UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

APLICAÇÃO DA ARQUITETURA MULTICAMADAS UTILIZANDO JAVA

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

RAQUEL SCHLICKMANN

BLUMENAU, NOVEMBRO/1999

1999/2-31

APLICAÇÃO DA ARQUITETURA MULTICAMADAS UTILIZANDO JAVA

RAQUEL SCHLICKMANN

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof. Everaldo A. Grahl

_____

Prof. Maurício Capobianco Lopes

Ao meu namorado Sérgio

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ACL	ACCESS CONTROL LISTS
API	APPLICATION PROGRAMMING INTERFACE
CORBA	COMMON OBJECT REQUEST BROKER ARCHITECTURE
DCE	DISTRIBUTED COMPUTING ENVIROMENT
DCOM	DISTRIBUTED COMPONENT OBJECT MODEL
EJB	ENTERPRISE JAVABEANS
GUI	GRAPHICAL USER INTERFACE
HTML	HIPER TEXT TRANSFER PROTOCOL
HTTP	HIPER TEXT MARKUP LANGUAGE
IIOP	INTERNET INTER OBJECT REQUEST BROKER PROTOCOL
JAR	ARQUIVO JAVA
JDBC	JAVA DATABASE CONNECTIVITY
JNDI	JAVA NAMING AND DIRECTORY INTERFACE
JRMP	JAVA REMOTE METHOD PROTOCOL
JTA	JAVA TRANSACTION INTERFACE
JTS	JAVA TRANSACTION SERVICE
LAN	LOCAL AREA NETWORK
OR	OBJETO-RELACIONAL
PC	POWER COMPUTER
RMI	REMOTE METHOD INVOCATION
SGBD	SISTEMA GERENCIADOR DE BANCO DE DADOS
UML	UNIFIED MODELING LANGUAGE
WAN	WIDE AREA NETWORK

RESUMO

Este trabalho visa estudar e aplicar a arquitetura de software em multicamadas através da especificação e implementação de uma aplicação do sistema de identificação pessoal da FURB. Foi utilizado na implementação a linguagem Java, explorando os recursos de *Enterprise JavaBeans* (EJB), utilizando como servidor de aplicação o Voyager 3.11 e acessando o banco de dados 8.0.5.

ABSTRACT

This work aims to study and to apply the multi-tier software architecture through the specification and implementation of an application of the FURB's personal identification system. The Java language was used in the implementation, exploiting Enterprise JavaBeans (EJB), using Voyager 3.11 as application server and Oracle 8.0.5 as database management system.

1 INTRODUÇÃO

Hoje em dia, os gerentes de sistemas de informação defrontam-se com um dilema: criar uma vantagem competitiva para a organização, desenvolvendo, distribuindo e gerenciando aplicações distribuídas, que sejam escaláveis através de uma *Local Area Network* (LAN), *Wide Area Network* (WAN) e Internet, enquanto se preservam investimentos em sistemas, aplicações, informações e recursos humanos.

Dentro deste contexto, uma solução seria a implementação de uma arquitetura de computação distribuída em multicamadas (*multi-tier*), pois esta tem o potencial de prover melhores resultados para a organização a um custo mais baixo do que a combinação de uma PC LAN, cliente/servidor ou aplicações baseadas em *mainframes* [HAM99].

Segundo [HAM99], a arquitetura multicamadas é composta basicamente por três camadas, que se referem às três partes lógicas que compõem uma aplicação, e não ao número de máquinas usadas pela aplicação. O modelo multicamadas divide a aplicação, normalmente, nas seguintes camadas, porém, pode existir qualquer número de qualquer das camadas em uma aplicação:

a) lógica de apresentação: apresenta a informação a uma fonte externa e obtém entradas daquela fonte. Na maioria dos casos, a fonte externa é um usuário final trabalhando em um terminal ou estação de trabalho, embora a fonte externa possa ser um sistema robótico, um telefone ou algum outro dispositivo de entrada;
b) lógica do negócio: contém a lógica da aplicação que governa as funções do negócio e os processos executados pela aplicação;
c) lógica de acesso a dados: contém a lógica que fornece à interface um sistema de armazenamento de dados ou com algum outro tipo de fonte de dados externos, como uma aplicação externa.

As três camadas são, na verdade, componentes da aplicação que se comunicam utilizando uma interface abstrata, que funciona como um contato, em termos do que está sendo tornado público. Essa mesma camada abstrata esconde os detalhes da implementação das funções desempenhadas por um componente. Esse tipo de infra-estrutura possibilita serviços de localização, segurança e comunicação entre os componentes da aplicação [HAM99].

A intenção do modelo em três camadas é suportar uma forma mais poderosa de modelagem, projeto e programação orientada a objeto do que aquelas que, historicamente, tem caracterizado a arquitetura de modelagem de aplicações. Esta abordagem é conhecida como um processo de desenvolvimento dirigido a arquitetura e iterativo. A abordagem arquitetural em três camadas significa a criação de uma arquitetura de software em níveis, na qual há a completa separação dos serviços de dados e dos serviços de negócios (modelo do domínio) dos serviços dos usuários (interface do usuário), onde cada camada possui regras de negócio dentro de classes e operações em objetos de negócio, comunicando-se através de mensagens [RAT98].

Uma das linguagens orientadas a objetos que suporta esta arquitetura é a linguagem Java. O aparecimento de Java, segundo [FUR98], revolucionou o desenvolvimento de software para Internet, Intranet e quase todas as redes distribuídas. Além disso, Java é uma linguagem totalmente orientada a objeto, poderosa, dinâmica, independente de plataforma tecnológica, segura e relativamente fácil de usar.

1.1 OBJETIVOS

Este trabalho tem como objetivo estudar e aplicar a arquitetura multicamadas através da especificação e implementação em *Java* de uma aplicação do sistema de identificação pessoal da FURB. O desenvolvimento desta aplicação seguirá os passos descritos na metodologia de desenvolvimento do Núcleo de Informática da FURB, adaptada pela utilização da técnica UML - *Unified Modeling Language*.

1.2 ORGANIZAÇÃO DO TEXTO

No primeiro capítulo é apresentado o assunto do trabalho como um todo, assim como os seus objetivos.

No segundo capítulo é apresentado um breve histórico sobre as arquiteturas de software, abordando separadamente cada uma delas, desde o *mainframe* até a arquitetura multicamadas, dando mais enfoque a esta última.

No terceiro capítulo são apresentadas as ferramentas que auxiliam no desenvolvimento da arquitetura multicamadas como *Enterprise JavaBeans* (EJB) que é uma arquitetura de

componentes para desenvolvimento de aplicações distribuídas e orientada a objetos e o *Voyager* que é um servidor de aplicações que implementa *Enterprise JavaBeans.*

No quarto capítulo é apresentado o cenário do protótipo, sua especificação e sua tela principal.

No quinto capítulo são apresentadas as conclusões obtidas no decorrer do desenvolvimento do mesmo e sugestões para futuros trabalhos.

2 HISTÓRICO DA ARQUITETURA DE SOFTWARE

Como a maioria dos campos de empenho humano, a indústria de Tecnologia de Informação evoluiu periodicamente por “troca de paradigmas” - transformações fundamentais tais como, desenvolvimento, usuários, acesso e integração com sistemas de computação e aplicações. Estas ondas de tecnologia tendem a ocupar de 10 a 15 anos ou muito mais tempo até progredir de experimentação inicial em laboratórios de pesquisa de computação para uso em indústrias. Dado que esta indústria só é de aproximadamente 45 anos, isto significa que muitas áreas de tecnologia estão agora na terceira onda de evolução. De maneira interessante, também é dito que a civilização humana está entrando em sua terceira fase fundamental: da sociedade agrária de 10.000 anos atrás, para a sociedade industrial de 200 anos atrás, para a sociedade de informação do século 21 [LEF98].

Em alguns casos, um paradigma novo aumenta, mas não substitui um paradigma existente (*mainframes* ainda existem, apesar da era PC). Em outros casos (cartões perfurados), o paradigma novo substituiu o paradigma velho efetivamente. O enfoque desta composição é a troca de paradigma à terceira onda de desenvolvimento de software aplicativo que logo estará alcançando o ponto de inflexão depois dos 10-15 anos típicos de investimento de tecnologia [LEF98].

Um breve histórico sobre essas três “ondas”, segundo [LEF98]:

a) primeira onda - aplicações monolíticas (anos 50 a 70): do começo da indústria de computadores nos anos cinqüenta, até o fim dos anos setenta, todas aplicações de software eram monolíticas, com programas e dados firmemente entrelaçados. Cada desenvolvedor de aplicação escolhia como estruturar e armazenar dados e usava freqüentemente técnicas inteligentes para minimizar o caro armazenamento (conseqüentemente, surgiu o problema do “ano 2000” ou “*bug* do milênio”). Esta integração estreita entre programa e dados dificultou o modelo e reuso da informação incorporada;
b) segunda onda - aplicações cliente/servidor (anos 80 a 90): com a viabilidade comercial de sistemas de administração de banco de dados no início dos anos 80, empresas começaram a modelar a informação incorporada e criar repositórios de dados de empreendimento que eram acessíveis através de programas múltiplos. Esta separação de programa e dados causou uma troca fundamental em como

companhias dirigiram seus negócios. Pela primeira vez foram criados departamentos denominados “Administração de Sistemas de Informação” para modelar dados corporativos e prover múltiplas aplicações que manipularam informações comuns. Este modelo foi muito útil enquanto as interfaces dos usuários para dados eram interfaces baseadas em caracter simples, acessível por um número pequeno de processamneto de dados. Porém, a adoção difundida de interfaces gráficas nos anos noventa e a extensão de acesso de informação de tempo real para milhões de *desktops*, conduziu a um aumento dramático no tamanho e complexidade de aplicações cliente - clientes crescentemente “pesados” que executam em milhares de PC, todos tentando ter acesso a um repositório de dados central. E, da perspectiva de desenvolvimento, enquanto havia reuso de dados de empreendimento, havia pequeno reuso de lógica empresarial encapsulada;

c) terceira onda - aplicações distribuídas (metade dos anos 90): durante os anos 80 havia muito experimentação na comunidade de pesquisa de computação em computação distribuída - escrevendo aplicações que não tiveram acesso há poucos dados distantes em um servidor de SGBD, mas há vários pedaços da própria aplicação por sistemas múltiplos. Esta pesquisa conduziu ao desenvolvimento de padrões de computação distribuída como *Distributed Computing Enviroment* (DCE) da *Open Software Foundation and Open Group,* e a *Common Object Request Broker Architecture* (CORBA) de *Object Management Group.* Durante os anos noventa, companhias começaram o processo de construção e desdobramento de aplicações distribuídas, denominadas multicamadas que usam *frameworks,* com um cliente leve (somente a interface é executada no cliente, as regras de negócio são executadas no servidor de aplicação) *Graphical User Interfae* (GUI) que se comunica com um servidor de aplicação de camada mediana que executa a lógica empresarial centralizada, este por sua vez se comunica com um servidor SGBD tradicional. Esta arquitetura está se estabelecendo agora como o empreendimento da arquitetura de aplicação de software dominante para o fim dos anos 90 e início do século 21.

A tabela 1 mostra algumas das trocas de paradigmas que foram vivenciados em tecnologia da informação e o que está no horizonte, segundo [LEF98].

TABELA 1 TROCA DE PARADIGMAS.

Tecnologia	Primeira Onda	Segunda Onda	Terceira Onda	Quarta Onda?
Sistemas de Computadores	Sistemas *mainframes* (50s – 60s)	Mini Computadores (70s – 80s)	Computadores Pessoais (80s – 90s)	Computadores de "*Intimate/Wearable*"? (00s)
Componentes	*Vacuum Tubes* (50s)	Transições Discretas (60s – 70s)	Circuitos Integrados (80s – 90s)	Eletrônica de *Quantum* ? (10s)
Interface de Usuário	Cartões Perfurados (50s – 60s)	Tela Verde Interfaces Baseadas em Caracter (70s – 80s)	Interfaces de Usuário Vívidas (80s – 90s)	Interfaces "Natural" (Linguagem, Visão) ? (00s)
Comunicações Digitais	"*Sneaker Net*" (50s – 70s)	Redes Locais (70s – 80s)	Internet (90s)	Infra-estrutura de Informação Global? (00's)

2.1 ARQUITETURA DE UMA CAMADA

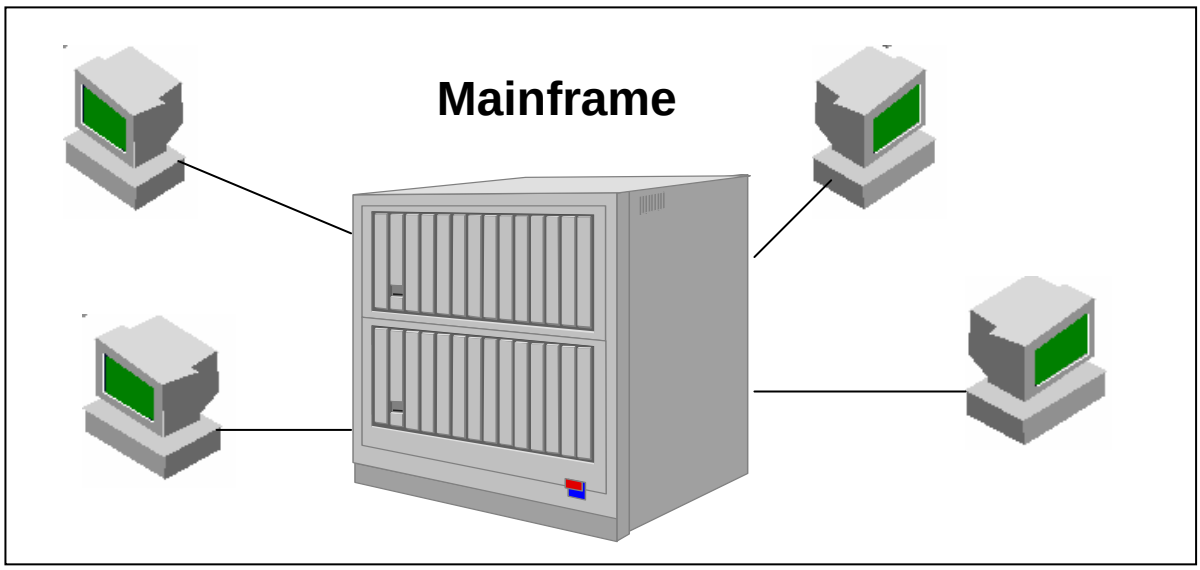
O núcleo do modelo de computação usado durante muitos anos era um *mainframe* com vários dispositivos de entrada e saída conectados a ele, arquitetura de uma camada, segundo [BOC95]. No início da informática, a entrada era fornecida por cartões perfurados. Depois que os dados eram alimentados no computador através desses cartões, eles eram freqüentemente copiados em fitas magnéticas. Essa saída em fita poderia ser usada como entrada em outros sistemas. Enquanto a saída dos primeiros programas de computador eram armazenados em fitas magnéticas ou discos, a saída a que os usuários finais tinham acesso consistia em toneladas de papel.

Segundo [BOC95], algumas vezes, os terminais eram conectados a *mainframes*. O terminal mais usado para *mainframes* IBM era o 3270. Muitos terminais semelhantes foram comercializados, criando o termo "tipo 3270", visto que sua funcionalidade era parecida,

senão idêntica. Os operadores podiam digitar dados diretamente nesses terminais em vez de usar os cartões perfurados, e os programadores podiam escrever instruções nestes terminais em vez de usar folhas de codificação que então seriam transformadas em cartões perfurados. Além disso, os usuários finais podiam acessar sistemas do *mainframe* através desses terminais. Eles podiam entrar com uma transação usando um terminal e solicitar ao *mainframe* que fornecesse informações relativas aos seus aplicativos. No entanto, esses terminais não realizavam nenhum processamento relacionado a essas solicitações. Todo processamento referente a uma transação ou consulta fornecida em um terminal do tipo 3270 era executado no *mainframe*. Um terminal do tipo 3270 não tinha processador interno e por isso não podia realizar nenhuma computação extensiva. Ele era freqüentemente chamado de "terminal burro". Durante alguns poucos anos após a introdução dos terminais 3270, a configuração típica dos computadores da IBM consistia em um *mainframe* e dúzias ou centenas de terminais do tipo 3270 conectados a ele.

Segundo [HAM99], a arquitetura de uma camada é baseada em um ambiente onde todos os componentes são combinados num simples programa integrado, rodando somente em uma máquina. Essa arquitetura, totalmente centralizada, corresponde ao tradicional ambiente *mainframe* (figura 1).

FIGURA 1 MAINFRAME



A alternativa de uma camada oferece quantidade significativa de vantagens. Sendo a aplicação centralizada em um único ambiente, é fácil gerenciar, controlar e garantir a segurança de acesso a essas aplicações, além do que, esses sistemas são seguros, confiáveis e suportam vários usuários. [HAM99]

Por outro lado, existe uma grande quantidade de desvantagens associadas a essa arquitetura. A primeira delas é escalabilidade. Se a máquina corrente ficar sobrecarregada, a única saída é fazer um *upgrade* para uma máquina de maior capacidade. Aplicações de uma camada são também extremamente dependentes do *hardware* e do ambiente operacional. Essencialmente as companhias ficam presas a um fornecedor específico da sua plataforma de *hardware*. Como resultado, não se pode tirar proveito de novas tecnologias, até que elas sejam disponibilizadas por aquele fornecedor específico. [HAM99]

2.2 ARQUITETURA DE DUAS CAMADAS

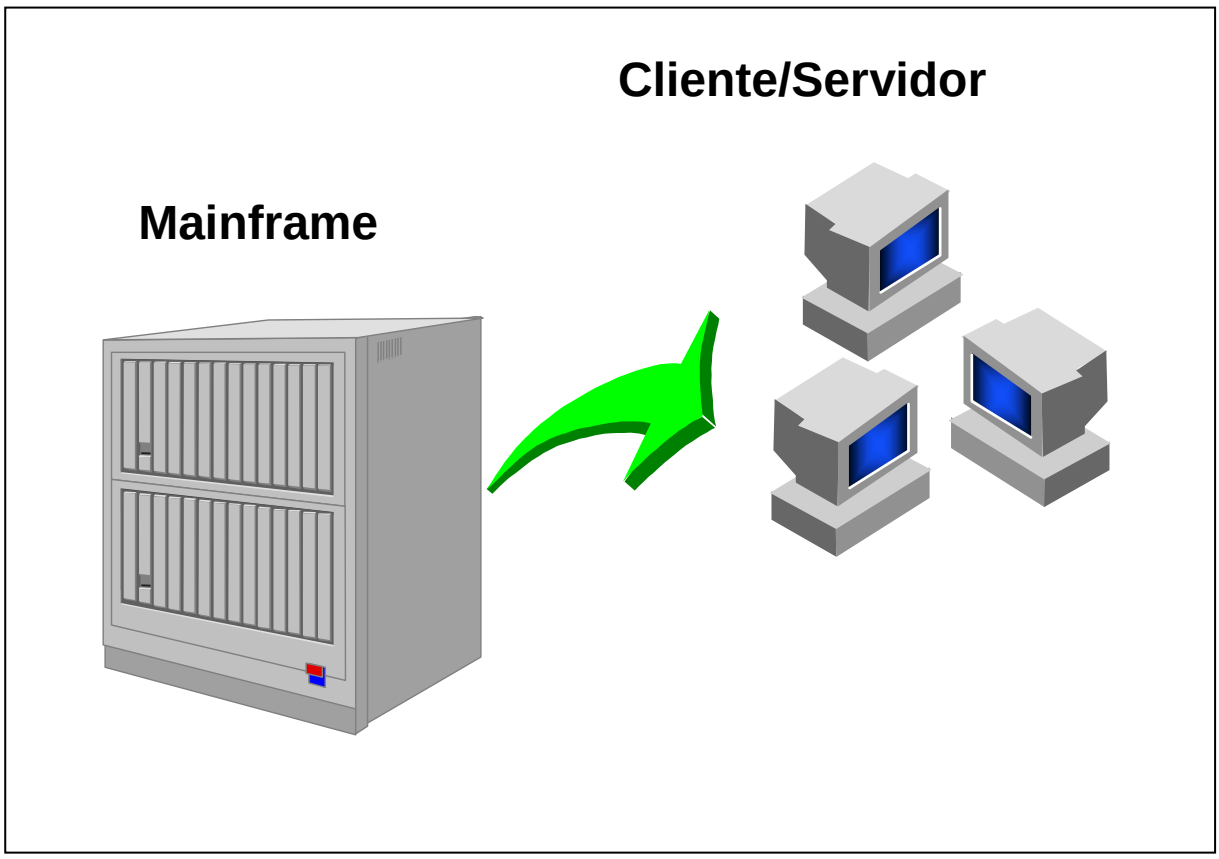
Segundo [BOC95], em uma arquitetura cliente/servidor, processos independentes residem em diferentes plataformas e interagem através de redes, para atender aos objetivos da computação.

Em um sistema cliente/servidor, um processo do cliente faz solicitações ao processo do servidor e este atende a essas solicitações. Os processos de cliente e servidor em geral ocorrem em plataformas diferentes, permitindo que compartilhem recursos enquanto aproveitam ao máximo as vantagens das plataformas e dispositivos diferentes (figura 2). Embora um *mainframe* possa ser utilizado como servidor nessa arquitetura, os sistemas cliente/servidor mais eficientes em termos de custo consistem apenas em micros conectados em rede [BOC95].

Com o advento dos computadores pessoais, redes locais, bancos de dados relacionais, aplicações e ferramentas do *desktop*, a indústria de computação direcionou-se para o mundo dos sistemas “abertos” e cliente/servidor. Tomadores de decisão podem gerar seus próprios relatórios e manipular dados com ferramentas *desktop* em suas próprias estações de trabalho. Esse tipo de arquitetura permite a manipulação de funções, anteriormente impossíveis de serem manipuladas [BOC95].

A arquitetura cliente/servidor em duas camadas divide o processamento entre uma estação *desktop* e uma máquina servidora. O ambiente cliente/servidor mais popular usa um PC (*Windows-based*) com uma ferramenta de desenvolvimento *Graphical User Interface* (GUI) e um servidor de banco de dados UNIX ou Windows NT.

FIGURA 2 PASSAGEM DE MAINFRAME PARA CLIENTE/SERVIDOR.



