

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

Romualdo Rubens de Freitas

**Um estudo sobre Arquiteturas de Componentes de
Software para o Desenvolvimento de Sistemas
Distribuídos para Aplicações Multicamadas para Web**

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

Professor Rosvelter João Coelho da Costa

Florianópolis, Junho 2003

Um estudo sobre Arquiteturas de Componentes de Software para o Desenvolvimento de Sistemas Distribuídos para Aplicações Multicamadas para Web

Romualdo Rubens de Freitas

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

Banca Examinadora

Fernando Alvaro Ostuni Gauthier, Dr.
Coordenador do Curso de Pós-Graduação em
Ciência da Computação

Prof. Rosvelter João Coelho da Costa, Dr.
Presidente da Banca (Orientador)

Prof. Mario Antonio Ribeiro Dantas, Dr.
Membro da Banca

Prof. Vitório Bruno Mazzola, Dr.
Membro da Banca

Agradecimentos

Antes de qualquer coisa, o agradecimento maior deve ser dirigido a **Deus** e a **Nossa Senhora**, seres supremos de luz e para os quais devemos nos voltar não somente nos momentos de necessidade e angústia, como também nos momentos de alegria e de conquistas para exaltá-los em agradecimento.

Ao meu orientador, Prof. Rosvelter João Coelho da Costa, por sua paciência, sabedoria e por um dom cultivado por poucos: a amizade.

À Maria Odete Zanoni de Freitas, pelo suporte e orações e, principalmente, por ser minha mãe.

À minha amada, Deise Costacurta, por sua compreensão nos momentos em que não pude estar presente.

Aos professores do CPGCC.

Aos funcionários da secretaria do CPGCC, em especial a Sra. Vera Lúcia Sodré Teixeira, conhecida carinhosamente como “Verinha”, por sua paciência no atendimento dispensado, não somente a mim, mas a todos que, de alguma forma, precisam de seu auxílio.

Agradeço a todos aqueles que, de uma forma ou de outra, ajudaram com sugestões, críticas e comentários. Dentre estes, o amigo Arthur Cattaneo Zavadski.

Dedicatória

Da mesma forma que o agradecimento maior está dirigido a **Deus** e a **Nossa Senhora**, este trabalho é dedicado totalmente a **Eles**, por serem a força que precisamos quando as nossas parecem esvair-se.

A minha mãe, Maria Odete, e a minha querida Deise, por serem as mulheres, as quais são dotadas naturalmente de um poder místico e universal, que persistiram ao meu lado durante a realização deste trabalho, ora com suas palavras de ânimo, ora me dando o apoio necessário para continuar nos momentos em que as dificuldades faziam-se de obstáculos ao meu caminho.

Dedico este trabalho ao esforço de vários meses de trabalho árduo, às várias horas de viagem de ida e volta a Florianópolis.

Resumo

Este trabalho de dissertação apresenta um estudo sobre arquiteturas de componentes de software e descreve as principais tecnologias que compreendem a Plataforma 2, Edição Corporativa, da linguagem Java. É abordada a tecnologia de componentes *Enterprise JavaBeans* (EJB). Outros modelos de componentes de software tais como COM, DCOM e CORBA, são abordados. Também são levadas em conta as características de cada um destes modelos, instalação e administração de componentes, segurança, ciclo de vida e formas de acesso. Aplicações cliente que acessam componentes EJB podem ser de qualquer natureza, indo desde aplicações isoladas, passando por aplicações *web*, até aplicações para dispositivos portáteis. Por fim, apresenta-se um estudo de caso que utiliza a linguagem Java como linguagem alvo e algumas das principais tecnologias da Plataforma 2, como Servlets, *JavaServer Pages* e JDBC, além do *framework* Struts para o desenvolvimento de aplicações *web* seguindo o modelo 2 da especificação JSP.

Palavras-chaves: Componentes distribuídos, software para web, aplicações distribuídas.

Abstract

This work shows a study about some of the software components architectures, as well as describes the main J2EE technologies. The Enterprise JavaBeans component technology is studied. Another software component models such as COM, DCOM, and CORBA are studied. Features as of installation, component management and security, lifecycle, and access forms are also studied. Client applications accessing EJB components can be any ranging from desktop to Web applications, even applications to mobile devices. The case study uses Java language as its target language e some of the main J2EE technologies, like Servlets, JavaServer Pages and JDBC, as well as Struts framework for Web application development accordingly to the Model 2 from JSP specification.

Keywords: Distributed components, web applications, distributed applications.

Sumário

LISTA DE FIGURAS	IX
LISTA DE ABREVIATURAS	X
CAPÍTULO 1 INTRODUÇÃO	12
CAPÍTULO 2 A PLATAFORMA JAVA 2 EDIÇÃO CORPORATIVA.....	14
2.1 INTRODUÇÃO	14
2.2 JAVA DATABASE CONNECTIVITY	15
2.3 SERVLETS	16
2.4 JAVASERVER PAGES	19
2.5 ENTERPRISE JAVABEANS.....	24
2.5.1 Funcionamento de um sistema EJB cliente/servidor.....	25
2.5.2 Constituição de um Enterprise JavaBean.....	27
2.5.3 Ciclo de vida de um bean.....	30
2.6 JAVA NAMING AND DIRECTORY INTERFACE.....	37
2.7 REMOTE METHOD INVOCATION.....	38
2.7.1 Arquitetura RMI.....	39
2.7.2 Comunicação RMI.....	41
2.8 JAVA MESSAGE SERVICE	42
2.8.1 Arquitetura da API JMS	42
2.8.2 Domínios de Mensagens.....	44
2.9 CONCLUSÃO	46
CAPÍTULO 3 ARQUITETURAS DE COMPONENTES DE SOFTWARE	48
3.1 INTRODUÇÃO	48
3.2 COMPONENT OBJECT MODEL – COM.....	51
3.2.1 Interfaces.....	51
3.2.2 Classes e Servidores.....	52
3.2.3 Ciclo de Vida de Objetos.....	53
3.2.4 Interoperabilidade Binária.....	54
3.2.5 Transparência de Empacotamento.....	54
3.3 DISTRIBUTED COMPONENT OBJECT MODEL – DCOM.....	55
3.3.1 Transparência de Localização e Empacotamento.....	55
3.3.2 Modelo de Fluxo de Execução Livre	56
3.3.3 Segurança.....	56
3.3.4 Contagem de Referência e Pinging	57
3.3.5 Administração.....	57
3.4 COMMON OBJECT REQUEST BROKER ARCHITECTURE – CORBA.....	58
3.4.1 CORBA no lado do Servidor.....	59
3.4.2 CORBA no lado do Cliente.....	61
3.4.3 Protocolo Internet Inter-ORB.....	62
3.5 JAVABEANS	62
3.6 CONCLUSÃO	64
CAPÍTULO 4 ESTUDO DE CASO: SISTEMA DE CONTROLE IMOBILIÁRIO.....	65
4.1 INTRODUÇÃO	65
4.2 DESCRIÇÃO DA APLICAÇÃO.....	65
4.3 MODELAGEM.....	68
4.4 PROJETO	73
4.5 CONCLUSÃO	74

CAPÍTULO 5 CONCLUSÃO E TRABALHOS FUTUROS	76
REFERÊNCIAS	79

Lista de Figuras

Figura 2.1: Arquitetura de aplicação em múltiplas camadas.....	15
Figura 2.2: Tradução de um arquivo JSP em um Servlet.....	20
Figura 2.3: Ciclo de vida de um arquivo JSP.....	21
Figura 2.4: Modelo 1 para aplicações utilizando JavaServer Pages.....	22
Figura 2.5: Modelo 2 para aplicações utilizando JavaServer Pages baseado no padrão de projeto MVC.....	23
Figura 2.6: Ciclo de vida de um Stateless Session Bean.....	30
Figura 2.7: Ciclo de vida de um Stateful Session Bean.....	32
Figura 2.8: Ciclo de vida de um bean com persistência gerenciada a bean.....	33
Figura 2.9: Ciclo de vida de um bean com persistência gerenciada a contêiner.....	35
Figura 2.10: Ciclo de vida de um Message-driven bean.....	37
Figura 2.11: Stubs e Skeletons.....	39
Figura 2.12: Arquitetura JMS.....	43
Figura 2.13: Point-to-Point Messaging.....	44
Figura 2.14: Publish/Subscribe Messaging.....	46
Figura 3.1: Evolução do Desenvolvimento baseado em Componentes.....	49
Figura 3.2: Cliente COM interagindo com objetos COM empacotados de várias formas.....	53
Figura 3.3: Mecanismo COM da Classe Factory.....	53
Figura 3.4: Requisição de um objeto via ORB.....	58
Figura 3.5: Descrição de uma Interface em IDL.....	59
Figura 3.6: Geração de <i>stubs</i> e interface em linguagem nativa a partir da descrição IDL.....	60
Figura 4.1: Interface administrativa do Sistema de Controle Imobiliário.....	66
Figura 4.2: Interface pública do Sistema de Controle Imobiliário.....	67
Figura 4.3: Resultado da pesquisa por imóveis na interface pública.....	68
Figura 4.4: Diagrama de Casos de Uso para o ator funcionário.....	69
Figura 4.5: Interface do Usuário para Inclusão de um Imóvel.....	70
Figura 4.6: Relação dos imóveis existentes.....	71
Figura 4.7: Operação de alteração dos dados de um determinado imóvel.....	72
Figura 4.8: Operação de exclusão dos dados de um determinado imóvel.....	72

Lista de Abreviaturas

ACL	A ccess C ontrol L ist
API	A pplication P rogramming I nterface
BMP	B ean- m anaged p ersistence
CGI	C ommon G ateway I nterface
CLSID	C lass I dentifier
CMP	C ontainer- m anaged p ersistence
COM	C omponent O bject M odel
CORBA	C ommon O bject R equest B roker A rchitecture
DCOM	D istributed C omponent O bject M odel
DLL	D ynamic L ink L ibrary
DNS	D omain N ame S ervices
EJB	E nterprise J ava B eans
GIOP	G eneral I nter- O RB P rotocol
HTTP	H yper T ext T ransport P rotocol
HTML	H yper T ext M arkup L anguage
IBM	I nternational B usiness M achines
IDL	I nterface D efinition L anguage
IID	I nterface I dentifier
IIOP	I nternet I nter- O RB P rotocol
IOR	I nteroperable O bject R equest
IP	I nternet P rotocol
J2EE	J ava 2 E nterprise E dition
J2SE	J ava 2 S tandard E dition
JDBC	J ava D atabase C onnectivity
JDK	J ava D evelopment K it
JMS	J ava M essage S ervice
JNDI	J ava N aming and D irectories I nterface
JNI	J ava N ative I nterfaces
JSP	J ava S erver P ages
LDAP	L ightweight D irectory A ccess P rotocol
MIDL	M icrosoft I nterface D efinition L anguage
MOM	M iddleware O riented- M essage

ODBC	O pen D atabase C onnectiv y
OMG	O bject M anagement G roup
ORB	O bject R equest B roker
PDA	P ersonal D isplay A ssistent
RMI	R emote M ethod I nvocation
RPC	R emote M ethod C all
SGBD	S istema G erenciador de B anco de D ados
SQL	S tructured Q uery L anguange
TCP	T ransfer C ontrol P rotocol
UDP	U ser D atagram P rotocol
URL	U niform R esource L ocator
XML	E xtensible M arkup L anguage
WWW	W orld W ide W eb

Capítulo 1

Introdução

Desde que a primeira versão de Java foi apresentada em maio de 1995, ela tornou-se amplamente aceita pela comunidade devido ao seu forte interesse pela WWW (*World Wide Web*), fornecendo serviços para a construção de sistemas distribuídos. Atualmente Java é utilizada para a criação de páginas *Web* com conteúdo interativo e dinâmico, para o desenvolvimento de aplicativos corporativos de larga escala, para melhorar os servidores *Web* e para o fornecimento de aplicativos para dispositivos destinados ao consumidor final, tais como telefones celulares, *paggers* e PDAs (*Personal Display Assistant*).

A rápida adoção de Java deve-se ao fato de ser uma linguagem para o desenvolvimento de aplicações do lado do servidor para a *Web*. Assim, Java torna-se a viga-mestre de um conjunto de tecnologias para a construção de sistemas baseados em componentes e distribuídos devido ao fato de que o desenvolvimento de aplicações ser centrado em rede.

A constante busca por padrões, técnicas e ferramentas que permitam agilizar a produção de software confiável e de qualidade em um prazo cada vez menor é incessante. Com a crescente necessidade de aplicações complexas, que estejam distribuídas por redes locais, metropolitanas ou geograficamente divididas, como a *world wide web*, a utilização da tecnologia de componentes de software para o desenvolvimento de aplicações destaca-se por sua grande confiabilidade e alto grau reusabilidade e flexibilidade.

Assim, a motivação para este trabalho é o estudo do emprego de componentes de software no desenvolvimento de sistemas distribuídos multicamadas para *web*.

Objetivando estudar o desenvolvimento de tais sistemas através do emprego da tecnologia EJBs (*Enterprise JavaBeans*), disponibilizada na versão corporativa da Plataforma Java 2 – J2EE (*Java 2 Enterprise Edition*), será feito um levantamento bibliográfico à cerca dos trabalhos já realizados que abordam estas tecnologias e o estudo da tecnologia *Enterprise JavaBeans* por meio de sua especificação.

Com a utilização de EJBs, o projetista de software pode concentrar-se na lógica de negócios do domínio de suas aplicações, enquanto faz uso de características importantes, tais como: distribuição de objetos e componentização de software no lado do servidor, que também fornece outros serviços, como, por exemplo: transações e serviço de nomes acessíveis através da API (*Application Programming Interface*) JNDI (*Java Naming and Directory Interface*).

Como proposta de estudo de caso, o Capítulo 4 apresenta um estudo abordando a arquitetura EJB de componentes de software em uma aplicação voltada ao ramo imobiliário e que pode ser executada via Internet, Intranet ou Extranet. Além da arquitetura de componentes EJB, outras tecnologias da Plataforma Java 2 são utilizadas, bem como outros artefatos de software destinados tanto ao desenvolvimento de aplicações desta natureza, tais como o *framework* Struts, o servidor JBoss, que reúne um servidor de aplicações – para a execução de componentes EJB, um servidor *web* – para disponibilizar conteúdo via navegador e um contêiner Servlet/JSP (*JavaServer Pages*).

Os capítulos que se seguem estão assim divididos: Capítulo 2: descreve a Plataforma Java 2 em sua edição corporativa, Capítulo 3: apresenta algumas das principais arquiteturas de componentes de software, Capítulo 4: aborda o estudo de caso de um sistema de controle imobiliário que emprega a arquitetura de componentes de software EJB. Para finalizar, o Capítulo 5 apresenta as conclusões obtidas com a realização deste trabalho, bem como sugere trabalhos a serem realizados.

