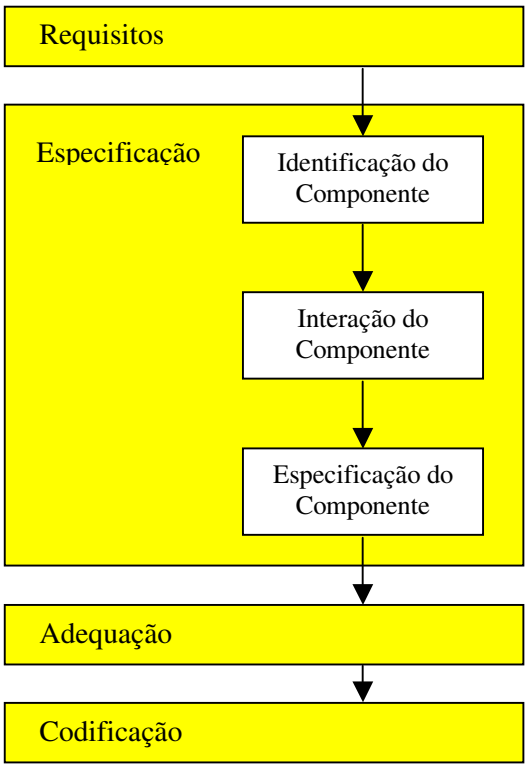

Figura 5.2 – Detalhamento das atividades do processo de desenvolvimento baseado em componentes utilizando UML *Components* (Nascimento, 2005, p. 32)

A atividade **Especificação** recebe como entrada o modelo conceitual do negócio e o modelo de casos de uso da aplicação, representados, respectivamente, na Figura 5.4 e na Figura 5.5, ambos da Seção 5.4.1. No Quadro 5.3, também localizado na Seção 5.4.1, apresenta-se a descrição dos casos de uso Registrar Venda e Cadastrar Cliente, que compõem a realização de uma venda da aplicação.

De acordo com Pagano (2004), as fases que compõem a atividade Especificação geram como saída um conjunto de arquiteturas e especificações de componentes. Essa saída será usada pela atividade **Adequação**, que segundo Nascimento (2005), vai garantir que "os componentes necessários estarão disponíveis, sejam eles comprados, construídos do zero, integrados com outros componentes, reutilizados de outros componentes ou outro software".

A **Codificação** integra todos os componentes com os possíveis softwares já existentes, utilizando uma interface apropriada com o usuário, formando uma aplicação que atenda às necessidades dos requisitos levantados (NASCIMENTO, 2005).

As seguintes Seções abordam de forma detalhada as três fases da atividade Especificação. Segundo Cheeseman e Daniels (apud BARROCA; GIMENES; HUZITA, 2005), essa atividade é voltada para sistemas desenvolvidos em camadas, conforme Figura 5.3, mais especificamente para a camada de sistema, que contêm o que é específico à aplicação, ou seja, trata das funcionalidades, e para a camada de negócios, que trata de informações e fornecem operações que serão necessárias aos serviços do sistema. As fases serão exemplificadas com base na aplicação de uma loja comercial, que foi composta pelos componentes desenvolvidos nesse trabalho.

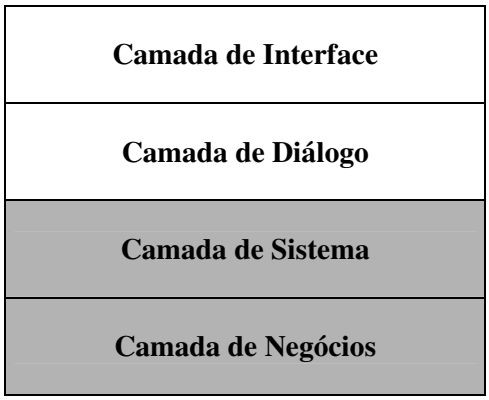

Figura 5.3 – Camadas da Arquitetura de uma Aplicação.

A descrição do mini-mundo da aplicação é: a aplicação é operada por funcionários que lidam com as vendas de produtos e/ou de serviços solicitados pelos clientes. Uma venda é realizada para um cliente cadastrado e é verificada a existência dos produtos em estoque. Existem diversos produtos para venda, mas no caso de ser uma chave, este possui para seu cadastro informações da seção, onde se localizará, e sua marca. A compra para a oficina é realizada quando a falta de produtos é verificada, sendo que diferentes fornecedores podem fornecer o mesmo produto.

5.4.1 Identificação do Componente

Essa fase, segundo Barroca; Gimenes; Huzita (2005), produz uma especificação e arquitetura inicial. Tem como entrada o modelo conceitual do negócio, Figura 5.4, e o modelo de casos de uso, Figura 5.5, que possibilitam:

- identificar interfaces para os componentes de negócio;
- identificar interfaces de sistema para os componentes de sistema e;
- gerar uma arquitetura de componentes inicial.

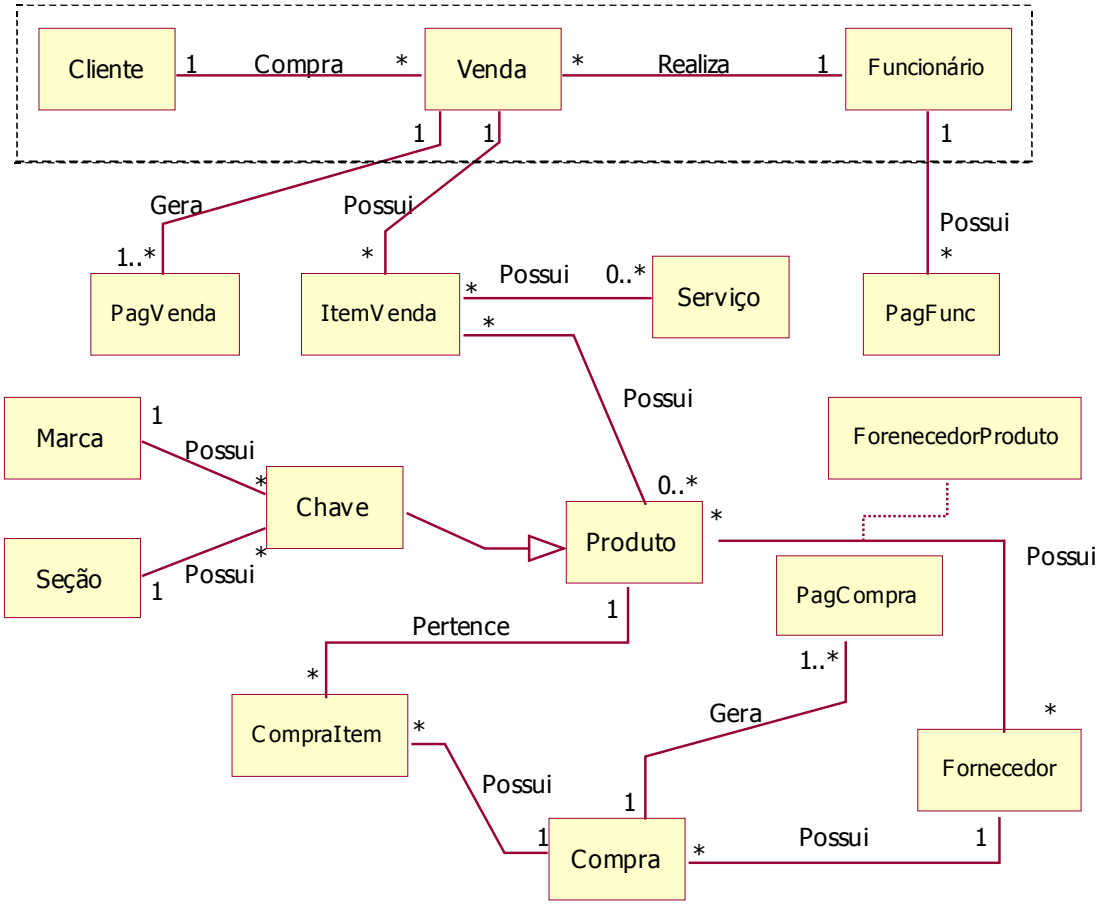

Figura 5.4 – Modelo Conceitual da aplicação.

A parte destacada na Figura 5.4 especifica o que foi implementado tanto na aplicação, quanto no Modelo de Casos de Uso da Realização de uma Venda, Figura 5.5.

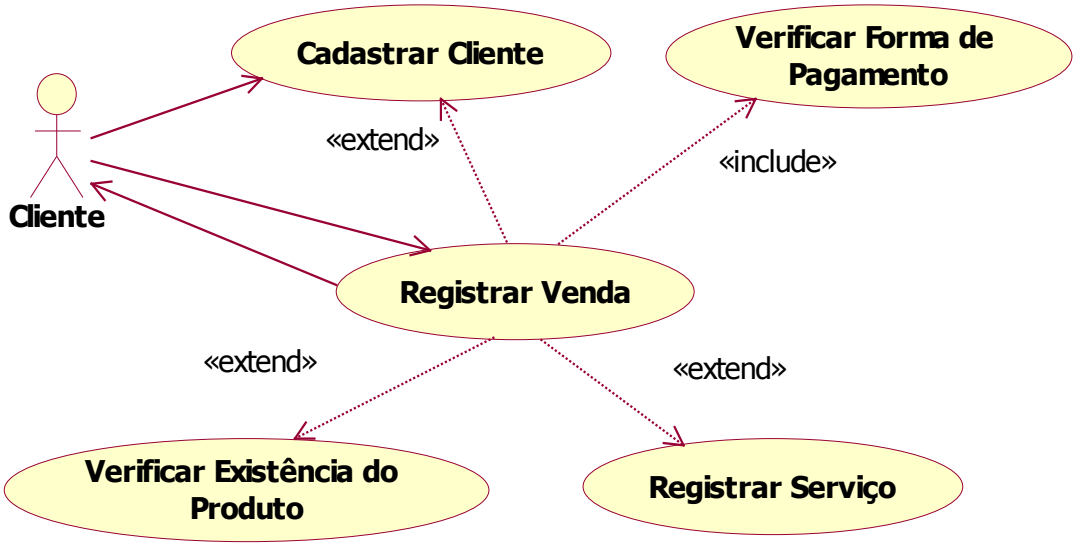

Figura 5.5 – Modelo de Casos de Uso da Realização de uma Venda.

A especificação do Modelo de Casos de Uso da Realização de uma Venda é representada pelo Quadro 5.3:

Ator: Cliente. Nome: Registrar Venda. Descrição: Caso de uso que registra as vendas solicitadas pelos clientes. Fluxo Normal: 1. Cliente solicita produto e/ou serviço. 2. Sistema verifica que cliente possui cadastro. 3. Para cada produto solicitado. 3.1 Chamar caso de uso "Verificar Existência do Produto". 4. Para cada serviço solicitado. 4.1 Chamar caso de uso "Registrar Serviço". 5. Cliente informa forma de pagamento. 6. Chamar caso de uso "Verificar Forma de Pagamento". 7. Sistema registra a venda e emite mensagem: "Venda efetuada com sucesso". Fluxo Alternativo: 2.1 Sistema verifica que cliente não possui cadastro. 2.2 Chamar caso de uso "Cadastrar cliente".
Ator: Cliente. Nome: Cadastrar Cliente. Descrição: Caso de uso que efetua o cadastro dos clientes. Fluxo Normal: 1. Cliente informa seus dados: nome, endereço, bairro, tel1, tel2, cidade, cnpj, enscrestadual,

cep. 2. Sistema registra o cliente e emite mensagem: "Cliente cadastrado com sucesso".

Quadro 5.3 – Modelo de Casos de Uso da Realização de uma Venda.

As Seções seguintes abordam uma descrição sobre Interfaces de Sistema, Interfaces de Negócio, baseadas no estudo de caso realizado neste trabalho e a Especificação da Arquitetura do Componente, utilizando os componentes, desenvolvidos neste trabalho.

5.4.1.1 Interfaces de Sistema

Para identificar interfaces do sistema é necessário que os casos de uso sejam observados a cada passo de execução e, a partir deles, pode-se obter informação de quais operações o sistema deve prover para atingir as responsabilidades definidas nos requisitos.

Cada caso de uso é mapeado para um **tipo de interface** (<<interface type>>) da camada de sistema. Um tipo de interface será realizado por uma interface de um componente quando o mesmo for implementado. Para cada passo de um caso de uso, de seu fluxo normal e alternativo, verifica-se se a ação do processo representa uma responsabilidade do sistema. Caso represente, o passo é mapeado para uma ou mais operações do tipo de interface definida para o caso de uso.

Nascimento (2005) ressalta que essas operações não são mapeadas diretamente com os passos de um para um, ou seja, para atender às necessidades, cada passo pode ter zero ou muitas operações providas pelo sistema.

“Como a proposta é obter uma solução de software para o problema, o modelo de negócios levará em consideração a estrutura do software” (BARROCA; GIMENES; HUZITA, 2005). As autoras descrevem que devem ser identificados os objetos de domínios

que terão representação de software, os atributos dos objetos que devem ser definidos em classes e os relacionamentos entre as classes.