Segundo [HAM99], algumas vantagens surgem dessa arquitetura:

a) as ferramentas GUI possibilitam um desenvolvimento e distribuição de aplicações muito mais rápidos;

b) por se ter um processamento distribuído entre o cliente e o servidor, as máquinas servidoras não necessitam ser tão potentes, o que resulta em custos mais baixos do que os sistemas *mainframe*;

c) os sistemas de bancos de dados, por independerem de plataforma, permitem portabilidade mais fácil entre sistemas, efetivamente quebrando a dependência no fornecimento do *hardware*;
d) considerando a facilidade de uso das ferramentas GUI, o nível de qualificação dos desenvolvedores não precisa ser alto.

Em contrapartida, surgem algumas desvantagens: perda de segurança, confiança e controle é uma delas. Esse modelo é extremamente eficaz para aplicações de médio porte, acessando poucos bancos de dados e não suportando uma grande quantidade de usuários. Sem as facilidades de controle e segurança, disponíveis nos sistemas centralizados, cada aplicação cliente deve cuidar do seu próprio processo de segurança. Com isso, muitas companhias ainda relutam em mover suas aplicações de missão crítica para PCs [HAM99].

2.3 ARQUITETURA MULTICAMADAS

Felizmente, segundo [HAM99], existe uma forma de se obter o melhor das arquiteturas anteriores, sem as suas desvantagens. Além de suportar os benefícios de ambas as arquiteturas (uma e duas camadas), a arquitetura multicamadas também suporta os benefícios de uma arquitetura bastante flexível.

Com a adoção difundida da Internet como um canal para trocar informação e administrar transações empresariais, a demanda para desenvolvimento escalável e aplicações distribuídas que usam tecnologia de rede está aumentando. Outras tendências na comunidade empresarial, com a freqüência crescente de fusões empresariais e subsidiárias, estão direcionando demandas para tecnologias de integração de sistemas mais sofisticadas e arquiteturas de sistema flexíveis. O sucesso de uma organização depende freqüentemente de sua habilidade para se adaptar depressa a condições de negócio variáveis [ONT99].

Mais do que nunca, organizações precisam entregar aplicações novas ou atualizar a funcionalidade de aplicações sob restrições de tempos menores e exigências contínuas para confiabilidade de sistema, escalabilidade, integridade e desempenho. Como resultado, organizações de desenvolvimento estão constantemente procurando tecnologias de desenvolvimento novas que possibitam o cumprimento de exigências de aplicações variáveis enquanto minimizam o desenvolvimento e os custos de manutenção.

Muitas destas organizações têm trocado seu enfoque estratégico sobre o projeto de arquiteturas de aplicação multicamadas colocando maior ênfase no desenvolvimento de serviços reutilizáveis de objetos distribuídos na camada mediana. Arquiteturas de aplicação multicamadas oferecem maior escalabilidade, reusabilidade, flexibilidade e manutenção mais fácil, ou seja, todas as exigências essenciais de qualquer aplicação de comércio eletrônico.

2.3.1 COMPONENTES DE APLICAÇÃO

Uma arquitetura de aplicações dita o caminho através do qual estas serão criadas e como os seus componentes serão distribuídos através do sistema. A maioria das aplicações é feita de três tipos fundamentais de componente, segundo [HAM99] e [THO98]:

a) um componente de apresentação, que contém a lógica que apresenta a informação a uma fonte externa e obtém entradas daquela fonte. Na maioria dos casos, a fonte externa é um usuário final trabalhando num terminal ou estação de trabalho, embora a fonte externa possa ser um sistema robótico, um telefone ou algum outro dispositivo de entrada. A lógica de apresentação geralmente provê opções que permitem ao usuário navegar a partes diferentes da aplicação, manipular entradas e saídas de gráficos e o *display* de informações. Freqüentemente, componentes de apresentação também executam alguma validação simples de usuário. Um benefício de separar serviços de apresentação de serviços empresariais é a flexibilidade de interface de usuário, de modo que múltiplas interfaces, de *web browsers* para *desktop* ou dispositivos de *handheld*, podem ter acesso a aplicação;

b) um componente de negócio, que contém a lógica da aplicação que administra as funções do negócio e os processos executados pela aplicação. Essas funções e processos são executados ou por um componente de apresentação, quando um usuário executa uma opção, ou por uma outra função de negócio. Componentes de negócio também executam manipulação de dados transformando os dados para um nível de abstração que é significante à camada de serviços de apresentação. Tendo uma camada de serviços empresariais separada melhora a gerenciabilidade e escalabilidade;

c) um componente de acesso a dados, que contém a lógica que fornece à interface um sistema de armazenamento de dados, como um banco de dados relacional ou com algum outro tipo de fonte de dados externos, como uma aplicação externa. Métodos

de acesso a dados geralmente são invocados por um componente de negócio, embora em aplicações simples eles possam ser invocados diretamente por um componente de apresentação.

Segundo [HAM99], as três camadas referem-se às três partes lógicas que compõem uma aplicação, e não ao número de máquinas usadas pela aplicação.

O termo multicamadas implica em camadas adicionais dentro de pelo menos uma das três divisões principais, normalmente na camada de negócio.

Em uma arquitetura multicamadas, componentes de aplicação podem ser compartilhados por qualquer número de sistemas de aplicação e podem ser desenvolvidos usando a melhor ferramenta para o desenvolvimento. Os componentes de aplicação podem ser desdobrados em um ou mais sistemas físicos. A figura 3 exemplifica um projeto físico da arquitetura multicamadas. Eles se comunicam usando uma interface abstrata que funciona como um contrato, em termos do que está sendo tornado público.

FIGURA 3 ARQUITETURA MULTICAMADAS : PROJETO FÍSICO

Uma interface abstrata esconde a lógica de aplicação atual executada dentro do objeto de aplicação. A interface abstrata permite ver o objeto como uma caixa preta pelo mundo externo. Além do que, a lógica de aplicação dentro do objeto pode ser modificada ou substituída sem alterar os outros objetos da aplicação que interfaceiam com este objeto. Se a interface do componente não muda, nenhuma modificação precisa ser feita em qualquer outro componente.

Essa mesma camada abstrata esconde todos os detalhes da implementação das funções desempenhadas por um componente. Ela identifica a operação a ser realizada e define os parâmetros de entrada e saída necessários à execução da operação. Esse tipo de infra-estrutura possibilita serviços de localização, segurança e comunicação entre os componentes da aplicação.

Uma aplicação multicamadas é uma aplicação que foi dividida em componentes de aplicação múltiplos. Segundo [THO98], esta arquitetura possui várias vantagens significantes sobre as arquiteturas cliente/servidor tradicionais, inclusive melhorias em escalabilidade, desempenho, confiabilidade, gerenciabilidade, reusabilidade e flexibilidade.

Em uma aplicação cliente/servidor tradicional, a aplicação cliente contém lógica de apresentação (janela e manipulação de controle), lógica empresarial (algoritmos e regras de negócio) e lógica de manipulação de dados (conexões de banco de dados e *SQL queries*). O servidor é geralmente um banco de dados relacional administrativo do sistema (não de fato, uma parte da aplicação). Em uma arquitetura multicamadas, a aplicação do cliente contém somente a lógica de apresentação para o cliente. A lógica empresarial e de dados têm acesso lógico e dividido em componentes separados e desdobrados em um ou mais servidores.

Como arquiteturas multicamadas são baseadas em componentes, eles provêem uma maior flexibilidade para construir, desdobrar e manter aplicações. As partições de projeto da aplicação, a lógica de apresentação, lógica empresarial e dados têm acesso lógico em componentes que podem ser desdobrados dentro do *hardware* físico e infra-estrutura de *software* que conectam o sistema *desktop* para o *mainframe*. Tecnologia de desenvolvimento de *software* baseada em componentes permite ao desenvolvedor de aplicação criar e testar componentes individuais antes do sistema estar completo. O sistema é desdobrado e simplifica esforços de manutenção depois dos componentes serem desdobrados [THO98].

2.3.2 ESCALABILIDADE E PERFORMANCE

Movendo a lógica do negócio e a lógica de manipulação de dados para um servidor, uma aplicação pode tirar proveito do poder de *multithreaded* e sistemas de multiprocessamento. Componentes de servidor podem agrupar parte de recursos escassos, como processos, *threads*, conexões de banco de dados e sessões. Com o aumento da demanda de sistemas, componentes altamente ativos podem ser replicados e distribuídos em sistemas múltiplos. Embora sistemas cliente/servidor modernos possam suportar facilmente centenas de usuários simultaneamente, sua escalabilidade tem limites. Podem ser construídos sistemas multicamadas com essencialmente nenhum limite de escalabilidade. Se o projeto é eficiente, mais ou maiores servidores podem ser adicionados essencialmente ao ambiente para melhorar a performance ou suportar usuários adicionais. Sistemas multicamadas podem escalar para apoiar centenas de milhares ou milhões de usuários simultaneamente [THO98].

2.3.3 SUPORTE A SISTEMAS CRÍTICOS

Um ambiente multicamadas também pode apoiar muitos níveis de redundância. Por replicação e distribuição, uma arquitetura multicamadas elimina qualquer gargalo ou pontos fraco. A arquitetura multicamadas possibilita maior confiabilidade e disponibilidade para suportar operações empresariais críticas [THO98].

2.3.4 GERENCIABILIDADE

Uma aplicação multicamadas é mais fácil de administrar do que aplicações cliente/servidor tradicionais. Pouquíssimo código é desdobrado de fato nos clientes. A maioria da aplicação lógica é desdobrada, administrada e mantida nos servidores. Dificuldades, melhorias, novas versões e extensões passam a ser controlados por um ambiente de administração centralizado [THO98].

2.3.5 FLEXIBILIDADE

A arquitetura de aplicação multicamadas permite sistemas de aplicação extremamente flexíveis. A maioria da lógica de aplicação é implementada em componentes modulares pequenos. A lógica empresarial nos componentes é encapsulada atrás de uma interface abstrata bem definida. O código dentro de um componente individual pode ser modificado

sem haver uma mudança na interface. Então, um componente pode ser mudado sem alterar os outros componentes dentro da aplicação. Aplicações multicamadas podem facilmente se adaptar para refletir exigências empresariais variáveis [THO98].

2.3.6 REUSABILIDADE E INTEGRAÇÃO

Segundo [THO98], pela natureza de sua interface, um componente de servidor é um bloco de software reutilizável. Cada componente executa um conjunto específico de funções que são disponibilizados e acessíveis a qualquer outra aplicação pela interface. Uma função particular pode ser implementada e usada de novo em outra aplicação uma vez que esta, necessite desta função. Se uma organização mantém uma biblioteca de componentes, o desenvolvimento de aplicação se torna um assunto de montagem de componentes em uma configuração que executa as funções exigidas pela aplicação.

2.3.7 SUPORTE A MULTI-CLIENTES

Qualquer número de ambientes cliente pode ter acesso ao mesmo componente de servidor por sua interface, segundo [THO98]. Um único sistema de aplicação multicamadas pode apoiar uma variedade de dispositivos cliente e pode incluir tradicionais estações de trabalho *desktop*, clientes de rede ou os clientes mais exóticos, tal como eletrodomésticos de informação, *smartcards* ou os assistentes de dados pessoais.

2.3.8 VANTAGENS

Embora componentes de servidor e conceitos multicamadas sejam discutidos há quase uma década, relativamente poucas organizações os puseram em prática. Até recentemente, a maioria das organizações não sentia as pressões de escalabilidade que requeressem uma arquitetura multicamadas. Mas com a computação *web-based* está havendo um interesse crescente na arquitetura multicamadas. Infelizmente, construir aplicações multicamadas não é tão fácil quanto construir aplicações cliente/servidor. Aplicações multicamadas têm que interagir com uma variedade de serviços de *middleware*. Para atingir a escalabilidade, desempenho e confiabilidade, as aplicações têm que apoiar *multithreading*, compartilhamento de recursos, replicação e balanceamento de carga.

Segundo [HUE99] e [HAM99], arquiteturas multicamadas bem projetadas oferecem as seguintes vantagens:

a) escalabilidade e desempenho: os componentes em uma aplicação multicamadas são tipicamente projetados diferentes para se comunicar entre si em uma rede, assim eles podem ser distribuídos. Considerando que os componentes não são limitados a um único processador, eles podem ser escalados em diversas aplicações, assim como em uma aplicação que monitora e verifica gargalos. Se qualquer máquina é sobrecarregada em uso, existe a opção de substituir por uma máquina maior, reconfigurando a distribuição de componentes ou reproduzindo os componentes de servidor sobrecarregados em outra máquina;
b) desenvolvimento mais rápido: uma arquitetura multicamadas bem projetada provê a habilidade para desenvolver e desdobrar aplicações mais rapidamente. Considerando que funções de aplicação estão isoladas dentro de componentes de aplicação relativamente pequenos, a lógica de aplicação pode ser desenvolvida e testada como unidades independentes antes de integrar e desdobrar o sistema de aplicação como um todo. Considerando que podem ser testados componentes assim que estejam prontos, o processo de testes inicia antes que a aplicação esteja pronta para usuários. Dividindo o desenvolvimento de um projeto em componentes menores também reduz o risco de fracasso do projeto;
c) reutilização de objetos por outras aplicações: as vantagens de um ambiente multicamadas estendem além do ciclo de vida de uma única aplicação. Objetos podem ser compartilhados por diversas aplicações. De fato, o que está sendo construído não é propriamente uma aplicação, mas uma coleção de módulos (objetos) clientes e servidores que se comunicam através de uma interface padronizada e abstrata e que, ao serem combinados, funcionam como um sistema integrado de aplicações. Cada módulo é, na realidade, um objeto que pode ser reutilizado e compartilhado por diversos sistemas. Essa versatilidade *plug-and-play* é útil quando o setor de tecnologia da informação necessita suportar partes diferentes, mas relacionadas, do negócio. Por exemplo, três ou quatro aplicações *front-end* diferentes podem chamar um mesmo conjunto de objetos empresariais;
d) facilidade de manutenção do sistema: a vantagem mais óbvia dessa arquitetura é a facilidade de manutenção. Desde que as funções da aplicação são isoladas em

objetos granulares, a lógica da aplicação pode ser modificada muito mais facilmente do que antes – por exemplo, uma função que é realizada por uma aplicação financeira diz respeito a cálculo de taxas e impostos. O algoritmo para esse tipo de cálculo muda periodicamente, por força legal do cálculo de taxas. Ao isolar essas regras de negócio em objetos autônomos, os algoritmos podem ser trocados, sem afetar o resto da aplicação;

e) a aplicação passa a independer do fornecedor de banco de dados: uma grande vantagem da arquitetura multicamadas reside no fato de que a lógica da aplicação não é mais vinculada diretamente à estrutura de banco de dados ou a um SGBD particular. Objetos individuais da aplicação trabalham com as suas próprias estruturas de dados, que podem corresponder a uma estrutura do banco ou podem ser estruturas derivadas de um diferente número de fontes de dados. Quando os objetos na aplicação se comunicam, eles somente necessitam passar os parâmetros, como especificado na interface abstrata, em vez de passar os registros de um banco de dados, reduzindo assim o tráfego de rede. Os objetos de acesso a dados são os únicos componentes da aplicação que fazem uma interface diretamente com o banco de dados. Dessa forma, um banco de dados pode ser migrado de um fornecedor para outro, sem afetar a aplicação como um todo. Somente a camada da lógica de acesso aos dados precisará ser alterada. Isso representa uma autonomia para se reagir melhor a uma mudança tecnológica ou de negócio. Um outro fato importante é que, na camada lógica de apresentação, ou seja, o lado cliente da aplicação, não há a menor necessidade de ter configurações para acesso a dados, como existe hoje nas aplicações cliente/servidor com a instalação dos chamados clientes de base de dados;

f) a abstração da lógica de acesso a dados leva a um outro significativo benefício: o conceito de que os dados podem ser estendidos para incluir arquivos seqüenciais, indexados, bancos não-relacionais e mesmo sistemas de aplicação legados. Um conjunto de módulos de acesso a dados pode ser desenvolvido para prover acesso a esses ambientes legados, com um conveniente conjunto de interfaces abstratas que são acessíveis de qualquer lugar para toda a organização. Num mundo multicamadas, um módulo de acesso a dados que acessa dados legados hoje, pode

ser substituído por um outro módulo que acesse dados relacionais amanhã, sem afetar o resto da aplicação. O segredo disso tudo chama-se interfaces;

g) alta produtividade de desenvolvimento através de especialização: com a tecnologia cliente/servidor, cada programador deve desenvolver todos os aspectos de uma aplicação, incluindo apresentação, negócio e lógica de acesso a dados. Com isso, o fato de que muitos programadores se superam em determinadas tarefas, e não em outras, não é considerado, bem como o fato de que eles são mais produtivos quando especializados. No mundo multicamadas, programadores com excelente habilidade para interface de usuários podem se concentrar em desenvolver componentes de apresentação e não necessitam saber sobre os detalhes internos da lógica de negócios ou como dados são acessados de um banco de dados. Isso vale também para analistas de banco de dados, que conhecem a melhor maneira de acessar dados, ou analistas de negócio, que podem se concentrar em desenvolver algoritmos de negócio. Tudo que eles precisam saber são as interfaces, e aqui talvez a figura do contrato faça sentido. Quando um componente de negócio precisar de dados, é só fazer a chamada apropriada ao componente de acesso a dados;
h) infra-estrutura distribuída de computação: uma infra-estrutura distribuída de computação provê os serviços que permitem aos componentes da aplicação serem transparentemente distribuídos por qualquer número de sistemas físicos, um conceito normalmente conhecido como particionamento (*partitioning*).

Para se ter sucesso com essas novas tecnologias, a empresa necessita não somente usar novas ferramentas, mas também trocar a forma como as aplicações são desenvolvidas. Na realidade, a tecnologia multicamadas obrigatoriamente força a utilização de novas metodologias para o desenvolvimento de sistemas.

Para poder se desenvolver, a organização deve definir a sua arquitetura de aplicações como um todo. Uma arquitetura ajuda a definir como sistemas, subsistemas, ferramentas ou aplicações se encaixam num ambiente de negócio. Ela provê métricas que podem ser utilizadas para selecionar tecnologias que serão usadas para o desenvolvimento de sistemas. Uma arquitetura precisa especificar os produtos que serão utilizados. Ela deve basear-se em princípios corporativos e, portanto, deve unicamente refletir a corporação.

Componentes de desenvolvimento dentro de uma arquitetura multicamadas permitem a companhias construir aplicações maiores sem a necessidade de atualizar continuamente ou implementar mudanças. Componentes de desenvolvimento permitem aos desenvolvedores e projetistas planejar o futuro e permitir escalabilidade e desenvolvimento de protótipo.

Em uma arquitetura multicamadas, a programação da interface para o cliente apresenta níveis mais altos de abstração. Pode ser implementada ou chamando a interface de componente de negócio, um objeto de negócio ou a interface de componente de dados, um objeto de dados.

3 FERRAMENTAS PARA DESENVOLVIMENTO

3.1 EJB – CONCEITO

Enterprise JavaBeans (EJB) é uma arquitetura de componentes para desenvolvimento de aplicações distribuídas e orientada a objetos, segundo [MON99]. Aplicações que utilizam a arquitetura EJB são escaláveis, transacionais e multi-usuários. Após desenvolvidas, estas aplicações executam em qualquer plataforma de servidor que suporta a especificação de EJB.

Segundo [THO98], *Enterprise JavaBeans* (EJB) é uma tecnologia que define um modelo para o desenvolvimento de componentes reutilizáveis do lado do servidor. São pedaços de código de aplicação pré-desenvolvidos que podem ser unidos em sistemas de aplicações diversos.

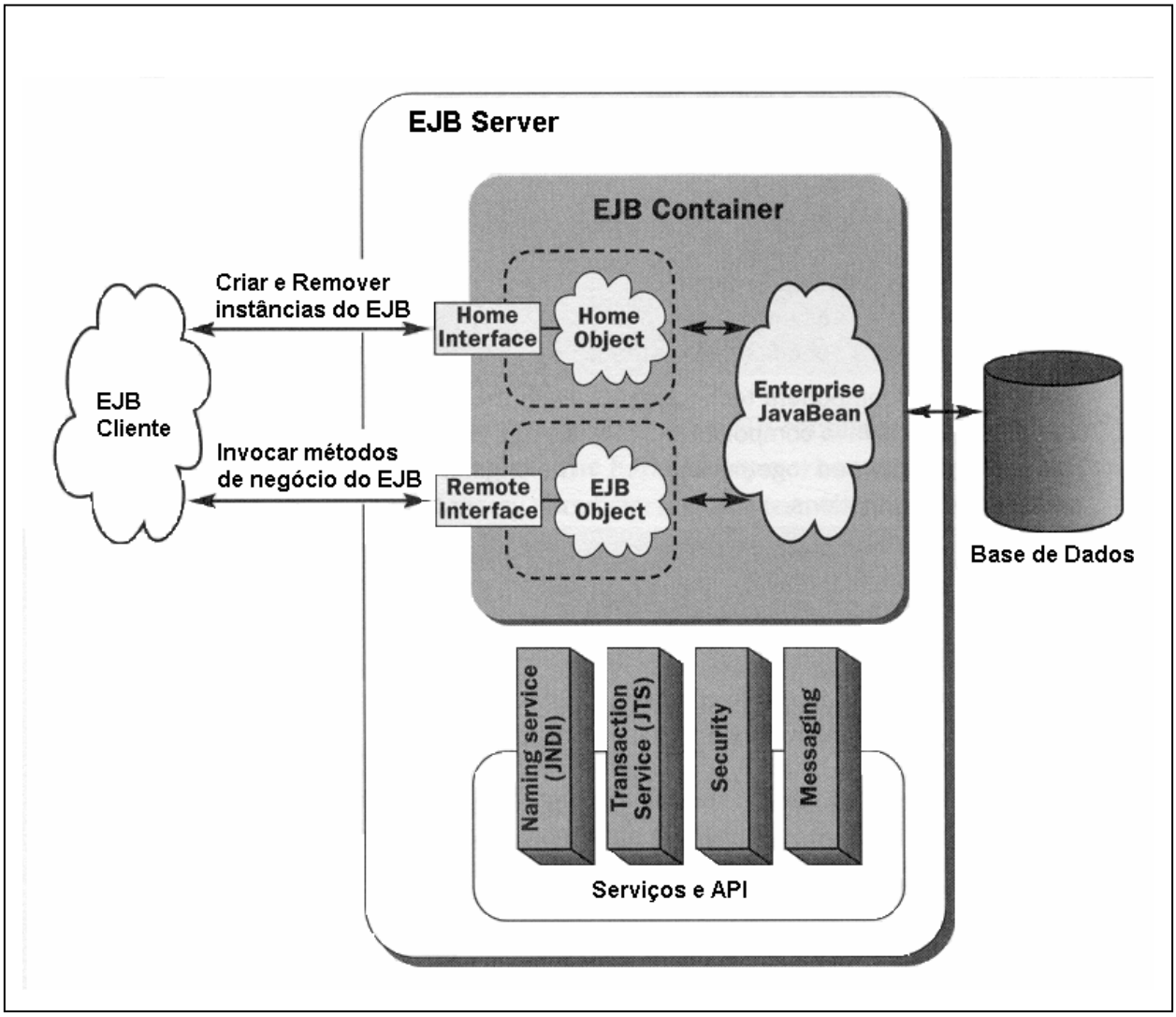
EJB permite ao desenvolvedor de software construir objetos de negócio reutilizáveis (*enterprise beans*) do lado do servidor. Porém, EJB leva a noção de objeto reutilizável mais adiante permitindo a programação baseada em atributos para dinamicamente definir ciclo de vida, transação, segurança e comportamento de persistência em aplicações EJB. Por exemplo, usando técnicas baseadas em atributos, o mesmo *enterprise bean* pode exibir comportamento transacional diferente em aplicações diferentes. Adicionalmente, o método de persistir um *enterprise bean* pode ser alterado durante o desenvolvimento sem ter que alterar o *enterprise bean*.