Capítulo 2

A Plataforma Java 2 Edição Corporativa

2.1 Introdução

A Plataforma Java 2, Edição Corporativa, é construída no topo da Edição Padrão. Assim, as características da J2SE (*Java 2 Standard Edition*), tais como: JDBC (*Java DataBase Connectivity*) para acesso a bases de dados, CORBA (*Common Object Request Broker Architecture*) para acesso a recursos corporativos e o modelo de segurança, acrescentado suporte a componentes EJB, Servlets, JSP, XML (*eXtensible Markup Language*), dentre outras. Desta forma, a Plataforma J2EE consiste de um conjunto de serviços, interfaces de programação de aplicações e protocolos que fornecem a funcionalidade para o desenvolvimento de soluções *Web* multicamadas e soluções do lado do servidor com força corporativa.

A Figura 2.1 ilustra como as várias tecnologias de J2EE podem ser utilizadas no desenvolvimento de um sistema distribuído.

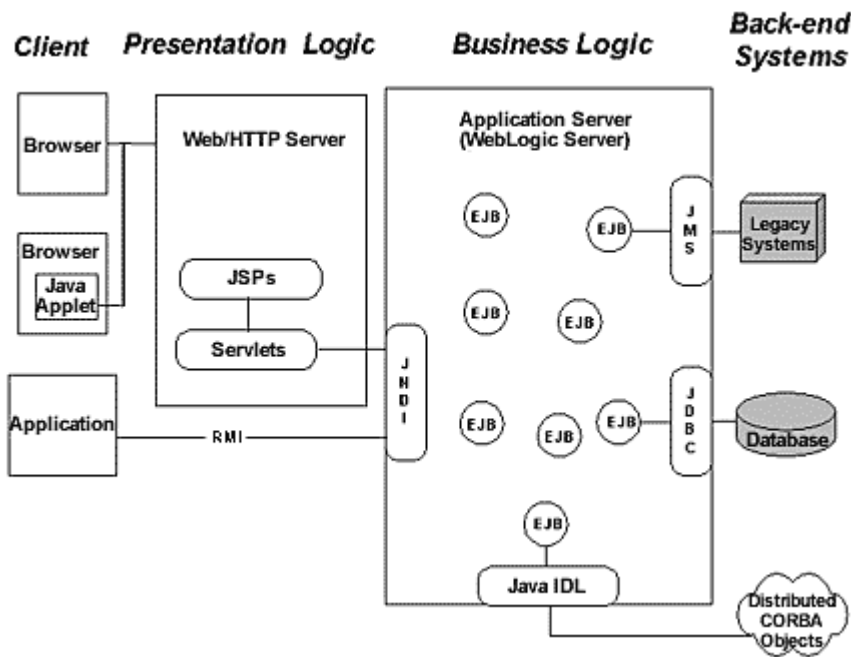


Figura 2.1: Arquitetura de aplicação em múltiplas camadas.

A seguir, são apresentadas as principais tecnologias que compõem a Plataforma Java 2, Edição Corporativa e que serão utilizadas, como foco principal, no desenvolvimento deste trabalho de dissertação.

2.2 Java DataBase Connectivity

Mesmo não estando ligada diretamente a Plataforma J2EE, a tecnologia JDBC é muito utilizada por componentes EJB para acesso ao sistema de persistência de dados. A interface JDBC está baseada ao nível de chamada SQL (*Structured Query Language*) X/Open – **X/Open SQL Call Level Interface**. A interface de programação de aplicações JDBC fornece uma interface de programação para comunicação com bases de dados de uma maneira uniforme semelhante ao componente ODBC (*Open DataBase Connectivity*) da Microsoft [SHA96].

Devido ao fato de ser desenvolvida em Java, JDBC permite acesso independente de plataforma a diversas bases de dados e consiste de duas camadas principais: a API JDBC fornece suporte de comunicação da aplicação com o Gerente

JDBC, enquanto a API do *Driver* JDBC fornece suporte de comunicação do Gerente JDBC com o *Driver* do banco de dados. Cada *driver* deve implantar o conjunto de classes virtuais de acesso ao banco de dados para o qual ele foi criado.

Para permitir o acesso a bases de dados, JDBC define quatro tipos de *drivers*:

- **Tipo 1: ponte JDBC-ODBC:** é possível acessar fontes de dados utilizando ODBC. Isto requer que um *driver* ODBC esteja instalado na máquina cliente e sacrifica a independência de plataforma;

- **Tipo 2: ponte *driver* nativo-JDBC:** fornece uma interface JDBC construída sobre uma *driver* de banco de dados nativo. Neste caso é necessário que exista o *driver* do banco de dados instalado na máquina cliente;

- **Tipo 3: ponte rede-JDBC:** este tipo exclui a necessidade de *drivers* de banco de dados no lado do cliente. O acesso ao banco de dados é feito através da camada intermediária do servidor de rede. Desta forma, tecnologias como balanço de carga, *pool* de conexões e *cache* de dados tornam-se possíveis. *Drivers* deste tipo são mais bem utilizados em aplicações para a Internet;

- **Tipo 4: *driver* Java puro:** fornece acesso direto ao banco de dados utilizando *driver* Java puro. Devido à forma como este tipo de *driver* executa no cliente e acessa diretamente o banco de dados, ele implica uma arquitetura de duas camadas. Para usá-lo em aplicações multicamadas, o ideal é utilizar a tecnologia EJB, vista mais adiante, para acesso ao banco de dados e, assim, fornecer a seus clientes serviços de banco de dados independente do SGBD (Sistema Gerenciador de Bancos de Dados) em uso.

2.3 Servlets

Servlets são programas Java executados no servidor *Web*. Estes programas que realizam a interface `Servlet`, atuam como uma camada intermediária entre o navegador ou outro cliente HTTP (*HyperText Transfer Protocol*) e bancos de dados ou aplicações no servidor HTTP.

[HAL02] descreve que os servlets utilizam o modelo de processamento pedido – parâmetro `ServletRequest` no método `service()` – e resposta – parâmetro `ServletResponse` no método `service()`. Neste modelo, o cliente envia uma mensagem na forma de um pedido ao servidor e este, por sua vez, retorna uma mensagem de resposta. O método `service()` será chamado sempre que uma requisição ao servlet for feita. Dois outros métodos existentes na interface `Servlet` e que são codificados na classe que a realiza são utilizados para a inicialização e finalização de execução do servlet. O método `init()` permite que quaisquer ajustes iniciais, conexões com bancos de dados, por exemplo, sejam feitos antes que qualquer requisição seja atendida pelo servlet. Este método recebe como parâmetro uma referência para a interface `ServletConfig`. Esta referência possui o método `getServletContext()` que retorna o contexto no qual o servlet está sendo executado através da interface `ServletContext`, definida pelo servidor *Web*. Quando o servlet está para ser encerrado, o método `destroy()` é chamado para que o servlet tenha a chance de finalizar sua execução, por exemplo, liberando uma conexão com banco de dados. Estes três métodos constituem o ciclo de vida de um servlet.

Pelo fato de um servlet executar processamento na camada intermediária, esta tarefa pode ser retirada do cliente, tornando-o mais rápido, e do servidor, fazendo que ele cumpra sua missão de apenas servir conteúdo.

Além de processamento na camada intermediária entre cliente e servidor, um servlet pode agir como um *proxy* para o cliente ou expandir as características da camada intermediária proporcionando suporte a novos protocolos ou a novas funcionalidades.

É muito comum servlets utilizarem o protocolo HTTP. Assim, a classe `HttpServlet` proporciona suporte a servlets deste tipo. As principais funções destes servlets são:

- Ler quaisquer dados enviados pelo usuário;
- Obter quaisquer informações sobre o pedido do usuário que possam fazer parte de um pedido HTTP;
- Gerar resultados;

- Formatar resultados através de documentos;
- Ajustar os parâmetros de resposta HTTP adequados;
- Enviar o documento de volta ao cliente.

Um servlet com suporte ao protocolo HTTP possui os métodos `init()` e `destroy()` que controlam a inicialização e finalização do servlet e também métodos que são realizados para o processamento de métodos¹ HTTP. Estes métodos podem ser: GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT e OPTIONS. Normalmente os métodos comumente realizados são: GET, POST e HEAD.

O método GET processa informações que são anexadas a URL (*Uniform Resource Locator*) pelo servidor *Web*. Este método é utilizado para o processamento de uma pequena quantidade de informações. O servidor *Web* acrescenta as informações a serem passadas ao servlet seguindo um formato específico. Inicialmente o servidor acrescenta o caractere ? (interrogação) ao final da URL, em seguida os dados são acrescentados na forma de par valor/nome, utilizando o formato **nome=valor**, separados pelo caractere & (E comercial). O final da URL é marcado com o caractere de espaço. Assim, se algum nome ou valor contiver este caractere, ele é substituído por %20, que o representa no formato hexadecimal².

No caso do método POST, o servidor *Web* fornece as informações de processamento através do fluxo de entrada do servlet também na forma de par valor/nome, permitindo ao servlet lidar com uma quantidade bem maior de informações, se comparado ao método GET.

Felizmente, o desenvolvedor não precisa se preocupar com a forma como os dados serão recebidos, se através da URL ou pelo fluxo de entrada, já que a classe `HttpServletRequest` os decodifica e os torna acessíveis através do método `getParameter()`. É comum que o desenvolvedor codifique, por exemplo, o método Java de processamento do método POST para que ele também processe os dados de

¹ Métodos HTTP representam um conjunto de mensagens baseadas em texto e não devem ser confundidos com métodos Java.

² Um dos sistemas numéricos mais utilizado na área da computação.

uma chamada GET. O método Java para processar o método GET simplesmente invoca o método Java para processamento do método POST repassando os objetos `HttpServletRequest` e `HttpServletResponse` recebidos do servidor.

O método HEAD é semelhante ao método GET com a diferença de que este método retorna somente informações de cabeçalho. Este método geralmente é utilizado para se obter informações, tais como: a data da última atualização do documento no servidor, o tamanho do documento antes que uma transferência seja feita, o tipo do servidor ou o tipo do documento sendo pedido.

2.4 JavaServer Pages

A tecnologia de JSP permite a combinação de código HTML estático com conteúdo gerado dinamicamente por meio de servlets. O código Java é inserido juntamente com o código HTML por meio da utilização de *tags* especiais em um arquivo com extensão `.jsp`. Antes que a página seja servida, ela é processada pelo servidor *Web*.

Conforme mostra a Figura 2.2, um arquivo JSP é automaticamente convertido para um servlet na primeira vez em que ele for requisitado. Assim, o código HTML estático simplesmente é enviado para o fluxo de saída associado com o método `service()` do servlet, enquanto o código Java é convertido para um servlet e compilado, de acordo com [SM00].

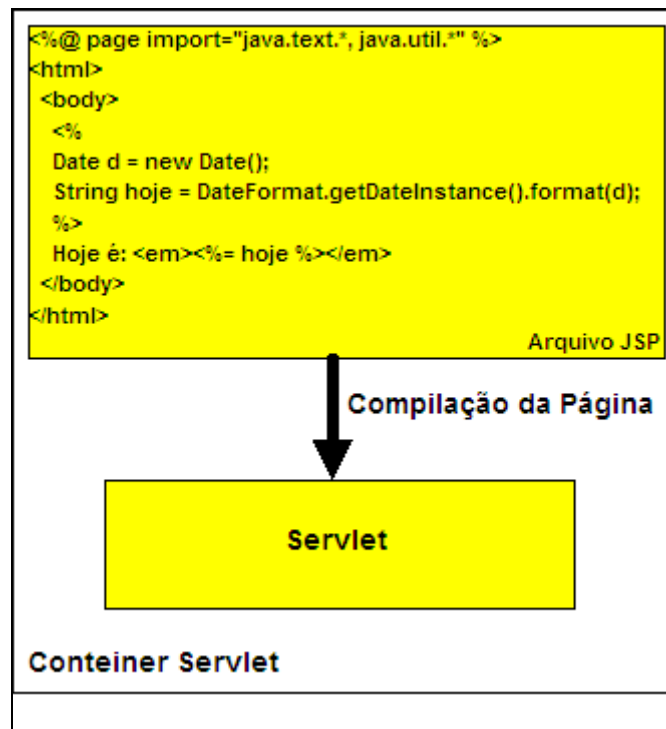


Figura 2.2: Tradução de um arquivo JSP em um Servlet.

O arquivo representado pela Figura 2.2 é convertido para um servlet e a parte do código fonte representando a página JSP, gerado pelo motor JSP do servidor Tomcat, é mostrado a seguir.

```

.
.
.
// begin
out.write("\r\n<html>\r\n<body>\r\n");
// end
// begin [file="E:\\jsp\\jsptest.jsp";from=(3,2);to=(5,0)]
Date d = new Date();
String hoje = DateFormat.getDateInstance().format(d);
// end
// begin
out.write("\r\nHoje é: \r\n<em> ");
// end
// begin [file="E:\\jsp\\jsptest.jsp";from=(7,8);to=(7,13)]
out.print(hoje);</b>
// end
// begin
out.write(" </em>\r\n</body>\r\n</html>\r\n");
// end
.
.
.

```

Uma vez que o arquivo JSP tenha sido compilado e carregado no contêiner JSP, seu ciclo de vida tem início. O ciclo de vida de um arquivo JSP é semelhante ao ciclo de vida de um servlet, visto que o primeiro é convertido e compilado para um servlet.

A Figura 2.3 mostra o ciclo de vida do servlet gerado a partir do arquivo JSP.

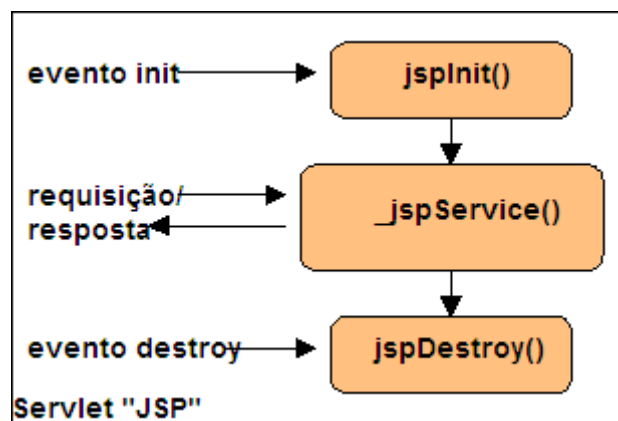


Figura 2.3: Ciclo de vida de um arquivo JSP.

A criação de aplicações *Web* com servlets e JSP pode ser feita utilizando-se um dos dois modelos propostos na especificação *JavaServer Pages*. O Modelo 1 sugere que uma solicitação seja processada da seguinte forma:

- O pedido é direcionado diretamente para um arquivo JSP;
- O arquivo JSP obtém acesso à lógica de negócios usando *JavaBeans*; e
- Os *JavaBeans* acessam os Sistemas de Informação Corporativos e retornam dados dinâmicos para o arquivo JSP.

De acordo com [SHA02], a estrutura descrita no Modelo 1 e visualizada na Figura 2.4 é mais adequada a aplicações simples, pois se houver algum processamento que não possa ser executado por *JavaBeans*, ele precisará ser executado com *scriptlets*, na página JSP, o que pode tornar a aplicação demasiadamente complexa.