No caso de haver restrições na aplicação, também deverão ser especificadas. Segundo Barroca; Gimenes; Huzita (2005), para fazer a especificação, as restrições são classificadas como: invariantes ou pré e pós-condições das operações realizadas na aplicação.

A partir das especificações é possível identificar as operações necessárias para as interfaces de sistema. Segundo Sousa (2004), as interfaces são específicas para cada sistema, mas devem ser comuns para várias aplicações. Analisando o Modelo de Caso de Uso da Realização de uma Venda, Quadro 5.3, é possível identificar as operações para as interfaces RegistrarVenda, e CadastrarCliente.

A Seção seguinte apresenta as Interfaces de Negócio, baseando-se na aplicação desenvolvida neste trabalho, explicando como identifica-las e como especificá-las.

5.4.1.2 Interfaces de Negócio

Para a identificação de interface do negócio, que segundo Nascimento (2005) são abstrações da informação manipulada pelo sistema, se dá pelo refinamento do modelo conceitual, dando assim origem ao modelo de tipos de negócio. Este refinamento se dá pela remoção de entidades que não precisam ser gerenciadas pelo sistema de software.

A Figura 5.6 é um refinamento do Modelo Conceitual da aplicação, Figura 5.4, em que foi feita uma análise detalhada dos requisitos e casos de uso da aplicação, referente à parte destacada.

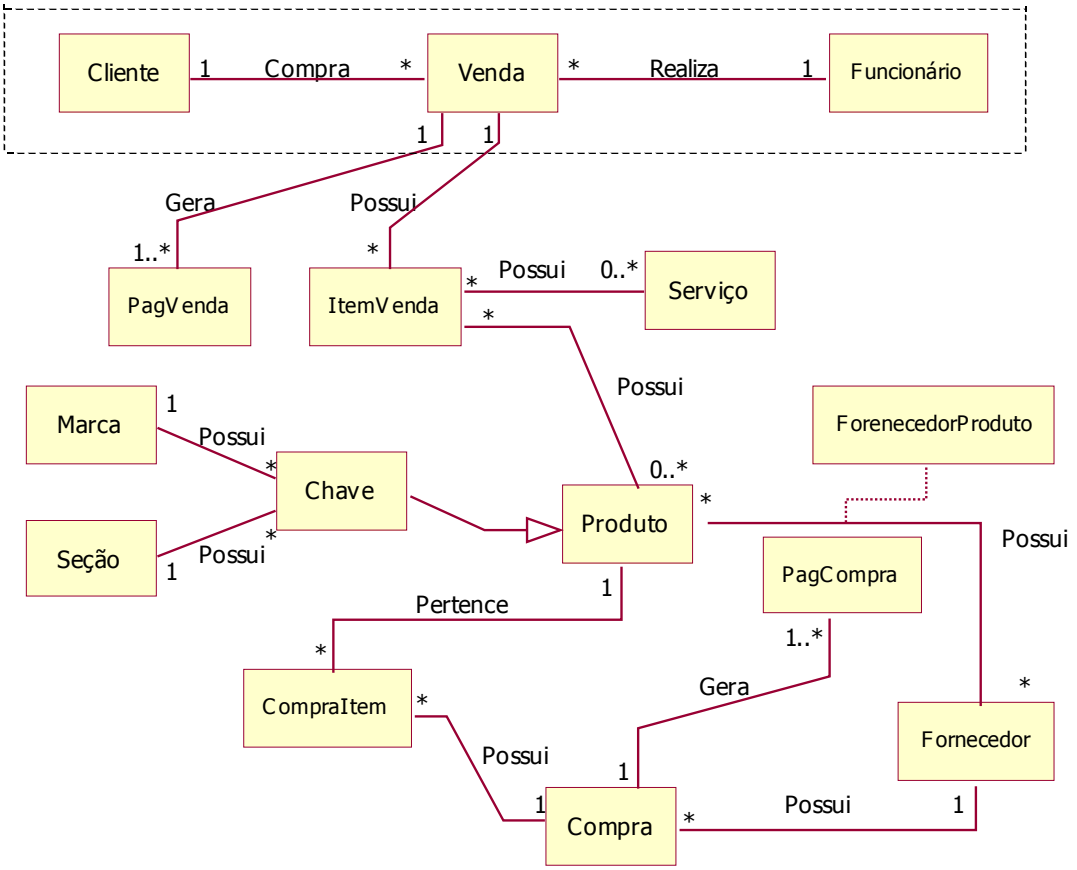

Figura 5.6 – Refinamento do Modelo Conceitual da aplicação resultando no Modelo de Tipos de Negócio.

A partir dessa análise, somente PagFunc foi considerado irrelevante por não fazer parte do contexto da aplicação, por isso houve a sua eliminação. Esse refinamento resulta na Figura 5.7 em que apresenta o Diagrama de Tipos de Negócio, referente à parte destacada.

Pagano (2004): Barroca; Gimenes; Huzita (2005) ressaltam que, as classes são consideradas como tipo e não implementações, ou seja, para essa fase é utilizado o estereótipo <<type>> para distinguir “Classe de Projeto” de “Classe de Implemantação”.

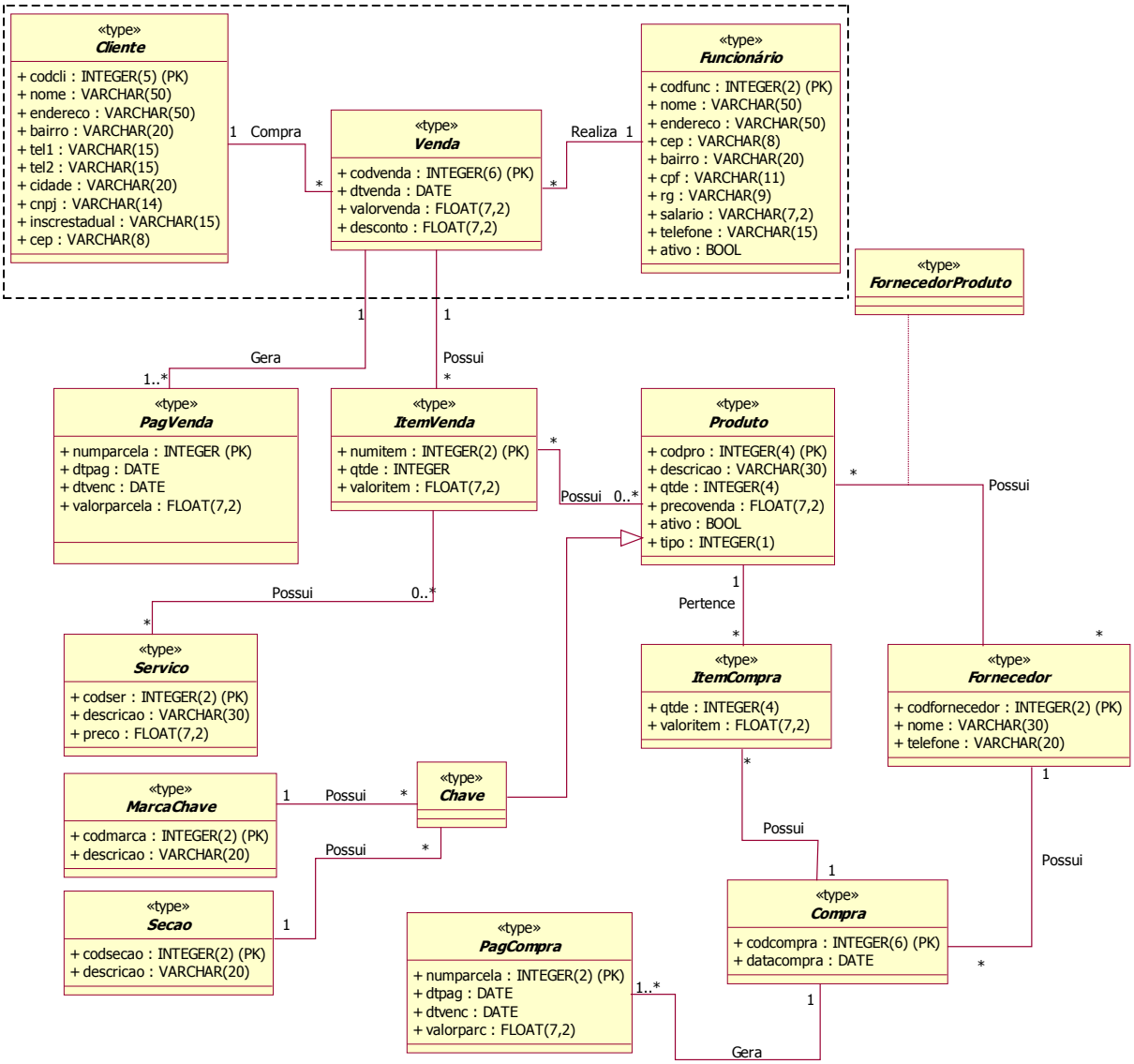

Figura 5.7 - Diagrama de Tipos de Negócio.

O refinamento do modelo conceitual resulta na identificação dos tipos principais e na criação de interfaces de cada tipo. Segundo Barroca; Gimenes; Huzita (2005), para os tipos que não fazem parte de um conjunto dos principais é necessário identificar a quais interfaces eles pertencem.

Cheesman e Daniels (2001 apud BARROCA;GIMENES e HUZITA, 2005) "identificam o conceito de tipos principais como sendo os tipos que têm existência independente", ou seja, um tipo principal (núcleo) não tem associações obrigatórias com outros tipos, exceto com um tipo de categorização.

A partir do Diagrama na Figura 5.7 pode-se identificar os tipos <<core>>, que conforme Seção 5.3.6, indica o tipo central do negócio. O autor ressalta que pelo fato da UML não permitir que uma entidade tenha mais de um estereótipo, o tipo <<core>> subentende o <<type>>. De acordo com a Figura 5.7, os tipos <<core>> são identificados nas entidades Cliente e Funcionário, pois um Cliente pode estar cadastrado na aplicação sem necessariamento ter comprado um produto ou solicitado um serviço e um Funcionário, pode estar cadastrado e não ter sido registrado em suas vendas realizadas, pois a aplicação não trata isso. No caso da Venda, essa não pode existir sem um Cliente.

Esta fase preocupa-se apenas com os atributos e os métodos são abordados na fase Especificação do Componente, próxima fase. Portanto, não se preocupa com o comportamento, e sim com as propriedades das entidades.

5.4.1.3 Especificação da Arquitetura do Componente

Neste estágio uma arquitetura começa a ser concebida, pois 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 (SOUSA, 2004). Ao final deste estágio são geradas as especificações iniciais das interfaces e da arquitetura de componentes.

A arquitetura dos componentes desenvolvidos neste trabalho são apresentados na Figura 5.8, em que cada componente é associado com cada interface.

As interfaces de negócio envolvidas com o gerenciamento das informações estão separadas das outras interfaces de negócio. Aqui, as interfaces do sistema estão todas associadas a um único componente, no caso o Oficina, que se comunica com todas as interfaces dos componentes, pois elas são muito relacionadas representando uma unidade.

Além disso, segundo Barroca; Gimenes; Huzita (2005), não é muito recomendável criar muitas interdependências entre componentes.

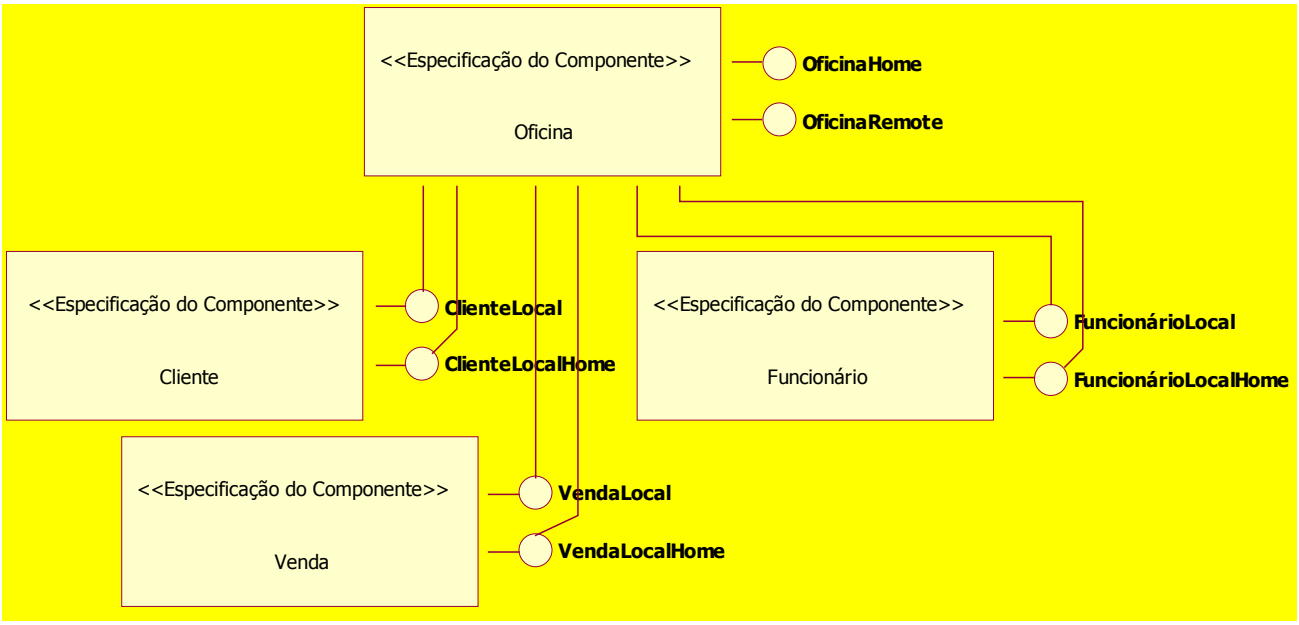

Figura 5.8 – Especificação da arquitetura dos componentes referentes à Venda da aplicação.