3.2 ARQUITETURA

Segundo [SES99], uma arquitetura básica EJB é mostrada na figura 4 e consiste de:

a) um servidor EJB;
b) EJB *containers* que executam dentro do servidor;
c) *home objects*, *remote EJBObjects* e *enterprise beans* que executam dentro de *containers;*
d) EJB clientes;
e) sistemas auxiliares como *Java Naming and Directory Interface* (JNDI), *Java Transaction Service* (JTS), serviço de segurança, entre outros.

FIGURA 4 ARQUITETURA BÁSICA DE ENTERPRISE JAVABEAN



Fonte: Enterprise javabeans by example, 1999, 158.

3.2.1 SERVIDOR EJB

O servidor EJB é o processo de alto nível ou aplicação que gerencia EJB *containers*, provendo acesso para serviços de sistemas. Um servidor é requerido para disponibilizar JNDI e serviços de transação[CRE99].

O servidor EJB disponibiliza serviços de sistema para EJB *containers* como multiprocessamento, acesso a dispositivos, entre outros. Os EJB *containers* executam dentro do servidor EJB e este permite que os EJB *containers* sejam visíveis ao mundo externo [SES99].

3.2.2 EJB *CONTAINERS*

O EJB *container* atua como uma interface entre um *enterprise bean* e a funcionalidade de uma plataforma específica que suporta o *bean* (baixo-nível). Em essência, o EJB *container* é uma abstração que gerencia uma ou mais classes ou instâncias EJB, enquanto executa os serviços disponíveis exigido pelas classes EJB por interfaces padrão definidas na especificação de EJB. O fabricante de *containers* é livre para prover serviços adicionais implementados no *container*. Um EJB cliente nunca acessa um *bean* diretamente. Qualquer acesso do *bean* é terminado pelos métodos das classes geradas pelo *container* que, em troca, invoca os métodos do *bean* [SES99].

Ele é responsável em prover ao *bean* serviços como controle de transações, gerenciamento de ciclo de vida e segurança. O *container* não é visível para o cliente ou para o *bean* contido. Porém, todos os métodos invocados pelo *bean* são interceptados pelo *container*, se for preciso, o *container* provê vários serviços para o *bean* transparentemente [CRE99].

Segundo [THO98], um servidor de aplicação provê um *container* para administrar a execução de um componente. Quando um cliente invoca um componente de servidor, o *container* aloca uma linha de processo automaticamente e inicia o componente. O *container* administra todos os recursos em nome do componente e administra todas as interações entre o componente e os sistemas externos.

Existem dois tipos de EJB *containers*: *session containers* que podem conter EJB passageiros, não persistentes, cujos estados não são salvos e *entity containers* que contém EJB persistentes cujos estados são salvos entre as invocações [SES99].

3.2.3 HOME INTERFACE E *HOME OBJECT*

Métodos para localizar, criar e remover instâncias de classes EJB são definidos na *home interface*. A *home object* é a implementação da *home interface*. O desenvolvedor EJB deve primeiro definir a *home interface* para seu *bean* [SES99] e [CRE99].

3.2.4 *REMOTE INTERFACE* E EJB *OBJECT*

A *remote interface* lista os métodos de negócio disponíveis para o *enterprise bean*. O *EJBObject* é a visão do cliente do *enterprise bean* e implementa a *remote interface* [CRE99] e [SES99].

3.2.5 EJB - *ENTERPRISE JAVABEANS*

O real *enterprise bean* é contido dentro de um EJB *container* e nunca deve ser acessado diretamente por qualquer um a não ser pelo *container*. Embora seja possível acessar diretamente, isto é desaconselhado pois quebra o contrato entre o *bean* e o *container* [SES99].

O EJB *container* intermedia todos os acessos ao *enterprise bean*. Por esta razão, o desenvolvedor do *enterprise bean* não implementa a *remote interface* dentro do *enterprise bean*. O código da implementação é gerado automaticamente pelas ferramentas de *container* [SES99].

3.2.6 O CLIENTE EJB

Clientes EJB localizam o EJB *container* específico que contém o *enterprise bean* pelo *Java Naming and Directory Interface* (JNDI). Eles fazem uso do EJB *container* para invocar métodos do *bean*. O EJB cliente somente pega uma referência para uma instância do *EJBObject* e nunca realmente uma referência para sua atual instância do *enterprise bean.* Quando o cliente invoca um método, o *EJBObject* recebe a requisição e delega para a instância do EJB, provendo alguma funcionalidade de envoltura necessária no processo [CRE99] e [SES99].

O cliente usa a *home object* para localizar, criar ou destruir instâncias de um *enterprise bean*. Ele usa a instância do *EJBObject* para invocar os métodos de negócio de uma instância de um *bean* [SES99].

3.3 MODELO DE COMPONENTES

Segundo [HEM99], a arquitetura EJB estende logicamente o modelo de componentes do JavaBeans para permitir o desenvolvimento de componentes de servidor.

Já [MON99] afirma que EJB não estende ou usa o modelo de componentes original do *JavaBeans*. A proposta original de *JavaBeans* é ser usado para processos externos, enquanto a proposta de EJB é para ser usado como componentes de processos internos. Uma vez um componente definido, ele se torna um pedaço de software que pode ser distribuído e usado em outras aplicações. Um componente é desenvolvido para uma proposta específica mas não uma aplicação específica.

EJB é designado para administrar assuntos envolvendo gerenciamento de objetos de negócio distribuídos em uma arquitetura três camadas.

Segundo [JUB99], um modelo de componentes define a arquitetura básica de uma aplicação baseada em componentes, especificando a estrutura das interfaces e os mecanismos pelos quais interagem com seu ambiente. O modelo de componentes provê diretrizes para criar e implementar componentes que podem operar juntos e formar uma aplicação maior. Estas diretrizes permitem construir aplicações combinando componentes de diferentes desenvolvedores ou diferentes vendedores.

Um componente é um “bloco” de software reutilizável que tem uma interface definida e provê a funcionalidade da aplicação. Componentes podem ser combinados com outros componentes e “colados” para rapidamente produzir uma aplicação customizada [JUB99] e [THO98].

3.3.1 COMPONENTES DE SERVIDOR

Segundo [JUB99], componentes de servidor são componentes de aplicação que executam em um servidor. Estes componentes implementam pequenas exigências e são designados para combinar com outros componentes da mesma arquitetura para formar uma solução total.

Um modelo de componente de servidor simplifica o processo de desenvolvimento movendo a lógica de negócios para o servidor. Também ajuda dividindo a aplicação em vários componentes logicamente bem definidos para requerimentos de reusabilidade, escalabilidade e performance.

Componentes de servidor são componentes de aplicação que executam em um servidor de aplicação. A tecnologia EJB suporta as demandas rigorosas de larga escala, distribuição,

sistemas de aplicação missão-crítica, além do desenvolvimento de aplicações baseadas em arquiteturas multicamadas na qual a maioria da lógica da aplicação é movida do cliente para o servidor. A lógica de aplicação é dividida em um ou mais objetos empresariais que são desdobrados em uma aplicação servidor.

Uma aplicação Java servidor provê um ambiente de execução aperfeiçoado para o lado do servidor de aplicação de componentes. Uma aplicação Java servidor possibilita um alto desempenho em um ambiente de execução robusto, além de serem altamente escaláveis, apoiando sistemas de aplicação para Internet.

3.4 PORQUE UTILIZAR EJB

Segundo [HEM99] e [THO98] as vantagens excedem em valor as desvantagens, especialmente para aplicações mais complexas:

a) produtividade: EJB aumenta produtividade, pois os desenvolvedores não precisam se preocupar com a programação de baixo nível (como conexão, segurança, administração de transação, gerência de estados, persistência, número de clientes e *multithreading*); eles simplesmente se concentram em escrever a lógica empresarial e desenvolvem o *enterprise bean* como se este fosse usado por um único cliente;
b) arquitetura de componente do lado do servidor aberta: EJB provê portabilidade para plataformas. Porque EJB fixa um caminho claro para fabricantes de aplicação, tudo tem que prover a mesma funcionalidade mínima nos produtos de servidor e os construtores de componentes têm que facilitar a construção de componentes do lado do servidor, não só componentes de interfaces gráficas (GUI) do lado do cliente;
c) programando no servidor orientado a objeto: graças às raízes da orientação a objetos de Java e o modelo de componente de EJB, organizações podem criar e usar mais facilmente componentes reutilizáveis, gastando assim, menos tempo escrevendo código. Um fator que ajuda esta reusabilidade é que são empacotados a lógica e os dados em um mesmo objeto. Adicionalmente, *containers* de EJB podem traduzir dados relacionais automaticamente em objetos. Isto elimina a distinção entre ter acesso a dados de um banco de dados *versus* qualquer outro objeto;

d) Java na camada mediana: EJB traz todas as características de Java (como segurança, serviço de diretório e serialização) para a camada mediana, ou seja, tira do cliente estes serviços permitindo que este fique mais leve, colocando na camada intermediária, entre o cliente e o banco de dados os serviços de segurança, entre outros;

e) apoio para outras linguagens e CORBA: EJB provê apoio a outras linguagens e CORBA. É o fabricante de *middleware*, não o desenvolvedor de *enterprise bean*, quem entende sobre os assuntos de protocolos, dessa forma, pode ser usado qualquer protocolo distribuído para apoiar muitos tipos de clientes (como COM/DCOM, *Servlets*, por exemplo).

3.5 METAS

Segundo [HEM99], as especificações de EJB 1.0 definem certas metas para fabricantes de *java middleware* que implementarão este padrão. Algumas destas metas:

a) padrão de arquitetura de componentes distribuída para Java;

b) portabilidade para plataformas e vendedores;

c) aumento de produtividade por simplicidade (o desenvolvedor não se preocupa com gerenciamento de estados, *multithreading*, conexões de rede e protocolos e assim por diante) ;

d) compatibilidade com outras linguagens de programação e CORBA;

e) compatibilidade com infra-estrutura ou investimentos de plataforma existentes.

3.6 CARACTERÍSTICAS

Para cumprir as metas acima, EJB possui várias características que permitem a arquitetura de componentes distribuída. A tabela 2 cita estas características:

TABELA 2 CARACTERÍSTICAS DE EJB

CARACTERÍSTICA	APOIADA POR
Modelo de Componentes	• *Session beans* • *Entity beans*

Persistência de objetos	• *Entity beans* (*containers* de EJB)
Administração de Transação	• JTS/JTA • `javax.jts.UserTransaction` • Possa ser vendedor proprietário
Manipulação de Exceção	• Do lado do cliente e do lado do servidor
Segurança	• `javax.security` • Métodos segurança – relacionados em `javax.ejb.EJBContext` • Propriedades de *deployment descriptor*
Nomeando e serviço de diretório	• *Java Naming and Directory Interface* (JNDI)
Protocolos	• RMI/JRMP • IIOP (CORBA) • Qualquer outro protocolo distribuído
Apoio para CORBA	• CORBA (ejb.idl)
Programação baseada em atributos	• Arquivo de descritor de desenvolvimento
Desenvolvimento	• Arquivo EJB .JAR

3.6.1 TIPOS DE *ENTERPRISE BEANS*

EJB possui dois tipos de *enterprise bean*:

a) *entity beans*;

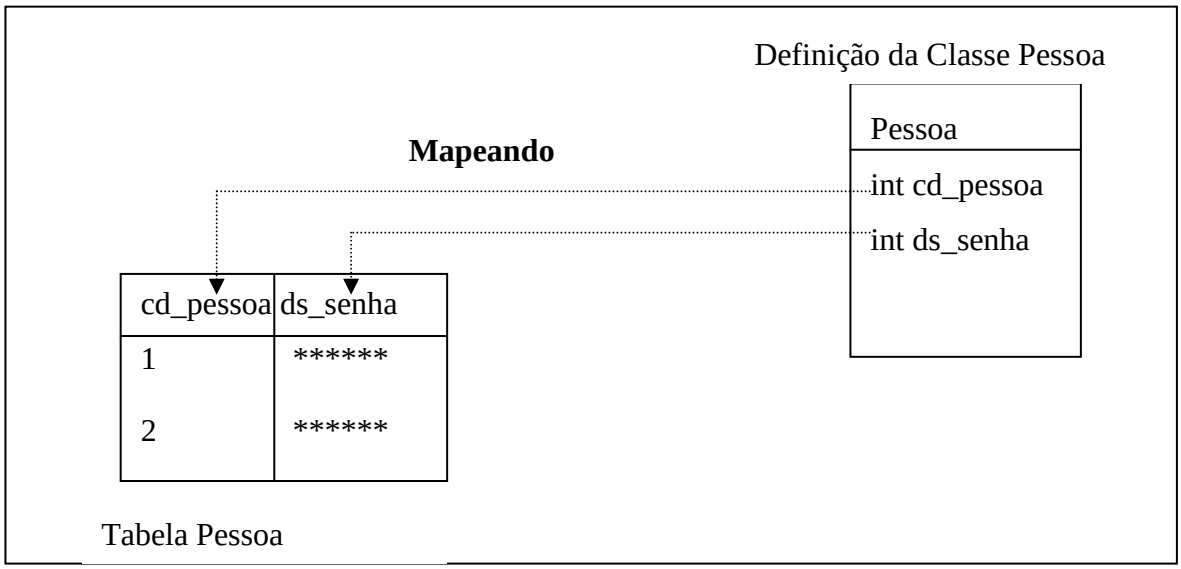
b) *session beans*.

3.6.1.1 ENTITY BEANS

Segundo [MON99], *entity beans* modelam conceitos empresariais que podem ser expressados como substantivos. Isto não é uma regra, mas ajuda a determinar quando um conceito empresarial é um candidato para ser implementado como um *entity bean*. Assim como na gramática, *entity beans* podem descrever pessoas, lugares ou coisas (reais ou abstratas). *Entity beans* descrevem o estado e o comportamento de objetos do mundo real e permitem aos desenvolvedores encapsular os dados e as regras de negócio associadas com conceitos específicos.

Entity beans sempre têm estados que podem ser persistidos e armazenados por múltiplas invocações. Múltiplos EJB clientes podem, porém, compartilhar um *entity bean*. O tempo de vida de um *entity bean* não está limitado ao tempo de execução da máquina virtual. Um *crash* da máquina virtual pode resultar em um *rollback* da transação corrente, mas não destruirá o *entity bean* nem invalidará as referências que outros clientes têm a este *entity bean.* Além disso, um cliente pode conectar depois ao mesmo *entity bean* que usa sua referência de objeto porque encapsula uma chave primária única e permite ao *entity bean* ou seu *container* recarregar seu estado [SES99].

FIGURA 5 MAPEAMENTO DE UM *ENTITY BEAN* PARA RELACIONAL



Entity beans proporcionam aos programadores mecanismos mais simples para acessar e modificar dados. Quando um novo *bean* é criado, um novo registro deve ser inserido na base de dados e uma instância do *bean* deve ser associada com este dado (figura 5). Como um *bean* é usado e seu estado é modificado, esta mudança deve ser sincronizada com os dados na base de dados: entradas devem ser inseridas, alteradas e removidas. O processo de coordenar os dados representados por uma instância de um *entity bean* com o banco de dados é chamado persistência. [MON99]

Existem dois tipos de *entity beans* :

a) *containers-managed beans*;

b) *beans-managed.*

Eles são distinguidos pela maneira como administram a persistência. *Containers-managed beans* tem a persistência automaticamente administrada pelo EJB *container*. O *container* sabe como a instância de um campo de um *bean* está mapeado para a base de dados e automaticamente insere, apaga ou altera os dados associados com a entidade na base de dados. *Beans-managed* fazem todo esse trabalho explicitamente: o desenvolvedor do *bean* tem que escrever o código para manipular a base de dados. O EJB *container* chama a instância do *bean* quando é seguro inserir, alterar ou apagar os dados da base de dados, mas não provê nenhuma outra ajuda. A instância do *bean* faz todo o trabalho de persistência [MON99].

Entity beans são objetos persistentes e representam uma visão de objeto de dados armazenada em algum meio de armazenamento permanente, segundo [HEM99]. Para entender melhor o que é um *entity bean*, deve-se pensar nele como se fosse uma linha em um banco de dados relacional. Ao longo dessas linhas, pode ser criado, localizado ou removido um *entity bean* - usando os métodos `create()`, `findxxx` ou `remove()`- da mesma maneira que uma linha de banco de dados pode ser inserida, selecionada ou apagada de um banco de dados SQL.

Um *entity bean* vive em um EJB *container* assim como um registro vive em um banco de dados. *Stateful session beans* distintos e *entity beans* podem acessar concorrentemente clientes múltiplos – a concorrência é administrada pelo EJB *container*. Considerando que *entity beans* provêem uma visão de objeto dos dados, eles podem criar dados através do método `create()` do *entity bean*, alterar ou apagar dados através do método `remove()`,ou ainda podem ser usados para devolver um único objeto ou coleção de objetos através de um dos métodos `find()` [HEM99].

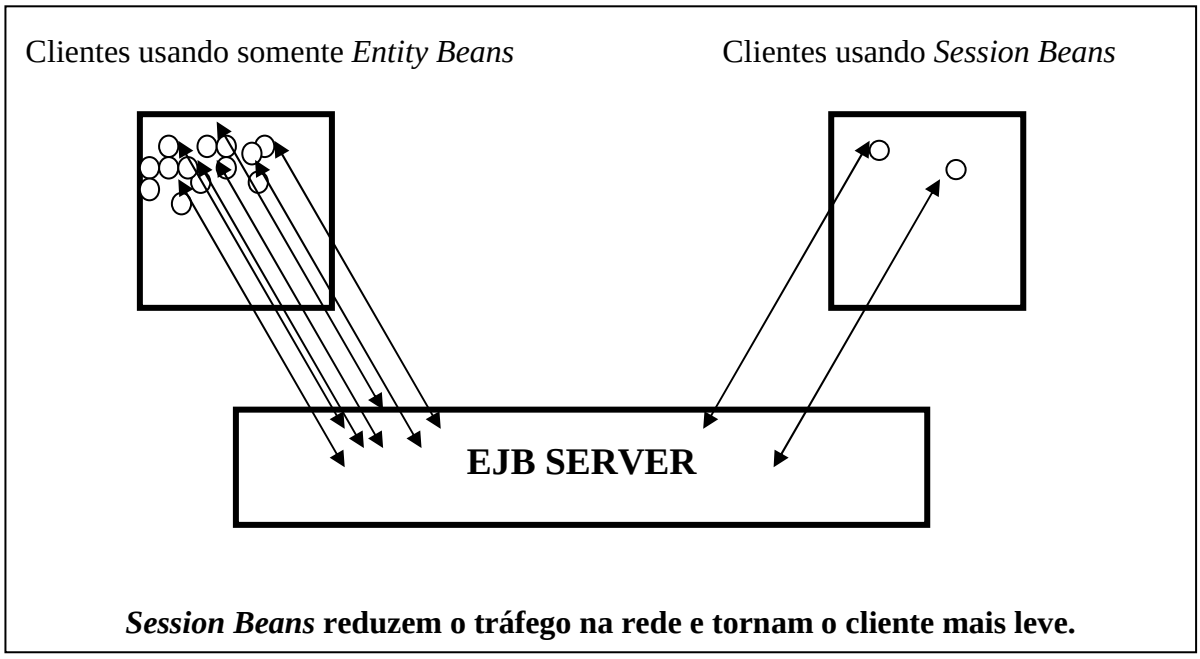
3.6.1.2 *SESSION BEANS*

Segundo [JUB99], um *session bean* é um objeto EJB que representa uma conversação passageira com o cliente. É uma extensão lógica do programa cliente que executa operações no servidor, como a execução de funções de negócio ou manipulação de dados em transações seguras, em nome do cliente.

Segundo [JUB99] e [CRE99] um *session bean* é privado para a conexão do cliente e não pode ser compartilhado com outros clientes. Isto permite ao *bean* manter informações da sessão de um cliente específico, chamado conversão de estado. Um *session bean* que mantém conversão de estados é chamado *stateful session bean*.

Segundo [SES99], um *session bean* é criado pelo cliente e, em muitos casos, existe somente para a duração de uma única sessão. Embora *session beans* possam ser transacionais, eles não são recuperáveis com um *crash* do sistema. E, ao contrário de *entity beans*, *session beans* não são persistentes (figura 6).