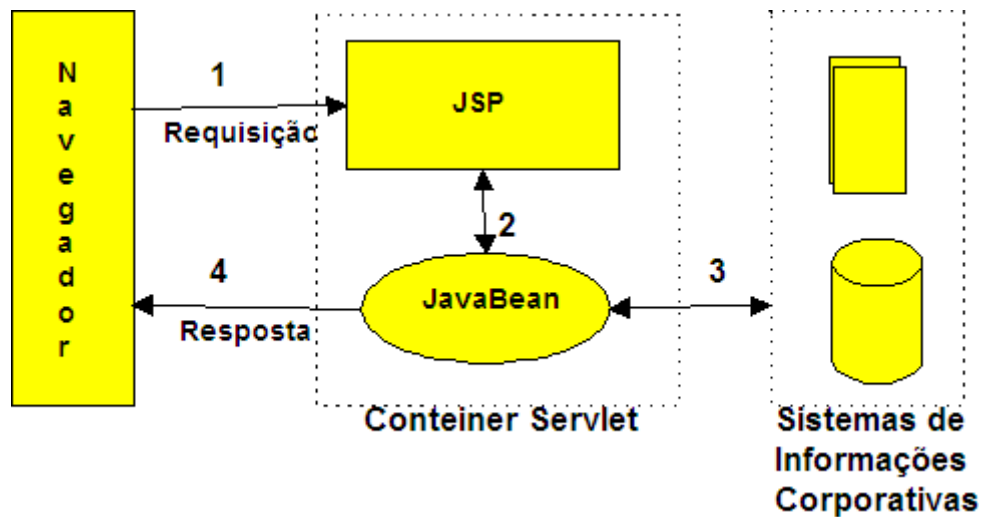


Figura 2.4: Modelo 1 para aplicações utilizando JavaServer Pages.

Ainda de acordo com [SHA02], o Modelo 2³ mostra uma clara separação entre os componentes da aplicação responsáveis pela apresentação e pelo processamento de requisições HTTP. Componentes que tratam da apresentação são escritos na forma de páginas JSP (Visão – *View*), enquanto os componentes que processam os pedidos HTTP podem ser escritos como servlets ou páginas JSP (Controlador – *Controller*) e os componentes que lidam com a lógica de negócios podem ser JavaBeans ou *Enterprise JavaBeans* (Modelo – *Model*). O Modelo 2 é apresentado na Figura 2.5, abaixo.

³ Este modelo descreve como uma aplicação pode ser criada utilizando-se o padrão de projeto Modelo/Visão/Controlador (MVC – *Model/View/Controller*).

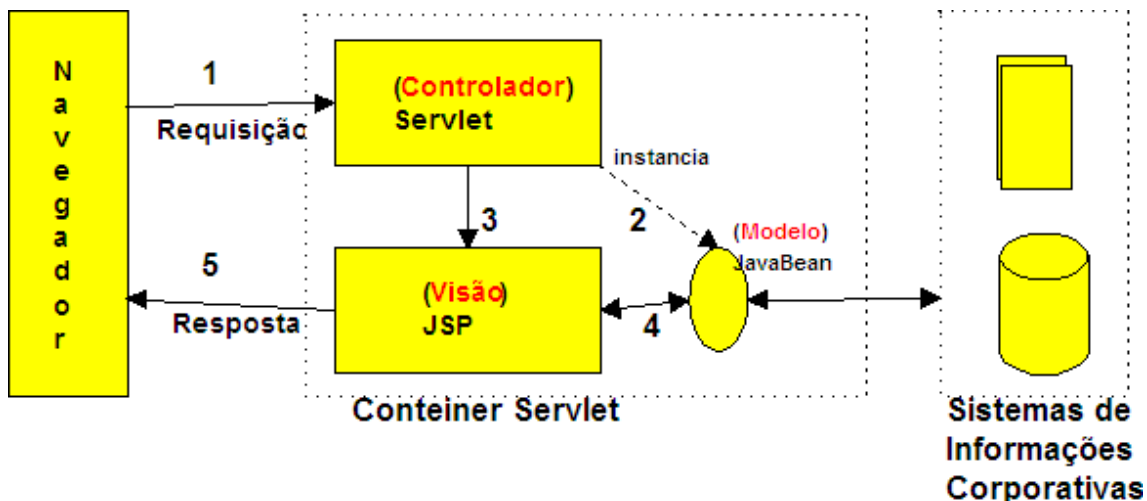


Figura 2.5: Modelo 2 para aplicações utilizando JavaServer Pages baseado no padrão de projeto MVC.

A criação de um arquivo JSP é feita utilizando-se conteúdo estático, por exemplo, HTML (*HyperText Markup Language*), e elementos de sintaxe JSP, descritos a seguir, conforme [SHA02]:

Elementos de script que permitem a inserção de código Java que será convertido em um servlet. Estes elementos podem ser os seguintes: **expressões** que são avaliadas e incluídas na saída do servlet; **scriptlets** que são incluídos no método `_jspService()`, chamado pelo método `service()`, do servlet; e **declarações** que são incluídas no corpo da classe do servlet, fora de quaisquer métodos.

Diretivas que afetam a estrutura geral do servlet resultante da página JSP. Atualmente três diretivas estão disponíveis para uso: **page**, permite a utilização de atributos tais como `import`, `contentType`, `session`, `buffer`, `errorPage` e `isErrorPage`; **include**, possibilita a inclusão de outros arquivos, tais como arquivos JSP utilizando o atributo `file`, arquivos necessários em um determinado momento com a ação `jsp:include` e *applets* ou JavaBeans utilizando o elemento `jsp:plugin`; e **taglib**, introduzida a partir da versão 1.1 da especificação JSP. Esta diretiva permite ao projetista de software definir *tags* que cuidam de comportamentos complexos do lado do servidor. Estas *tags* são agrupadas formando bibliotecas de *tags*.

2.5 *Enterprise JavaBeans*

Sua idéia básica é a de fornecer um *framework* para componentes que podem ser “conectados” a um servidor, o Servidor de Aplicações, além de estender a funcionalidade do servidor. Assim, a tecnologia de componentes corporativos *Enterprise JavaBeans* incorpora conceitos de várias áreas tais como computação distribuída, bancos de dados, segurança e aplicações modeladas a componentes.

A especificação EJB define os vários participantes no sistema cliente/servidor EJB, explica a compatibilidade EJB com CORBA e define as responsabilidades para os vários componentes do sistema.

Em [SM01] observa-se que a especificação EJB procura atingir os seguintes objetivos:

- EJBs são projetados para facilitar a criação de aplicações por parte dos desenvolvedores, liberando-os de detalhes de baixo nível do sistema para gerenciamento de transações, *threads*, balanceamento de carga e assim por diante. Desta forma, os desenvolvedores de aplicações podem concentrar-se na realização da lógica de negócios;
- A especificação define as maiores estruturas do *framework* EJB e então, especificamente, define os contratos entre elas. As responsabilidades do cliente, do servidor e de componentes individuais também são definidas;
- EJB almeja ser o meio padrão para aplicações cliente/servidor serem construídas na linguagem Java;
- Finalmente, EJBs são compatíveis com outras APIs Java, podem interoperar com aplicações não Java e são compatíveis com CORBA.

Quando o projetista cria aplicações de negócios de grande porte, ele não se preocupa ou tem pouco a se preocupar com questões, tais como: invocação remota de métodos, balanço de carga, tolerância a falhas, integração com sistemas de retaguarda, transações, *clustering*, registro e auditoria, gerenciamento de sistemas, *threading*, camada intermediária orientada a mensagens, ciclo de vida de objetos, *pooling* de recursos, segurança e *caching*.

2.5.1 Funcionamento de um sistema EJB cliente/servidor

Para compreender como um sistema EJB Cliente/Servidor opera, é necessário entender as partes básicas de um sistema EJB: o **componente EJB**, o **contêiner EJB** e o **objeto EJB**, conforme descrito em [SM01]:

- Um EJB é um componente que é executado em um Contêiner EJB, que por sua vez é executado em um Servidor EJB. Um componente EJB é uma classe Java, escrita por um projetista EJB que implanta lógica de negócios. Todas as outras classes no sistema EJB ou suportam acesso cliente a classes componentes EJB ou fornecem serviços (persistência, por exemplo) a estas classes;

- O Contêiner EJB fornece serviços tais como gerenciamento de transação e de recursos, controle de versão, escalabilidade, mobilidade, persistência e segurança a componentes EJB que ele contém. Levando em consideração que um Contêiner EJB cuida de todas estas funções, o projetista de componentes EJB pode concentrar-se nas regras de negócios e deixar a manipulação de bases de dados, em se tratando de um *Entity Bean* do tipo CMP (*Container-managed persistence*) e outros detalhes para o contêiner;

- Programas clientes executam métodos em EJBs remotos através de um Objeto EJB. O Objeto EJB realiza a **interface remota** do componente EJB no servidor. Esta interface representa os métodos de “negócios” do componente EJB. Objetos EJB e Componentes EJB são classes separadas. Um Componente EJB é executado no servidor em um Contêiner EJB e implanta a lógica de negócios. O Objeto EJB é executado no cliente e remotamente executa métodos do Componente EJB.

Além dos objetivos da especificação EJB citados anteriormente, esta mesma especificação define três tipos fundamentais de *beans*:

- **Session beans**: agem como “verbos”, pois executam ações. Estes *beans* modelam processos de negócios tais como cálculos de preços, autorização de crédito ou processamento de catálogos de produtos;

- **Entity beans**: agem como “substantivos”, pois representam objetos de dados. Estes *beans* modelam dados de negócios, tais como: produto, pedido ou cliente;

- ***Message-driven beans***: são similares aos *Session Beans*, pois executam ações. A diferença reside no fato de que *Message-driven Beans* são utilizados somente para o envio de mensagens a outros *beans*, por exemplo, mensagem para autorização de crédito.

Os *Session Beans* possuem, por sua vez, dois subtipos:

- ***Stateless session beans***: fornecem serviço de uso único, não mantêm qualquer estado, não “sobrevivem” a quedas do servidor e são de vida relativamente curta, dependendo da sessão do cliente que os utiliza. Este tipo de *bean* pode servir a vários clientes;

- ***Stateful session beans***: fornecem interação conversacional com o cliente e, como tal, armazenam estado em nome do cliente. Ele não sobrevive a quedas do servidor, também possuem vida relativamente curta e cada instância somente pode ser usada por uma única *thread* (cliente).

Quanto aos *Entity Beans*, os dados de objetos por eles representados normalmente são armazenados em uma base de dados utilizando a tecnologia JDBC. Este armazenamento pode ser feito de duas maneiras: pelo próprio *bean* ou pelo Contêiner EJB.

Beans que contêm a lógica de manipulação de bancos de dados são criados como *beans* com persistência gerenciada a *bean* (BMP – *Bean-managed persistence*). Para que o *bean* seja capaz de manipular as informações de e para a base de dados, o desenvolvedor deve codificar toda a lógica de armazenamento, recuperação, alteração e exclusão dos dados via chamadas a API JDBC.

Na outra forma de armazenamento dos objetos de dados, o Contêiner EJB cuida da geração do código necessário ao armazenamento. A geração deste código é específica de cada vendedor, que fornece ferramentas que permitem o mapeamento dos atributos do *bean* com os atributos no banco de dados. *Entity Beans* que deixam a cargo do contêiner o armazenamento dos dados são conhecidos como *beans* com persistência gerenciada a contêiner (CMP – *Container-managed bean*).

2.5.2 Constituição de um *Enterprise JavaBean*

Em [RAJ02] é constituído de uma realização, denominada “classe de realização do *bean*”. Esta classe Java realiza uma interface bem definida e obedece a regras necessárias à sua execução no contêiner EJB. Para um *Session Bean*, a classe define a lógica de negócios de processos, para um *Entity Bean*, a classe define a lógica relacionada a dados e para um *Message-driven Bean*, a classe contém lógica orientada a mensagens.

Todos os *beans* devem realizar algumas interfaces padrões para que eles possam expor certos métodos definidos no modelo de componentes EJB. A interface `EnterpriseBean` é a mais básica delas. Esta interface age como “marcadora” identificando o *bean* como um *enterprise bean* e estende a interface `Serializable`, indicando que o *bean* pode ser serializado⁴.

Os *Session Beans*, *Entity Beans* e *Message-driven Beans* realizam, cada um, uma interface mais específica. Todos os *Session Beans* devem realizar a interface `SessionBean`, todos os *Entity Beans* devem realizar a interface `EntityBean` e todos os *Message-driven Beans* devem realizar a interface `MessageDrivenBean`.

O acesso a um *bean*, executado no contêiner EJB, jamais é feito diretamente, ao invés disso, o cliente envia um pedido de acesso que é interceptado pelo contêiner que então transfere o pedido ao *bean*. A interceptação de pedidos permite que o contêiner possa executar algumas ações implícitas na camada intermediária tais como:

- Gerenciamento de transação distribuída;
- Segurança;
- Gerenciamento de recursos e ciclo de vida do componente;
- Persistência;
- Acessibilidade remota;
- Suporte a concorrência;
- Transparência de localização do componente;
- Monitoramento.

⁴ Habilidade de armazenar e recuperar objetos Java utilizando o conceito de fluxos de dados.

O contêiner age como uma camada intermediária entre o código do cliente e o código do *bean*. Esta camada intermediária manifesta-se como um **objeto EJB**. Um objeto EJB replica e expõe todos os métodos de negócios definidos na classe de realização do *bean*.

O objeto EJB é gerado automaticamente pelo contêiner. O contêiner “sabe” como criar o objeto EJB de acordo com a interface remota. Esta interface, fornecida pelo projetista, realiza a interface `EJBObject` de acordo com regras definidas pela especificação EJB. O código cliente que deseja acessar um determinado *bean* deve invocar métodos na interface remota do *bean*, pois além de duplicar os métodos definidos na classe de realização do *bean*, a interface remota expõe os métodos que serão gerados automaticamente pelo contêiner.

Para que uma referência a um objeto EJB possa ser obtida, o código cliente utiliza, de acordo com a especificação EJB, uma **fábrica** de objetos EJB. Esta fábrica, denominada **objeto Home**, é responsável pela criação e remoção de objetos EJB e, para *Entity Beans*, por encontrar objetos EJB. O contêiner cria automaticamente objetos *Home*, isto é feito através da interface `EJBHome`, que deve ser estendida para fornecer métodos específicos para as operações de criação, busca e remoção de objetos *Home*.

O acesso a um *bean* é feito executando-se os seguintes passos:

- O código cliente aciona o *stub*;
- O *stub* “empacota” as informações de chamada em uma forma aceitável pela rede;
- As informações de chamada trafegam pela rede do *stub* até chegar ao *skeleton*;
- O *skeleton* “desempacota” as informações de chamada;
- O *skeleton* aciona o objeto EJB;
- O objeto EJB executa tarefas implícitas na camada intermediária citadas anteriormente;
- O objeto EJB chama o *bean*, que realiza a função desejada.

Stubs e *Skeletons*⁵ são detalhados na Seção 2.7.1.

Após o *bean* executar seu trabalho, todos os passos mencionados acima são executados novamente na ordem inversa para que o resultado da operação do *bean* possa ser retornado ao código cliente. Isto implica em um tráfego de rede alto, principalmente se for considerado que muitos clientes possam estar acessando muitos *beans*.