Depois de feita a especificação da arquitetura dos componentes, a Seção seguinte descreve como é realizada a interação entre os componentes, mas não é exemplificada com o estudo de caso de teste.

5.4.2 Interação do Componente

Depois de realizada uma especificação inicial de interfaces na fase de identificação do componente, este é o momento em que as interações entre os componentes do sistema são identificadas. Através de modelos de interação, são descobertas mais operações com suas assinaturas completas e as interfaces podem ser refinadas, agrupadas ou divididas (PAGANO, 2004).

Segundo Barroca; Gimenes; Huzita (2005), a análise da interação entre componentes inicia-se com:

- um conjunto de interfaces de negócio;
- um conjunto de interfaces de sistema e;
- uma especificação inicial dos componentes e suas arquiteturas.

De acordo com as autoras, nesse estágio podem ser identificadas novas operações de negócio, pois o sistema, através de diagramas de colaboração, será detalhado em termos de suas interações.

Para atingir o objetivo desta fase, alguns passos devem ser seguidos (NASCIMENTO, 2005):

- **Desenvolver modelos de interação para cada operação do sistema**: todos os casos de uso são analisados e são verificados quais passos precisam de uma operação correspondente no sistema. Essas operações descobertas podem ser auxiliadas pela UML, utilizando diagramas de colaboração.
- **Descobrir operações das interfaces do sistema e suas assinaturas**: a partir de diagramas de colaboração, é possível identificar as operações necessárias, e com isso, especificar as assinaturas de cada operação, mas não necessariamente todas, podendo algumas passar por outros refinamentos.
- **Refinar responsabilidades**: uma vez encontradas as operações das interfaces e suas assinaturas, as responsabilidades dos tipos de negócio podem, não obrigatoriamente, serem refinadas.
- **Definir restrições necessárias**: esse é um passo importante para definir as regras de negócio do sistema. São ressaltadas as restrições que levam a definição de pré e pós-condições das operações do sistema na próxima fase de especificação do componente.

Nascimento (2005) ressalta que os passos acima devem ser seguidos na ordem que foram apresentados.

5.4.3 Especificação do Componente

Nesta fase é feita uma especificação detalhada definindo as operações do componente, denotadas como pré e pós-condições, as invariantes para os componentes, as restrições nas interações dos componentes e as restrições nas interações das interfaces (SOUSA, 2004). Segundo Pagano (2004), isso ocorre com base nas informações definidas nos estágios anteriores.

Segundo Barroca; Gimenes; Huzita (2005), "o que já foi feito para as operações mais gerais de interface de sistema deve ser repetido para cada operação de interface de sistema e para cada operação de interface de negócio".

Na Figura 5.9 especifica-se a interface de negócio do componente Oficina, descrevendo todos os seus métodos definidos para a realização de suas funções. Para, por exemplo, inserir um novo cliente, o componente Oficina, que representa o *Session Façade*, precisa invocar o componente Cliente, que representa os dados da tabela.

Esse mesmo processo deve ser seguido para especificar as demais interfaces de negócio do sistema. Essas especificações se referem ao uso dos componentes e representam para o desenvolvedor o que ele precisa saber sobre os componentes (BARROCA; GIMENES e HUZITA, 2005).

A especificação se dá como completa quando forem consideradas também as interfaces que o componente demanda de outros componentes, que podem ser obtidas pelos diagramas de colaboração.

Segundo as autoras, se fosse necessário o estabelecimento de alguma restrição adicional sobre a comunicação entre os componentes, as especificações deveriam ser mais detalhadas.

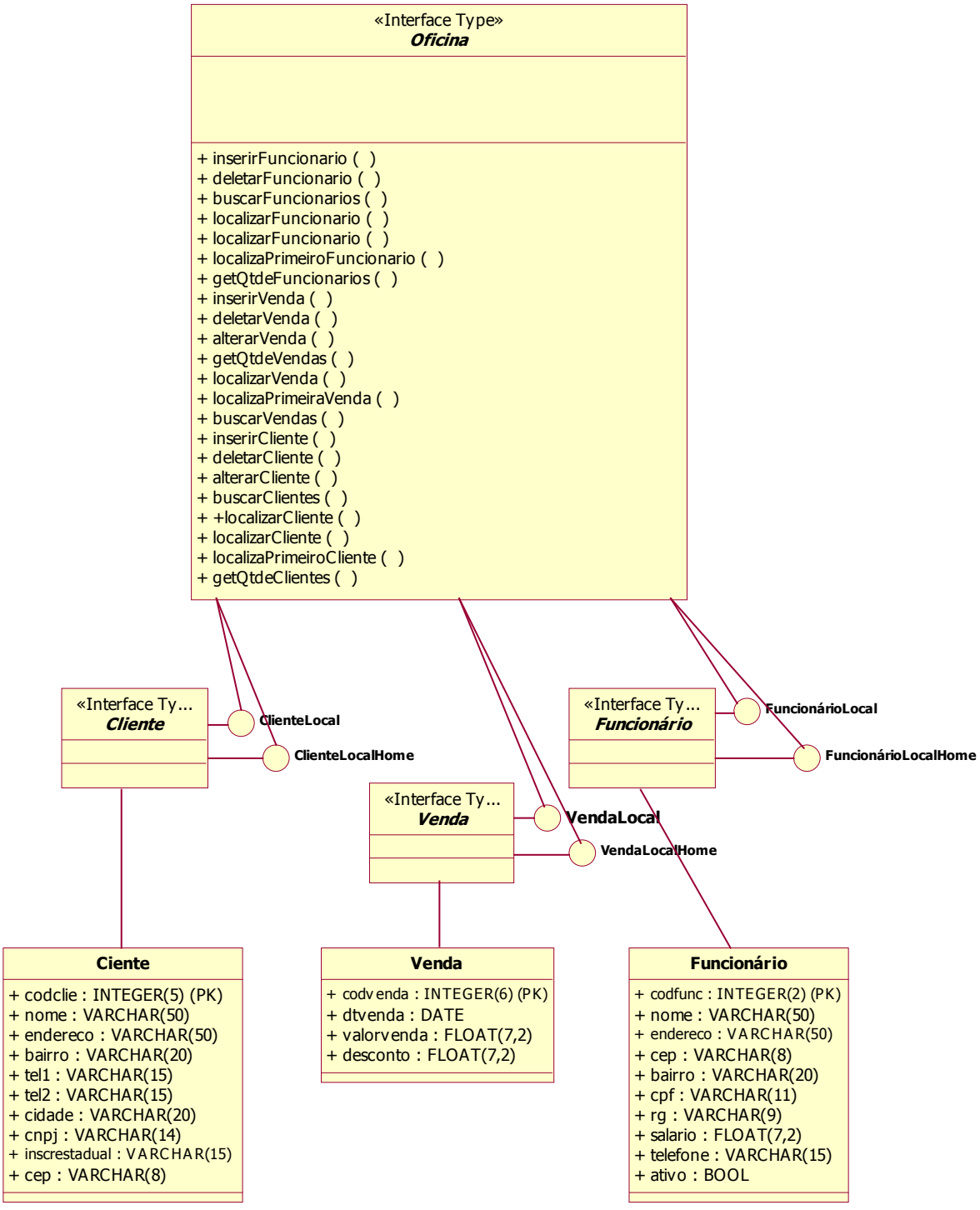

Figura 5.9 – Especificação da Interface do componente Oficina.

5.5 Considerações Finais

Neste capítulo foi abordado o método UML *Components*. Primeiramente foi descrito o método e apresentado os estereótipos utilizados pelo método, pois como já mencionado o método é uma extensão da linguagem UML. Em seguida foi apresentado o Processo do UML *Components*, dando ênfase somente nas atividades de Especificação, Adequação e Codificação. As atividades abordadas nesse capítulo substituem diretamente as atividades de Análise, Projeto e Implementação do RUP.

A descrição do Processo do UML *Components* foi feita se baseando no estudo de caso de uma loja comercial, que foi implementada com a composição dos componentes desenvolvidos neste trabalho.

Para o reúso, deve haver a documentação dos componentes, pois abrange todas as catacterísticas necessárias para o desenvolvedor. O UML *Components* com a utilização de estereótipos, faz a documentação de modo bastante viável, deixando o desenvolvedor livre para a implementação em qualquer plataforma.

6 COMPARAÇÃO ENTRE IMPLEMENTAÇÕES DE COMPONENTES DE SOFTWARE

6.1 Considerações Iniciais

Este capítulo aborda o desenvolvimento de uma aplicação três camadas utilizando componentes EJB. Na Seção 6.2 é mostrada parte da análise de projeto da aplicação que foi desenvolvida com a utilização de componentes. Na Seção 6.3 são apresentadas as ferramentas utilizadas para a criação da aplicação. Em um primeiro momento foi construída a aplicação sem a utilização do padrão de projeto, abordada na Seção 6.4. Depois, construiu-se a mesma aplicação, abordada na Seção 6.5, com o padrão *Session Façade*. As duas implementações são descritas com o intuito de se verificar as principais diferenças e características de cada e, por fim, realizar uma discussão, apresentada na Seção 6.6, da utilização dos componentes com e sem o padrão *Session Façade*.

6.2 Estudo de Caso

A aplicação desenvolvida abrange o cadastro de clientes, funcionários e a realização das vendas de uma loja comercial. Esta funcionalidade abrange uma parte de toda a análise e projeto da aplicação apresentada parcialmente na Seção 5.4 do Capítulo 5. Na Figura 6.1 mostra-se o Diagrama Entidade-Relacionamento (DER) (CHEN 1976 apud YOURDON, 1992) de toda a aplicação bem como a parte desenvolvida com os componentes EJB, que está delimitada por uma linha tracejada.

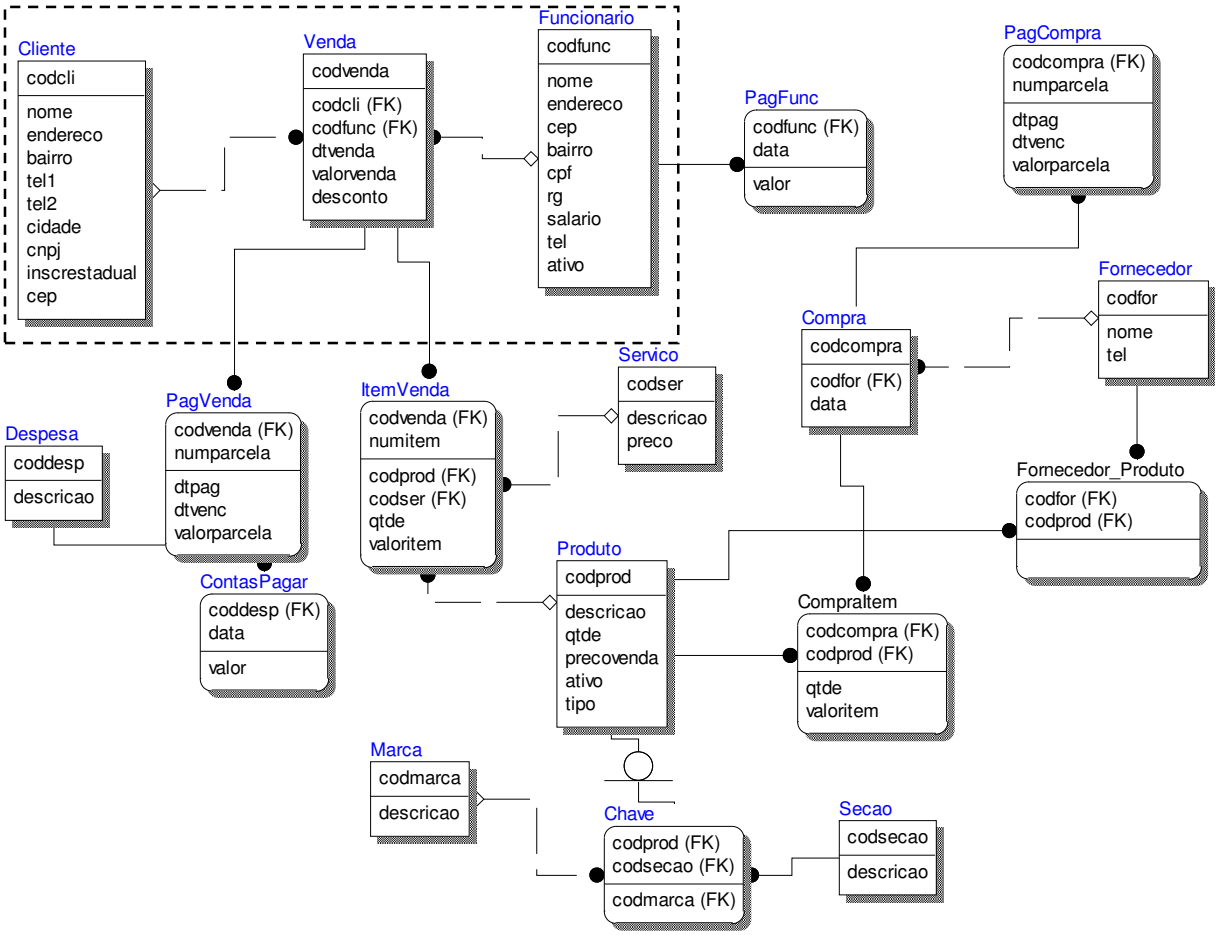

Figura 6.1 – Diagrama Entidade-Relacionamento.