FIGURA 6 *SESSION BEANS*



Session beans são usados para descrever interações entre *beans* ou para implementar tarefas particulares. Ao contrário de *entity beans*, *session beans* não representam compartilhamento de dados no banco de dados, mas eles podem acessar estes dados compartilhados. Para acessar estes dados diretamente, o *session bean* pode representar *workflow*. *Workflow* descreve todos os passos requeridos para realizar uma tarefa particular. *Session beans* são parte da mesma API de negócios como *entity beans*, mas como

componentes *workflow*, eles apresentam uma proposta diferente. *Session beans* podem gerenciar a interação entre *entity beans,* descrevendo como eles devem trabalhar para realizar determinada tarefa. O relacionamento entre *entity beans* e *session beans* é como o relacionamento entre um *script*, um jogo e os atores que executam este jogo. Enquanto *entity beans* são os atores, o *session bean* é o *script*. Atores sem um *script* podem cada um executar uma função específica, mas somente no contexto de um *script* é que eles podem contar a história [MON99].

Existem dois tipos de *session beans*: *stateless session beans* e *stateful session beans*.

3.6.1.2.1 *STATELESS SESSION BEANS*

Stateless session beans são uma coleção de serviços relacionados, cada um representado por um método; o *bean* não mantém o estado da invocação de um método para o próximo. Quando um método é invocado de um *stateless session bean*, ele executa o método e retorna o resultado sem saber ou se preocupar se foram feitos outros pedidos antes ou se surgiram novos. Um *stateless session bean* poderia ser comparado a um procedimento ou programas *batchs* que executam uma requisição baseados em alguns parâmetros e retornam o resultado. *Stateless session beans* tendem a ser propósito geral ou reutilizável, como um serviço de software [MON99].

Stateless session beans têm vida mais longa porque eles não retêm qualquer estado e não são dedicados a um cliente, também não salvam dados em um banco de dados, porque não representam qualquer dado para ser salvo. Uma vez um *stateless session bean* tenha terminado um método invocado por um cliente, ele pode ser designado novamente para qualquer outro objeto EJB para servir a um novo cliente.

Stateless session beans freqüentemente executam serviços que são bastante genéricos e reutilizáveis. Os serviços podem ser relacionados, mas eles não são mutuamente dependentes. Isto significa que tudo que um *stateless session bean* precisa saber tem que ser passado via parâmetro para o método, com exceção da informação obtida pelo `SessionContext`. Isto provê uma interessante limitação, s*tateless session beans* não se “lembram” de qualquer coisa da invocação de um método para o próximo [MON99].

Este tipo de sessão EJB não tem nenhum estado interno, porque são *stateless*, eles não precisam ser passivos e podem ser agrupados em serviços múltiplos de clientes [SES99].

Segundo [MON99], um *stateless session bean* é fácil de desenvolver e também muito eficiente. Eles requerem poucos recursos do servidor porque não são nem persistentes nem dedicados ao cliente. Por eles não serem dedicados ao cliente, muitos objetos EJB podem usar poucas instâncias de um *stateless bean*. Um *stateless session bean* não mantém o estado relativo aos serviços do objeto EJB, assim pode ser trocado livremente entre objetos EJB. Assim que uma instância *stateless* invoca um método, ele pode ser trocado para outro EJB objeto imediatamente. Como não há estados, um *stateless session bean* não requer passivação ou ativação, reduzindo o *overhead*.

Segundo [HEM99], s*tateless session beans* são simples, quer dizer, fácil de desenvolver, com baixas exigências de recurso de *runtime* no servidor para escalabilidade de componentes. Qualquer estado, se preciso for, é mantido pelo cliente e altamente escalável para o servidor. Considerando que nenhum estado é mantido neste tipo de *enterprise bean*, não são amarrados s*tateless session beans* a qualquer cliente específico, consequentemente qualquer instância disponível de um s*tateless session bean* pode ser usada para um serviço de um cliente.

Stateless session beans podem ser usados para a geração de relatórios, processamento *batch* ou qualquer outro serviço como validação de um cartão de crédito [MON99].

3.6.1.2.2 *STATEFUL SESSION BEANS*

Stateful session bean é uma extensão da aplicação cliente. Ele executa tarefas e mantém o estado relacionado com o cliente. Este estado é chamado *conversational state* porque ele representa uma continuidade da relação entre o *stateful session bean* e o cliente. Métodos invocados em um *stateful session bean* podem escrever e ler dados neste estado que é compartilhado entre todos os métodos do *bean* [MON99].

Um s*tateful session bean* é definido como um *session bean* que contém estado que deve ser retido por chamadas de múltiplos métodos e transações. *Conversational state* inclui os valores dos campos do *bean* e fechamento transitivo, quer dizer, todas as referências aos objetos que seriam armazenados seriando a instância do *bean* [JUB99].

O *conversational state* pode até mesmo conter recursos abertos, como arquivos abertos, *socket descriptors* ou conexão de banco de dados que não podem ser salvos quando um *bean* é desalocado da memória. Nestes casos, o desenvolvedor fecha e reabre os recursos pelos métodos `ejbPassive` e `ejbActive`, respectivamente [JUB99].

Stateful session beans são dedicados a um cliente e possuem um período de *timeout.* O *timeout* é especificado no *deployment descriptor*, definido em segundos e aplicado entre as invocações dos métodos de negócio pelo cliente. Se o cliente falhar para usar o *stateful bean* antes do tempo final, a instância do *bean* é destruída e a referência remota é invalidada. Isto impede o *stateful session bean* de demorar muito tempo depois que um cliente foi desligado. Um cliente também pode explicitamente remover um *stateful session bean* chamando um dos métodos necessários [MON99].

Stateful session beans permitem gerenciamento de estado fácil e transparente no lado de servidor. Como o estado é mantido neste tipo de *enterprise bean*, o servidor de aplicação administra pares de *client-bean*. Em outras palavras, cada instância de um determinado *enterprise bean* é criada em nome de um cliente e é um recurso privado àquele cliente (embora pudesse ser compartilhado por clientes que usam a instância do *enterprise bean*). Em essência, um *stateful session beans* é uma extensão lógica do cliente, exceto a carga do cliente que é distribuída entre ele e o *enterprise bean* no servidor. Qualquer estado relacionado a dados nas variáveis do objeto não sobrevive a um *shutdown* do servidor ou *crash*, embora um fabricante pudesse prover uma implementação transparente para o cliente mantendo o estado do *enterprise bean*. *Stateful session bean* podem ter acesso a recursos persistentes (como bancos de dados e arquivos) em nome do cliente, mas eles não representam os dados de fato, como *entity beans*. Por exemplo, um *session bean* poderia ter acesso a dados por JDBC ou por um *entity bean* [HEM99].

3.6.2 COMPARAÇÃO ENTRE *ENTITY* E *SESSION BEANS*

Segundo [SES99] as maiores diferenças entre *entity beans* e *session beans* estão descritas na tabela 3.

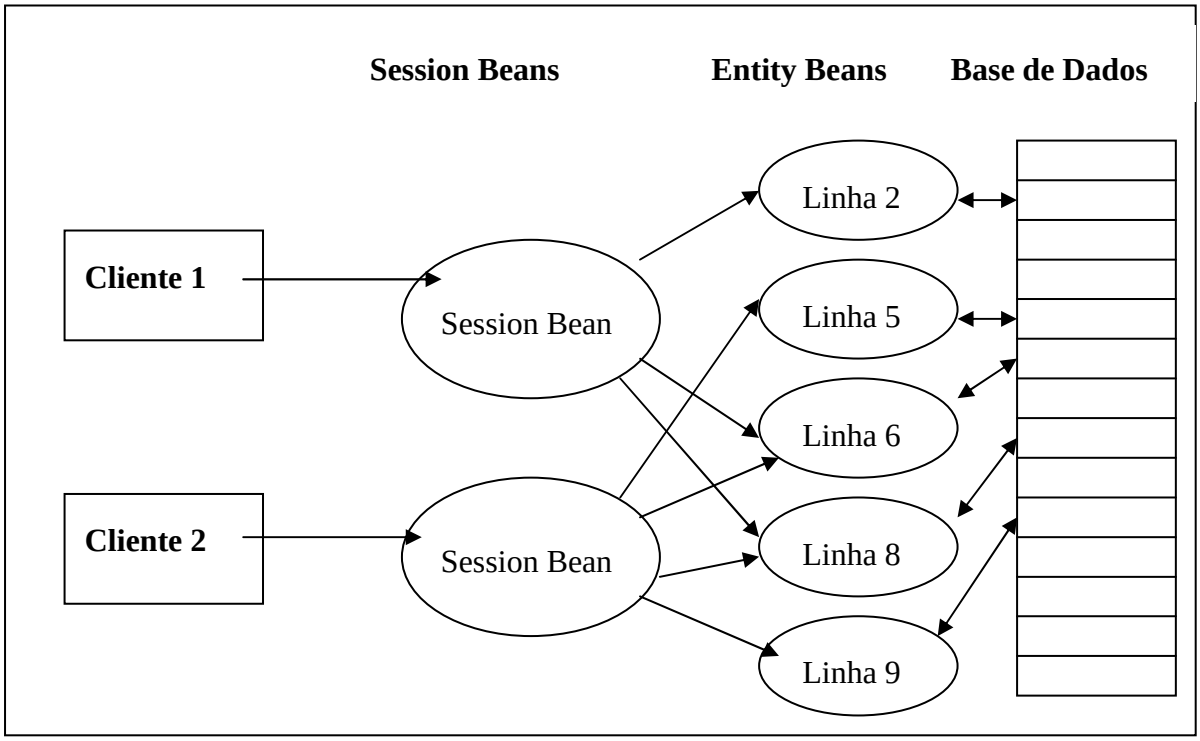
TABELA 3 DIFERENÇAS ENTRE *ENTITY BEANS* E *SESSION BEANS*

SESSION BEAN	ENTITY BEAN

Os membros de dados do *session bean* contêm *conversational state.*	Os membros de dados do *entity bean* representam os dados atuais no modelo de domínio.
Um *session bean* pode acessar a base de dados para um único cliente.	*Entity beans* compartilham acessos a banco de dados com múltiplos clientes.
Porque *session beans* possuem *conversational state* com um único cliente, é permitido que o *session bean* armazene por cliente o estado da informação.	Porque *entity beans* são compartilhados entre múltiplos clientes, eles não permitem armazenamento de informação pelo estado do cliente.
O relacionamento entre um *session bean* e o cliente é um-para-um.	O relacionamento entre um *entity bean* e uma linha do modelo de domínio é um-para-um.
A vida do *session bean* é limitada a vida do cliente.	Um *entity bean* persiste contanto que os dados existam no banco de dados.
Session beans podem ser *transaction aware.*	*Entity beans* são transacionais.
Session beans não sobrevivem a *crashes* do servidor.	*Entity beans s*obrevivem a *crashes* do servidor.

A figura 7 mostra uma comparação entre *entity* e *session beans*. O conveniente é que os clientes acessem somente *session beans* para evitar tráfego de rede, os *session beans* por sua vez, podem acessar qualquer *entity* bean e, estes, acessam a base de dados, inserem, alteram e apagam dados.

FIGURA 7 COMPARAÇÃO ENTRE *ENTITY BEANS* E *SESSION BEANS*



Apesar das diferenças, segundo [HEM99] *session* e *entity beans* compartilham as seguintes características:

a) um *handle* (`javax.ejb.Handle`) pode ser obtido à instância de um *enterprise bean* que usa o método `getHandle();`
b) um *enterprise bean* pode estar customizado no momento do desenvolvimento editando suas propriedades de ambiente;
c) instâncias de um *enterprise bean* são criadas e administradas em tempo de execução pelo servidor de aplicação;
d) podem ser manipulados certos atributos do *enterprise bean*, como modo de transação e segurança, em momento de desenvolvimento usando as ferramentas de desenvolvimento EJB *server*;
e) podem ser criados ou apagados pelos métodos `create()` e `remove()` usando a *home interface*. O método `create()` é usado para inicializar os campos do objeto.

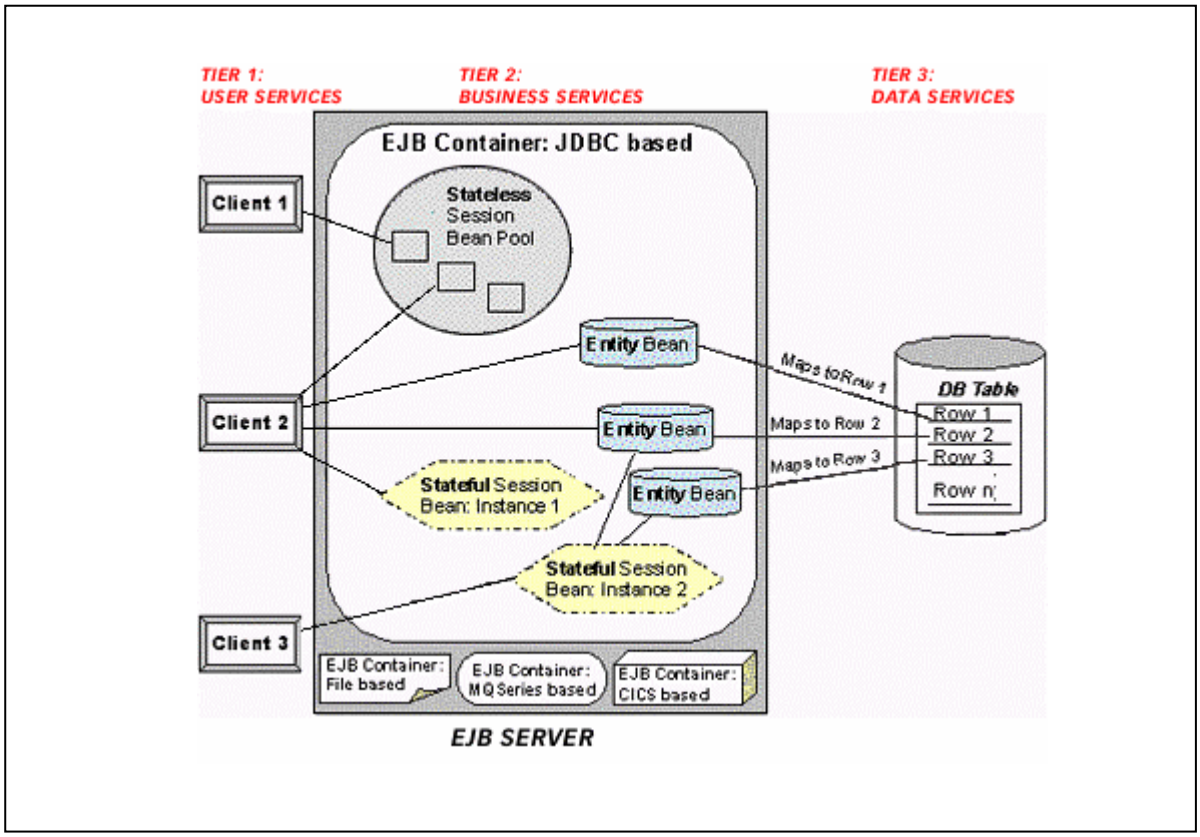
Decidir que tipo de *enterprise bean* usar é um assunto para o projeto de desenvolvimento. [HEM99] cita duas regras gerais:

a) usar *entity beans* em vez de *session beans* se não quiser embutir chamadas a banco de dados nas classes *enterprise bean* mas quer tirar proveito das características de EJB como administração de transação automática, recurso *pooling* e gerenciamento de persistência de *container*. O *session bean* deve ser considerado como uma extensão lógica, ciclo de vida pequeno para o cliente - o objeto desaparece assim que o cliente ou o servidor fecha. Já e*ntity beans* são objetos persistentes, duradouros que podem ser compartilhados por múltiplos clientes, talvez durante anos (semelhante a dados em um banco de dados);

b) para usar *session beans*, escolhe-se entre *stateless* e *stateful* que definem quanta carga (*load*) colocar no cliente ou no servidor. Em outras palavras, para manter estado no servidor (quer dizer, um *stateful session bean*), o servidor será taxado, já um *stateless bean*, minimiza a carga do servidor em conseqüência deixa para o cliente gerenciar o estado.

A figura 8 retrata um servidor EJB hipotético.

FIGURA 8 INTERAÇÃO ENTRE CLIENTES, *BEANS* E CAMADAS



Fonte: The state of java middleware, part 2: enterprise javabeans, 1999.

Ao observar a figura 8 pode-se analisar o seguinte:

a) um servidor EJB pode ter *containers* múltiplos;
b) um *container* pode ter *beans* múltiplos de tipos diferentes;
c) um *entity beans* mapea uma linha em um banco de dados;
d) um cliente pode usar qualquer tipo de *enterprise bean*;
e) são amarrados *stateful session beans* a um cliente específico;
f) não são amarrados *stateless session beans* a um cliente específico; eles são ao invés usados em um agrupamento se necessário;
g) *entity beans p*odem ser compartilhados por clientes e ter acesso concorrente.

3.6.3 PERSISTÊNCIA

Um das características de EJB é persistência embutida, que é obtida através de *entity beans*. A persistência ou pode ser gerenciada pelo *bean* ou pelo *container*.

Para persistência gerenciada pelo *bean*, o desenvolvedor reescreve (*overrides*) os métodos `ejbLoad()` (dentro deste método o desenvolvedor provê o código que extrairá os dados apropriados da base de dados para o *bean*) e `ejbStore()` (quando o *container* deseja salvar (*commit*) a transação ele primeiro invoca este método, que é responsável por escrever os dados de volta na base de dados) e com o código necessário (por exemplo, usando JDBC) para transferir os dados dos campos do objeto para seu armazenamento permanente e vice-versa. Enquanto gerenciamento de persistência pelo *bean* provê muita flexibilidade, amarra o *entity bean* a um método de persistência, como JDBC exige para o desenvolvedor reescrever e manter o código de persistência. Em alguns casos, porém, o desenvolvedor pode ter que empregar uma solução de gerenciamento pelo *bean* (por exemplo, se um EJB *container* provedor permite que um *entity bean* referencie uma única tabela mas o *entity bean* precisa de dados de múltiplas tabelas). Muitos vendedores de servidores de aplicação estão começando a implementar a abordagem objeto-relacional (OR) permitindo que ferramentas possam administrar complexos *joins* e outras relações de entidade e ainda proporcionam os benefícios de gerenciamento de persistência no *container*. De forma alternativa, pode-se usar uma visão do banco de dados relacional para representar tabelas múltiplas e suas relações como uma tabela. O problema é que muitos bancos de dados não suportam atualização de dados em tabelas múltiplas por visões [HEM99].

No gerenciamento pelo *container*, o provedor do *container* permite associar os campos do *entity bean* com os campos dos dados armazenados (como as colunas de uma tabela de banco de dados relacional).

Enquanto o processo de mapeamento pode ser tedioso, as vantagens de usar gerenciamento de persistência pelo *container* excedem suas desvantagens secundárias. Por exemplo, o provedor de *container* gera o código automaticamente para mover os dados entre os campos do *entity bean* e os dados armazenados para aplicações de banco de dados relacional, isto é um mapeamento objeto-relacional automático. Adicionalmente, o *entity bean* simplesmente pode ser persistido em uma fonte de dados diferente usando um *container* de EJB diferente e remapeando os campos [HEM99], [MON99] e [JUB99].

3.6.4 GERENCIAMENTO DE TRANSAÇÕES

Para transações, nenhuma distinção é feita entre um *session bean* e um *entity bean* com exceção de uma diferença sutil que permite só *stateful session beans* suspender uma transação gerenciada pelo usuário (usando o atributo de transação `TX_BEAN_MANAGED`). Adicionalmente, o desenvolvedor não é exposto à complexidade de transações que poderiam ser distribuídas potencialmente por fontes de dados múltiplos em plataformas múltiplas - esta responsabilidade fica a cargo do *middleware* de Java (ou servidor de aplicação).

Demarcação de transação em um ambiente EJB pode ser de dois modos. Primeiro, o desenvolvedor pode programar o estado da transação usando a interface `javax.jts.UserTransaction` ao cliente ou servidor. Segundo, o provedor do servidor de aplicação administra os limites da transação que os desenvolvedores usam para mudar os atributos de transação, que pode aplicar ao *enterprise bean* inteiro ou seus métodos específicos.

A tabela 4 mostra uma lista de valores válidos para a atributos de transação EJB.

TABELA 4 VALORES DE ATRIBUTOS VÁLIDOS PARA TRANSAÇÕES EJB

Valor de atributo	Como o *enterprise bean* é invocado
TX_NOT_SUPPORTED	Invocado sem um estado de transação
TX_BEAN_MANAGED	Um *enterprise bean* pode usar a interface `javax.jts.UserTransaction` para demarcar limites de transação. A interface que usa o método pode ser obtida. `UserTransaction javax.ejb.SessionContext.getUserTransaction()`. Uma instância de um *stateless session bean* ou um *entity bean* não é permitida reter uma associação com uma transação por múltiplas chamadas de um cliente.
TX_REQUERED	Se o cliente é associado com um contexto de transação, o *enterprise bean* é invocado no mesmo contexto, caso contrário o *container* começa uma transação nova antes de invocar um método do *enterprise bean* e comete a transação ao voltar do método.
TX_SUPPORTS	Invoca no estado da transação do cliente, se tem um. Caso contrário, invoca sem um contexto de transação.
TX_REQUIRES_NEW	Sempre é invocado em uma transação nova.
TX_MANDATORY	Sempre invocado no estado da transação do cliente. Se o cliente não tem um, é lançado (*thrown*) ao cliente uma exceção `javax.transaction.TransactionRequiredException`

Segundo [MON99], [JUB99] e [HEM99] EJB também suporta isolamento de transação por níveis, um conceito que permite que as mudanças feitas por uma transação sejam visíveis a outras transações. Os valores válidos são os seguintes:

a) TRANSACTION_READ_UNCOMMITTED : a transação pode ler dados que foram modificados por uma diferente transação que ainda está em processo;
b) TRANSACTION_READ_COMMITTED : a transação não lê dados não salvos; não podem ser lidos dados que estão sendo modificados por uma transação diferente;

c) TRANSACTION_REPEATABLE_READ : a transação não pode mudar dados que estão sendo lidos por uma transação diferente;
d) TRANSACTION_SERIALIZABLE : a transação tem leitura exclusiva e privilégios para alterar os dados; transações diferentes não podem nem ler nem escrever os mesmos dados.

Para *session beans* e *entity beans* com gerenciamento de persistência pelo *bean*, o nível de isolamento especificado é fixo na conexão com o banco de dados. Para *entity beans* gerenciados pelo *container*, o fabricante tem que permitir esta funcionalidade.