Para amenizar o alto tráfego de rede na chamada a um *bean*, a especificação EJB 2.0 introduziu duas novas interfaces, a interface *local* (`EJBLocalObject`) e a interface *local home* (`EJBLocalHome`). Estas interfaces podem ser utilizadas para substituir ou complementar a interface remota (`EJBObject`) e a interface *home* (`EJBHome`), respectivamente.

A interface *local* é utilizada na realização do **objeto local**. Quando o código cliente necessita chamar um *bean*, as etapas necessárias são minimizadas da seguinte forma:

- O cliente chama o objeto *local*;
- O objeto *local* executa código na camada intermediária;
- O objeto *local* chama o *bean* para que este possa executar o método desejado.

Os passos citados acima ajudam a diminuir o tráfego de rede quando a chamada a um *bean* é feita. Este ganho de desempenho tem uma desvantagem considerável, pois é importante salientar que as interfaces *local* e *local home* podem ser utilizadas somente por *beans* que chamam outros *beans* no mesmo espaço do processo em execução, por exemplo, um *bean* que aciona outro *bean* no mesmo servidor de aplicações.

⁵ *Stubs* e *Skeletons* são estruturas geradas a partir da compilação de classes utilizando o programa `rmic`, uma ferramenta de software disponível no *kit* de desenvolvimento Java (JDK – *Java Development Kit*).

2.5.3 Ciclo de vida de um bean

Todos os *beans*, além da interface remota, possuem uma interface *home*. Esta interface fornece métodos que controlam o ciclo de vida dos *beans*.

De acordo com [CK02], os vários ciclos de vida dos vários *beans* são mostrados a seguir.

O ciclo de vida de um *Stateless Session Bean* é apresentado na Figura 2.6.

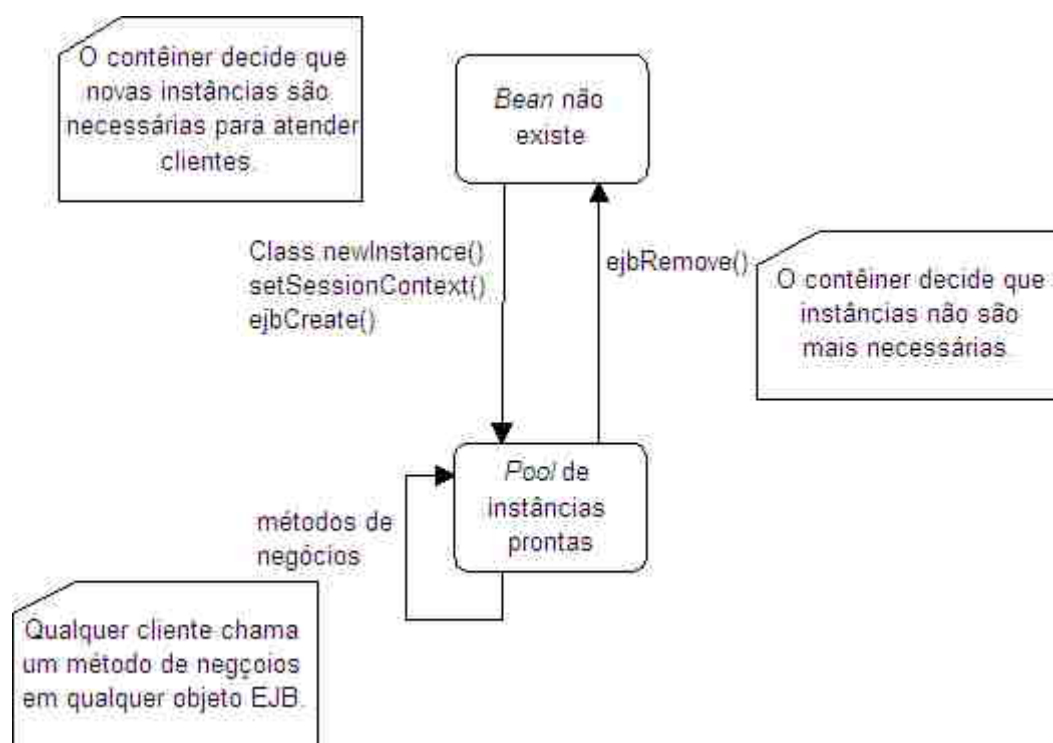


Figura 2.6: Ciclo de vida de um Stateless Session Bean.

A Figura 2.6 mostra que o cliente jamais chama métodos diretamente no *bean*. Os objetos *home* e remoto, no contêiner, são responsáveis por invocar métodos no *bean* em nome do cliente. Os passos executados durante o ciclo de vida de um *Stateless Session Bean* são descritos a seguir:

- Inicialmente o *bean* não existe, talvez o servidor de aplicações acabou de ser iniciado ou uma determinada instância de *bean* ainda não foi sido criada;

- O contêiner cria *beans* de acordo com sua política de *pooling*, baseado nas informações fornecidas por meio de arquivos de configuração;

- O contêiner instancia o *bean* do cliente invocando o método `Class.newInstance("MeuBean.class")`, que é equivalente a `new MeuBean()`. O *bean* é instanciado utilizando a primeira forma, pois o contêiner trabalha com qualquer *bean*, desta forma não é necessário codificar a chamada que dependeria do nome da classe para ser realizada. Quando o *bean* é instanciado seu construtor é chamado e quaisquer inicializações que se fizerem necessárias para a execução do *bean* podem ser realizadas neste momento;

- O contêiner invoca o método `setSessionContext()` que associa a aplicação com contexto do objeto e permite que métodos possam ser chamados no contêiner;

- O contêiner chama o método `ejbCreate()` que inicializa o *bean*;

- O contêiner chama o(s) método(s) de negócios no *bean*;

- Finalmente, o contêiner chama o método `ejbRemove()`. Este método permite que o *bean* possa executar qualquer ação antes que ele seja removido pelo contêiner.

Cuidado deve ser tomado quando os métodos `ejbCreate()` e `ejbRemove()` forem chamados, pois o contêiner pode, na verdade, não criar o *bean* literalmente na memória ou removê-lo dela. O contêiner pode decidir utilizar um *bean* anteriormente criado e que se encontra no *pool*. Isto se deve ao fato de que o *bean* não mantém **estado conversacional** com o cliente permitindo que o contêiner manipule-o de forma que vários clientes utilizem a instância de um mesmo *bean*, por exemplo.

A Figura 2.7 mostra o ciclo de vida de um *Stateful Session Bean*. É possível observar que tanto o ciclo de vida de um *Stateless Session Bean* quanto de um *Stateful Session Bean* são semelhantes, tendo como diferenças o fato de não haver *pool* de instâncias, pois o *bean* mantém o estado conversacional e a há de transições para passivar e ativar estado.

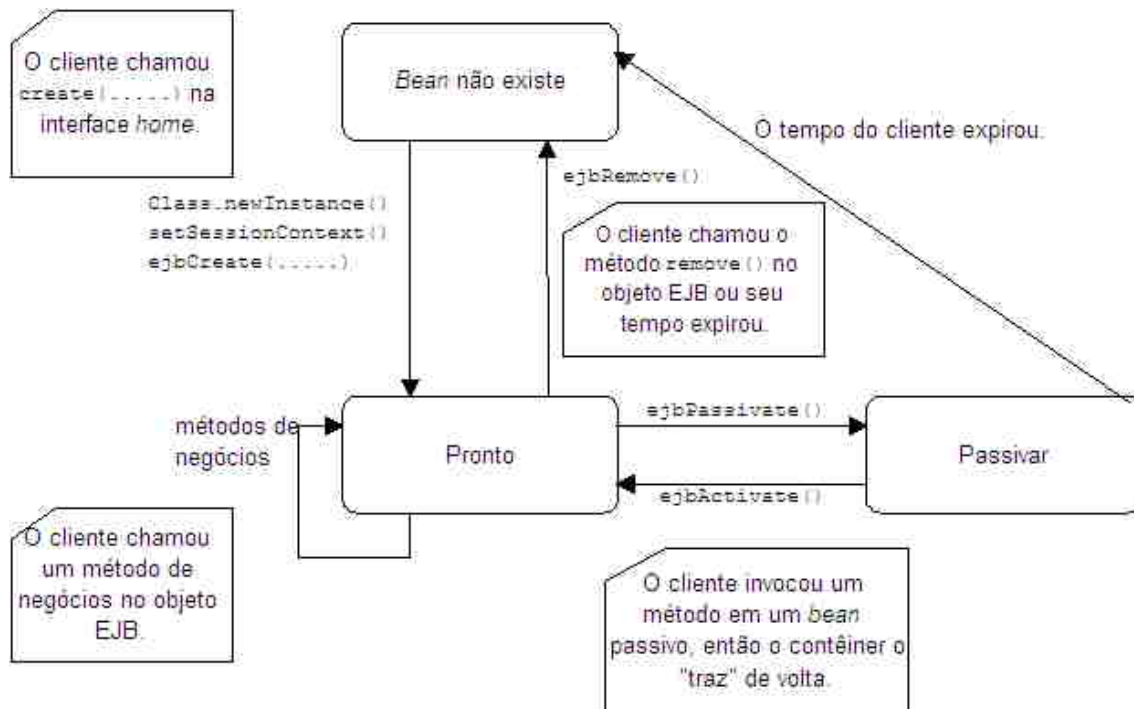


Figura 2.7: Ciclo de vida de um Stateful Session Bean.

O processo de passivação é utilizado pelo Contêiner EJB quando, por exemplo, o cliente aciona um método em um *bean* para o qual não existe um objeto EJB na memória. Quando exemplos como este acontecem, o contêiner, geralmente utilizando o método “Último Recentemente Usado” (LRU – *Last Recently Used*) para selecionar qual *bean* passará do estado ativo para passivo. Imediatamente antes de ocorrer o processo – o estado conversacional do *bean* deve ser armazenado na memória secundária – o contêiner chama o método `ejbPassivate()` para que o *bean* tenha a possibilidade de liberar quaisquer recursos alocados anteriormente. Posteriormente, quando o contêiner recuperar o *bean* da memória secundária, o método `ejbActivate()` será chamado para que os recursos que necessitam ser utilizados sejam novamente alocados para o *bean*.

Os ciclos de vida descritos acima dizem respeito aos *Session Beans*. A seguir, são vistos os ciclos de vida para os *Entity Beans*.

O primeiro ciclo de vida, mostrado na Figura 2.8, trata de um *Entity Bean* cuja persistência é controlada pelo *bean*.

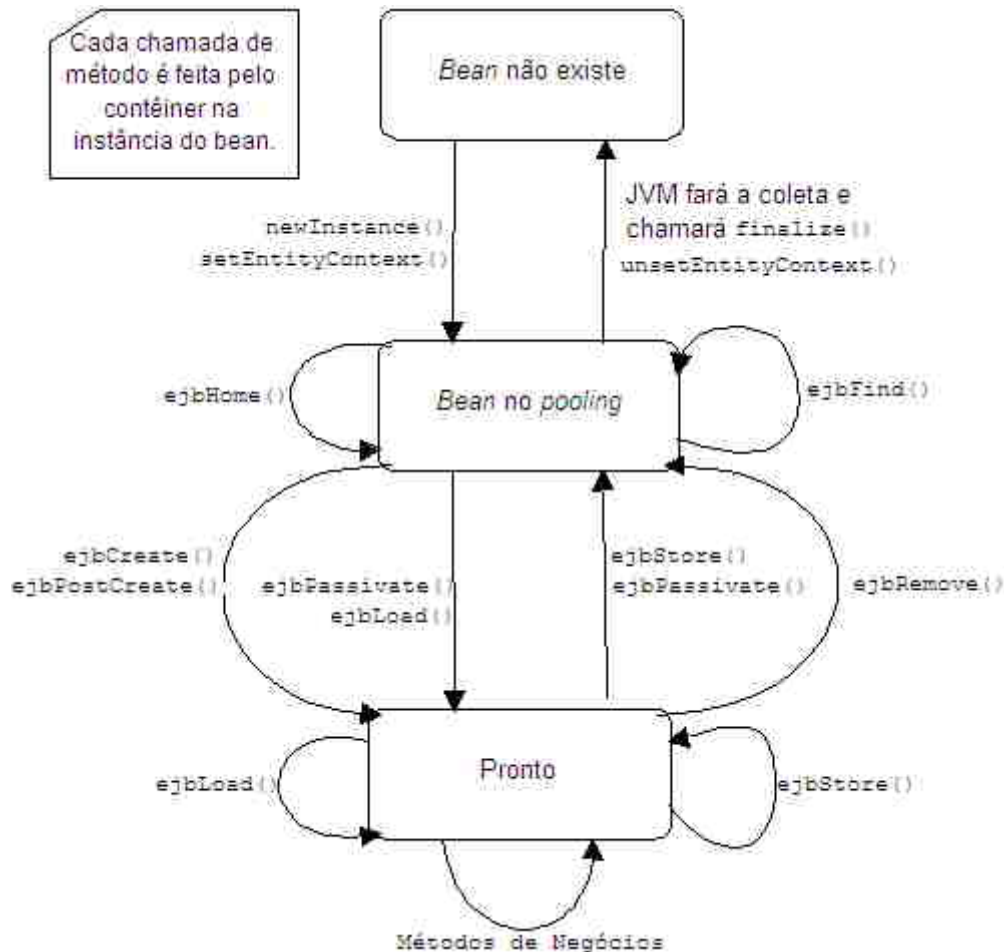


Figura 2.8: Ciclo de vida de um bean com persistência gerenciada a bean.

Durante o seu ciclo de vida, um *bean* do tipo BMP pode estar em um dos três estados possíveis: **não existente**, **no pool de instâncias** ou **pronto**. Para que estes estados sejam alcançados, o *bean* passa pelas seguintes etapas:

- Inicialmente instâncias dos *Entity Beans* não existem;
- A seguir, o container chama o método `Class.newInstance()` passando como argumento o *bean*. Uma vez que ele seja criado, o container invoca o método `setEntityContext()`, no *bean*, passando como argumento um objeto que identifica o seu contexto;

- Uma vez criado, o *bean* é colocado no *pool* de instâncias. Neste estado, o *bean* pode ser utilizado para executar métodos *finder* – *Entity Beans* devem realizar, ao menos, um método *finder*, aquele que localiza dados de objetos do *bean* pela sua chave primária⁶ – ou para executar métodos `ejbHome()` em nome do cliente. Aqui o contêiner pode destruir o *bean* caso seja necessário. Antes que esta remoção ocorra, o contêiner chamará o método `unsetEntityContext()` notificando-o da operação e, assim, permitindo que quaisquer recursos alocados em `setEntityContext()` sejam liberados;

- O cliente chamará o método `create()` na interface remota quando desejar incluir informações na base de dados. A chamada do método `create()` fará com que o contêiner chame o método `ejbCreate()` na interface *home* do *bean*. O método `ejbCreate()` inicializa suas variáveis membro com os dados recebidos como parâmetros e cria uma entrada na base de dados;

- Depois que `ejbCreate()` finaliza sua execução, o *bean* entra no estado pronto. Ele representa um conjunto de dados e um objeto EJB. Eventualmente várias instâncias do *bean* podem representar o mesmo conjunto de dados. Quando isto ocorre, o contêiner necessita sincronizar as várias instâncias. Esta sincronização é realizada através dos métodos `ejbStore()` e `ejbLoad()`. As chamadas a estes métodos são realizadas de acordo com a configuração de transações definidas pelo desenvolvedor;

- Durante o estado de pronto, o *bean* pode retornar ao *pool* de instâncias se o cliente invocar o método `remove()` no objeto *home* – os dados são descartados e a ligação que o *bean* mantinha com o objeto EJB é perdida, ou o tempo do *bean* expira ou o contêiner está com recursos escassos – o contêiner chamará o método `ejbStore()` para garantir que os dados sejam armazenados na base de dados e, sem seguida, o método `ejbPassivate()` permitindo que o *bean* possa liberar quaisquer outros recursos alocados anteriormente;

- Se o *bean* encontra-se no *pool* de instâncias após ter entrado no estado passivo, o contêiner pode reativá-lo novamente chamando os métodos `ejbActivate()` permitindo ao *bean* obter recursos que serão necessários em algum momento de sua existência, e `ejbLoad()` para que os dados sejam carregados da base de dados.