6.3 Ferramentas Utilizadas

As ferramentas utilizadas para a construção da aplicação neste trabalho foram:

- Banco de dados MySQL Server 5.0[7]: A principal razão para a escolha deste banco é por ele ser gratuito. É um banco de dados muito utilizado, eficiente, multi-plataforma e de simples instalação.
- Servidor de aplicação JBOSS na versão 4.0.1sp1[8]: Além de estar de acordo com a especificação J2EE, este servidor é gratuito, flexível e

[7] Disponível em <www.mysql.com>.
[8] Disponível em <www.jboss.com>.

robusto. Pode ser integrado com outras ferramentas *open source* para a criação de ambientes de desenvolvimento.

- Versão 1.4.2 da linguagem de programação Java: para a criação das interfaces gráficas.

6.4 Aplicação sem o Padrão *Session Façade*

Para a aplicação sem o padrão *Session Façade*, foram desenvolvidos três componentes *Entity Beans* do tipo BMP: `Funcionário`, `Cliente` e `Venda`. Eles representam os dados do banco e são acessados diretamente pelo cliente da aplicação. Cada componente possui interfaces *Home* e *Remote* para permitir o acesso remoto do cliente com os mesmos.

6.4.1 Arquitetura

A aplicação sem o uso do padrão *Session Façade*, possui uma arquitetura de três camadas. Na Figura 6.2 ilustra-se a arquitetura da aplicação.

A primeira camada consiste na aplicação cliente (`Aplicação desktop`, Figura 6.2), ou seja, é a camada relacionada à lógica de apresentação que providencia a interface gráfica para a interação do usuário com a aplicação. Nesta camada tem-se a aplicação `desktop` desenvolvida em Java na versão 1.4.2.

A segunda camada consiste na camada de negócios (`Servidor de aplicação`, Figura 6.2) que realiza as operações relacionadas às lógicas comerciais da aplicação. Esta camada é representada pelo servidor de aplicação JBOSS que contém os componentes EJB.

Estes componentes são os três *beans* de entidade que representam os dados da aplicação (`Cliente`, `Venda` e `Funcionário`).

A terceira camada (`Banco de Dados`, Figura 6.2) está associada com a lógica de acesso aos dados que se relaciona diretamente com o banco de dados. Ela está representada nesta aplicação pelo banco de dados MySQL.

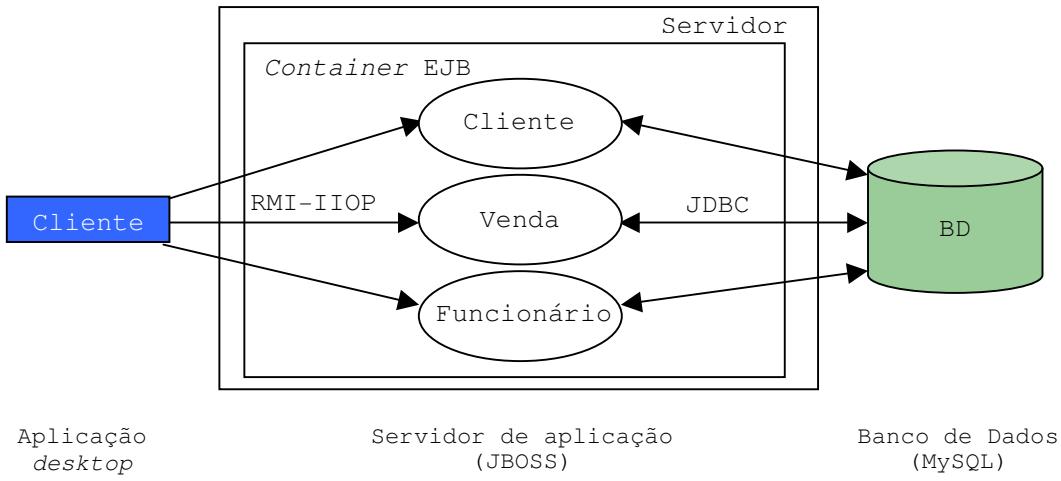

Figura 6.2 – Arquitetura da aplicação sem o padrão *Session Façade*.

Apesar dos *beans* de entidade `Cliente`, `Venda` e `Funcionário` estarem dispostos no servidor de aplicação JBOSS, a principal finalidade que possuem é de representar os dados do banco de dados. Portanto, nesta implementação sem o padrão, a lógica comercial deve ser implementada juntamente com o aplicativo cliente.

Nota-se maior dependência do cliente com a estrutura do banco de dados representados pelos *beans* de entidade, pois o cliente possui acesso direto a estes *beans*. A lógica de negócio está localizada juntamente com o aplicativo cliente e, caso haja a necessidade de alterá-las, o aplicativo cliente também haveria de ser alterado.

O cliente se relaciona com as interfaces dos três *beans* de entidade e, por não existir uma interface uniforme de acesso ao sistema, o cliente necessita conhecer o sistema mais profundamente para realizar a comunicação com o mesmo.

O acesso do cliente acontece via RMI-IIOP e quanto maior o número de chamadas remotas, maior o tráfego na rede. Os *beans* de entidade utilizam a JDBC para o acesso aos dados.

6.4.2 Interface Gráfica do Cliente

A camada que providencia a apresentação dos dados é constituída pela aplicação *desktop* Java. Esta aplicação é composta por três módulos que realizam todas as operações para a manipulação dos dados por meio dos componentes. Estes módulos correspondem aos dados do funcionário, cliente e venda. Nas Figuras 6.3 e 6.4 ilustra-se a interface relacionada ao funcionário.

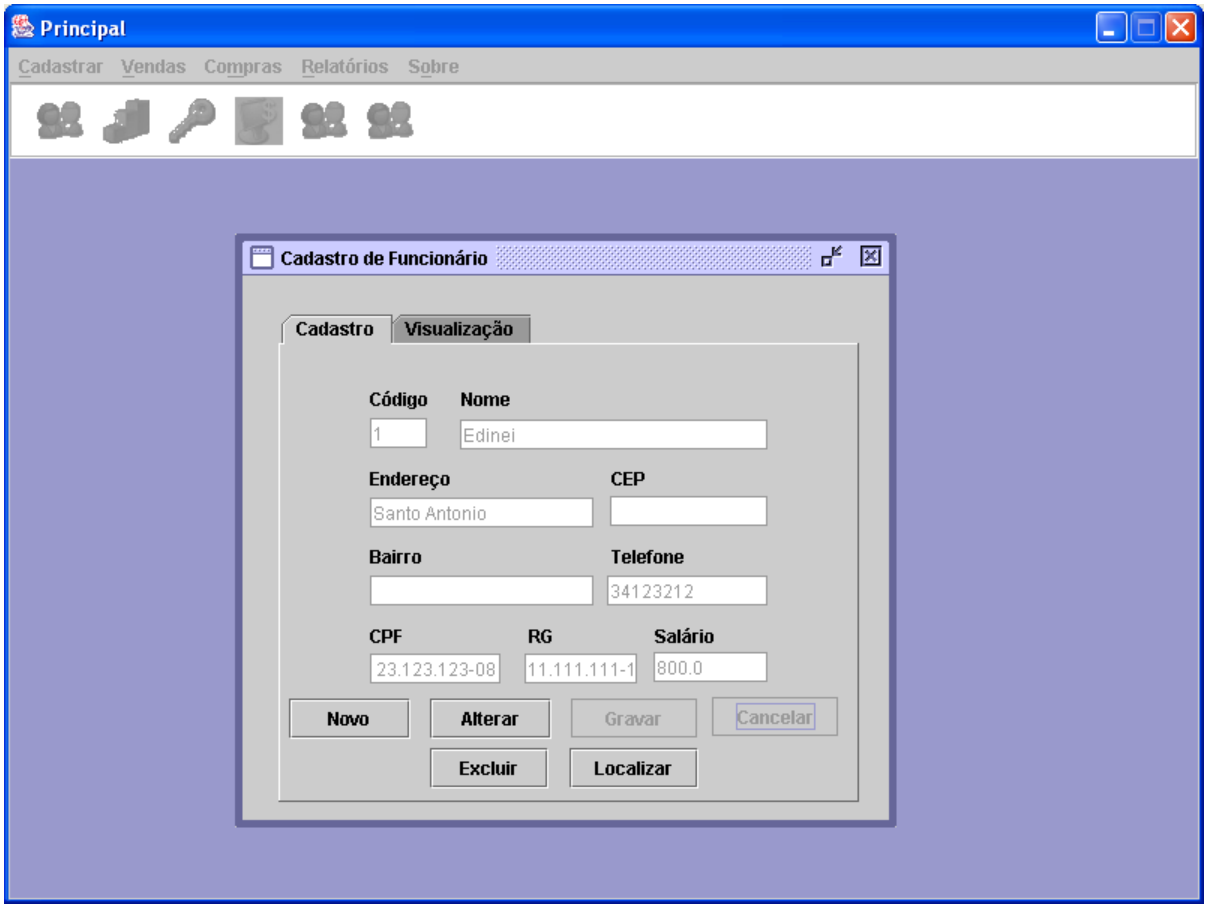

Figura 6.3 – Cadastro de Funcionário.

Na parte de cadastro de funcionário (Figura 6.3), pode-se realizar as operações de cadastro, alteração e exclusão de um funcionário.

Na parte de visualização dos dados (Figura 6.4), pode-se realizar operações de busca pelo código ou pelo nome do funcionário.

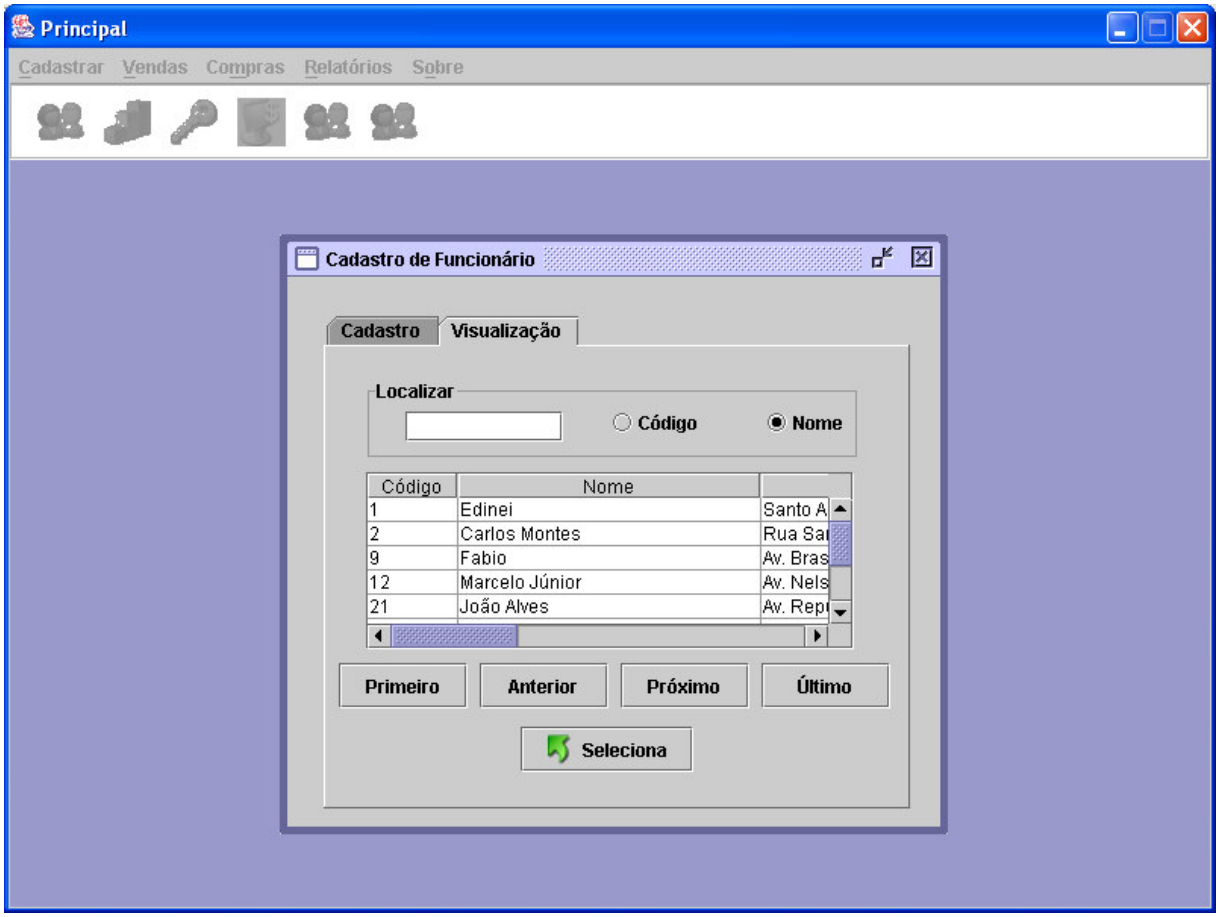

Figura 6.4 – Visualização de Funcionário.