3.6.5 MANIPULAÇÃO DE EXCEÇÃO

As especificações de EJB 1.0 definem dois tipos de exceções: a nível de aplicação e a nível de sistema. Considerando que um cliente tem acesso aos métodos de um *enterprise bean* por suas *home* e *remote* interfaces, todos os métodos definidos nestas interfaces têm que incluir um `java.rmi.RemoteException`. Quando uma exceção remota é lançada, um cliente pode assumir que é uma falha a nível de sistema. Todas as outras exceções, inclusive `javax.ejb.CreateException`, `javax.ejb.RemoveException` e `javax.ejb.FindException`, são consideradas falhas a nível de aplicação [HEM99], [MON99] e [JUB99].

Em relação a transações, um cliente pode assumir que uma transação executou um *rollback* se for lançado uma exceção do tipo `javax.jts.TransactionRolledbackException` (Nota: O método `javax.ejb.SessionContext.getRollbackOnly()` também pode ser chamado para testar se uma transação que usou o método de *rollback* foi ou não executada.).

3.6.6 SEGURANÇA

EJB usa a classe `java.security.Identity` para descrever papéis (*roles*) ou usuários. O servidor de aplicação (ou *container* de EJB) de fato executa o mapeamento, normalmente de uma plataforma específica e provê esta informação para a instância do *enterprise bean* pelos métodos `getCallerIdentity` e `isCallerInRole` na classe `javax.ejb.EJBContext` [HEM99], [MON99] e [SES99].

EJB *security* usa listas de controle de acesso (ACLs) no *deployment descriptor* para gerenciar a segurança em nome do *enterprise bean*. EJB também provê atributos `RunAsMode` e

`RunAsIdentity` para definir a identidade de segurança a ser associada com a execução de métodos no *enterprise bean* e para qualquer chamada para outros gerenciadores de recurso (como JDBC). Os valores válidos para estes atributos são SPECIFIED_IDENTITY, CLIENT_IDENTITY e SYSTEM_IDENTITY (conta privilegiada).

3.6.7 JNDI

EJB usa o *Java Naming and Directory Interface* (JNDI) para nomear serviços. Em outras palavras, um cliente observa uma *home* interface de EJB que usa JNDI. É responsabilidade do *container* fazer com que as classes do *enterprise bean* estejam disponíveis para o cliente por JNDI.

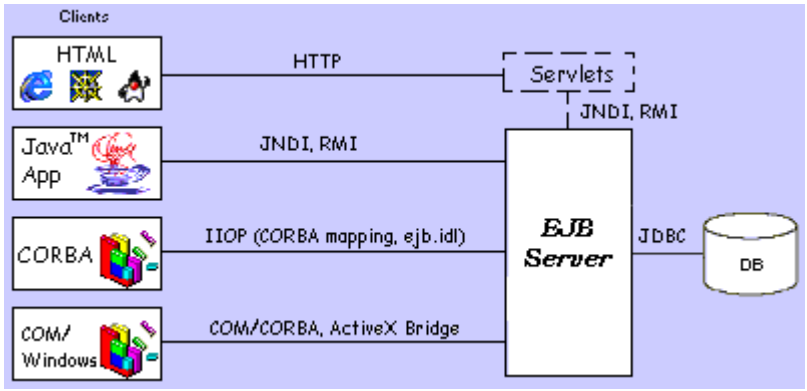
3.6.8 PROTOCOLOS

As especificações de EJB definem o protocolo Java RMI (JRMP) como o protocolo *default* para acessar *enterprise beans* em rede. Adicionalmente, as especificações provêem CORBA/IIOP que mapeam como os clientes de CORBA podem invocar *enterprise beans*. Porém, EJB não é limitado usando só os protocolos JRMP e IIOP. De fato, EJB pode usar qualquer protocolo (como HTTP ou DCOM) para apoiar uma variedade de tipos de cliente [HEM99].

A figura 9 mostra os vários modos como os *enterprise beans* podem ser acessados onde são descritos alguns padrões de arquitetura:

a) uma página HTML pode invocar um *servlet* que em então, invoca o *enterprise bean*;
b) um *Java applet* ou aplicação *standalone* podem invocar o *enterprise bean* diretamente;
c) uma aplicação *Visual Basic* pode usar CORBA ou o ActiveX para invocar o *enterprise bean*.

FIGURA 9 PADRÕES DE ARQUITETURA EJB BASEADA EM PROTOCOLOS CORRESPONDENTES



Fonte: The state of java middleware, part 2: enterprise javabeans, 1999.

Adicionalmente, o *middleware* de Java poderia proporcionar apoio para outros protocolos, como DCOM ou HTTP, para invocar o *enterprise bean*.

3.6.9 DESENVOLVIMENTO BASEADO EM ATRIBUTOS

Segundo [HEM99], EJB não só aumenta a produtividade de um desenvolvedor provendo uma arquitetura simples de componente do lado do servidor, mas também introduzindo programação declarativa, ou baseada em atributo. Por exemplo, atributos para transações, isolamento de níveis, segurança e mapeamento da fontes de dados de EJB *container* podem ser manipulados em tempo de desenvolvimento. Consequentemente, o desenvolvedor não tem que se preocupar com detalhes de baixo nível quando desenvolve componentes; ele pode se concentrar completamente na lógica empresarial.

3.6.10 DESENVOLVIMENTO

Segundo [HEM99], [SES99],[MON99] e [JUB99], são desdobrados *enterprise beans* em um arquivo JAR que provê um modo agradável e limpo para incluir os seguintes requerimentos e arquivos opcionais:

a) os arquivos com as classes *enterprise bean* (*home interface*, *remote interface* e *enterprise bean* atual) ;

b) um *deployment descriptor* para cada *enterprise bean*. Um *deployment descriptor* é uma instância serializada de um *entity bean* ou de um *session bean;*
c) um arquivo de manifesto que lista o nome do *deployment descriptor* e também indica quais são os *enterprise beans*;
d) propriedades de ambiente para o *enterprise bean.*

3.6.11 PAPÉIS (*ROLES*) E RESPONSABILIDADES

A especificação de EJB 1.0 define seis papéis diferentes envolvidos no desenvolvimento de EJB, conforme a tabela 5. Estes papéis variam baseados no tamanho da empresa ou projeto. Por exemplo, em um projeto pequeno, a mesma pessoa poderia ser o provedor de *enterprise bean*, montador de aplicação, desenvolvedor e administrador de sistema. Semelhantemente, um vendedor poderia ser o servidor de EJB e o provedor de *container* (por exemplo, os construtores de informação incluem vários *containers* de EJB com seu servidor de aplicação que atenda a EJB).

TABELA 5 PAPÉIS E RESPONSABILIDADES ENVOLVIDOS NO PROJETO E DESENVOLVIMENTO DE EJB

PAPEL	RESPONSABILIDADE
Provedor de *enterprise bean*	• classes e interfaces; • propriedades de ambiente; • *deployment descriptor*; • arquivo de manifesto; • EJB *packaging* (arquivo JAR)
Montador de Aplicação	• reúne todas as camadas (o cliente, GUI, servlets, e assim por diante)
Deployer	• ferramentas de *container* de usos para mapear *enterprise bean* para *container* (se preciso); • modifica os atributos de segurança (se preciso);

	• as propriedades de *enterprise bean* (se preciso)
EJB servidor provedor	• pode ser *middleware* ou um banco de dados; • provê *container* para *session beans;* • possivelmente provê *containers* para *entity beans* ou publica suas interfaces de baixo nível para permitir *plug-in* em *containers*
EJB *container* provedor	• responsável pela persistência de *entity beans*; • ferramentas para mapear *entity beans* (por exemplo, objeto para relacional); • gera código para mover dados de instância variáveis de *entity beans* para armazenamento secundário; • provê ferramentas para administrar *container* e *beans* que executam naquele *container*
Administrador de sistema	• sistema de monitores

3.7 *VOYAGER* (SERVIDOR DE APLICAÇÃO)

Voyager é o *container* e servidor de aplicação. Através dele é criado o arquivo .JAR, o arquivo de manifesto e o *deployment descriptor*. São definidas as classes e suas transações, bem como a segurança de acesso ao banco de dados.

Através do EJB *Studio*, uma ferramenta auxiliar do *voyager*, são inseridas as classes, a *home* e a *remote interface*. Define-se o tipo do *bean* (*entity bean*, *stateless session bean* ou *stateful session bean*) e atribui-se as transações aos *beans* inseridos.

Todas as configurações feitas no *voyager* são explicadas e exemplificados no item 4.5.

4 IMPLEMENTAÇÃO

Para testar o desenvolvimento e a aplicação da arquitetura multicamadas, será desenvolvido uma aplicação do sistema de Identificação Pessoal da FURB - a solicitação por parte do aluno, da emissão da segunda via do cartão de identificação única.

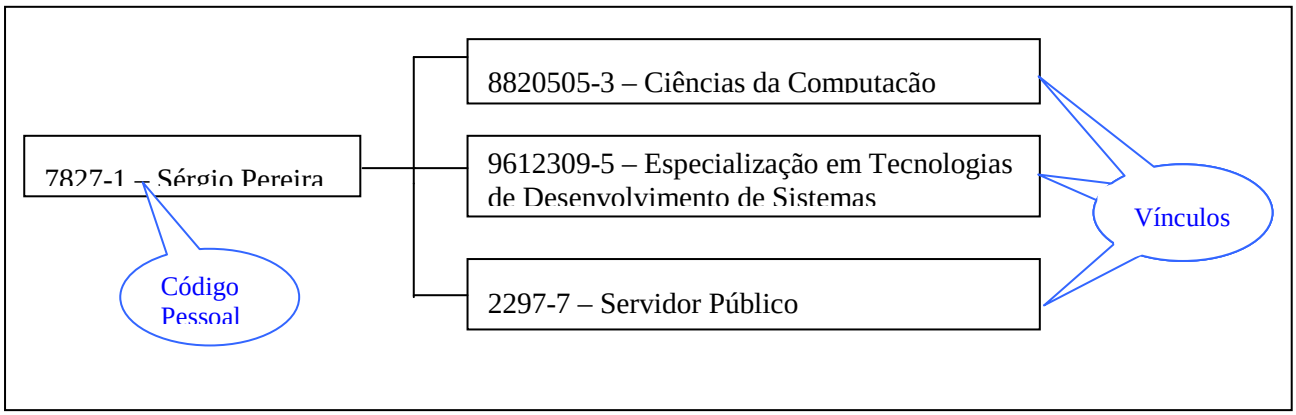
4.1 CENÁRIO

A identificação pessoal trata-se da identificação centralizada de todas as pessoas ligadas à Universidade Regional de Blumenau (FURB), através de um código pessoal. Este código é fornecido pela FURB no momento da primeira matrícula do aluno, em qualquer curso.

O objetivo é unificar todas as informações pessoais (figura 10) e facilitar o acesso a serviços (biblioteca, consulta de informações acadêmicas, etc.), através do código pessoal e da senha (que deve ser cadastrada na Biblioteca Central). Para facilitar o acesso a estes serviços, a FURB entrega um cartão de identificação a todos os alunos, onde consta no seu verso o código pessoal.

Além do código pessoal, os alunos receberão um outro código, denominado vínculo. Cada vínculo identifica o ingresso do aluno em um determinado curso da FURB (ETEVI, Laboratório de Línguas, Cursos de Extensão, Graduação, Pós-Graduação). Assim um código pessoal poderá possuir diversos vínculos.

FIGURA 10 EXEMPLO DO CÓDIGO PESSOAL UTILIZADO NA FURB



Com exceção do servidor, qualquer pessoa que necessitar solicitar nova via do cartão de identificação deverá dirigir-se aos quiosques localizados nos campi da FURB, informar seu código pessoal e senha. Após o prazo de 3 (três) dias úteis a pessoa poderá retirar a nova via na Biblioteca Central. A cobrança será efetuada na próxima fatura em um dos vínculos ativos.

4.2 METODOLOGIA DEFINIDA PELO NI PARA A IMPLEMENTAÇÃO DE MULTICAMADAS

Para a implementação de *Enterprise JavaBeans*, o Núcleo de Informática da FURB onde será desenvolvida a aplicação, definiu uma nova metodologia para o desenvolvimento de aplicações distribuídas e orientadas a objetos.

A definição desta metodologia foi baseada nas sugestões de [COA99] referentes a especificação e implementação em UML e [OBJ99] sobre a implementação de *Enterprise JavaBeans* e a configuração do servidor de aplicação.

Estão a seguir os passos para o desenvolvimento de uma aplicação distribuída e orientada a objetos:

a) especificação:
   - definição do diagrama de casos de uso;
   - definição do diagrama Entidade Relacionamento - modelo físico (tabelas da aplicação);
   - definição dos diagramas de classes;
   - definição dos diagramas de seqüência;

b) implementação:
   - criação de classes auxiliares: as classes auxiliares são responsáveis pelo acesso ao servidor de aplicação (ou *container*), pela implementação da interface *EntityBean* ou *Session Bean* e pelo acesso ao banco de dados;
   - definição das interfaces (*home* e *remote*);
   - criação das classes *bean* (*entity* e *session beans*): são as classes *beans* que irão implementar os métodos definidos nas interfaces;
   - criação das classes e seus atributos;

c) configuração do servidor de aplicação;

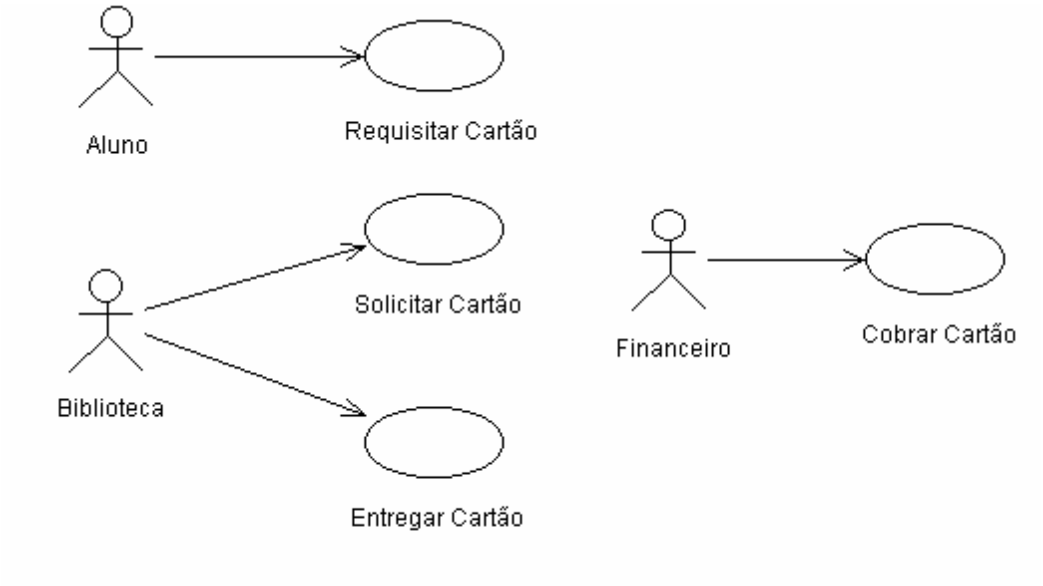
d) implementação dos clientes, que acessam as interfaces.

Desta forma, na primeira camada é executado o cliente, na segunda, as interfaces, classes beans e classes auxiliares, na terceira camada há o banco de dados. O cliente só acessa as interfaces, ele não possui nenhum acesso direto a base de dados, nem às classes *beans*.

4.3 ESPECIFICAÇÃO DO PROTÓTIPO

Após o estudo da aplicação foi desenvolvido o diagrama de casos de uso (figura 11), que exemplifica o cenário da aplicação, ou seja, o aluno irá solicitar uma nova via do cartão de identificação pessoal nos quiosques. A Biblioteca Central deverá solicitar a confecção do cartão e efetuar a entrega no terceiro dia útil. O Setor Financeiro irá cobrar esta nova via na próxima fatura do aluno em um de seus vínculos ativos.

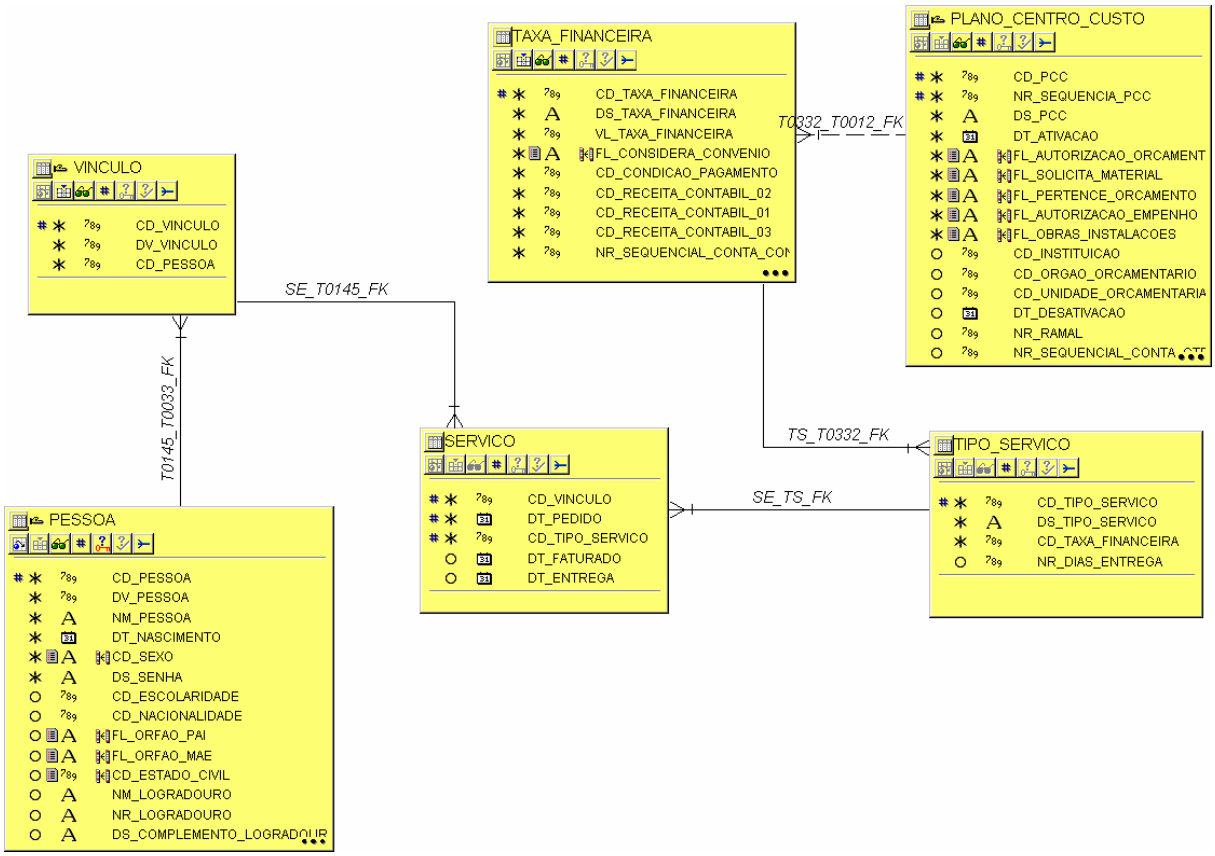
FIGURA 11 DIAGRAMA DE CASOS DE USO



Depois foi especificado o Diagrama Entidade Relacionamento - modelo físico (figura 12), ou seja, as tabelas da aplicação, que armazenarão as informações no banco de dados ORACLE 8.0.5, que já é utilizado pela FURB.

Para a especificação foi utilizado a ferramenta Designer/2000, seguindo os padrões de modelagem desta ferramenta e a metodologia do Núcleo de Informática para a especificação estruturada.

FIGURA 12 DIAGRAMA ENTIDADE RELACIONAMENTO - MODELO FÍSICO



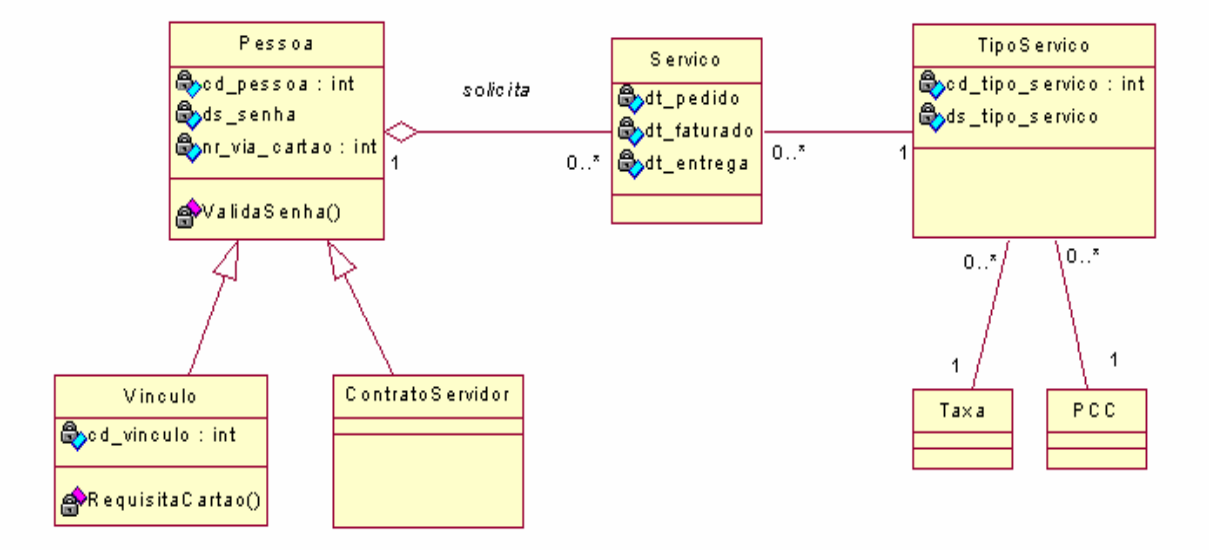
Detalhando o diagrama Entidade Relacionamento temos: na FURB uma PESSOA pode ter vários VÍNCULOS (servidor, graduação, pós-graduação, etc), estes VÍNCULOS terão SERVIÇOS associados a eles (solicitação da segunda via do cartão de identificação única, solicitação de emissão histórico escolar, etc). Estes SERVIÇOS serão cobrados através de uma TAXA FINANCEIRA, cadastrada no Setor Financeiro da FURB. As despesas e receitas oriundas destas taxas destinar-se-ão a um CENTRO DE CUSTO (PCC).

No modelo estruturado não foi representada a tabela contrato_servidor.