⁶ Conceito aplicado à banco de dados utilizado para compor um conjunto de um ou mais atributos em uma tabela que unicamente identifica uma tupla daquela tabela.

Os passos mostrados acima descrevem, de forma geral, o ciclo de vida de um *Entity Bean* com persistência gerenciada a *bean*. Durante seu ciclo de vida, outras etapas, como sincronização de transações, podem ocorrer.

A seguir, a Figura 2.9 mostra o ciclo de vida de um *Entity Bean* CMP.

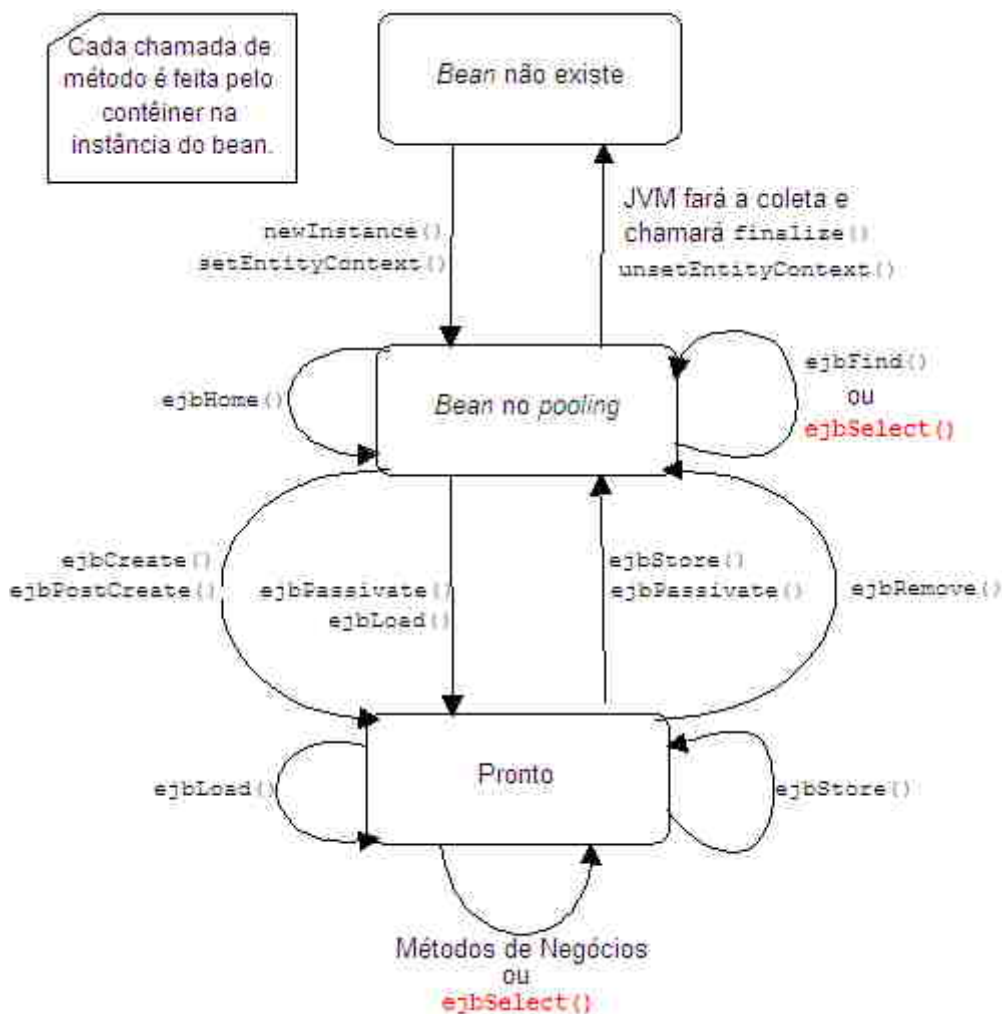


Figura 2.9: Ciclo de vida de um bean com persistência gerenciada a contêiner.

É possível observar que os ciclos de vida de um *bean* BMP e de um *bean* CMP são semelhantes na maioria dos aspectos, devendo ser ressaltado que um *bean* CMP pode possuir o método `ejbSelect()`. Este método é utilizado quando um *Entity Bean*

relaciona-se com outro *Entity Bean* e não é disponibilizado aos clientes do *bean*, pois é utilizado internamente. A utilização deste método pode ser exemplificada quando um *Entity Bean* representando uma conta bancária necessita do saldo da conta e para isto utilizada o método `ejbHomeGetSaldo()`. Não é preciso que o projetista escreva o código utilizando a API JDBC para obter o resultado, basta que ele declare como isto é feito no arquivo descritor e o contêiner se incumba de gerar o código necessário à execução do método `ejbSelect()`.

Finalmente, a especificação EJB versão 2.0 apresenta o *Message-driven bean*. Este *bean* assemelha-se a um *Stateless Session Bean* e suas principais características são as seguintes:

- Uma mensagem pode chegar ao *bean* a partir de vários destinos: do MSMQ (Microsoft), do MQSeries (IBM – (*International Business Machines*)) ou de uma aplicação utilizando a API JMS – detalhada na Seção 2.8. Este *bean* processa mensagens, por isto não possui interfaces *home* ou *local home*, interfaces remota ou *local*;

- Este *bean* possui somente um método de negócios, `onMessage()`, que recebe um único argumento, um objeto `Message JMS`;

- *Beans* deste tipo não retornam valores e não podem levantar exceções capazes de serem “pegas” pelo cliente devido à sua natureza de independência dos produtores de mensagens, mas é possível ao *bean* lançar uma exceção do sistema que será “pega” pelo contêiner;

- Este *bean* pode ser um **assinante durável** ou **não durável**. Se o assinante JMS for um assinante durável ele receberá todas as mensagens mesmo que não esteja ativo. Isto é possível através da persistência da mensagem e posteriormente o seu envio quando o assinante tornar-se ativo. Caso o assinante seja não durável, ele receberá mensagens somente enquanto estiver ativo. As mensagens enviadas ao assinante enquanto ele estiver inativo serão perdidas.

O ciclo de vida de um *Message-driven bean* pode ser visto na Figura 2.10.

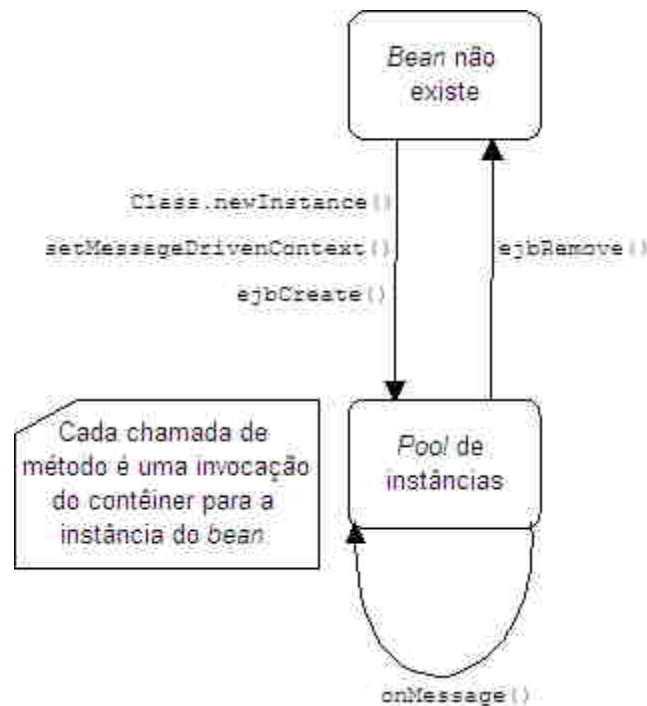


Figura 2.10: Ciclo de vida de um Message-driven bean.

2.6 Java Naming and Directory Interface

Uma aplicação Java tem acesso aos serviços de nomes e de diretório através da API JNDI. Ela fornece um modelo consistente de acesso e manipulação de recursos corporativos tais como DNS (*Domain Name Services*), LDAP (*Lightweight Directory Access Protocol*), sistemas de arquivos locais ou objetos em um servidor de aplicações.

[GOU00] descreve que em JNDI, todo nó em uma estrutura de diretório é denominado **contexto**. Todo nome JNDI é relativo a um contexto; não há noção de nome absoluto. Uma aplicação pode obter seu primeiro contexto utilizando a classe `InitialContext`, da seguinte forma: `Context ctxInicial = new InitialContext();`

A partir do contexto inicial, a aplicação “caminha” pela árvore de diretórios para localizar os recursos ou objetos desejados. Um bom exemplo do funcionamento deste processo é um EJB cuja interface `home` possui o nome

`minhaAplicacao.meuEJB`. Um cliente deste EJB, após obter um contexto inicial, pode localizar a interface *home* utilizando a seguinte:

```
MeuEJBLocal local = ctxInicial.lookup("minhaAplicacao.meuEJB");
```

Uma vez obtida uma referência para o objeto, é possível invocar métodos sobre ele.

2.7 Remote Method Invocation

Invocação Remota de Métodos (RMI – *Remote Method Invocation*) é uma camada de rede que permite a uma aplicação cliente invocar métodos de objetos em máquinas virtuais diferentes não importando suas localizações. RMI não é específica da linguagem Java. O conceito RMI não é recente e tem sido realizado em muitas outras linguagens. Invocação Remota de Métodos pode ser considerada equivalente à Chamada de Procedimentos Remotos (RPC – *Remote Procedure Call*). Com RMI é possível transferir objetos inteiros, incluindo comportamentos – realização, entre cliente e servidor e entre servidor e cliente. Em sistemas baseados em RPC, como DCOM (*Distributed Component Object Model*) e sistemas RPC baseados em objetos, como CORBA, a passagem de argumentos é feita somente com dados.

No contexto da linguagem Java, RMI é considerada como um protocolo que permite a um objeto remoto ser usado como se este estivesse na máquina local. RMI fornece um modelo simples e direto para computação distribuída com objetos Java. Além de permitir a conexão com sistemas existentes ou legados, é possível conectar uma aplicação Java com servidores de bancos de dados através das combinações RMI/JNI (*Java Native Interface*)⁷ e RMI/JDBC, conforme [SM02].

Dentre as várias vantagens do uso da Invocação Remota de Métodos, as seguintes destacam-se:

⁷ Tecnologia que permite a aplicações Java acessarem aplicações e bibliotecas escritas em outras linguagens, C e C++, por exemplo.

- Orientada a objeto;
- Comportamento móvel;
- Utilização de padrões de projeto;
- Segurança;
- Conexão com sistemas existentes e legados;
- Coletor de lixo distribuído;
- Computação paralela;

2.7.1 Arquitetura RMI

O sistema RMI é projetado para ser uma fundamentação direta e simples para a computação orientada a objetos distribuída.

Conforme mostra a Figura 2.11, quando um cliente recebe uma referência a um servidor, a Invocação Remota de Métodos transfere a chamada de um método remoto a um *stub*, que traduz chamadas para aquela referência em chamadas remotas para o servidor. O *stub* organiza os argumentos ao método utilizando serialização de objetos. No lado do servidor, a chamada é recebida pelo sistema RMI e transferida a um *skeleton*, que é responsável por reorganizar os argumentos e invocar a realização do método no servidor. Quando a realização no servidor é concluída, retornando um valor ou levantando uma exceção, o *skeleton* organiza o resultado e envia uma resposta ao *stub* do cliente. O *stub* reorganiza a resposta e devolve o valor de retorno ao cliente.

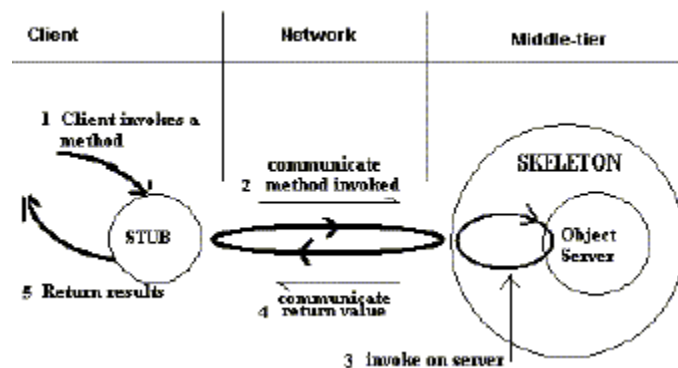


Figura 2.11: Stubs e Skeletons.

Para que o objeto remoto seja localizado no computador no qual reside, é necessário que ele esteja registrado junto a um serviço de nomes. Antes que o cliente possa efetuar chamadas aos métodos do objeto remoto, ele se conecta ao registro de nomes e pede uma referência ao serviço registrado. Se o serviço for localizado, o registro de nomes retorna uma referência remota ao objeto listado sob um dado nome. A obtenção de um objeto remoto através de seu nome é feita através da interface de nomes e diretórios Java, descrita na Seção 2.6.

A seguir, é apresentado um exemplo de uma aplicação que utiliza RMI para comunicar-se com um objeto remoto:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Cliente extends Remote {
    public int remoteHash(String s) throws RemoteException;
}
```

No trecho de código acima, a classe `Cliente` estende a interface `Remote`, indicando que a classe sendo declarada é acessível remotamente. Todas as assinaturas de métodos presentes na interface remota (`Cliente`) devem, no mínimo, ser declaradas como passíveis de lançar a exceção `RemoteException`.

A codificação do objeto remoto deve estender a classe `RemoteObject` ou uma de suas subclasses. Geralmente a classe `UnicastRemoteObject` é utilizada como superclasse. `UnicastRemoteObject` fornece uma realização típica para objetos remotos. O exemplo a seguir mostra a realização da interface `Cliente`, definida acima.