A interface gráfica para o cadastro de cliente, funcionário e realização da venda foi implementada uma única vez durante a construção da aplicação sem o uso do padrão *Session Façade* e reutilizada durante a construção da aplicação com o uso do padrão. Porém, existem diferenças no código de cada implementação ao relacionarem-se com os componentes.

No Quadro 6.1 são ilustrados trechos de código existentes na aplicação *desktop* referentes aos métodos `setContexto()`, que realiza a localização dos componentes, e `selecionaFuncionarios()`, que realiza a busca de todos os funcionários.

Para a aplicação cliente acessar o serviço de nomes JNDI foi necessária a criação de um `InitialContext()` com os parâmetros (Rótulo 1):

- `Context.INITIAL_CONTEXT_FACTORY`: especifica qual a classe que fabrica objetos `InitialContext`.
- `Context.PROVIDER_URL`: especifica a localização do servidor de nomes.

Os valores destes parâmetros foram configurados para funcionar com o servidor de aplicação JBOSS utilizado nesta aplicação, pois cada servidor de aplicação possui uma classe diferente para fabricar os objetos `InitialContext`.

São localizados os três componentes via JNDI pelo método `lookup()` (Rótulo 2) e, posteriormente, adquire-se as referências das três interfaces *Home* (Rótulo 3) pelo método `narrow(ref, class)`. Este método assegura que o objeto `ref` possa ser convertido (*cast*) para a classe `class`, ou seja, transforma um objeto geral (do tipo `Object`) em um objeto de uma determinada classe (por exemplo, da classe `FuncionarioHome`). Java RMI-IIOP fornece este mecanismo pela classe `PortableRemoteObject`.

No rótulo 4 ocorre a criação de uma referência da interface `Funcionário` para cada registro encontrado no método `findAll()` da interface *Home*. Este método está especificado na interface *Home* e seleciona todos os funcionários existentes na tabela funcionário.

```
. . .
private void setContexto()    {
       // criando um objeto properties para construir um contexto inicial
1      Properties properties = new Properties();
       properties.put(Context.INITIAL_CONTEXT_FACTORY,
                   "org.jnp.interfaces.NamingContextFactory");
       properties.put(Context.PROVIDER_URL, "10.17.227.110:1099");

  try {
        jndiContext = new InitialContext(properties);

2      Object ref1 = jndiContext.lookup("Funcionario");
       Object ref2 = jndiContext.lookup("Cliente");
       Object ref3 = jndiContext.lookup("Venda");

3      home1 = (FuncionarioHome)PortableRemoteObject.narrow (ref1, FuncionarioHome.class);
       home2 = (ClienteHome)PortableRemoteObject.narrow (ref2, ClienteHome.class);
       home3 = (VendaHome)PortableRemoteObject.narrow (ref3, VendaHome.class);
  }
  catch(Exception e) {
        System.out.println(e.toString());
  }
}

public Vector[] selecionaFuncionarios()
{
        try{
           Iterator iterator = home1.findAll().iterator();

           //enquanto tem registros
           while (iterator.hasNext()) {
4            Funcionario funcionario = (Funcionario)javax.rmi.PortableRemoteObject.narrow(
                     iterator.next(), Funcionario.class);

             line[indice] = new Vector();
             line[indice].add(new Integer(funcionario.getCodFunc()));
             line[indice].add(new String(funcionario.getNome()));
             line[indice].add(new String(funcionario.getEndereco()));
             if(funcionario.getCep() == null)
                    line[indice].add(new String(""));
             else
                    line[indice].add(new String(funcionario.getCep()));
             . . .
        }
        return line;
}
. . .
```

Quadro 6.1 – Código Cliente sem o Padrão.

Nota-se que o cliente necessita interagir com as interfaces dos três *beans* de entidade. Com isso, o cliente fica acoplado com o sistema, pois se relaciona diretamente com os *beans* de entidade. Além disso, verifica-se a existência de muitas chamadas remotas ao *bean* `Funcionário` (Rótulo 4), o que aumenta o tráfego da rede.

6.4.3 *Beans* de Entidade

Como mencionado anteriormente, os três *beans* de entidade representam os dados da aplicação e são acessados diretamente pelo cliente. São do tipo BMP, o que permitiu a criação de consultas SQL dentro dos métodos do *bean* para a inserção e recuperação dos dados entre o banco de dados e o *bean*.

Cada *bean* de entidade é composto por quatro arquivos que são: a classe principal *bean*, as interfaces *Home* e *Remote* e uma classe que representa sua chave primária. Cada arquivo é abordado a seguir e, para isto, o componente Venda será descrito.

6.4.3.1 Classe VendaPK

A classe VendaPK é fundamental para a criação do *bean* de entidade, pois ela representa a chave primária de cada *bean* instanciado. Assim, o *container* consegue controlar todas as instâncias dos *beans* por meio desta classe. No Quadro 6.2 a classe `VendaPK` , que implementa a interface `Serializable`, é apresentada.

```
Package br.fundanet.lucio.oficina;
import java.io.Serializable;

public class VendaPK implements Serializable {

   public String codVenda;

   public VendaPK(String codigo) {
      this.codVenda = codigo;
   }
   public VendaPK() { }

   public boolean equals(Object vend) {
      if (!( vend instanceof VendaPK))
        return false;
      else
        return ((VendaPK)vend).codVenda.equals(codVenda);
  }
  public int hashCode() {
    return codVenda.hashCode();
  }
}
```

Quadro 6.2 – Classe `VendaPK`

Nota-se que a classe `VendaPK` possui o atributo público `codVenda` que representa a chave primária. Isto é um requisito da especificação EJB. Nesta classe devem ser implementados os métodos `equals()` e `hasCode()`. O método `equals()` é utilizado pelo *container* para verificar a igualdade entre as instâncias dos *beans*. O método `hasCode()` tem como objetivo gerar um inteiro que possa ser utilizado pelo *container* em tabelas indexadas.

6.4.3.2 Interface VendaHome

A interface `VendaHome` estende a interface `EJBHome` e todos os métodos devem lançar a exceção `RemoteException`, pois ela é acessada remotamente pelo cliente. Esta interface, mostrada no Quadro 6.3, contém a assinatura dos métodos para a criação e localização dos *beans* `Venda`. Estes métodos são implementados pela classe principal do componente chamada `VendaBean`. Dentre todos os métodos destacam-se:

- `create()`: método responsável por criar um *bean* de entidade e que corresponde ao método `ejbCreate()` da classe principal `VendaBean`.
- `findByPrimaryKey()`: método que localiza os *beans* pela chave primária.
- `findAll()`: método que seleciona todas as vendas existentes.

```
package br.fundanet.lucio.oficina;
. . .
public interface VendaHome extends EJBHome {
        public Venda create(int codVenda, int codCli, int codFunc,
                        String dtVenda, float valorVenda, float desconto)
                    throws RemoteException, CreateException,SQLException;

        public Venda findByPrimaryKey(VendaPK chavePK)
           throws RemoteException, FinderException,SQLException,NumberFormatException;

        public Collection findAll() throws RemoteException,FinderException;
. . .
}
```

Quadro 6.3 – Interface `VendaHome`.

6.4.3.3 Interface Venda

A interface `Venda` estende a interface `EJBObject` e todos os métodos devem lançar a exceção *RemoteException*, pois ela é acessada remotamente pelo cliente. No Quadro 6.4 a interface `Venda` é ilustrada. Ela possui métodos para obter (Rótulo 1) e métodos para alterar (Rótulo 2) os dados de uma venda.

```
package br.fundanet.lucio.oficina;
. . .
public interface Venda extends EJBObject {

      public int getCodVenda() throws RemoteException;
      public int getCodCli() throws RemoteException;
  1   public int getCodFunc() throws RemoteException;
      public String getDataVenda() throws RemoteException;
      public float getValorVenda() throws RemoteException;
      public float getDesconto() throws RemoteException;

      public void setCodVenda(String s)throws RemoteException;
      public void setCodCli(String s) throws RemoteException;
  2   public void setCodFunc(String s) throws RemoteException;
      public void setDataVenda(String s) throws RemoteException;
      public void setValorVenda(String s) throws RemoteException;
      public void setDesconto(String s) throws RemoteException;
}
```

Quadro 6.4 – Interface `Venda`.

6.4.3.4 Classe VendaBean

A classe `VendaBean` implementa a interface `EntityBean` e é a principal parte do componente `Venda`, pois realiza a implementação dos métodos especificados nas interfaces `VendaHome` e `Venda`. Esta classe, parcialmente ilustrada no Quadro 6.5, possui os atributos necessários de uma venda (Rótulo 1). Ressalta-se a existência do atributo `contexto` (Rótulo 2) que representa o contexto de execução do *container*. No método `setEntityContext()` (Quadro 6.6) ocorre a associação do *bean* ao contexto de execução e, com isso, o *bean* consegue obter informações do ambiente, como por exemplo, o valor da chave primária (Quadros 6.9 e 6.10, Rótulo 2).

```
. . .
public class VendaBean implements EntityBean {

      private int codVenda;
      private int codCli;
  1   private int codFunc;
      private String dtVenda;
      private float valorVenda;
      private float desconto;

  2   private EntityContext contexto;
. . .
```

Quadro 6.5 – Atributos de `VendaBean`.

Além da implementação dos métodos das interfaces `Venda` e `VendaHome`, a classe `VendaBean` também deve implementar os métodos especificados na interface `EntityBean` que são utilizados pelo *container* e correspondem à: `setEntityContext()`, `ejbFindByPrimaryKey()`, `ejbCreate()`, `ejbRemove()`, `ejbStore()` e `ejbLoad()`. Tais métodos são ilustrados a seguir.

O principal trecho do método `setEntityContext()` é ilustrado no Quadro 6.6. O contexto de execução do *container* é atribuído ao atributo `contexto` do bean (Rótulo 1).

```
. . .
public class VendaBean implements EntityBean {

        public void setEntityContext(EntityContext ctx)throws EJBException
        {
  1             contexto = ctx;
          . . .
```

Quadro 6.6 – Método `setEntityContext()` de `Venda`.

No Quadro 6.7 ilustra-se o método `ejbCreate()`. Este método é responsável pela criação do *bean* e pela inserção dos dados no banco de dados. Ele recebe como parâmetro os valores referentes à venda e verifica a existência do funcionário e do cliente que participam da venda (Rótulo 1). Neste método existe o comando SQL necessário para a inserção dos dados no banco de dados (Rótulo 2).

```
. . .
      public VendaPK ejbCreate(int codVenda, int codCli, int codFunc,String dtVenda,
          float valorVenda, float desconto) throws CreateException,SQLException
      {
            this.codVenda = codVenda;
            this.codCli = codCli;
            this.codFunc = codFunc;
            this.dtVenda = dtVenda;
            this.valorVenda = valorVenda;
            this.desconto = desconto;

            /*verificando se o funcionário e o cliente existem*/
            try{
            clienteHome.findByPrimaryKey(new ClientePK(String.valueOf(codCli)));

            }catch(Exception fe){
                  throw new CreateException("cliente não existe.");
            }
      1     try{
            funcionarioHome.findByPrimaryKey(new FuncionarioPK(String.valueOf(codFunc)));

            }catch(Exception fe){
                  throw new CreateException("funcionario não existe.");
            }

            Connection con = null;
            PreparedStatement ps = null;
            try {
                  String sql = "INSERT INTO Venda" +
      2           "(codvenda,codcli,codfunc,dtvenda,valorvenda,desconto) " +
                  "VALUES" + " (?,?,?,str_to_date(?,'%d/%m/%Y'),?,?)";
                  con = getConnection();
                  ps = con.prepareStatement(sql);
                  ps.setInt(1, codVenda);
                  ps.setInt(2, codCli);
                  ps.setInt(3, codFunc);
                  ps.setString(4, dtVenda);
                  ps.setFloat(5, valorVenda);
                  ps.setFloat(6, desconto);
                  ps.executeUpdate();
            }
            catch (SQLException e) { throw e; }
            finally {
                  try {
                    if (ps != null)
                      ps.close();
                    if (con != null)
                      con.close();
                  } catch (SQLException e) {}
            }
            return new VendaPK(Integer.toString(codVenda));
      }
```

Quadro 6.7 – Método `ejbCreate()` de `VendaBean`.