O próximo passo foi a especificação do modelo orientado a objetos (figura 13) seguindo a metodologia proposta por [COA99] e aplicada pelo Núcleo de Informática.

As classes ContratoServidor, Taxa, PCC e TipoServico (figura 13) apenas exemplificam o cenário, mas não serão implementadas no protótipo.

FIGURA 13 DIAGRAMA DE CLASSES



Detalhando o diagrama de classes temos: um VINCULO ou um CONTRATO SERVIDOR herdam todas as características da classe PESSOA. Dentro da FURB uma PESSOA pode ter vários vínculos, mas o código de pessoa é único. Cada PESSOA, graduando, pós-graduando, servidor (vinculos) pode solicitar um ou vários SERVICOS (solicitação da segunda via do cartão de identificação única, solicitação de emissão histórico escolar, etc). Estes SERVIÇOS serão cobrados através de uma TAXA FINANCEIRA, cadastrada no Setor Financeiro da FURB. As despesas e receitas oriundas destas taxas destinar-se-ão a um CENTRO DE CUSTO (PCC).

Após a definição dos diagramas de casos e de classes foi necessário a especificação dos diagramas de seqüência referente aos métodos ValidarSenha (figura 14) e RequisitarCartão (figura 15).

FIGURA 14 DIAGRAMA DE SEQÜÊNCIA DO MÉTODO VALIDARSENHA

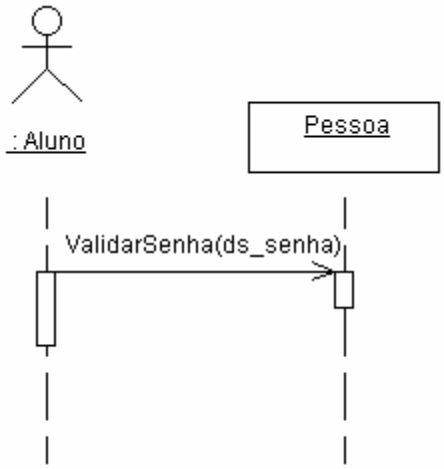
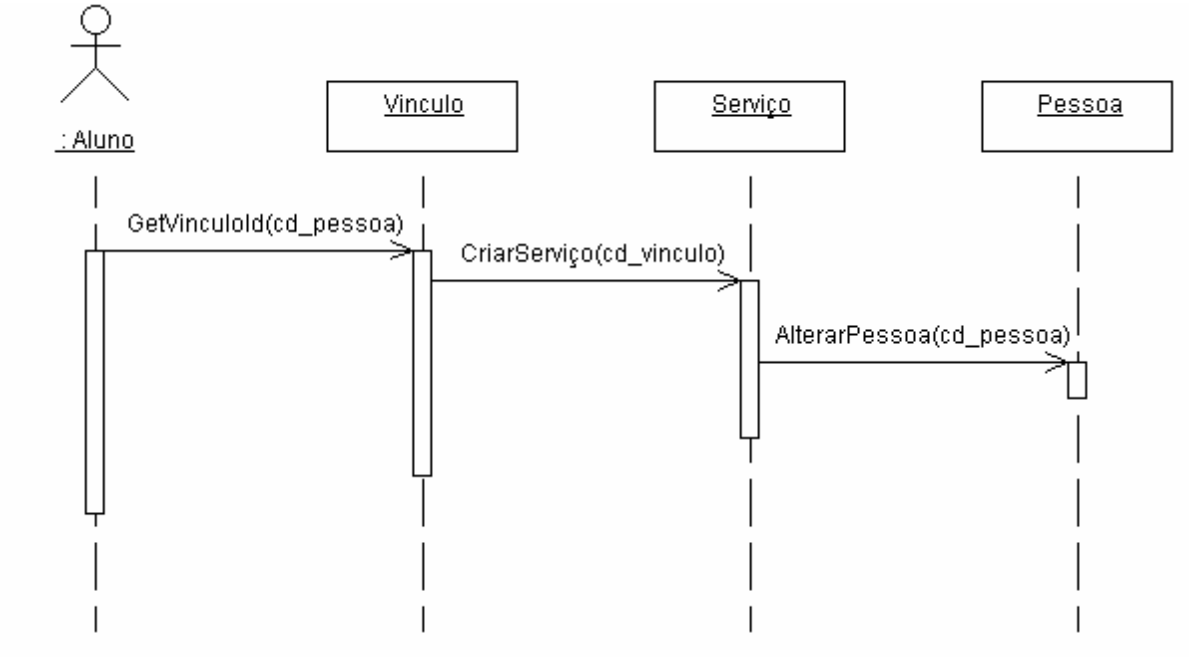


FIGURA 15 DIAGRAMA DE SEQÜÊNCIA DO MÉTODO REQUISITARCARTAO



4.4 IMPLEMENTAÇÃO

Para a implementação das classes é necessário que se conheça bem o funcionamento da aplicação.

4.4.1 CRIAÇÃO DAS CLASSES AUXILIARES

As classes auxiliares foram definidas a partir de sugestões de [OBJ99]. Desta forma, qualquer acesso tanto ao servidor de aplicação quanto ao banco de dados são de responsabilidade destas classes.

Foram definidas três classes auxiliares:

a) *AbstractBean*: responsável pelo acesso ao servidor de aplicação (ou *container*);

b) *AbstractEntity*: responsável pela implementação da interface *EntityBean*;

c) *DbHelper* (Anexo 7): responsável pelo acesso ao banco de dados.

4.4.2 DEFINIÇÃO DAS INTERFACES

Todas as classes *beans* devem ter duas interfaces, a *home interface* e a *remote interface*. Na *home interface* (Anexo 1 e Anexo 2) obrigatoriamente deve ser definido um método `create()` e um método `findByPrimaryKey()`, porém, outros métodos `findByxxx()` podem ser criados, por serem obrigatórios, na especificação do diagrama de classes, estes métodos não estão definidos.

A *remote interface* (Anexo 3 e Anexo 4) define os métodos de negócio

Todos os métodos definidos tanto na *home* quanto na *remote inetrface* deverãso ser implementados nas classes *beans*.

Se, por algum motivo, a aplicação tiver que ser modificada, altera-se as interfaces, modificando a chamada dos métodos, se for necessário, senão, apenas, modifica-se o código nas classes *beans*. Não há a necessidade de se modificar a classe cliente que, após definida, só será modificada por motivos de estética ou ergonomia. As regras de negócio ficam no servidor de aplicação, em classes pequenas e de fácil manutenção, pois todas as validações de tela (verificar se o usuário digitou certo, etc) encontram-se na classe cliente e não nas classes *beans.*

4.4.3 CRIAÇÃO DAS CLASSES *BEANS*

Na implementação das classes *beans* (Anexo 5 e Anexo 6), os métodos definidos na *home interface* passam a ser denominados de **`ejb`**`Create()`, **`ejbF`**`indByPrimaryKey()` e `ejbFindBxxx()` é desta maneira que o servidor de aplicação identifica os métodos definidos na *home interface* e que estão implementadas nas classes *beans.*

Já os métodos definidos na *remote interface* são implementados nas classes *beans* com o mesmo nome.

Se algum dos métodos definidos nas *inetrfaces* não foi implementado na classe *bean*, o servidor de aplicação informa que o método correspondente não foi implementado. Há dois pontos a considerar:

a) a linguagem java considera que dois métodos que possuem o mesmo nome são diferentes se o tipo ou a quantidade de parâmetros que eles recebem são diferentes.

Portanto, se um método foi definido na *home interface* ou na *remote interface* para receber um parâmetro inteiro e na classe *bean* este método não recebe parâmetros eles são considerados diferentes e o servidor não permite que o arquivo .jar seja criado;

b) uma outra exceção é levantada pelo servidor de aplicação se a classe *bean* não implementar para cada método `ejbCreate()`, um método `ejbPostCreate()` com os mesmos parâmetros.

4.4.4 CRIAÇÃO DAS CLASSES E SEUS ATRIBUTOS

As classes, como a classe Pessoa (que não é um *entity bean*), por exemplo, são acessadas apenas pelas classes *beans* e também podem implementar regras de negócio, desde que:

a) método esteja definido na *remote interface*;
b) a classe cliente acesse a *remote interface*;
c) a classe *bean* crie uma nova instância da classe propriamente dita e o método esteja definido também na classe *bean*;

Por exemplo, a classe Pessoa:

a) na *remote interface* (Anexo 3) foi definido o método ValidaSenha que recebe uma *String*, o código da pessoa não é passado como parâmetro porque ele é armazenado no atributo cd_pessoa da classe Pessoa no momento em que é executado o método `ejbCreate()` da classe PessoaBean (Anexo 5);
b) a classe cliente chama este método através da *remote interface* passando a senha digitada pelo usuário;
c) o servidor de aplicação localiza o método na classe PessoaBean (Anexo 5). Esta classe já possui uma nova instância da classe Pessoa e acessa o método implementado na classe Pessoa, como mostra a figura 16.

FIGURA 16 EXEMPLO DO MÉTODO VALIDARSENHA NA CLASSE PESSOABEAN

```
public boolean ValidaSenha(String ds_senha) throws RemoteException
    {
     return pessoa.ValidaSenha(ds_senha);
    }
```

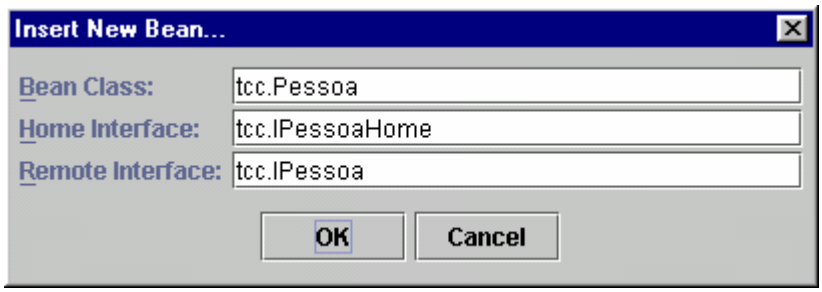
Na figura 16 pessoa é uma variável privada do tipo Pessoa. Quando o método na classe Pessoa terminar sua execução ele irá retornar o resultado para a classe PessoaBean que irá retornar para o cliente.

4.5 CONFIGURAÇÃO DO SERVIDOR DE APLICAÇÃO

Após a implementação de todas as classes é necessário a criação de um arquivo .jar. Este arquivo é gerado na ferramenta EJB *Studio*, ferramenta auxiliar do *voyager*.

Todos os *beans* devem ser inseridos neste arquivo, informando a classe *bean*, a *home interface* e a *remote interface*, como mostra a figura 17.

FIGURA 17 INSERINDO AS CLASSES NO SERVIDOR DE APLICAÇÃO

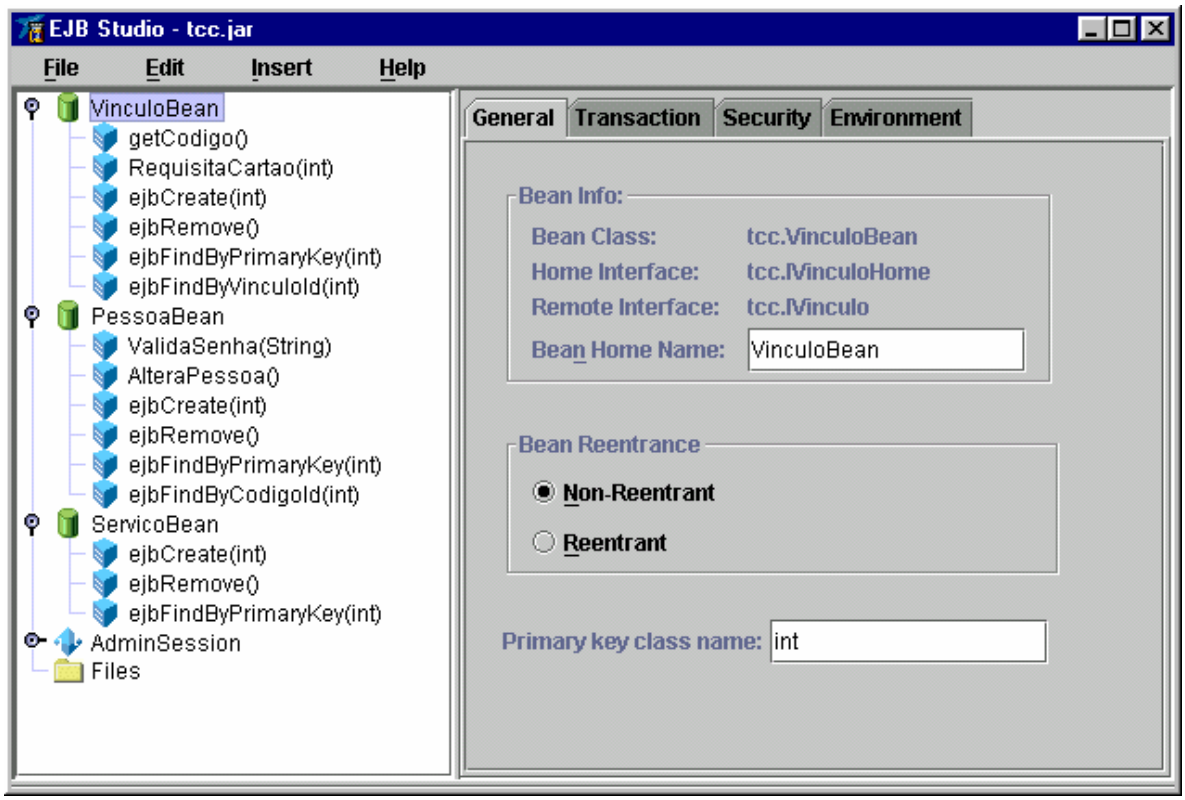


Obs.: tcc. especifica em que *package* (pacote) a classe PessoaBean está inserida.

O processo deve ser repetido para todas as classes da aplicação.

Automaticamente o servidor de aplicação (figura 18) identifica os métodos de cada classe. Se algum método definido em uma das duas interfaces não foi implementado na classe *bean* o EJB *Studio* lança uma exceção.

FIGURA 18 MÉTODOS IDENTICADOS PELO SERVIDOR DE APLICAÇÃO

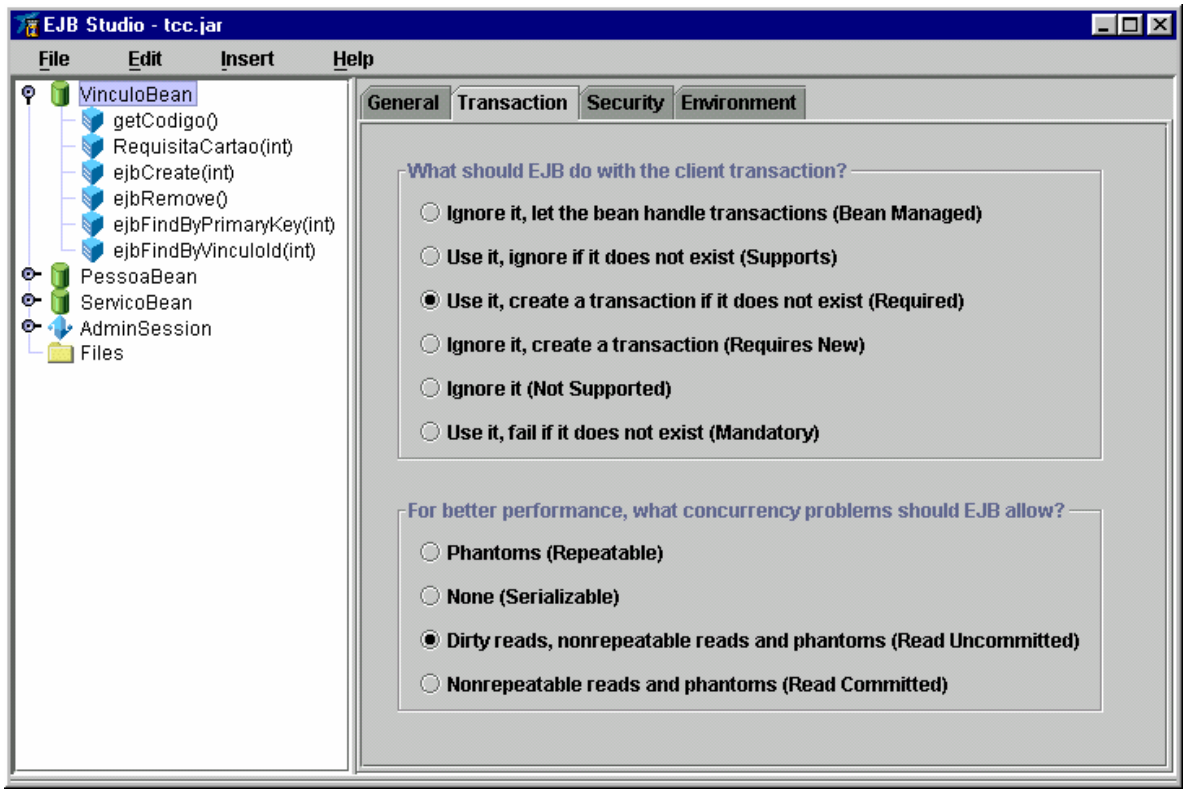


O próximo passo é configurar três itens em cada bean: *home name*, *transaction type* e *environment properties*.

O item *bean home name* deve conter o nome da classe *bean* (figura 18).

Todas as transações associadas aos métodos são requeridas (figura 19), ou seja, se a chamada do método tem uma transação associada a ela, esta transação é utilizada. Senão, o *container* cria e usa uma nova transação. No acesso a base de dados a transação foi definida como READ_UNCOMMITTED, ou seja, a transação pode ler dados que foram modificados por uma diferente transação que ainda está em processo.

FIGURA 19 DEFINIÇÃO DO TIPO DAS TRANSAÇÕES



Apesar de [OBJ99] sugerir a utilização da transação de acesso ao banco de dados como READ_COMMITTED, na hora da execução da aplicação é lançado uma exceção de acesso pelo servidor de aplicação informando que este não suporta este tipo de transação.

O último passo é definir a propriedade *environment properties* (figura 20). Esta propriedade irá definir o nome do acesso à base de dados. O método `getDbHelper()` (figura 21) pertencente à classe *AbstractBean* chama o método construtor da classe DbHelper, passando o valor desta propriedade. A classe DbHelper usa, então, o nome desta propriedade para recuperar o nome da fonte de dados através do JNDI.

FIGURA 20 DEFINIÇÃO DO NOME DO ACESSO AO BANCO DE DADOS

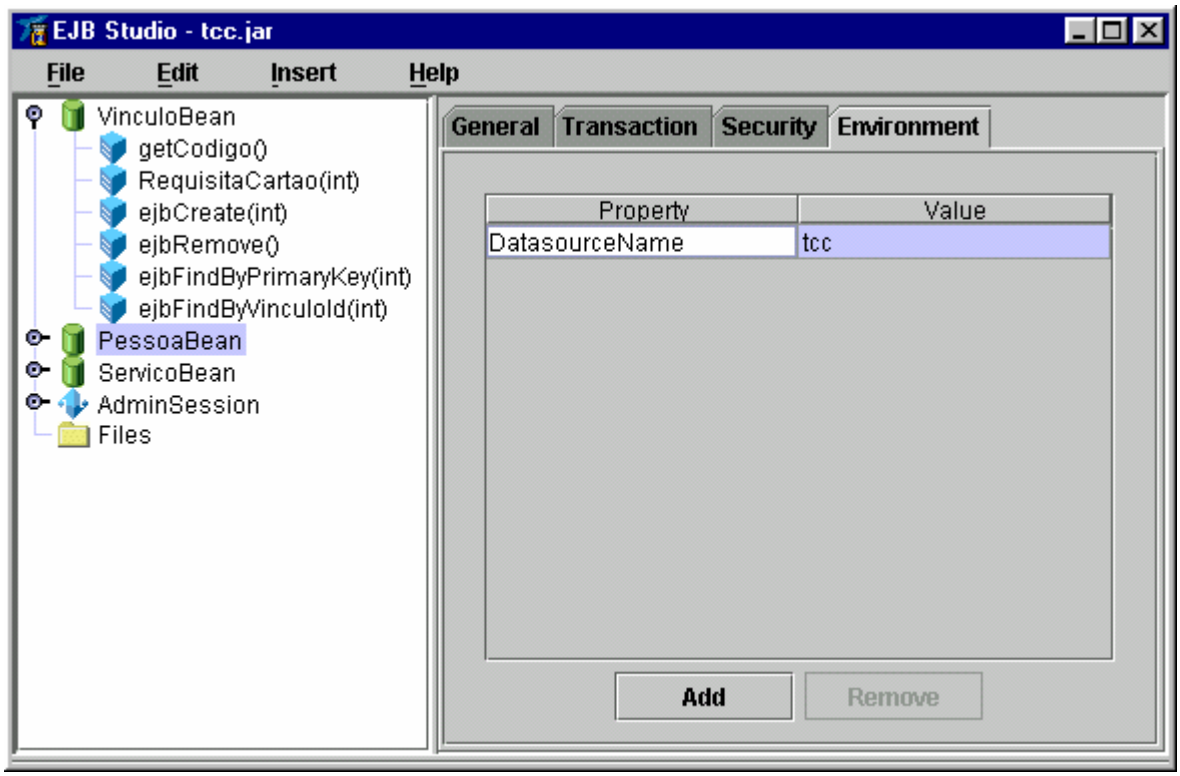
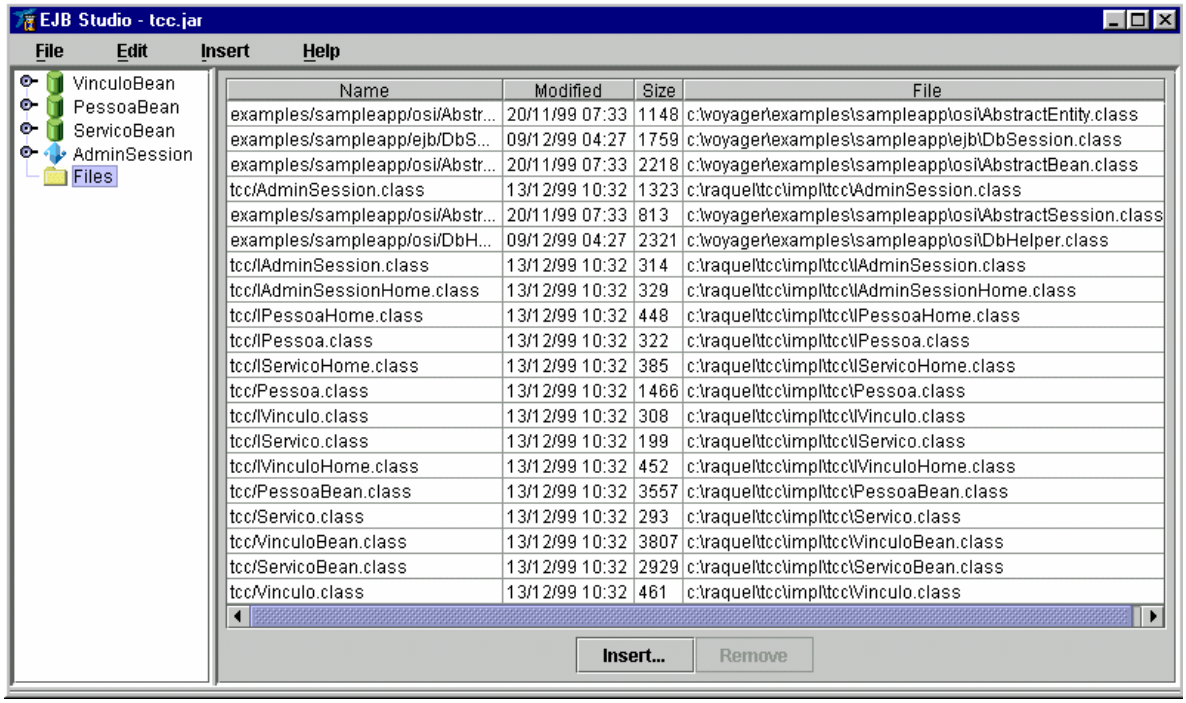


FIGURA 21 UTILIZAÇÃO DA PROPRIEDDADE *ENVIRONMENT*

```
protected DbHelper getDbHelper() throws RemoteException
    {
    try
      {
      if( dbHelper == null )
        dbHelper = new DbHelper( getEnvProperty( cDatasourceName ) );
      }
    catch( NamingException e )
      {
      throw new RemoteException( ( "Erro ao acessar datasource <" +
cDatasourceName + ">" ), e );
      }
      return dbHelper;
    }
```

Antes de salvar o arquivo .jar é necessário incluir as classes auxiliares (figura 22) *AbstractBean*, *AbstractEntity* e *DbHelper* e as demais classes, como a classe Pessoa, por exemplo, pois estas classes serão utilizadas na aplicação e os acessos a elas gerenciados pelo *container* (ou servidor de aplicação).