```
import java.rmi.*;
import java.rmi.server.*;

public class ClienteImpl extends UnicastRemoteObject
    implements Cliente {
    public ClienteImpl() throws RemoteException {}
    public int remoteHash(String s) { return s.hashCode(); }
}
```


Uma vez que a interface `Cliente` e a classe `ClienteImpl` estejam prontas, os objetos *stub* e o *skeleton* devem ser gerados. Utilizando o programa `rmic`, que acompanha o Java, a classe `ClienteImpl` é passada como seu argumento e os arquivos `ClienteImpl_Stub.class` e `ClienteImpl_Skel.class` são criados.

As classes `ClienteImpl_Stub` e `ClienteImpl_Skel` contêm o código necessário para a serialização das informações de chamada de método no objeto remoto, invocação dinâmica do método e serialização dos resultados retornados ao cliente. Para executar a chamada remota, estas classes utilizam classes especializadas, `ObjectOutputStream` e `ObjectInputStream`, para envio e recebimento das informações de invocação de métodos e obtenção de resultados pela rede através de um *socket*⁸ TCP/IP (*Transfer Control Protocol/Internet Protocol*).

2.7.2 Comunicação RMI

A comunicação entre cliente e servidor é feita utilizando a técnica mais rápida possível. Esta operação é realizada na primeira tentativa que o cliente faz de conectar-se com servidor através de uma de três possibilidades:

- Utilização de *sockets* para comunicação direta com a porta do servidor;
- Se a opção anterior falhar, uma URL é criada contendo o hospedeiro e a porta do servidor e, usando uma requisição POST⁹ HTTP sobre esta URL, as informações são enviadas ao *skeleton* como parte do POST. Se a requisição POST for completada com sucesso, seu resultado será a resposta ao *stub* do cliente;
- Se esta opção também falhar, uma URL é criada contendo o hospedeiro do servidor e a porta 80, porta padrão do protocolo HTTP, usando um *script*

⁸ Entidade abstrata que permite a comunicação entre duas aplicações, no mesmo computador ou não, via o protocolo TCP ou UDP (*User Datagram Protocol*).

⁹ Uma das duas formas básicas de comunicação entre uma aplicação que utiliza o protocolo HTTP com o servidor.

CGI¹⁰ (*Common Gateway Interface*) que encaminhará o pedido RMI ao servidor.

A primeira das técnicas citadas acima que for completada com sucesso será utilizada em todas as comunicações subsequentes com o servidor. Se qualquer uma destas técnicas falhar, a Invocação Remota de Métodos também falhará.

2.8 Java Message Service

O Serviço de Mensagem Java é uma API que permite que aplicações criem, enviem, recebam e leiam mensagens. A API JMS define um conjunto comum de interfaces que permite a uma aplicação Java comunicar-se com outras realizações de mensagens.

Introduzida em 1998, a API JMS tem como propósito permitir que aplicações Java acessassem sistemas *middleware* orientados a mensagens (MOM – *Middleware Oriented-Message*), como MQSeries da IBM.

Integrada à plataforma J2EE, a API JMS tem as seguintes características:

- Aplicações cliente, EJB e aplicações *Web* podem enviar uma mensagem JMS ou receber uma mensagem sincronamente;
- Aplicações cliente também podem receber mensagens assincronamente;
- Um *Message-driven bean* pode consumir mensagens assincronamente;
- Envio e recebimento de mensagens pode ser realizado em transações distribuídas.

2.8.1 Arquitetura da API JMS

Com a utilização de uma ferramenta administrativa é possível conectar **fábricas** de conexões e destinos dentro do espaço de nomes da API JNDI. Um cliente

¹⁰ Uma das primeiras especificações descrevendo regras de conversação entre aplicações e servidores *Web*.

JMS pode procurar por objetos administrativos no espaço de nomes e então estabelecer uma conexão lógica com estes objetos através de um fornecedor JMS.

De acordo com [HAA02], uma aplicação JMS é composta das seguintes partes:

- Fornecedor JMS: um sistema de mensagens que realiza as interfaces JMS e fornece características administrativas e de controle. Um Fornecedor JMS acompanha a Plataforma J2EE versão 1.3;
- Clientes JMS: programas ou componentes Java que produzem e consomem mensagens;
- Mensagens: objetos utilizados para comunicação entre Clientes JMS;
- Objetos Administrativos: objetos pré-configurados pelo administrador para serem utilizados por clientes;
- Clientes Nativos: programas que utilizam a API de mensagens de outros produtos.

A Figura 2.12 demonstra como estas partes interagem.

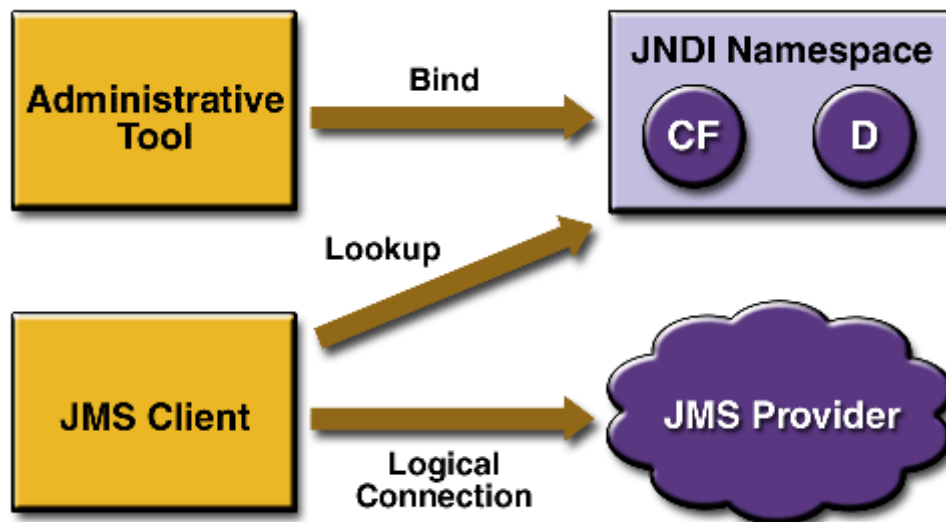


Figura 2.12: Arquitetura JMS.

2.8.2 Domínios de Mensagens

Os domínios *Point-to-Point Messaging* e *Publish/Subscribe Messaging* não eram, ambos, adotados pela maioria dos produtos existentes até o surgimento da API JMS. A especificação JMS fornece domínios distintos para estas abordagens. Um fornecedor JMS pode realizar qualquer uma delas ou ambas. O Fornecedor J2EE realiza ambos.

Um produto ou aplicação *Point-to-Point Messaging* é construída utilizando os conceitos de **fila**, **remetente** e **destinatário**. Cada mensagem é endereçada a uma determinada fila e os clientes destinatários extraem ou consomem as mensagens da fila que foi estabelecida para armazená-las.

A Figura 2.13 ilustra as características do domínio *Point-to-Point Messaging* que são descritas logo abaixo.

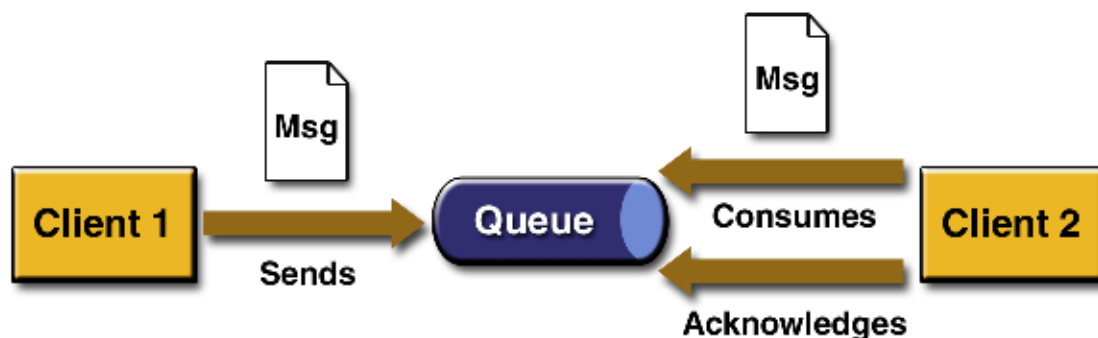


Figura 2.13: Point-to-Point Messaging.

- cada mensagem possui um único consumidor;
- o remetente e o destinatário de uma mensagem não possuem qualquer dependência temporal;
- o destinatário reconhece com sucesso o recebimento da mensagem.

O domínio *Point-to-Point Messaging* é usado quando uma mensagem enviada deve ser processada somente por um único consumidor.

Em uma aplicação *Publish/Subscribe Messaging*, os clientes endereçam mensagens a um **tópico**. Editores e assinantes geralmente são anônimos. O sistema cuida da distribuição de mensagens que chegam de um tópico de múltiplos editores para seus múltiplos assinantes. As mensagens permanecem retidas nos tópicos até que elas sejam distribuídas para seus respectivos assinantes.

As seguintes características são pertinentes ao domínio *Publish/Subscribe Messaging*:

- cada mensagem pode ter múltiplos consumidores;
- editores e assinantes possuem uma dependência temporal. Um cliente que assina um tópico somente pode consumir mensagens após sua inscrição e ele deve continuar ativo para consumi-las. Consumidores com esta dependência são conhecidos como tendo **assinaturas não duráveis**.

A API JMS relaxa esta dependência permitindo que clientes criem **assinaturas duráveis**. Assinaturas duráveis podem receber mensagens mesmo que os assinantes não estejam ativos.

A Figura 2.14 mostra que o domínio *Publish/Subscribe Messaging* é usado em situações nas quais cada mensagem pode ser processada por zero, um ou mais consumidores.

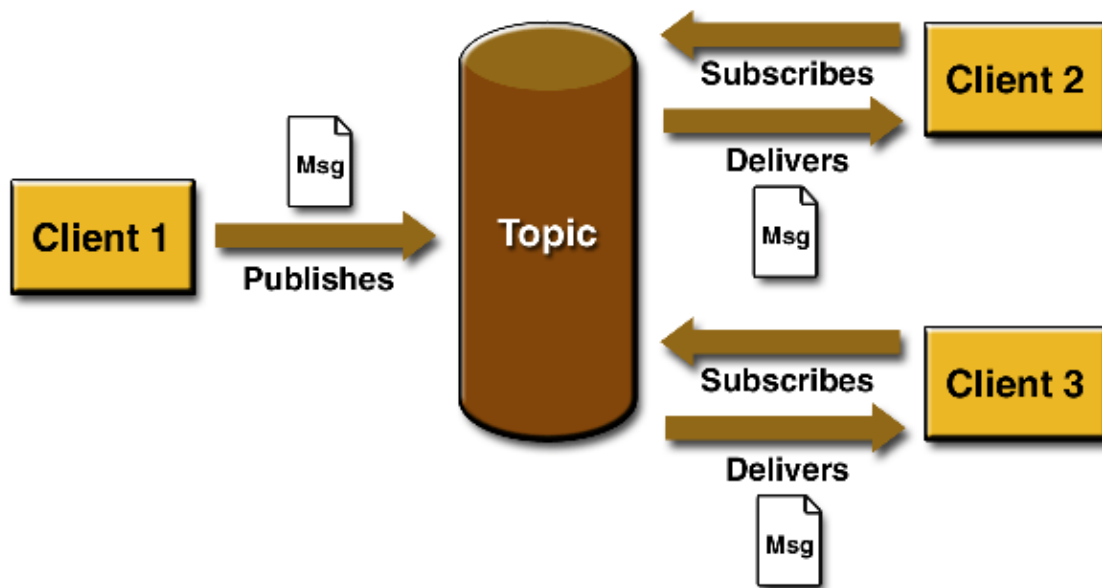


Figura 2.14: Publish/Subscribe Messaging.

A especificação JMS declara que mensagens podem ser consumidas em uma de duas formas:

- Sincronamente: um assinante ou destinatário recupera uma mensagem explicitamente do destino através do método `receive()`. Este método permanece bloqueado até que uma mensagem chegue ou até um tempo máximo seja alcançado caso a mensagem não tenha chegado;
- Assincronamente: um cliente pode registrar um “ouvinte de mensagens” com um consumidor. Sempre que uma mensagem chega ao seu destino, o fornecedor JMS entrega a mensagem através da chamada do método `onMessage()` do “ouvidor”, que age sobre o conteúdo da mensagem.

2.9 Conclusão

A Plataforma 2, Edição Corporativa, um dos três pacotes possível de serem utilizados com a linguagem Java, possui um vasto conjunto de tecnologias para o desenvolvimento de aplicações dos mais variados tipos, indo de simples aplicações *desktop*, passando por aquelas capazes de comunicarem-se via rede até aquelas que

permitem ao projetista valer-se da arquitetura de várias camadas utilizando como aplicações cliente, aplicações *desktop* e navegadores *web* na camada de apresentação, Servlets e *JavaServer Pages* e *JavaBeans* na camada *Web*, *Enterprise JavaBeans* na camada do servidor de aplicações e *Java DataBase Connectivity* para acesso, independente de plataforma, aos dados armazenados na camada de retaguarda – banco de dados.

Todas estas tecnologias associadas a ferramentas de produtividade tais como as de modelagem e ambientes integrados de desenvolvimento – IDEs, permitem ao projetista rapidez e produtividade na criação de aplicações.

Capítulo 3

Arquiteturas de Componentes de Software

3.1 Introdução

Ao longo do tempo, a indústria de software passou por uma evolução no desenvolvimento de aplicações. A criação de aplicações de software era desordenada e necessitava ser formalizada com novas técnicas de desenvolvimento. Na primeira fase, o desenvolvimento estruturado definiu padrões e antipadrões que estruturaram formalmente os programas. O objetivo desta técnica consistia na decomposição do problema em partes cada vez menores até elas fossem suficientemente simples para serem codificadas. A decomposição poderia ser invalidada ou não ser ótima caso os requisitos mudassem, visto que ela era realizada no início do processo de desenvolvimento. Outro problema era o fato de que a estrutura projetada não possuía relação direta com o domínio do problema, acarretando em dificuldades de manutenção e melhoramentos no projeto. Na segunda fase, o projeto orientado a objetos apresentou vantagens mais claras entre os modelos de análise do domínio do problema, os projetos e a codificação. Projetos orientados a objetos apresentam uma solução à cerca do domínio do problema, o que leva a uma melhor documentação e a sistemas melhor mantidos. O problema desta abordagem está na necessidade de um ambiente de desenvolvimento rigoroso e formal, pois abstrações corretas são difíceis de serem obtidas. Na fase atual encontra-se o desenvolvimento baseado em componentes. Nesta técnica um sistema é considerado como um conjunto de pequenos subsistemas [CG00]. Esta descrição é mostrada na Figura 3.1, a seguir.

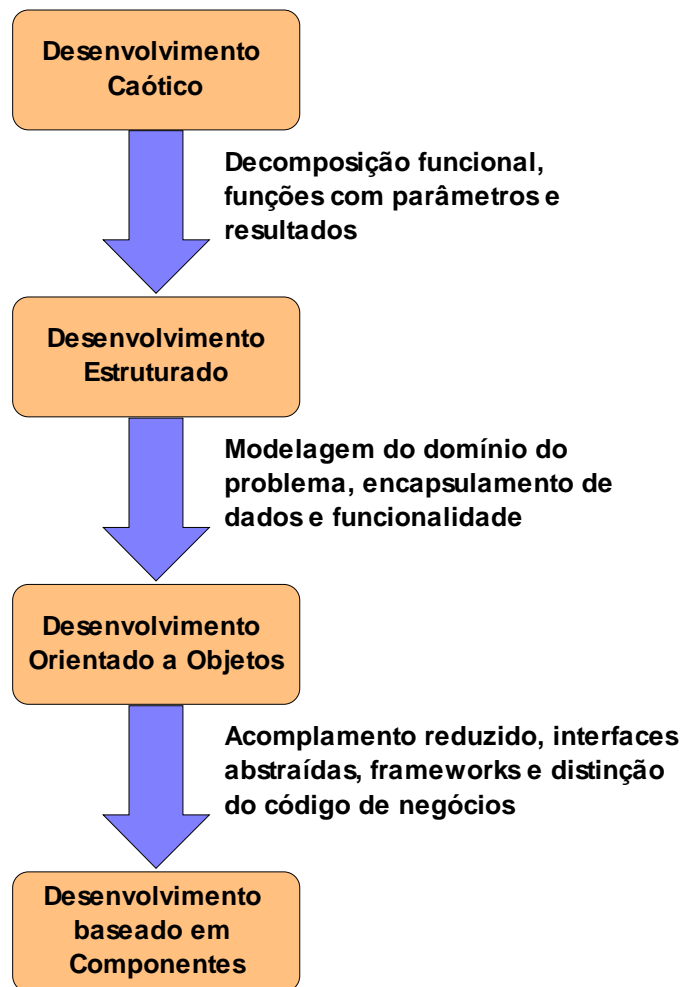


Figura 3.1: Evolução do Desenvolvimento baseado em Componentes.

A idéia por trás da Arquitetura de Componentes de Software está na criação de unidades de software reusáveis e independentes do contexto da aplicação – ou domínio do problema – na qual são utilizados. A independência de contexto é importante, pois um componente pode ser facilmente transportado de um contexto de aplicação para outro ou substituído por outro de igual funcionalidade, mas com diferentes ou melhores qualidades [ZMF99].

O objetivo dos componentes de software é permitir que aplicações possam ser “montadas” a partir de componentes de “prateleira”, cuja funcionalidade não precisa ser codificada a cada novo sistema.

Assim, características como funcionalidade e modularidade são essenciais em componentes de software. Partindo destes conceitos, pode-se agrupar um conjunto de unidades de software em um *framework* e este, por sua vez, utilizado em domínios de aplicação específicos, por exemplo, ambientes gráficos ou domínios de mais alto nível como em aplicações comerciais.

É importante observar que componentes podem ser definidos como tendo todas as características de objetos, serem limitados ao *framework* no qual eles existem, estarem isolados de outros componentes exceto daqueles componentes do *framework* por eles utilizados, possuem interface clara e comum e serem autodescritos; e *frameworks* criam instancias de componentes em tempo de execução, permitem que componentes possam encontrar e comunicar-se outros componentes e fornecem serviços de uso comum tais como: persistência, transação, independência de localização, segurança e monitoramento [CG00].

O desenvolvimento de componentes de software e sua conseqüente aceitação depende de um conjunto de padrões e de especificações que seja amplamente aceito pela comunidade de fabricantes e de desenvolvedores.

Tendo em vista que a arquitetura de componentes corporativos *Enterprise JavaBeans* foi descrita na Seção 2.5, a seguir serão abordadas algumas outras arquiteturas de componentes de software consideradas de relevância.

3.2 *Component Object Model* – COM

Este modelo de programação baseada em objetos permite o desenvolvimento de componentes de software que podem ser integrados no ambiente hospedeiro [RE97]. Alguns conceitos importantes no entendimento desta tecnologia são apresentados no decorrer desta seção.

3.2.1 Interfaces