No Quadro 6.8 ilustra-se o método `ejbFindByPrimaryKey()` que efetua a busca de um *bean* pela sua chave primária (Rótulo 1) e retorna a classe da chave primária encontrada (Rótulo 2). Pode lançar a exceção `ObjectNotFoundException` caso o *bean* não seja encontrado.

```
    public VendaPK ejbFindByPrimaryKey(VendaPK chavePK)
        throws FinderException,SQLException,NumberFormatException
    {
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
1         String sql = " SELECT codvenda FROM Venda" + " WHERE codvenda = ? ";

          int codigoVenda = Integer.parseInt(chavePK.codVenda);
          con = getConnection();
          ps = con.prepareStatement(sql);
          ps.setInt(1, codigoVenda);
          rs = ps.executeQuery();
          if (rs.next()) {
2           return chavePK;
          }
        }
        catch (SQLException e) { throw e; }
        finally {
          try {
            if (rs!=null)
              rs.close();
            if (ps!=null)
              ps.close();
            if (con!=null)
              con.close();
          }
          catch (SQLException e) {  }
        }throw new ObjectNotFoundException();
    }
```

Quadro 6.8 – Método `ejbFindByPrimaryKey()` de `VendaBean`.

No Quadro 6.9 é apresentado o método `ejbRemove()`. Este método realiza a exclusão dos dados do banco de dados com um comando SQL (Rótulo 1) e o *container* se encarrega de destruir a instância do *bean* corrente.

```
    public void ejbRemove() throws RemoveException,EJBException {

        Connection con = null;
        PreparedStatement ps = null;
        try {
1           String sql = "DELETE FROM Venda WHERE codvenda = ? ";

            /* recupera a PK corrente da instância do bean de entidade */
2         VendaPK busca = (VendaPK) contexto.getPrimaryKey();

          int codigo = Integer.parseInt(busca.codVenda);
          con = getConnection();
          ps = con.prepareStatement(sql);
          ps.setInt(1, codigo);
          ps.executeUpdate();
        }
        catch (SQLException e) {
          System.err.println(e.toString());
        }
        . . .
    }
```

Quadro 6.9 – Método `ejbRemove()` de `VendaBean`.

O método `ejbStore()`, ilustrado no Quadro 6.10, é chamado pelo *container* para que os dados do *bean* sejam atualizados no banco de dados. O *bean* é do tipo BMP, ou seja, o próprio *bean* é que controla sua persistência. Por isso, foi necessária a criação de um comando SQL (Rótulo 1) para persistir os dados contidos nos atributos do *bean* para o banco de dados. No rótulo 3, os dados que estão nos atributos do *bean* são carregados no comando SQL e este é executado.

Caso exista alguma alteração nos atributos do *bean*, o *container* invoca o método `ejbStore()` automaticamente, garantindo a integridade dos dados da aplicação.

```
    public void ejbStore() throws EJBException {
        Connection con = null;
        PreparedStatement ps = null;
        try {
          String sql = "UPDATE Venda" + " SET codcli = ?, codfunc = ?," +
1               " dtvenda = str_to_date(?,'%d/%m/%Y'), valorvenda = ?," +
                " desconto = ? WHERE codvenda = ?";

2           VendaPK chave = (VendaPK) contexto.getPrimaryKey();
            int codigoVenda = Integer.parseInt(chave.codVenda);

            con = getConnection();
            ps = con.prepareStatement(sql);
            ps.setInt(1, this.codCli);
            ps.setInt(2, this.codFunc);
3           ps.setString(3, this.dtVenda);
            ps.setFloat(4, this.valorVenda);
            ps.setFloat(5, this.desconto);
            ps.setInt(6, codigoVenda);
            ps.executeUpdate();
          }
          catch (SQLException e) { System.out.println(e.toString());
          }
          finally {

            try {
              if (ps!=null)
                ps.close();
              if (con!=null)
                con.close();
            }
            catch (SQLException e) { }
          }
    }
```

Quadro 6.10 – Método `ejbStore()` de `VendaBean`.

O método `ejbLoad()`, ilustrado no Quadro 6.11, realiza o inverso do método `ejbStore()`. Ele é chamado pelo *container* para recuperar os dados do banco (Rótulo 1) e carregá-los nos atributos do *bean* (Rótulo 2).

```
    public void ejbLoad() throws EJBException
    {
        VendaPK cod = (VendaPK) contexto.getPrimaryKey();
        this.codVenda = Integer.parseInt(cod.codVenda);
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
          String sql = "SELECT codcli,codfunc,date_format(dtvenda,'%d/%m/%Y')
1             data,valorvenda,desconto" + " FROM Venda WHERE codvenda = ?";

          con = getConnection();
          ps = con.prepareStatement(sql);
          ps.setInt(1, this.codVenda);
          rs = ps.executeQuery();
          if (rs.next()) {
                   this.codCli = rs.getInt(1);
                   this.codFunc = rs.getInt(2);
2                  this.dtVenda = rs.getString(3);
                   this.valorVenda = rs.getFloat(4);
                  this.desconto = rs.getFloat(5);
          }
        }
        catch (SQLException e) { System.out.println(e.toString()); }
        finally {
          try {
            if (rs!=null)
              rs.close();
            if (ps!=null)
              ps.close();
            if (con!=null)
              con.close();
          }
          catch (SQLException e) { }
        }
    }
```

Quadro 6.11 – Método `ejbLoad()` de `VendaBean`.

O método `getConnection()`, ilustrado no Quadro 6.12, desempenha uma função importante, pois localiza via JNDI (Rótulo 1) o *data-source* utilizado na aplicação. O *data-source* é responsável por estabelecer a conexão com o banco de dados e será abordado na Subseção 6.4.4. Todo método que necessite persistir, atualizar, recuperar ou remover dados do banco invoca o método `getConnection()` para obter a conexão.

```
public Connection getConnection()
{
    Connection conexao = null;
    DataSource dataSource = null;
    try {
       InitialContext contexto = new InitialContext();
1      dataSource = ( DataSource ) contexto.lookup( "java:jdbc/MySqlDS" );

    } catch ( NamingException namingException ) {
        System.err.println( namingException.getMessage() );
    }
    try {
        conexao = dataSource.getConnection();
    }
       catch (SQLException e) {
          System.out.println(e.toString());
       }
           return conexao;
  }
```

Quadro 6.12 – Método `getConnection()` de `VendaBean`.

6.4.4 Arquivo *Data-Source*

O *data-source* agrega um conjunto de atributos que viabilizam a conexão com o banco de dados sendo de grande importância na aplicação. Ele se encontra em um arquivo nomeado *myslq-ds.xml* que está localizado no diretório *deploy* do JBOSS. Os *beans* de entidade localizam este *data-source* pela JNDI. No Quadro 6.13 é mostrado o arquivo *myslq-ds.xml* utilizado para a conexão com o banco de dados.

O *data-source* possui diversos parâmetros utilizados na conexão (Quadro 6.13). Dentre os mais importantes destacam-se: nome a ser utilizado pela JNDI (`<jndi-name>`), *url* de conexão (`<connection-url>`), *driver* utilizado (`<driver-class>`), usuário (`<user-name>`) e senha (`<password>`).

Utilizou-se o *driver* Java para a conexão com o MySQL que corresponde ao arquivo *mysql-connector-java-3.0.11-stable-bin.jar* e que deve ser colocado na pasta *lib* do JBOSS. O mesmo *data-source* foi utilizado nas implementações com e sem o padrão *Session Façade*.

É importante ressaltar o benefício que o *data-source* agrega à aplicação, pois diferentes *data-sources* podem ser distribuídos no *container* EJB para fornecer aos *beans* de entidade a possibilidade de acesso a diferentes bancos de dados.

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/oficina</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>cliente</user-name>
    <password>cliente</password>
    <exception-sorter-class-name>
       org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
    </exception-sorter-class-name>
    <metadata>
       <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Quadro 6.13 – Arquivo *mysql-ds.xml*.

6.4.5 Descritor de Distribuição

O descritor de distribuição é o arquivo *ejb-jar.xml* que contém as metainformações dos componentes criados nesta aplicação. É importante ressaltar que esse descritor é essencial para que o *container* EJB obtenha os dados referentes aos componentes. No Quadro 6.14 é apresentado um fragmento do descritor de distribuição utilizado nesta implementação.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <ejb-jar
3     xmlns="http://java.sun.com/xml/ns/j2ee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
6                         http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
7     version="2.1">
8      <description>Beans de Entidade</description>
9      <display-name>Beans BMP</display-name>
10     <enterprise-beans>
11            <entity>
12                   <ejb-name>Venda</ejb-name>
13                   <home>br.fundanet.lucio.oficina.VendaHome</home>
14                   <remote>br.fundanet.lucio.oficina.Venda</remote>
15                   <ejb-class>br.fundanet.lucio.oficina.VendaBean</ejb-class>
16                   <persistence-type>Bean</persistence-type>
17                   <prim-key-class>br.fundanet.lucio.oficina.VendaPK</prim-key-class>
18                   <resource-ref>
19                          <res-ref-name>jdbc/MySqlDS</res-ref-name>
20                          <res-type>javax.sql.DataSource</res-type>
21                          <res-auth>Container</res-auth>
22                          <res-sharing-scope>Shareable</res-sharing-scope>
23                   </resource-ref>
24            </entity>
. . .
```

Quadro 6.14 – Descritor de Distribuição.

Dentre as características mais relevantes do Quadro 6.14, destacam-se:

- No início do arquivo existe o cabeçalho utilizado para a versão EJB 2.1 (Linhas 1 a 7).
- Os *beans* estão declarados dentro da *tag* `<enterprise-beans>`.
- A *tag* `<entity>` define o *bean* de entidade `Venda` e, no escopo desta *tag*, existe a especificação de informações acerca do *bean*, como por exemplo, o nome do componente (Linha 12) a ser utilizado pelo serviço de nomeação JNDI, o nome qualificado das interfaces *home* e *remote* (Linhas 13 e 14), a classe principal do *bean* (Linha 15), quem controla sua persistência (Linha 16) e o nome qualificado da classe de chave primária (Linha 17).
- É importante observar a *tag* `<resource-ref>` que contém a declaração de uma referência a um recurso externo que, neste caso, é o *data-source*. Com isto, os *beans* conseguem localizar o *data-source* via JNDI para o acesso ao banco de dados.

6.4.6 Arquivo de Distribuição

Após a criação de todos os arquivos dos componentes, a estrutura do diretório de projeto é ilustrada na Figura 6.5.

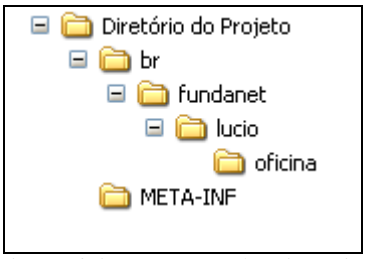

Figura 6.5 – Estrutura de Diretório.

Na pasta `META-INF` está o arquivo *ejb-jar.xml* e na pasta oficina existem todos os arquivos *.class* que compõem os três *beans* de entidade.

É necessário empacotar estes arquivos em um arquivo *.jar* para que eles sejam distribuídos na pasta *deploy* do JBOSS. No Quadro 6.15 ilustra-se o comando utilizado para a criação do arquivo *BeansDeEntidade.jar*.