FIGURA 22 INSERÇÃO DAS DEMAIS CLASSES NO SERVIDOR DE APLICAÇÃO

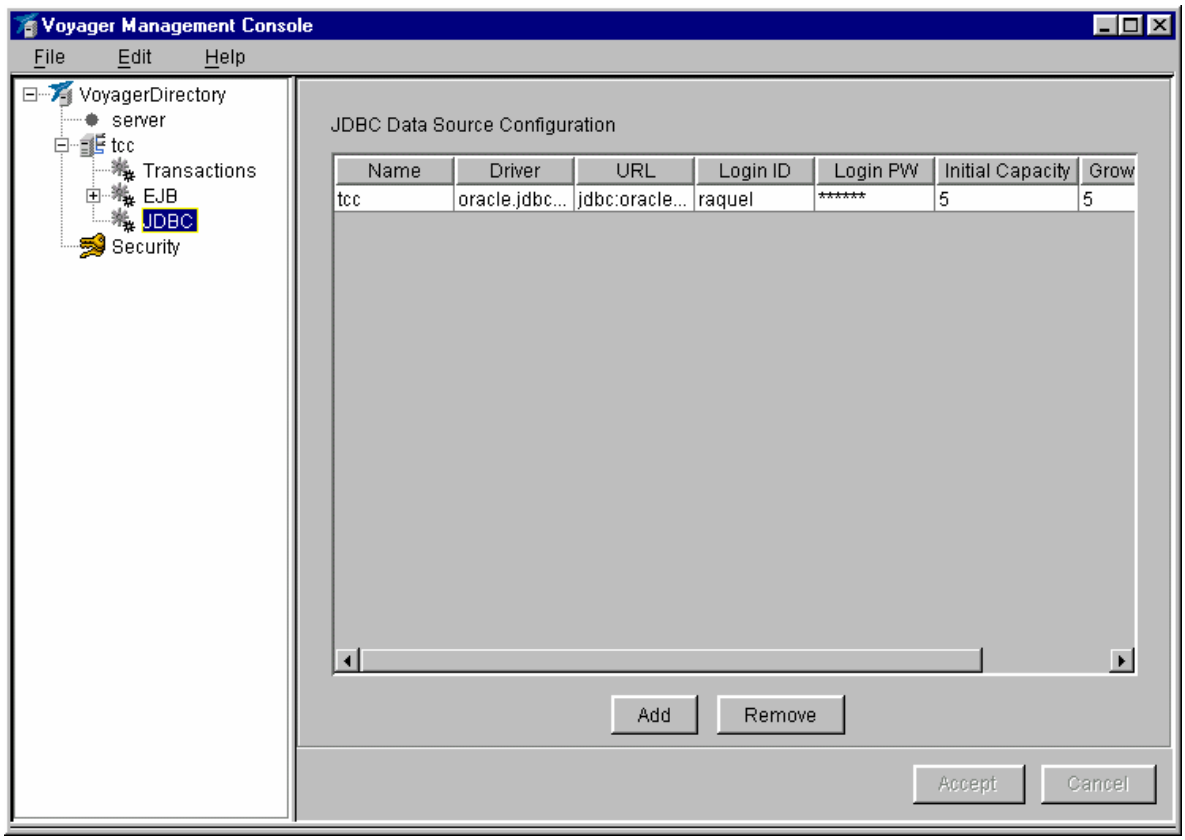


O próximo passo é a execução da console. Antes, é necessário iniciar um novo diretório servidor JNDI passando para o *voyager* os seguintes parâmetros: porta (8000), arquivo de armazenamento de diretório (jndi.db) e o servidor raiz (JNDI).

Logo após iniciar a execução do *voyager*, executa-se a console. Na console, define-se um novo perfil do servidor, neste caso, denominado de tcc (figura 23).

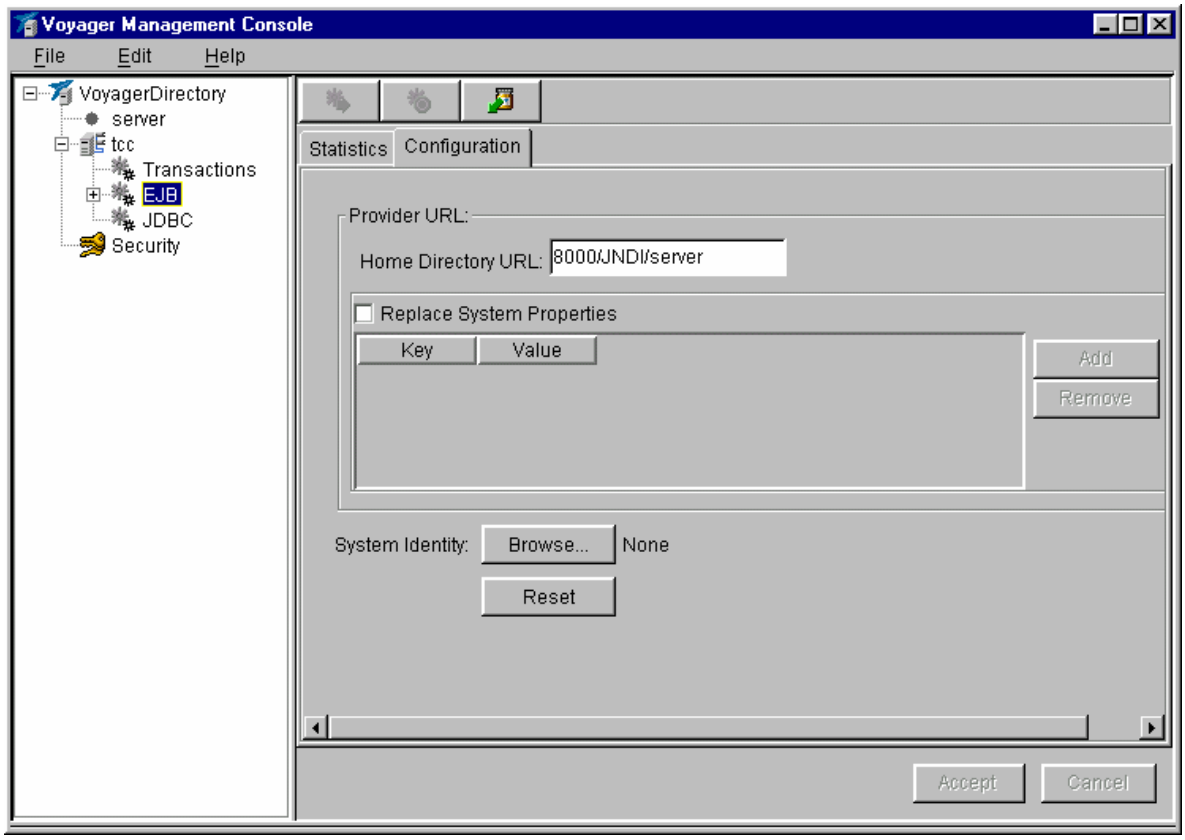
Para configurar o novo perfil de servidor (figura 23) inicia-se pela configuração do JDBC (acesso ao banco de dados). O nome deve ser igual àquele definido no *environment properties*: tcc (figura 20). Os demais valores dependem do *setup* de cada base de dados particular.

FIGURA 23 NOVO PERFIL DE SERVIDOR – CONFIGURAÇÃO DO JDBC



Depois da configuração do JDBC, deve-se configurar o EJB, setando *o Home Directory* URL como 8000/JNDI/server (figura 24). Logo após importa-se o arquivo .jar gerado anteriormente para o diretório JNDI especificado

FIGURA 24 NOVO PERFIL DE SERVIDOR – CONFIGURAÇÃO DO EJB



Após todas as configurações é necessário iniciar o servidor de aplicação *voyager*:

a) passando os mesmos parâmetros utilizados na execução da console, por exemplo, `voyager 8000 –r –f jndi.db JNDI`;

b) iniciar o *container* com a configuração URL (8000/JNDI/tcc). A configuração URL é composta de três itens:

- porta do diretório servidor (8000);
- diretório raiz (JNDI);
- nome do perfil do servidor conforme especificado na console (tcc);

c) executar o cliente.

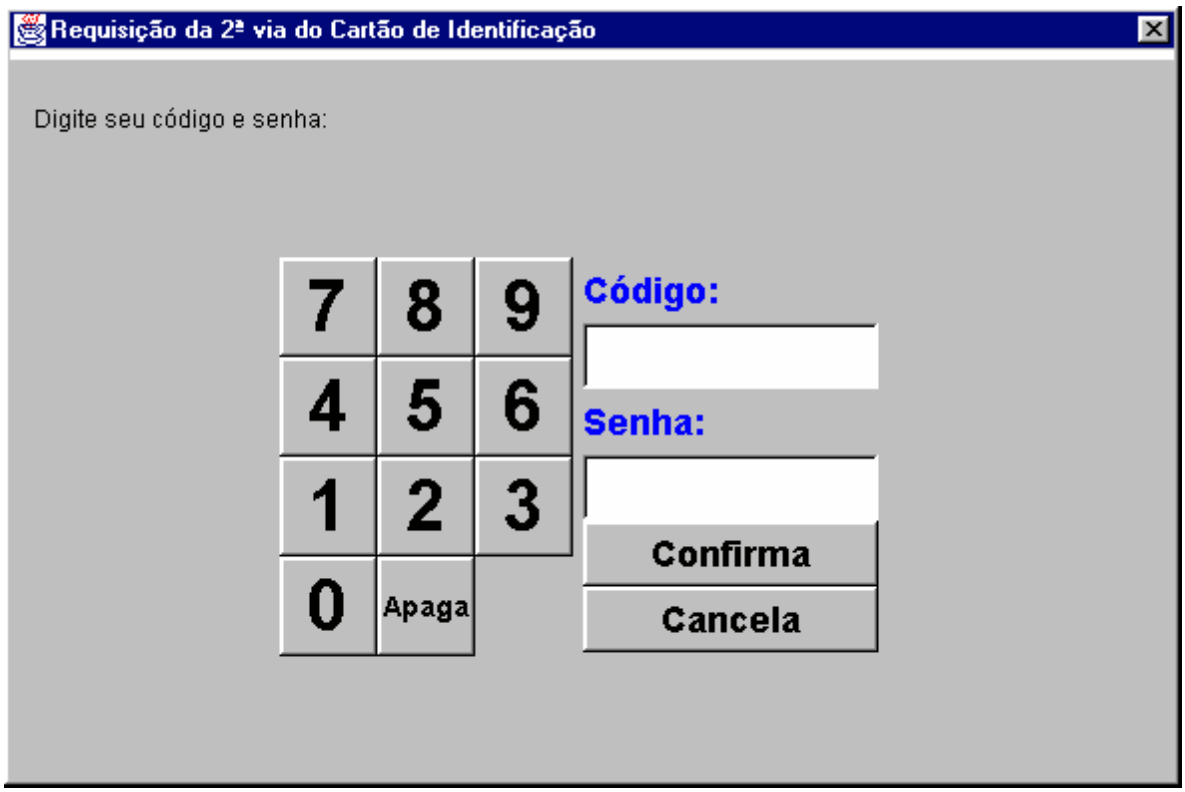
É importante enfatizar que o banco de dados e o servidor de aplicação devem estar ativos para que o cliente consiga executar.

4.6 TELAS DA APLICAÇÃO

A tela da aplicação (figura 25) apresenta como entradas o código de identificação pessoal e a senha do aluno solicitante da nova via do cartão de identificação.

Nesta tela é feito apenas uma validação: verifica-se se a senha digitada contém seis dígitos. As demais validações como código e senha corretos são feitas em classes que se encontram no servidor de aplicação.

FIGURA 25 TELA PRINCIPAL DA APLICAÇÃO



A segunda tela da aplicação (figura 26) mostra ao aluno informações necessárias para a confirmação dele na solicitação da segunda via da aplicação. Se ele discordar é mostrado a tela principal e a solicitação não é feita. Se ele aceitar, é mostrado uma terceira tela (figura 27) que apenas informa, após todos os procedimentos, que a solicitação foi efetuada.

FIGURA 26 TELA COM AS INFORMAÇÕES PARA O ALUNO

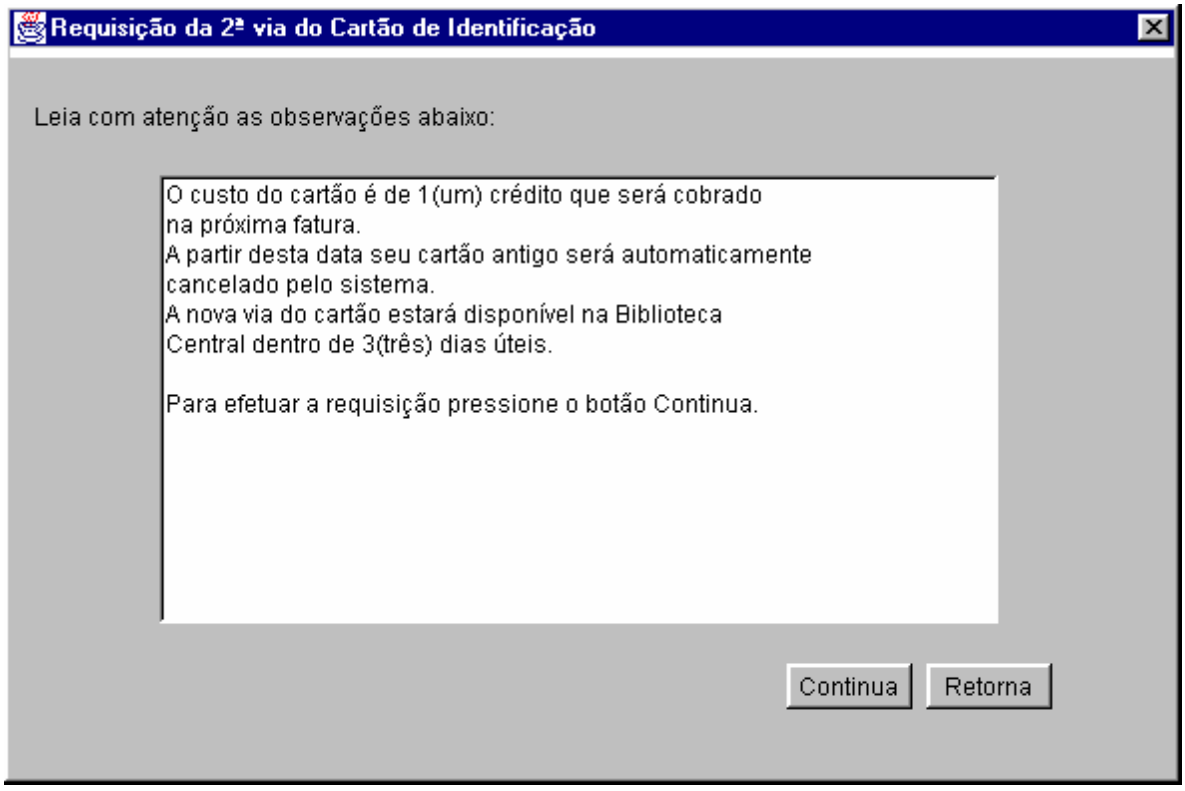
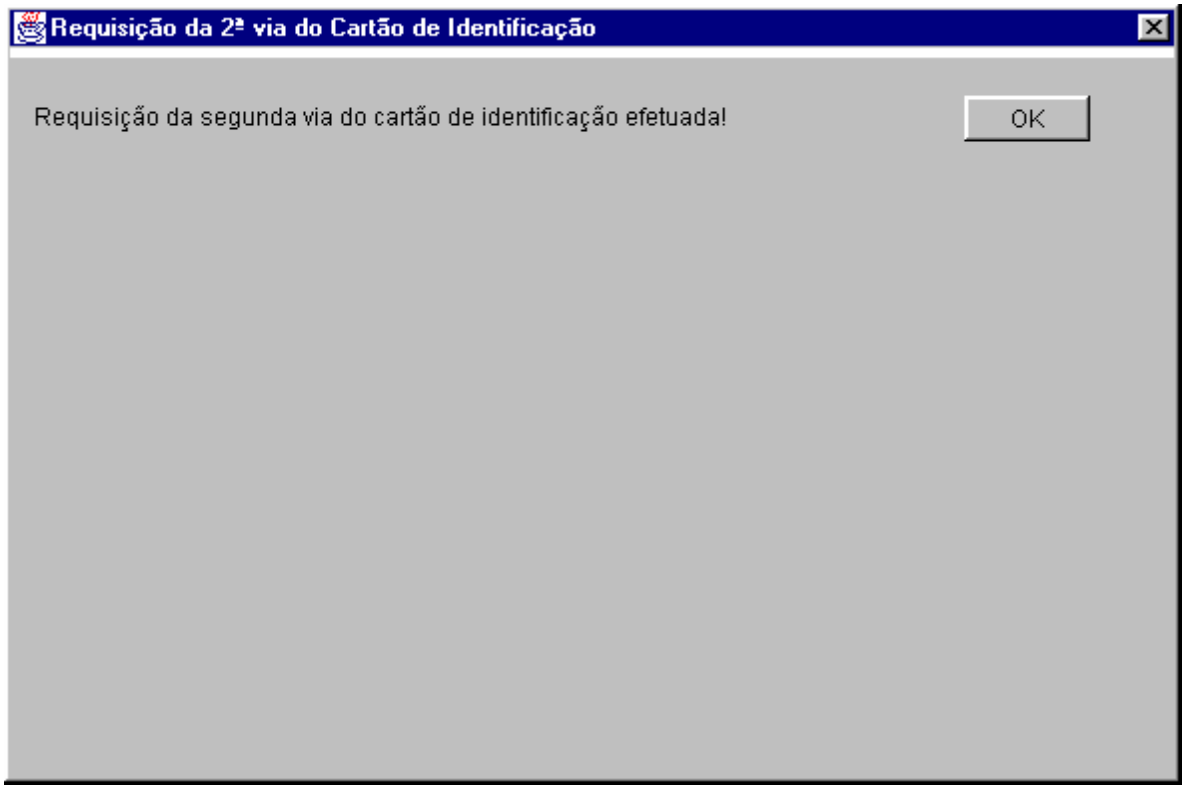


FIGURA 27 TELA CONFIRMANDO A REQUISIÇÃO DO CARTÃO



4.7 ESTRUTURA DA APLICAÇÃO

A aplicação está estruturada da seguinte forma (figura 28):