Uma interface define o comportamento ou capacidades de um componente de software como um conjunto de métodos e propriedades, e é um contrato que garante a consistência semântica dos objetos que a suportam. Embora um objeto COM possa suportar diversas interfaces, ele deve suportar, ao menos, a interface `IUnknown`. Esta interface fornece funcionalidade básica para o gerenciamento do ciclo de vida dos objetos. O método `QueryInterface` é utilizado por clientes para identificar se uma determinada interface, identificada pelo seu IID (*Interface Identifier*), é suportada por um objeto. O valor de retorno de `QueryInterface` é um ponteiro de interface que, por sua vez, aponta para uma estrutura de dados especificada pelo padrão de interoperabilidade binária. Este padrão estabelece como as funções da interface devem ser acionadas a despeito das diferenças de codificação dos programas cliente e servidor.

As interfaces são descritas através da Linguagem de Definição de Interfaces da Microsoft (MIDL – *Microsoft Interface Definition Language*). O compilador MIDL gera o código *proxy*, utilizado pela aplicação cliente, e *stub*, utilizado no servidor, em C ou C++ a partir da definição da interface.

Para evitar conflitos de nomes, um identificador universalmente único (UUID – *Universally Unique Identifier*) precisa ser criado para cada interface. Este identificador é denominado IID.

3.2.2 Classes e Servidores

Classes COM, entre outras coisas, são as realizações de um ou mais interfaces COM e fornecem funções em quaisquer linguagens que são suportadas. Cada classe COM possui um identificador único, da mesma forma que interfaces COM, denominado CLSID (*CLaSS IDentifier*). Para que uma aplicação cliente possa interagir com um componente é necessário que ela conheça, ao menos, um CLSID e um IID para uma interface que é realizada pela classe. Com esta informação, é possível criar um objeto e obter um ponteiro de interface.

Classes COM podem ser empacotadas em um servidor de várias formas. Um servidor COM pode ser empacotado para ser utilizado de duas maneiras: 1) como uma DLL (*Dynamic Linking Library*). Este tipo de servidor é denominado *in-process* e o arquivo DLL é carregado no mesmo espaço do processo cliente quando uma classe é acessada; 2) como um executável que pode ser carregado no mesmo computador que a aplicação cliente ou em um computador remoto. Neste caso, o servidor, denominado *out-of-process*, é acessado usando DCOM, a ser visto na Seção 3.3.

Em servidores *in-process*, as chamadas feitas pelo cliente utilizando um ponteiro de interface vão diretamente a um objeto, que é criado no processo do cliente, enquanto que em servidores *out-of-process*, a chamada passa por um objeto *proxy in-process* encarregado de realizar a requisição usando chamada remota de métodos para o servidor. No lado do servidor, um objeto *stub* recebe a requisição e a encaminha para o objeto COM apropriado. Estes processos são descritos pela Figura 3.2, a seguir.

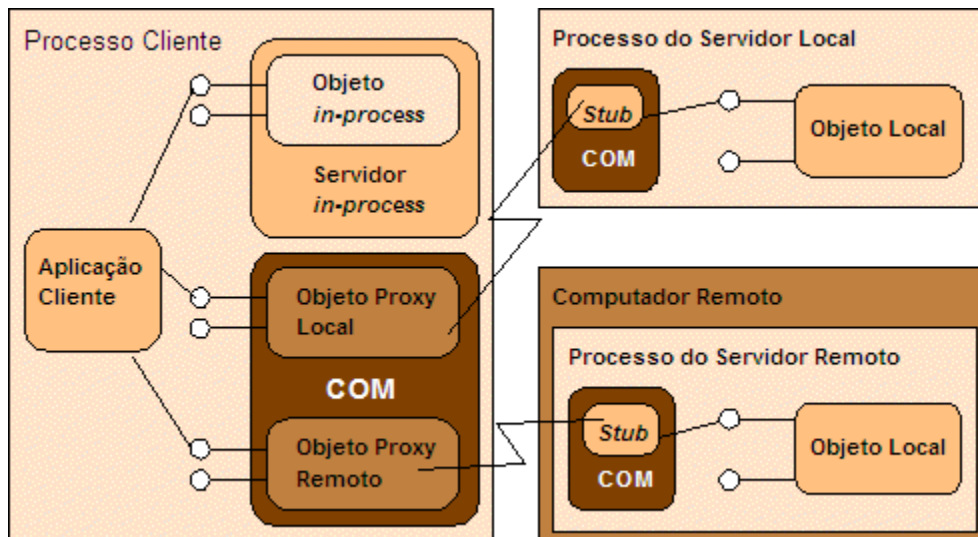


Figura 3.2: Cliente COM interagindo com objetos COM empacotados de várias formas.

3.2.3 Ciclo de Vida de Objetos

Na especificação COM cada classe COM está associada a outra classe COM denominada classe *factory*. Esta classe é incumbida de criar instâncias de classes COM. A classe *factory* realiza a interface padrão *IClassFactory*, definida na especificação COM. Este mecanismo é mostrado na Figura 3.3.

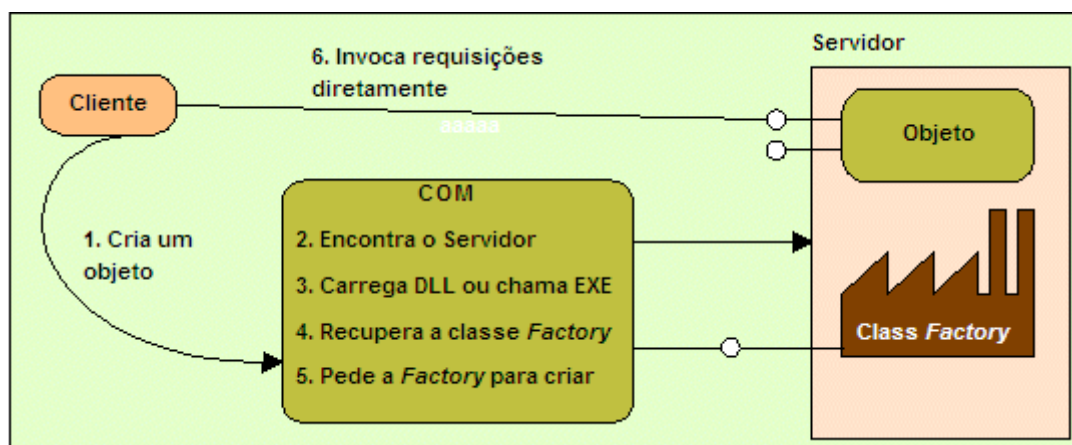


Figura 3.3: Mecanismo COM da Classe Factory.

O tempo de duração de um objeto COM é controlado por uma referência contada, que é manipulada pelos métodos `AddRef()` e `Release()`, definidos na interface `IUnknown`. Classes COM realizam estes métodos como forma de rastrear o número atual de suas instâncias. É possível para o projetista criar um único contador para o objeto ou contadores individuais para cada interface que o objeto suporta. O método `AddRef()` deve ser chamado por quaisquer métodos que retornam ponteiros para interface, incluindo `QueryInterface`, para indicar que uma nova referência a um objeto. Quando um ponteiro para interface não for mais usado, o método `Release()` deverá ser chamado.

3.2.4 Interoperabilidade Binária

Para permitir a escrita de componentes para diferentes ambientes, a especificação COM define um padrão de chamada binária. Este padrão descreve o *layout* da pilha de chamadas para todas as invocações de métodos.

O padrão binário define um ponteiro para interface como sendo um ponteiro que aponta para uma tabela de funções. Em servidores *in-process*, a tabela de funções lista as funções do objeto real, enquanto que, em servidores *out-of-process*, a tabela de funções lista as funções do objeto *proxy*.

3.2.5 Transparência de Empacotamento

A transparência de empacotamento garante que um cliente COM não precisa se preocupar com o tipo de empacotamento do servidor, se uma DLL ou um executável ou sua localização. Para a criação de um objeto COM, basta que a aplicação cliente forneça o CLSID da classe desejada, pois o código para acesso a servidores *in-process* e servidores *out-of-process* é idêntico.

A forma de localização de servidores dá-se por informações armazenadas em um registro local. Assim, a biblioteca COM pode encontrar a classe requisitada e estabelecer a conexão entre cliente e servidor por meio de informações como: o tipo do servidor e a localização de sua DLL ou executável.

Esta localização é realizada pelo Gerente de Controle de Serviço (SCM – *Service Control Manager*), que busca no registro local pelo CLSID para que o servidor possa ser ativado.

3.3 *Distributed Component Object Model* – DCOM

O modelo de objeto de componentes distribuídos estende o modelo visto na Seção 3.2 acrescentando uma melhor transparência de localização de servidores e seu empacotamento, modelos de linhas de execução adicionais, opções de segurança e capacidades administrativas, pois, neste modelo, os objetos estão distribuídos pela rede.

3.3.1 Transparência de Localização e Empacotamento

Na arquitetura COM, uma aplicação cliente é construída independente do empacotamento utilizado pelo servidor e seus objetos podem ser instanciados juntamente com o processo do cliente ou executados em um processo separado no mesmo computador. O modelo DCOM estende esta capacidade permitindo que objetos possam residir em qualquer ponto da rede.

Quando uma aplicação cliente requisita uma classe *factory* para um objeto COM, o SCM, visto na Seção 3.2.5, aciona o SCM no computador remoto, que localiza e ativa o servidor retornando uma conexão RPC à classe *factory* fornecida pelo servidor. Um objeto *proxy* para a classe *factory* é criado e a criação do objeto segue os métodos descritos nas Seções 3.2.3 e 3.2.5.

O acesso a objetos DCOM em computadores remotos pode ser feito sem codificação específica para comunicação via rede por meio de configurações administrativas. É possível também especificar o hospedeiro remoto no qual um objeto DCOM será criado por meio de configurações de aplicação, realizadas pelo usuário final.

3.3.2 Modelo de Fluxo de Execução Livre

A arquitetura DCOM permite a criação de servidores que suportem várias linhas de execução como extensão ao modelo proposto pela arquitetura COM, em todos os modos de empacotamento vistos na Seção 3.2.5.

Antes do advento do modelo COM distribuído, servidores COM estavam limitados a servidores que seguiam um modelo de linha de execução denominada *apartment*. Neste modelo um objeto COM pode ser acessado somente por uma linha de execução, aquela que o criou. A partir da versão 4 do Windows NT, um novo modelo, denominado modelo de linhas de execução livre, foi introduzido.

Com o modelo de linhas de execução livre, cada requisição a objetos é tratada por uma fluxo de execução e várias requisições a um único objeto são tratadas, cada uma, por sua própria linha de execução. Para assegurar-se que todos os objetos são *thread-safe*, o desenvolvedor necessita utilizar funções de baixo nível para garantir a concorrência, tais como **mutexes**¹¹.

3.3.3 Segurança

Como em qualquer arquitetura distribuída, a segurança é um item de suma importância. O modelo DCOM permite que a segurança de um componente possa ser ajustada pelo desenvolvedor ou pelo administrador do ambiente no qual o componente será utilizado. Desta forma, é possível a um mesmo componente ser executado em um ambiente sem nenhum requisito de segurança e em outro que exija um grau segurança muito maior.

A segurança em objetos COM é feita através de Listas de Controle de Acesso (ACL – *Access Control Lists*). Se um determinado usuário não possui permissão sobre um componente, o acesso a este será negado antes que qualquer código no componente seja executado. O desenvolvedor pode obter um alto grau de segurança codificando

¹¹ O termo mutex ou **mutual exclusion** objet – objeto de exclusão mútua – é utilizado em programação concorrente para denotar que um dado recurso será compartilhado por várias linhas de execução de um programa, mas seu acesso não será simultâneo.

políticas de segurança customizadas ao nível de chamada de métodos utilizando, para isto, chaves no registro juntamente com funções da API do Windows.

3.3.4 Contagem de Referência e *Pinging*

Pelo fato de utilizar contagem de referência nos objetos para determinar quando estes devem ser destruídos, a especificação DCOM executa periodicamente verificações para determinar se a aplicação cliente ainda está conectada. Para evitar o alto tráfego na rede, DCOM envia uma mensagem entre os computadores e se um determinado cliente pára de responder, o método `Release()` será invocado nos ponteiros de interface no servidor.

3.3.5 Administração

O gerenciamento e configuração de grandes ambientes de objetos distribuídos podem tornar-se difícil à medida que a quantidade de componentes e servidores aumenta. Mesmo com o auxílio de ferramentas dedicadas a estes trabalhos, tais tarefas não são triviais. Para estes ambientes o balanço de carga das requisições dos clientes entre vários servidores é a maior preocupação dos gerentes de rede. Infelizmente, DCOM não possui suporte para o controle do balanço de carga.

Se a aplicação cliente não possui balanço de carga dinâmico, fica a cargo do gerente de redes providenciar configuração necessária para que determinados computadores clientes utilizem um certo servidor. Quando um componente é transportado de um computador para outro, as referências para o novo computador precisam ser atualizadas. Da mesma forma, acontece com a propensão a falhas, DCOM não especifica uma maneira simples para um cliente procurar um servidor em um lista de servidores ao invés de um computador específico.

3.4 Common Object Request Broker Architecture – CORBA

No modelo de componentes CORBA um componente é um meta-tipo básico. Um meta-tipo componente é uma extensão e uma especialização de um meta-tipo objeto. Tipos componentes são especificados por meio da Linguagem de Definição de Interfaces (IDL – *Interface Definition Language*)¹² e representados no Repositório de Interfaces. Um componente é representado por uma referência de componente que é representado por uma referência de objeto. Seguindo esta correspondência, uma definição de componente é uma especialização e uma extensão de uma definição de interface [OMG02].

Como parte do modelo de referência da arquitetura CORBA, o *Object Request Broker* – ORB – permite que objetos possam fazer requisições e receber respostas em um ambiente distribuído. Com os ORBs é possível a aplicações que utilizam objetos distribuídos interagirem em ambientes homogêneos e heterogêneos. A Figura 3.4 demonstra a requisição de um objeto, através do ORB, por um cliente.

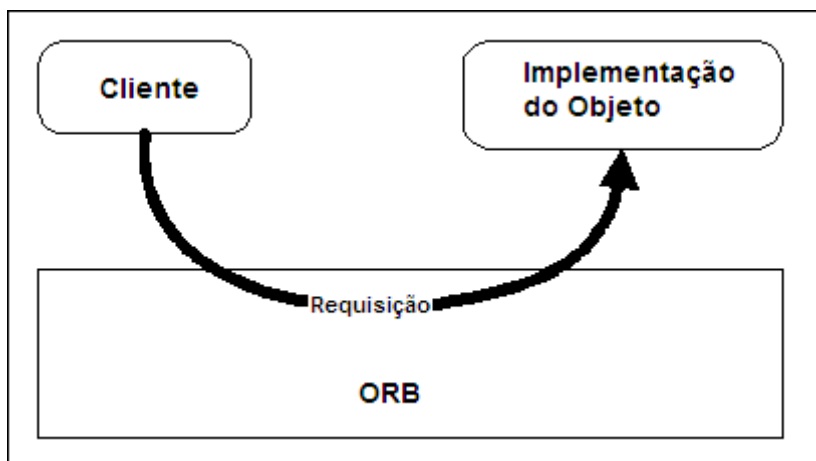


Figura 3.4: Requisição de um objeto via ORB.

O ORB é responsável por particionar as interações entre objetos fornecendo transparência de localização e acesso aos mesmos

¹² Linguagem descritiva orientada a objetos para especificação de interfaces no lado do servidor.

Na arquitetura de gerenciamento de objetos da OMG (*Object Management Group*), além do ORB, também existem especificações para Serviços de Objetos de baixo nível, Facilidades Comuns de alto nível para o desenvolvimento de aplicações, como por exemplo: serviços de nomes para localização de objetos, serviços de eventos para a assinatura e publicação de mensagens, segurança, suporte a transações, entre outros.

3.4.1 CORBA no lado do Servidor

A partir da especificação de uma interface, o compilador IDL mapeia as definições para uma linguagem de programação para o desenvolvimento do aplicativo servidor. A Figura 3.5 ilustra a descrição de uma interface em IDL.