```
C:\Diretório do Projeto> jar cfv BeansDeEntidade.jar br\fundanet\lucio\oficina\*.class META-INF\ejb-jar.xml
```

Quadro 6.15 – Criação do Arquivo de Distribuição.

6.5 Aplicação com o Padrão *Session Façade*

Para a criação da aplicação com a utilização do padrão *Session Façade*, foram desenvolvidos quatro componentes EJB: `Oficina`, `Funcionário`, `Cliente` e `Venda`. Os três últimos são componentes *Entity Beans* do tipo BMP que representam os dados do banco de dados. Possuem interfaces *LocalHome* e *Local* e só podem ser acessados localmente pelo componente `Oficina`, que é um *Session Bean* do tipo *Stateless*.

A diferença em relação à aplicação sem a utilização do padrão *Session Façade* é justamente a introdução do *bean* `Oficina` que atua entre o cliente e os três *beans* de entidade (Figura 6.6).

6.5.1 Arquitetura

Como já mencionado, a aplicação desenvolvida com o uso do padrão *Session Façade* é composta por quatro componentes e possui uma arquitetura de três camadas. Na Figura 6.6 ilustra-se esta arquitetura.

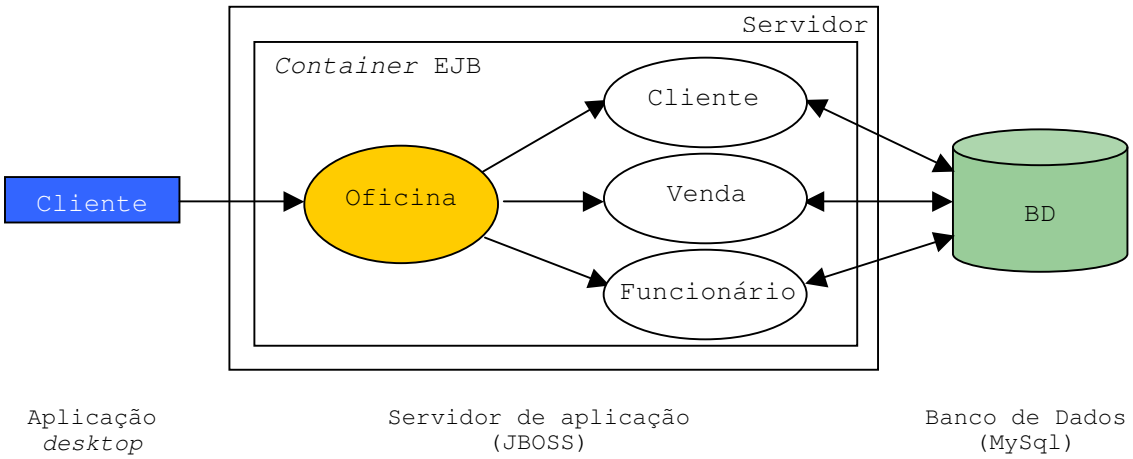

Figura 6.6 – Arquitetura da aplicação com o padrão *Session Façade*.

A arquitetura com o padrão *Session Façade* é semelhante em relação à arquitetura sem a utilização do padrão (Figura 6.2). O que diferencia é a inclusão do componente `Oficina`. O *bean* de sessão `Oficina` realiza a lógica comercial da aplicação enquanto que os *beans* de entidade representam os dados.

Nesta implementação, existe menor acoplamento do cliente com a aplicação, pois apenas o *bean* Oficina é exposto ao cliente. Nota-se que estas camadas proporcionam maior flexibilidade, pois elas podem ser alteradas mais independentemente, facilitando no momento em que houver a necessidade de manutenção. Isto ocorre porque o cliente possui a interface gráfica, o *bean* `Oficina` se concentra na lógica comercial e os *beans* de entidade representam os dados a serem manipulados.

6.5.2 Interface Gráfica do Cliente

A mesma interface gráfica foi utilizada nas duas implementações da aplicação. A diferença está no código do aplicativo cliente para interagir com os componentes.

No Quadro 6.16 são ilustrados trechos de código existentes no aplicativo do cliente. As três primeiras linhas do método `setContexto()` são idênticas à implementação sem o padrão (Quadro 6.1). Neste método localiza-se apenas um componente via JNDI pelo método `lookup()` (Rótulo 1) e, posteriormente, adquire-se a referência da interface *Home* do componente `Oficina` (Rótulo 2). Cria-se a referência da interface *Remote* (Rótulo 3) e, a partir dela, os métodos podem ser invocados (Rótulos 4, 5 e 6).

```
. . .
private void setContexto()
{
 Properties properties = new Properties();
 properties.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.NamingContextFactory");
 properties.put(Context.PROVIDER_URL, "10.17.227.110:1099");

    try {
        jndiContext = new InitialContext(properties);
 1 {    Object ref = jndiContext.lookup("Oficina");

 2 {    home = (OficinaHome)PortableRemoteObject.narrow (ref, OficinaHome.class);
        System.out.println("Pegou Referencia Home");
 3 {    oficina = home.create();

    }
    catch(Exception e) {System.out.println(e.toString());  }
}
. . .
public Vector[] selecionaTodosFuncionarios(){

    try{
 4 {    vetor1 = oficina.buscarFuncionarios();
    }catch(Exception ie){ . . . }
     return line;
}

public Vector[] selecionaTodosClientes(){

    try{
 5 {    vetor2 = oficina.buscarClientes();
    }catch(Exception ie){ . . . }
     return line;
}

public Vector[] selecionaTodasVendas(){

    try{
 6 {    vetor3 = oficina.buscarVendas();
    }catch(Exception ie){ . . . }
     return line;
}
. . .
```

Quadro 6.16 – Código Cliente com o Padrão.

Nota-se que o aplicativo cliente necessita interagir com as interfaces de apenas um componente, facilitando a interação do cliente com a aplicação e reduzindo a quantidade de código do aplicativo cliente em relação à implementação sem o padrão (Quadro 6.1).

6.5.3 *Bean* de Sessão

O *bean* de sessão `Oficina` é do tipo *stateless*, pois não há a necessidade de se manter um estado de comunicação contínua entre o cliente e o *bean*, pois este *bean* recebe a requisição do cliente, executa o método e fornece uma resposta. Foi criado com o objetivo de fornecer uma interface uniforme para a comunicação entre o cliente e os *beans* de entidade.

A comunicação entre o cliente e o *bean* de sessão `Oficina` ocorre via RMI-IIOP e a comunicação deste *bean* com os *beans* de entidade ocorre localmente.

Na aplicação sem a utilização do padrão *Session Façade* o *bean* de sessão `Oficina` não existe, pois o cliente acessa os *beans* de entidade diretamente. Este *bean* de sessão é constituído da classe principal `OficinaBean` e de suas interfaces `OficinaHome` e `Oficina`, que representam as interfaces *home* e *remote* respectivamente, descritas a seguir.

6.5.3.1 Interface OficinaHome

A interface `OficinaHome` contém a assinatura do método `create()`. Pela especificação EJB, deve existir apenas um método `create()` sem argumentos e que retorna a interface remota do *bean* de sessão *stateless*. Este método serve para criar uma instância do *bean* e corresponde ao método `ejbCreate()` definido na classe `OficinaBean`. A interface `OficinaHome` é mostrada no Quadro 6.17.

```
package br.fundanet.lucio.oficina;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface OficinaHome extends EJBHome {

	public Oficina create() throws RemoteException, CreateException;
}
```

Quadro 6.17 – Interface OficinaHome.

6.5.3.2 Interface Oficina

A interface `Oficina` possui os métodos de negócio necessários para a realização das seguintes operações:

- Inserção (como por exemplo, o método `inserirVenda()`, Rótulo 1 do Quadro 6.18);
- Exclusão (como por exemplo, o método `deletarVenda()`, Rótulo 2 do Quadro 6.18);
- Alteração (como por exemplo, o método `alterarVenda()`, Rótulo 3 do Quadro 6.18) e;
- Seleção (como por exemplo, o método `buscarVendas()`, Rótulo 4 do Quadro 6.18).

No Quadro 6.18 apresenta-se um fragmento do código relacionado às operações da tabela venda. Ressalta-se que as mesmas operações foram consideradas para as demais tabelas. Todos os métodos da interface `Oficina` devem obrigatoriamente lançar a exceção *RemoteException* no caso de existir algum erro durante a comunicação remota com o cliente.

```
package br.fundanet.lucio.oficina;
. . .

public interface Oficina extends EJBObject {

          /*Métodos relacionados à Venda*/
          public void inserirVenda(int codVenda, int codCli,     int codFunc,
   1      String dtVenda, float valorVenda, float desconto)
                 throws RemoteException,CreateException,SQLException;

   2      public void deletarVenda(String cod) throws RemoteException;

   3      public void alterarVenda(String s[]) throws RemoteException;

   4      public Vector[] buscarVendas()throws RemoteException,FinderException;
. . .
}
```

Quadro 6.18 – Interface `Oficina`.

6.5.3.3 Classe OficinaBean

A classe `OficinaBean` implementa a interface `SessionBean` e é principal parte do componente `Oficina`, pois possui a implementação dos métodos especificados nas interfaces `OficinaHome` e `Oficina`. Além destes, a classe `OficinaBean` implementa os métodos especificados na interface `SessionBean` que são utilizados pelo *container* para a interação com os componentes e são ilustrados no Quadro 6.19.

No método `setSessionContext()` as referências das interfaces *home* de cada *bean* de entidade são localizadas via JNDI (Rótulo 1). O *bean* `Oficina` interage com os *beans* de entidade localmente.

No método `ejbRemove()` (Rótulo 2) as referências são configuradas como nulas, pois quando este método é invocado, a instância do *bean* `Oficina` é removida.

```
Package br.fundanet.lucio.oficina;
. . .
public class OficinaBean implements SessionBean {
. . .
/*métodos utilizados pelo container*/

    public void ejbCreate()
    { System.out.println("ejbCreate Oficina");      }

    public void setSessionContext(SessionContext contexto)
       throws EJBException,RemoteException
    {
```

```
      System.out.println("setSessionContext Oficina");
      this.contexto = contexto;

      try {

          InitialContext jndiContext = new InitialContext();
  1       clienteHome = (ClienteLocalHome)jndiContext.lookup("local/Cliente");
          funcionarioHome = (FuncionarioLocalHome)jndiContext.lookup("local/Funcionario");
          vendaHome = (VendaLocalHome)jndiContext.lookup("local/Venda");

      }catch(NamingException ne)
       { System.out.println(ne.getMessage());
         ne.printStackTrace();}
    }

    public void ejbRemove() throws EJBException, RemoteException {
      clienteHome = null;
      cliente = null;
2     funcionarioHome = null;
      funcionario = null;
      vendaHome = null;
      venda = null;
    }
. . .
```

Quadro 6.19 – Métodos de `OficinaBean` utilizados pelo *container*.

A classe `OficinaBean` também possui métodos relacionados à lógica da aplicação. No Quadro 6.20 encontra-se a implementação do método de negócio `inserirVenda()`.

Nota-se que o *bean* `Oficina` possui referências às interfaces dos *beans* de entidade para a comunicação (Rótulo 1). No método `inserirVenda()` verifica-se que o *bean* `Oficina` delega o trabalho de inserção de dados ao *bean* de entidade `Venda` (Rótulo 2).

```
. . .
public class OficinaBean implements SessionBean {

      private SessionContext contexto;
      private ClienteLocalHome clienteHome;
      private ClienteLocal cliente;
 1    private FuncionarioLocalHome funcionarioHome;
      private FuncionarioLocal funcionario;
      private VendaLocalHome vendaHome;
      private VendaLocal venda;

      /*Métodos para a manipulação da Venda*/
      public void inserirVenda(int codVenda, int codCli,
             int codFunc, String dtVenda, float valorVenda,float desconto)
             throws CreateException,SQLException
             {
                try{
 2                   vendaHome.create(codVenda, codCli,
                            codFunc, dtVenda, valorVenda,desconto);

                    }catch(SQLException e)
                    { throw e; }
                    catch(CreateException e)
                    { throw e; }
      }
. . .
```

Quadro 6.20 – Método `inserirVenda()` de `OficinaBean`

6.5.4 *Beans* de Entidade

Na implementação com o padrão *Session Façade*, os *beans* de entidade representam os dados da aplicação e são acessados pelo *bean* de sessão `Oficina`. Estes *beans* são os mesmos utilizados na aplicação sem o padrão. Entretanto, ao invés de possuírem as interfaces *Home* e *Remote*, eles possuem as interfaces *HomeLocal* e *Local*, pois o acesso do *bean* `Oficina` aos *beans* de entidade ocorre localmente.

Cada *bean* de entidade é composto por quatro arquivos que são: a classe principal *bean*, as interfaces *HomeLocal* e *Local* e uma classe que representa sua chave primária. Destes arquivos, serão abordados apenas os que possuem diferenças em relação aos arquivos da aplicação sem o uso do padrão. Portanto, serão descritas apenas as interfaces *HomeLocal* e *Local* do componente `Venda` nesta implementação.

6.5.4.1 Interface VendaLocalHome

A interface `VendaLocalHome`, parcialmente ilustrada no Quadro 6.21, estende a interface `EJBLocalHome` e contém a assinatura dos métodos para a criação e localização dos *beans* de entidade `Venda`. Esta interface é semelhante à interface `VendaHome` da aplicação sem o uso de padrão (Quadro 6.3), entretanto, existem algumas diferenças. As principais diferenças são:

- A interface `VendaLocalHome` estende a interface `EJBLocalHome` ao invés de `EJBHome`.
- Os métodos não necessitam lançar a exceção *RemoteException*, pois eles são acessados localmente pelo *bean* `Oficina`.

```
package br.fundanet.lucio.oficina;
. . .
public interface VendaLocalHome extends EJBLocalHome {

        public VendaLocal create(int codVenda, int codCli, int codFunc,
                        String dtVenda, float valorVenda, float desconto)
                    throws CreateException,SQLException;

        public VendaLocal findByPrimaryKey(VendaPK chavePK)
              throws FinderException,SQLException,NumberFormatException;

        public Collection findAll() throws FinderException;
. . .
}
```

Quadro 6.21 – Interface `VendaLocalHome`.

6.5.4.2 Interface VendaLocal

A interface `VendaLocal` possui a assinatura dos métodos (Quadro 6.22) de negócio da aplicação que utiliza o padrão *Session Façade* e também é acessada localmente pelo *bean* `Oficina`. No Quadro 6.22 ilustra-se a interface `VendaLocal`.

Esta interface estende a interface `EJBLocalObject` e possui métodos para obter (Rótulo1) e alterar (Rótulo 2) os valores de cada atributo de uma venda. Ela é semelhante à interface `Venda` da aplicação sem o uso de padrão (Quadro 6.4). A principal diferença é que os métodos não necessitam lançar a exceção *RemoteException*.