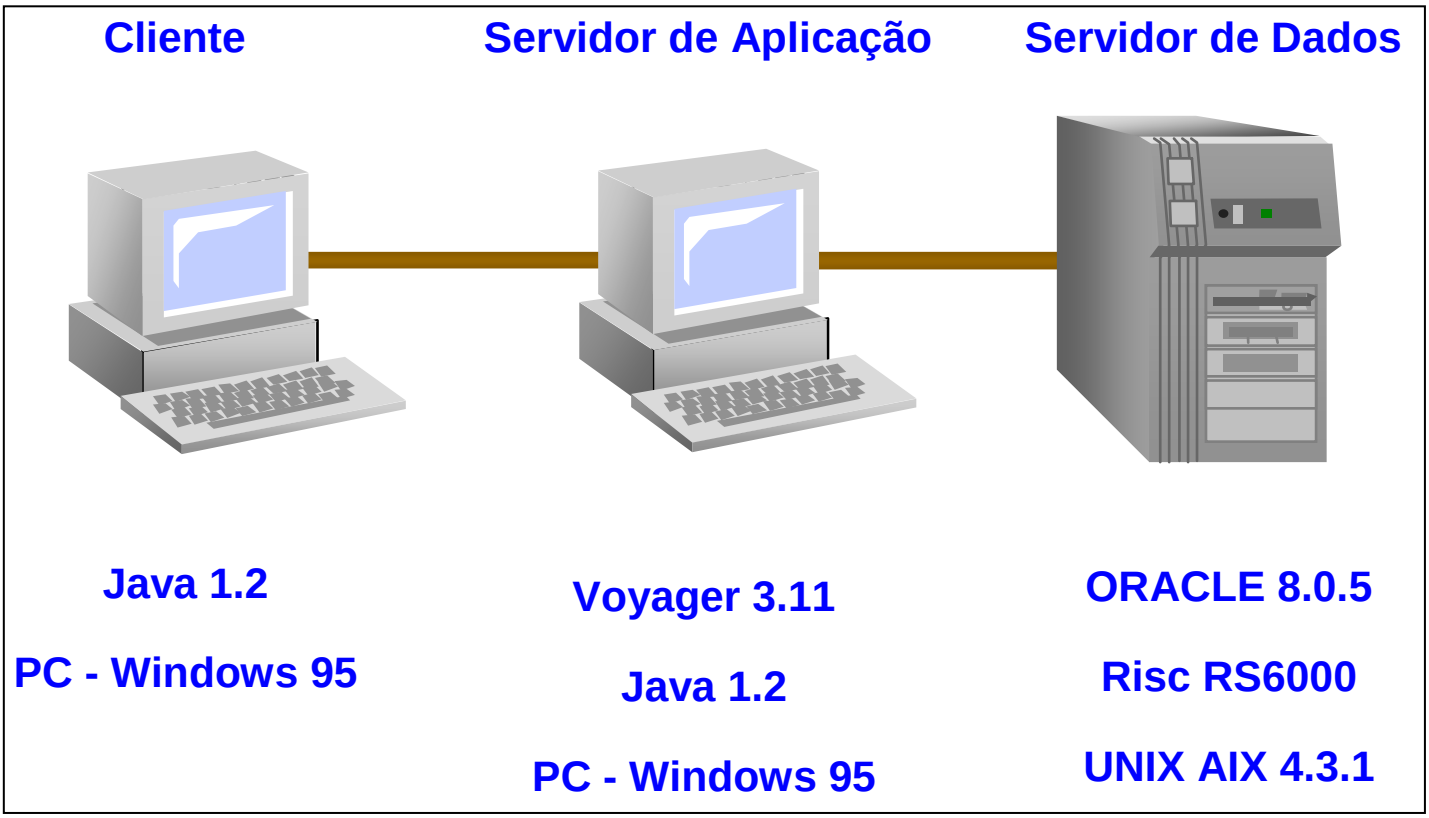


FIGURA 28 ESTRUTURA DA APLICAÇÃO

a) na camada de apresentação: há um computador PC – Windows 95 executando um *applet* java. Executam no cliente apenas duas classes: Numerico3 e cliente5. A classe Numerico3 implementa a interface para o cliente. Neste *applet* é feito uma única validação a nível de tela: verifica-se se o aluno digitou a senha com seis dígitos. A classe cliente inicia o servidor de aplicação *voyager* no cliente (figura 29) e inicializa as classes PessoaBean (Anexo 5) e VinculoBean (Anexo 6). As classes Numerico3 e cliente5 não são inseridas no servidor de aplicação pois não são gerenciadas por ele;

FIGURA 29 INICIALIZAÇÃO DO VOYAGER NO CLIENTE

```
public cliente5() {
  try { Voyager.startup();
     }
  catch( Exception exception ){
      exception.printStackTrace();}
  }
```

b) na camada da aplicação: também há um computador PC – Windows 95 executando o servidor de aplicação que acessa e gerencia todas as classes da aplicação;
c) na camada de dados: há um computador Risc RS6000 – UNIX AIX 4.3.1 executando o banco de dados ORACLE 8.0.3, onde estão as tabelas da aplicação que conterão as informações oriundas da aplicação.

4.8 EXEMPLO DA IMPLEMENTAÇÃO DO DIAGRAMA DE SEQÜÊNCIA REQUISITARCARTAO

Após a confirmação do aluno, é pesquisado o vínculo do aluno (figura 30) na classe Numerico3. Vinculo, na figura 30, é uma classe do tipo *remote interface* (Anexo 4). Neste momento, o *container* irá acessar a classe *bean* (Anexo 6) que implementa o método `getCodigo()`. Nenhum parâmetro é passado pois no momento da inicialização da classe VinculoBean foi criado uma nova instância da classe Vinculo e atribuído ao atributo cd_vinculo o código do vínculo do aluno que está requisitando o cartão .

FIGURA 30 PESQUISA DO VÍNCULO DO ALUNO

```
int cd_vinculo = vinculo.getCodigo();
```

Após a busca do código do vínculo do aluno, é executado o método `RequisitaCartao()` passando como parâmetro o vínculo encontrado. Na classe Numerico3 o código é o seguinte `vinculo.RequisitaCartao(cd_vinculo);`. Na classe VinculoBean o método é implementado conforme a figura 31.

FIGURA 31 IMPLEMENTAÇÃO DO MÉTODO REQUISITACARTAO

```
public void RequisitaCartao(int cd_vinculo) throws RemoteException
   {
    try
      {
/* Identifica o diretório JNDI servidor */
      String jndiDirectoryUrl = "8000/JNDI/server";
      Hashtable env = (Hashtable) System.getProperties().clone();
/* Retorna o nome do JNDI (tcc) */
      env.put( Context.PROVIDER_URL, jndiDirectoryUrl );
      InitialContext ctx = new InitialContext( env );
      IServicoHome home = (IServicoHome) ctx.lookup( "ServicoBean" );
/* Cria um novo serviço através da home interface */
      home.create( cd_vinculo );
      }
    catch( Exception exception )
      {
      System.err.println( "Erro ao acessar o Servico" );
      throw new RemoteException("Erro",exception);
      }
    }
```

Na classe VinculoBean (Anexo 6) é chamado o método `create(int)` da *home interface* da classe ServicoBean (figura 32). Nesta classe é inserido, na tabela serviço do banco de dados, uma nova linha com as informações obtidas anteriormente.

FIGURA 32 IMPLEMENTAÇÃO DO MÉTODO EJBCREATE DA CLASSE SERVICOBEAN

```
public int ejbCreate(int cd_vinculo )
    throws CreateException, RemoteException
    {
     String insert = " insert into SERVICO (cd_vinculo, dt_pedido,
cd_tipo_servico) values " + "(" + cd_vinculo + "," + "sysdate" + "," + 1 +
")";
    try
      {
      getDbHelper().executeUpdate( insert );
      servico = new Servico( cd_vinculo );
      getDbHelper().executeComplete();
      return cd_vinculo;
      }
    catch( SQLException e )
      {
      throw new RemoteException( ( "Impossível Criar Serviço" ), e );
      }
    finally
      {
       getDbHelper().executeCompleteLogException();
      }
    }
```

Se nenhuma exceção for lançada, a classe Numerico3 executa a chamada ao método `AlteraPessoa(int)`, que faz a alteração na tabela pessoa somando mais um ao número da via do cartão da pessoa.

Quando todas as operações terminam, é mostrado a tela final confirmando ao aluno a sua solicitação de nova via do cartão de identificação.

5 CONCLUSÕES

Com a conclusão deste trabalho observou-se que a arquitetura multicamadas oferece vantagens significativas sobre as demais arquiteturas de software existentes atualmente, visto que, com o advento da Internet, pela maioria das companhias mais e maiores aplicações (até mesmo aplicações críticas) devem ser disponibilizadas nesta “grande rede”. Portanto é fundamental que o cliente seja “leve” e as regras de negócio sejam facilmente gerenciadas e atualizadas.

Porém, a implementação da arquitetura multicamadas é muito complexa, muitas dificuldades foram encontradas. Por não ter uma biblioteca de componentes já pronta, a implementação de multicamadas foi muito trabalhosa. As classes auxiliares tiveram que ser criadas, todo o seu funcionamento teve que ser entendido, assim como o entendimento do servidor de aplicação.

Uma das vantagens encontradas, é que esta arquitetura é formada por componentes (camadas) permitindo a reutilização rápida e fácil das regras de negócio, uma vez que estas não estão implementadas no cliente. Dessa forma, uma nova interface pode conter parte das regras de negócio de outra interface. Se for bem projetada, uma aplicação passará a ser apenas a montagem de componentes de negócio.

Enterprise JavaBeans é um modelo de componentes reutilizáveis para o lado do servidor, ou seja, são utilizados para implementar as regras de negócio no servidor. Através deste foi possível mapear tabelas e executar transações em banco de dados de forma simples. Este modelo de componentes foi criado para facilitar a implementação da arquitetura multicamadas.

Em teoria, consegue-se ter o cliente acessando o servidor de aplicação através de uma página HTML ou um APPLET, que é rápido e fácil de carregar. Talvez pelo fato do servidor de aplicação ser uma cópia demonstração não foi possível iniciar o *Voyager* dentro do cliente na Internet, o que fez com que uma das maiores vantagens desta aplicação não pudesse ser visualizada.

Implementar a arquitetura multicamadas foi complicado. Mas provou, na prática, que a utilização desta arquitetura trará benefícios significativos a curto prazo, desde que a biblioteca de componentes esteja completa.

5.1 DIFICULDADES

Algumas dificuldades foram encontradas:

a) a implementação de *Enterprise JavaBeans* acessando o servidor de aplicação *Voyager*;
b) a inicialização do servidor de aplicação dentro de um *applet*. Como conseqüência não foi implementada a aplicação para a execução via Internet.

5.2 SUGESTÕES

Como sugestões para trabalhos futuros para um aproveitamento mais amplo das novas possibilidades das tecnologias apresentadas sugere-se:

a) implementar utilizando-se CORBA;
b) implementar *session beans*;
c) aprofundar o estudo sobre *Enterprise JavaBeans* e sobre as ferramentas que facilitam a sua implementação;
d) fazer com que a aplicação seja executada também na Internet;
e) estudar o funcionamento dos servidores de aplicação.

REFERÊNCIAS BIBLIOGRÁFICAS

[BOC95] BOCHENSKI, Barbara. **Implementando sistema cliente/servidor de qualidade**. São Paulo : Makron Books, 1995.

[COA99] COAD,Peter; MAYFIELD, Mark. **Java design: buildding better apps and applets**. United States of America : Yourdon Press, 1999.

[CRE99] CRENSHAW, Chris. **Developer's guide to understanding enterprise javabeans™ 1.1** 1999.

[FUR98] FURLAN, José Davi. **Modelagem de objetos através da UML :** the unified modeling language. São Paulo : Makron Books, 1998.

[HAM99] HAMPSHIRE, Paulo. **Utilizando java como ferramenta corporativa. Developers' Magazine**. Rio de Janeiro, v. 3, n. 34, p. 36–39, jun. 1999.

[HEM99] HEMRAJAI, Anil. **The state of java middleware, part 2: enterprise javabeans**. 22/10/1999. Endereço eletrônico: http://www.javaworld.com/javaworld/jw-04-1999/jw-04-middleware_p.htm

[HUE99] HUEBNER, Bob; ICENOGLE, Kathy. **Implementing a scalable architecture for e-commerce**. 05/1999. Endereço eletrônico: http://www.ontos.com/wp-arch.htm

[JUB99] JUBIN, Henri; FRIEDRICHS, Jürgen; TEAM, Jalapeño. **Enterprise javabeans by example**. United State of America : International Technical Support Organization, 1999.

[LEF98] LEFAIRE, Rick. **Multi-tier aplication servers and distributed object computing** 1998. Endereço eletrônico: http://www.borland.com/about/insigth/1998/rickmultitier.html

[MON99] MONSON, Richard Haefel. **Enterprise JavaBeans**. United States of America : O'Reilly & Associates, 1999.

[OBJ99] OBJECT SPACE VOYAGER. **Application server 3.1**. Developer guide, 1999.

[ONT99] ONTOS TECHNOLOGY. **Multi-tier, component-based applications** 24/09/1999. Endereço eletrônico: http://www.ontos.com/MultiTierApps.htm

[RAT98] RATIONAL SOFTWARE CORPORATION. **Rational rose98**. Rose Enterprise Evalution Edition, 1998.

[SES99] SESHADRI, Govind. **Enterprise java computing : applications and architecture**. Cambridge : Cambridge University : Sigs Books, 1999.

[THO98] THOMAS, Anne. **Enterprise javabeans™ technology**. Sun Microsystems : 1998.

ANEXO 1

Home interface da classe Pessoa

```
package tcc;

import javax.ejb.*;
import java.rmi.*;

public interface IPessoaHome extends EJBHome {
  IPessoa create()
  throws RemoteException,CreateException;
  IPessoa findByPrimaryKey( int cd_vinculo)
  throws RemoteException,FinderException;
  Enumeration findByCodigoId( int cd_vinculo )
  throws FinderException, RemoteException;
}
```

ANEXO 2

Home interface da classe Vinculo.

```
package tcc;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import java.util.Enumeration;

public interface IVinculoHome extends EJBHome {
  IVinculo create(int cd_vinculo) throws CreateException, RemoteException;
  IVinculo findByPrimaryKey( int primaryKey ) throws FinderException,
RemoteException;
  Enumeration findByVinculoId( int cd_vinculo ) throws FinderException,
RemoteException;
}
```

ANEXO 3

Remote interface da classe Pessoa

```
package tcc;

import javax.ejb.*;

import java.rmi.*;


public interface IPessoa extends EJBObject {

  public boolean ValidaSenha(int cd_pessoa) throws RemoteException;

}
```

ANEXO 4

Remote interface da classe Vinculo.

```
package tcc;
import javax.ejb.*;
import java.rmi.*;

public interface IVinculo extends EJBObject {
  public int getCodigo() throws RemoteException;
  public void RequisitaCartao(int cd_pessoa) throws RemoteException;
}
```

ANEXO 5

Entity bean da classe Pessoa

```
package tcc;

import javax.ejb.*;
import java.rmi.*;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.Connection;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Enumeration;
import java.util.Vector;

import examples.sampleapp.osi.*;

public class PessoaBean extends AbstractEntity {
  private Pessoa pessoa;

  /*
   *  --------------
   *  Metodo Create
   *  --------------
   */
  public int ejbCreate(int cd_pessoa)
    throws CreateException, RemoteException
    {
     String query = "select cd_pessoa from pessoa where cd_pessoa = " +
cd_pessoa;
     try
       {
        ResultSet rs = getDbHelper().executeQuery( query );
        if (rs.next())
          {
```

```
         pessoa = new Pessoa(rs.getInt("cd_pessoa"));
         cd_pessoa = rs.getInt("cd_pessoa");
         return cd_pessoa ;
        }
      else
        {
         throw new RemoteException ("Pessoa não encontrada");
        }
    }
  catch (SQLException exception)
    {
     throw new RemoteException ("Erro", exception);
    }
  finally
    {
     getDbHelper().executeCompleteLogException();
    }
  }

/*
 *  ----------------------------
 *  Metodo Find PrimaryKey
 *  ----------------------------
*/
public int ejbFindByPrimaryKey (int cd_pessoa)
  throws RemoteException, FinderException
  {
   return cd_pessoa;
  }
```

```
  /*
   *  ----------------------------
   *  Metodo Find cd_pessoa
   *  ----------------------------
  */
  public Enumeration ejbFindByCodigoId (int cd_pessoa)
    throws RemoteException, FinderException
    {
String query = "select cd_pessoa from pessoa where cd_pessoa = " +
cd_pessoa;
     try
       {
        ResultSet rs = getDbHelper().executeQuery( query );
        Vector vec = new Vector();

        while ( rs.next() )
           vec.addElement ( toPrimaryKey (rs.getInt(1)) );
        return vec.elements();
       }
     catch (Exception exception)
       {
        throw new RemoteException ("Erro", exception);
       }
     finally
       {
        getDbHelper().executeCompleteLogException();
       }
     }

   private Long toPrimaryKey (int cd_pessoa)
      {
       return new Long ( (long) cd_pessoa);
      }
```

```
 /*
  *  -----------
  *  Metodo Load
  *  -----------
  */
 public void ejbLoad()
   throws RemoteException
   {
    long key = ( (Long) getPrimaryKey() ).longValue();
    String query = "select cd_pessoa from pessoa where cd_pessoa = " +
key;
    try
      {
       ResultSet rs = getDbHelper().executeQuery( query );
       rs.next();
       pessoa = new Pessoa(rs.getInt("cd_pessoa"));
      }
    catch ( SQLException exception )
      {
       throw new RemoteException ("Pessoa não Encontrada",exception);
     }
   finally
     {
      getDbHelper().executeCompleteLogException();
     }
   }

  public boolean ValidaSenha(int cd_pessoa) throws RemoteException
   {
    return pessoa.ValidaSenha(cd_pessoa);
   }

 public void ejbStore() throws RemoteException {}
 public void ejbPostCreate(int cd_vinculo) throws RemoteException
   {}
 public void ejbActivate() throws RemoteException {}
 public void ejbRemove() throws RemoteException, RemoveException {}

}
```

ANEXO 6

Entity bean da classe Vinculo.

```
package tcc;

import javax.ejb.*;
import java.rmi.*;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.Connection;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Enumeration;
import java.util.Vector;

import examples.sampleapp.osi.*;

public class VinculoBean extends AbstractEntity {
  private Vinculo vinculo;

  /*
   *  --------------
   *  Metodo Create
   *  --------------
   */
  public int ejbCreate(int cd_vinculo)
    throws CreateException, RemoteException
    {
     String query = "select cd_vinculo from vinculo where cd_vinculo = " +
cd_vinculo;
     try
       {
        ResultSet rs = getDbHelper().executeQuery( query );
        if (rs.next())
          {
```

```
          vinculo = new Vinculo(rs.getInt("cd_vinculo"));
          cd_vinculo = rs.getInt("cd_vinculo");
          return cd_vinculo ;
         }
       else
         {
          throw new RemoteException ("Vinculo não encontrado");
         }
    }
  catch (SQLException exception)
    {
     throw new RemoteException ("Erro", exception);
    }
  finally
    {
     getDbHelper().executeCompleteLogException();
    }
  }

/*
 *  ----------------------------
 *  Metodo Find PrimaryKey
 *  ----------------------------
*/
public int ejbFindByPrimaryKey (int cd_vinculo)
  throws RemoteException, FinderException
  {
   return cd_vinculo;
  }
```

```
 /*
  *  ----------------------------
  *  Metodo Find cd_vinculo
  *  ----------------------------
 */
 public Enumeration ejbFindByVinculoId (int cd_vinculo)
   throws RemoteException, FinderException
   {
    String query = "select cd_vinculo from vinculo where cd_vinculo = " +
cd_vinculo;
    try
      {
       ResultSet rs = getDbHelper().executeQuery( query );
       Vector vec = new Vector();
       while ( rs.next() )
          vec.addElement ( toPrimaryKey (rs.getInt(1)) );
       return vec.elements();
      }
    catch (Exception exception)
      {
       throw new RemoteException ("Erro", exception);
      }
    finally
      {
       getDbHelper().executeCompleteLogException();
      }
    }

  private Long toPrimaryKey (int cd_vinculo)
     {
      return new Long ( (long) cd_vinculo);
     }
```

```
 /*
  *  -----------
  *  Metodo Load
  *  -----------
  */
 public void ejbLoad()
   throws RemoteException
   {
    long key = ( (Long) getPrimaryKey() ).longValue();
    String query = "select cd_vinculo from vinculo where cd_vinculo = " +
key;
    try
      {
       ResultSet rs = getDbHelper().executeQuery( query );
       rs.next();
       vinculo = new Vinculo(rs.getInt("cd_vinculo"));
      }
    catch ( SQLException exception )
      {
       throw new RemoteException ("Vínculo não Encontrado",exception);
     }
   finally
     {
      getDbHelper().executeCompleteLogException();
     }
   }
 public int getCodigo() throws RemoteException
   {
    return vinculo.getCodigo();
   }
 public void RequisitaCartao(int cd_vinculo) throws RemoteException
   {
   try
     {
     System.out.println(" Requisita Cartão - Dentro do Vinculo ");
     String jndiDirectoryUrl = "8000/JNDI/server";
     Hashtable env = (Hashtable) System.getProperties().clone();
     env.put( Context.PROVIDER_URL, jndiDirectoryUrl );
     InitialContext ctx = new InitialContext( env );
```

```
      IServicoHome home = (IServicoHome) ctx.lookup( "ServicoBean" );
      home.create( cd_vinculo );
      }
    catch( Exception exception )
      {
      System.err.println( "Erro ao acessar o Servico" );
      throw new RemoteException("Erro",exception);
      }
    }
  public void ejbStore() throws RemoteException {}
  public void ejbPostCreate(int cd_vinculo) throws RemoteException {}
  public void ejbActivate() throws RemoteException {}
  public void ejbRemove() throws RemoteException, RemoveException {}
}
```

ANEXO 7

Classe DbHelper – classe auxiliar que faz as conexões com o servidor de aplicação *Voyager*.

```
package tcc;

import java.sql.Connection;
import java.sql.Statement;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.DriverManager;
import javax.sql.DataSource;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;
import javax.naming.Context;

public class DbHelper
  {
  private Connection conn;
  private Statement stmt;
  private DataSource dataSource;

  public DbHelper( String dataSourceUrl ) throws NamingException
    {
    Hashtable env = new Hashtable();
    env.put( Context.PROVIDER_URL, "" );
    dataSource  =  (DataSource)  (  new  InitialContext(  env  ).lookup(
dataSourceUrl ) );
    }
```

```
/**
 * Recupere uma conexão da base de dados à base de dados apropriada.
 * Qualquer SQLException encontrada enquanto recuperando a conexão é
 * propagada para o método chamador; todas as outras excessões são
 * envolvidas na RemoteException.
 */
private void openStatement() throws SQLException
  {
  // connections are automatically closed when the transaction ends
  if( ( conn == null ) || conn.isClosed() )
    conn = dataSource.getConnection();

  stmt = conn.createStatement();
  }

public boolean execute( String query ) throws SQLException
  {
  openStatement();
  return stmt.execute( query );
  }

public ResultSet executeQuery( String sql ) throws SQLException
  {
  openStatement();
  return stmt.executeQuery( sql );
  }

public int executeUpdate( String sql ) throws SQLException
  {
  openStatement();
  return stmt.executeUpdate( sql );
  }
```

```
 public void executeCompleteLogException()
    {
    try
      {
      executeComplete();
      }
    catch( SQLException e )
      {
      System.err.println(   "Erro   enquanto   fechando   Statement:   "   +
e.getMessage() );
      }
    }

  public void executeComplete() throws SQLException
    {
    if( stmt != null )
      stmt.close();
    }
        }
```