```
package br.fundanet.lucio.oficina;
. . .

public interface VendaLocal extends EJBLocalObject {

    1 {
        public int getCodVenda();
        public int getCodCli();
        public int getCodFunc();
        public String getDataVenda();
        public float getValorVenda();
        public float getDesconto();
    }

    2 {
        public void setCodVenda(String s);
        public void setCodCli(String s);
        public void setCodFunc(String s);
        public void setDataVenda(String s);
        public void setValorVenda(String s);
        public void setDesconto(String s);
    }
}
```

Quadro 6.22 – Interface `VendaLocal`.

6.5.5 Descritor de Distribuição

Na implementação da aplicação com o uso do padrão, o descritor de distribuição também é representado pelo arquivo *ejb-jar.xml*, porém com algumas diferenças em relação ao descritor da implementação sem o padrão (Quadro 6.14). No Quadro 6.23 apresenta-se um fragmento do descritor de distribuição para a visualização das principais diferenças entre os descritores.

```
. . .
1       <enterprise-beans>
2          <session>
3                  <ejb-name>Oficina</ejb-name>
4                  <home>br.fundanet.lucio.oficina.OficinaHome</home>
5                  <remote>br.fundanet.lucio.oficina.Oficina</remote>
6                  <ejb-class>br.fundanet.lucio.oficina.OficinaBean</ejb-class>
7                  <session-type>Stateless</session-type>
8                  <transaction-type>Container</transaction-type>
9                  <ejb-local-ref>
10                     <ejb-ref-name>Venda</ejb-ref-name>
11                     <ejb-ref-type>Entity</ejb-ref-type>
12                     <local-home>br.fundanet.lucio.oficina.VendaLocalHome</local-home>
13                     <local>br.fundanet.lucio.oficina.VendaLocal</local>
14                     <ejb-link>VendaBean</ejb-link>
15                 </ejb-local-ref>
16                    . . .
17         </session>
18         <entity>
19             <ejb-name>Venda</ejb-name>
20             <local-home>br.fundanet.lucio.oficina.VendaLocalHome</local-home>
21             <local>br.fundanet.lucio.oficina.VendaLocal</local>
22             <ejb-class>br.fundanet.lucio.oficina.VendaBean</ejb-class>
23             <persistence-type>Bean</persistence-type>
24         </entity>
25      </enterprise-beans>
. . .
```

Quadro 6.23 – Descritor de Distribuição

Dentre as características mais relevantes do Quadro 6.23, destacam-se:

- A *tag* `<session>` define o *bean* de sessão `Oficina` e, no escopo desta *tag*, existe a especificação de informações acerca do *bean*, como por exemplo, o nome do componente a ser utilizado pelo serviço de nomeação JNDI (Linha 3), o nome qualificado das interfaces *home* e *remote* (Linhas

4 e 5), a classe principal do *bean* `Oficina` (Linha 6), o tipo do *bean* (Linha 7) e quem controla a transação (Linha 8).

- Entre as *tags* `<ejb-local-ref>` e `</ejb-local-ref>` existem as declarações das referências locais dos *beans* de entidade para que o *bean* de sessão possa utilizá-las.
- Em relação aos *beans* de entidade, é importante observar que as interfaces declaradas são locais (Linhas 20 e 21).

6.5.6 Arquivo de Distribuição

Após a criação de todos os arquivos que compõem os componentes, a estrutura do diretório de projeto mantém-se igual à Figura 6.5 referente à implementação sem o uso do padrão. Entretanto, todos os arquivos foram empacotados em um arquivo chamado *Façade.jar*. O Quadro 6.24 ilustra-se o comando utilizado para a criação do arquivo *Façade.jar*.

```
C:\Diretório do Projeto> jar cfv Façade.jar br\fundanet\lucio\oficina\*.class META-INF\ejb-jar.xml
```

Quadro 6.24 – Criação do Arquivo de Distribuição.

6.6 Discussão

Neste capítulo foi possível observar as diferenças entre a criação da aplicação com e sem o uso do padrão de projeto *Session Façade*.

Verifica-se que ao se utilizar o componente `Oficina` como interface de acesso à aplicação, o aplicativo cliente interage somente com uma interface uniforme o que facilita e

diminui o código do aplicativo cliente. Este processo torna mais simples a utilização do sistema pelo cliente.

Por outro lado, sem a utilização do padrão *Session Façade*, o cliente interage com as interfaces dos três componentes de entidade (`Cliente`, `Venda` e `Funcionário`). Isto resulta em uma maior dependência do aplicativo cliente com a aplicação, ou seja, o cliente necessita conhecer com maior profundidade suas características para a interação. Este cenário dificulta a interação entre o cliente e a aplicação e existe um número maior de linhas de código no aplicativo do cliente.

Com o uso do padrão reduz-se a quantidade de chamadas remotas, pois o *bean* `Oficina` se comunica com os *beans* de entidade por meio de suas interfaces locais. Esta redução de chamadas remotas pode melhorar o desempenho da aplicação. Verificou-se também que o sistema torna-se mais flexível, ou seja, mais uma camada é adicionada ao sistema reduzindo o acoplamento e facilitando a manutenção.

Uma desvantagem que se pôde observar com a utilização do padrão é o aumento da quantidade de chamadas de métodos, ou seja, sem o padrão o cliente acessa o *bean* de entidade com apenas uma chamada de método, já com o padrão existem duas chamadas de métodos para a requisição do cliente chegar ao *bean* de entidade. Esta desvantagem, entretanto, não se configurou como uma perda de desempenho da aplicação com o padrão.

7 CONCLUSÃO

O presente trabalho teve como objetivo o estudo acerca dos componentes de software explicitando suas principais características e do processo de desenvolvimento de software baseado em compontentes. O segundo objetivo foi construir uma aplicação e realizar uma comparação entre implementações de componentes de softwares com e sem o uso de padrões de projeto.

Com a construção da aplicação utilizando-se da tecnologia EJB (*Enterprise JavaBeans*) conclui-se que esta tecnologia deve ser utilizada para a construção de aplicações que necessitem ser distribuídas para que seu potencial seja utilizado de forma correta. É de grande importância a existência do *container* que controla o ciclo de vida dos componentes e oferece serviços importantes relacionados, como por exemplo, à transação. Com isso, o desenvolvedor pode se concentrar apenas na função que seu componente deve implementar.

Pôde-se concluir também que o tempo e o esforço gastos no aprendizado dos componentes podem ser recompensados com a posterior reutilização dos mesmos. Ou seja, o desenvolvedor gasta o tempo inicial para o desenvolvimento uma única vez e, após isso, quando se deseja utilizar novamente a funcionalidade que o componente implementa, ele não precisa ser recriado, pois já está pronto pra ser reutilizado.

Além de se ter uma aplicação baseada em componentes, é importante dispor estes componentes de uma maneira correta para que se obtenha uma aplicação melhor estruturada. Assim, nota-se a fundamental importância do uso de padrões de projeto para a criação de projetos mais elaborados.

Os *beans* de entidade podem ser reutilizados em outra aplicação, mas para que isso ocorra, as tabelas da nova aplicação devem possuir os mesmos campos, pois os *beans* representarão diretamente estes dados.

A documentação de componentes desenvolvidos com o intuito de serem reutilizados em futuras aplicações é de extrema importância para o DBC. Dentre os métodos disponíveis na literatura de Engenharia de Software Baseada em Componentes, o UML *Components* se apresenta muito interessante para engenheiros que trabalham com a linguagem UML, pois utiliza estereótipos para estender a linguagem. Assim, o método UML *Components* utiliza a linguagem UML para modelar todas as fases de desenvolvimento de sistemas baseados em componentes.

Este trabalho possibilita a condução de diferentes trabalhos a serem desenvolvidos futuramente:

- Desenvolvimento de outras partes da aplicação utilizando diferentes tipos de *beans*, como por exemplo, os *beans* de entidade CMP para representarem os dados da aplicação. A persistência destes *beans* é controlada pelo *container*, ou seja, o desenvolvedor não precisa inserir comandos SQL no código do *bean* para persistir os dados. Com isso, pode-se realizar uma comparação das principais diferenças encontradas.
- Com relação ao desempenho das aplicações, pode-se realizar uma análise do desempenho das aplicações desenvolvidas com e sem o padrão de projeto.
- Utilização de diferentes padrões de projeto que possam ser agregados para prover outros benefícios à aplicação.
- Desenvolvimento de outras aplicações pertencentes ao mesmo domínio para analisar a reusabilidade dos componentes construídos. Outros pontos importantes seriam a realização da Engenharia de Domínio e de testes específicos para desenvolver componentes com alto índice de reúso e para avaliar os componentes existentes, respectivamente.

REFERÊNCIAS

BACHMANN, F. et al. **Volume II: Technical Concepts of Component-Based Software Engineering**. Pittsburgh: Software Engineering Institute, May 2000. (Technical Report CMU/SEI-2000-TR-008).

BARROCA, L.; GIMENES, I. M. S.; HUZITA, E.H.M. Desenvolvimento Baseado em Componentes. In:_____. GIMENES, I. M. S.; HUZITA E. H. M. **Desenvolvimento Baseado em Componentes.** Rio de Janeiro: Ciência Moderna, 2005, cap. 2.

BASS, L. et al. **Volume I: Market Assessment of Component-Based Software Engineering**. Pittsburgh: Software Engineering Institute, May 2001. (Technical Note CMU/SEI-2001-TN-007).

BOND, M. et al. **Aprenda J2EE em 21 dias: Com EJB, JSP, Servlets, JNDI, JDBC e XML**. São Paulo: Pearson, 2003.

CAGNIN, M. I.; **PARFAIT: uma contribuição para a reengenharia de software baseada em linguagens de padrões e frameworks**. Tese (Doutorado em Ciência da Computação) – Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, São Carlos, 2005.

DEITEL, H. M.; DEITEL, P. J. **Java: Como Programar**. 4ª ed. Porto Alegre: Bookman, 2003.

DEMICHIEL, L. G. **Enterprise JavaBeans Specification, Version 2.1**. Disponível em: <http://java.sun.com/products/ejb/docs.html>. Acesso em: 20 abr. 2005.

D'SOUZA, D. F.; WILLS, A. W. **Objects, components and frameworks with UML – the catalysis approach**. Addison-Wesley, Reading, MA, 1999.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Reading, MA, 1995.

GIMENES, I. M. S.; HUZITA E. H. M. **Desenvolvimento Baseado em Componentes.** Rio de Janeiro: Ciência Moderna, 2005.

KURNIAWAN, B. **Java para a Web com Servlets, JSP e EJB**. Rio de Janeiro: Ciência Moderna, 2002.

METSKER, S. J. **Design Patterns Java Workbook.** Addison-Wesley, Reading, MA, 2002.

MONSON-HAEFEL R. **Enterprise JavaBeans**. 4ª ed. Capítulo 11. Sebastopol, CA: O'Reilly, 2004. Disponível em: <http://www.oreilly.com/catalog/entjbeans4/ >. Acesso em: 13 jul. 2005.

MORISSEAU-LEROY, N.; SOLOMON, M. K.; BASU J. **Oracle 8i: programação de componentes JAVA com EJB, CORBA e JSP**. Rio de Janeiro: Campus, 2001.

NASCIMENTO, L. B. **Componentização de Software em JAVA™ 2 Micro Edition - Um *framework* para Desenvolvimento de Interface Gráfica para Dispositivos Móveis**. Dissertação (Graduação em Ciência da Computação) – Centro de Informática. Universidade Federal de Pernambuco, Recife, 2005.

PAGANO, V. A. **Uma Abordagem Arquitetural com Tratamento de Exceções para Sistemas de Software Baseado em Componentes**. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação. Universidade Estadual de Campinas, Campinas, 2004.

PHAROAH, A.; ARNI F. **Creating Commercial Components (EJB 2.0)**. Disponível em: <http://www.componentsource.com>. Acesso em: 12 jun. 2005.

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

ROMAN, E.; SRIGANESH, R. P.; BROSE, G. **Mastering Enterprise Javabeans**. 3ª ed. Indianápolis: Wiley, 2005. Disponível em: <http://www.theserverside.com/books/wiley/masteringEJB/>. Acesso em: 15 maio 2005.

ROSSI, A. C. **Representação de Componentes de Software na FARCSoft: Ferramenta de Apoio à Reutilização de Componentes de Software**. Dissertação (Mestrado em Engenharia) - Departamento de Engenharia de Computação e Sistemas Digitais. Escola Politécnica da Universidade de São Paulo, São Paulo, 2004.

SOMMERVILLE, I. **Engenharia de Software**. 6ª ed. São Paulo: Addison Wesley, 2003.

SOUSA, M. C. F. **Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture**. Dissertação (Mestrado em Computação na área de Engenharia da Computação) – Instituto de Computação. Universidade Estadual de Campinas, Campinas, 2004.

TRIGAUX, J. C.; HEYMANS, P. **Software Product Lines: State of the art**. Disponível em: <www.info.fundp.ac.be/~jtr/ PLENTY/Files/productline0309.pdf>. Acesso em: 23 set. 2005.

WERNER, C. M. L.; BRAGA, R. M. M. A Engenharia de Domínio e o Desenvolvimento Baseado em Componentes. In:_____. GIMENES, I. M. S.; HUZITA E. H. M. **Desenvolvimento Baseado em Componentes.** Rio de Janeiro: Ciência Moderna, 2005, cap. 3.

YOURDON, E. Diagramas Entidades-Relacionamento. In:_____. **Análise Estruturada Moderna**. Rio de Janeiro: Campus, 1992. cap 12.