```
// IDL
interface ContaCorrente {
    // atributos
    attribute float saldo;
    readonly attribute string correntista;

    // operações
    void depositar(in float valor, out float novoSaldo);
    void sacar(in float valor, out float novoSaldo);
};
```

Figura 3.5: Descrição de uma Interface em IDL.

Conforme mostra a figura acima, em IDL *interface* corresponde a uma classe, *attribute* é mapeado para métodos de acesso – *getter/setter*, *readonly* é mapeado somente para o método que retorna o valor do atributo, *operation* corresponde a um método e parâmetros devem utilizar *in*, *out* ou *inout*, para indicar o modelo de passagem de parâmetros. Além destas características, outras como herança, módulos, definições de exceções e quais métodos podem lançá-las e códigos de tipos também estão presentes em IDL.

De posse da descrição de uma interface, o compilador IDL gera o *stub* do servidor, também conhecido como *skeleton*, que será ligado ao programa servidor, a interface em linguagem nativa para o desenvolvimento do programa servidor e do *stub* para a aplicação cliente. O *stub* do servidor fornece interfaces estáticas para a chamada de métodos nos objetos. Ele é responsável por obter os métodos e seus parâmetros que vêm do cliente via ORB. O processo de geração dos *stubs* e da interface em linguagem nativa é ilustrado na Figura 3.6.

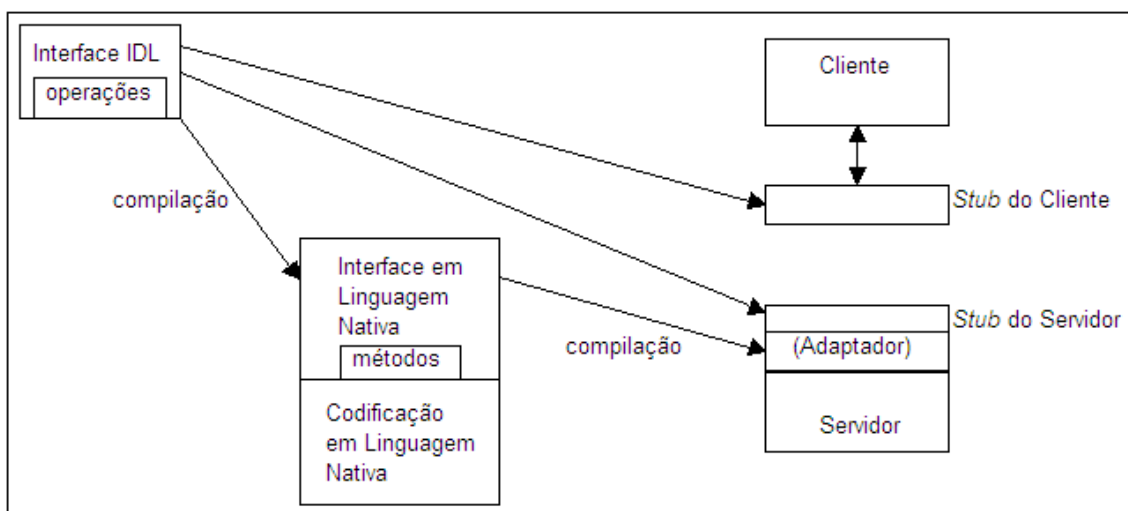


Figura 3.6: Geração de *stubs* e interface em linguagem nativa a partir da descrição IDL.

Um componente importante do ORB é o Adaptador de Objetos, responsável por gerenciar os objetos do servidor, bem como objetos relacionados.

CORBA utiliza o Repositório de realização para localizar o nome do servidor e executar comandos, identificar o modo de ativação e determinar as permissões de execução e de acesso. Os modos de ativação identificam as características em tempo de execução de um servidor, se um processo servidor pode conter um ou mais objetos e se vários clientes têm permissão de acesso ao mesmo objeto.

3.4.2 CORBA no lado do Cliente

O *stub* do cliente, gerado pelo compilador IDL, será utilizado para mapear um objeto CORBA no lado do servidor a um objeto na linguagem da aplicação cliente. Sua função é organizar os métodos e parâmetros a serem transmitidos pelo ORB.

Um servidor pode ser localizado obtendo uma referência de objeto diretamente do servidor, utilizando o serviço de nomes CORBA para localizar um objeto pelo seu nome ou para obter referências de objetos que combinem com um dado critério de busca. A especificação CORBA define o formato para um Referência de Objeto Interoperável (IOR – *Interoperable Object Request*), que pode ser utilizado entre os diferentes ORBs. Após obter a referência de objeto, uma aplicação cliente pode invocar métodos em objetos do servidor por meio dos seguintes modos de chamada: [EMC02]

- **sincronamente:** o cliente envia um pedido e aguarda até que uma resposta seja retornada;
- **mão-única / *poll*:** o cliente envia a requisição mas não espera pela resposta. O cliente continua até que o servidor finalize o processamento da requisição;
- **mão-única / *callback*:** o cliente envia uma requisição passando uma referência *callback* de objeto. Quando o servidor completar a requisição, ele invocará um método no objeto *callback*. Isto implica em que o cliente se comportará como um servidor dirigido a evento com uma interface IDL;
- **mão-única / serviço de eventos:** utilizando o serviço de eventos CORBA, o modelo publicar-assinar é empregado de tal forma que o cliente inicialmente pede para ser notificado pelo serviço de eventos quando uma mensagem de término é despachada. Desta forma, o cliente envia uma requisição ao servidor e quando este finalizar o processamento, ele envia uma mensagem de término ao serviço de eventos que, por sua vez, notifica o cliente.

3.4.3 Protocolo Internet Inter-ORB

Na versão 1.0 da especificação CORBA, a forma como os ORBs se comunicavam pela rede era deixada a cargo de seus respectivos fabricantes. Com a versão 2.0, uma metodologia para comunicação entre diferentes ORBs foi apresentada. O Protocolo Geral Inter-ORB (GIOP – *General Inter-ORB Protocol*), também conhecido como Arquitetura de Interoperabilidade ORB, descreve uma coleção de requisições de mensagens utilizadas pelos ORBs para comunicação pela rede.

O Protocolo Internet Inter-ORB (IIOP – *Internet Inter-ORB Protocol*) é uma versão padronizada da realização GIOP que mapeia mensagens GIOP sobre o protocolo TCP/IP, utilizando a Internet como barramento de comunicação ORB.

3.5 *JavaBeans*

Basicamente, *JavaBeans* são classes Java que seguem um conjunto de padrões de projeto que os torna fáceis de serem utilizados com ferramentas de desenvolvimento e com outros componentes [HDF03].

Uma classe Java torna-se um *JavaBean* quando ela é concreta e pública, possui um construtor sem parâmetros, podendo ser um construtor implícito, e permite acesso a alguns ou a todos os seus atributos, disponibilizando acesso ao seu estado interno por meio de padrões de projeto. Atributos que são expostos por um *JavaBean* também são conhecidos como **propriedades**.

A especificação *JavaBeans* descreve que o acesso às propriedades de um *JavaBean* é realizado por meio de métodos de acesso denominados *setters* e *getters*.

Os métodos de acesso às propriedades de um *JavaBean* possibilitam ajustar ou obter o valor de um determinado atributo. O método que ajusta o valor de uma propriedade deve, segundo a especificação, ter seu nome iniciado com *set*, seguido pelo nome do atributo, com sua primeira letra em maiúsculo, cujo valor será ajustado.

Estes métodos devem ser públicos e o valor de retorno deve ser `void`. Geralmente, estes métodos recebem apenas um único parâmetro como valor de entrada, sendo este o valor a ser atribuído ao atributo designado pelo nome do método. Como exemplo, tem-se um atributo hipotético cujo nome de identificador é **altura** e como tipo básico o tipo **double**. O método *setter* para este atributo deve ser `public void setAltura(double altura)`. Embora não esteja especificado, normalmente o nome de identificador do parâmetro é o mesmo nome do atributo.

Em contra-partida, o método que obtém o valor de uma propriedade deve, ainda segundo a especificação, ter seu nome constituído de `get` precedendo o nome do atributo, cuja primeira letra deve estar em maiúsculo. Além de ser declarado como público, um método *getter* deve possuir, como valor de retorno, um tipo compatível com o tipo usado na declaração do atributo. Utilizando o mesmo atributo exemplo, citado no parágrafo anterior, tem-se o método `public double getAltura()` para obter o valor do atributo especificado. Para atributos simples, métodos *getters* não tomam parâmetros em suas declarações.

Uma exceção a esta regra diz respeito a atributos que retornam valores lógicos do tipo `boolean` ou `Boolean`. Para estes atributos, ao invés de `get`, o prefixo `is` deve preceder o nome do atributo. Assim, o método de acesso a um atributo cujo nome de identificar é **pessoaFisica** deve ser `public boolean isPessoaFisica()`.

Estas convenções de nomeação de métodos, descritas na especificação *JavaBeans*, proporcionam a ferramentas de construção de interfaces gráficas e a diversos outros tipos de aplicativos e ferramentas de software de desenvolvimento de aplicativos a fazerem introspecção, utilizando a API de Reflexão Java, procurando por métodos com nomes iniciados com `set`, `get` ou `is`. Como exemplo de ferramentas, tem-se ambientes de desenvolvimento de aplicações gráficas em Java, como *JBuilder*, Projeto Eclipse, NetBeans, *JDeveloper*, *frameworks* para o desenvolvimento de aplicações para Internet, como *Struts* e a tecnologia de desenvolvimento de páginas para Internet *JavaServer Pages*, entre outros.

3.6 Conclusão

Dentre os modelos de componentes de software abordados neste capítulo, o que mais se destaca é aquele oferecido pela especificação CORBA. Este modelo apresenta uma distribuição de objetos em rede e que é independente da plataforma utilizada para a comunicação entre a aplicação cliente e os objetos que residem em um computador remoto.

Conforme descrito na Seção 3.4.3, até a versão 2.0 não era possível a ORBs de diferentes fabricantes conversarem “entre si”. A partir da versão 2.0, esta barreira foi transposta. É importante ressaltar, também, que a introdução do protocolo IIOP permitiu que as mensagens utilizadas pelos ORBs pudessem ser mapeadas sobre o protocolo TCP/IP permitindo que objetos remotos pudessem ser acessados utilizando a rede mundial – Internet – como meio de comunicação.

Apesar de ter sido apresentado na Seção 2.5, o modelo de componentes distribuídos descrito nesta seção pode ser considerado como uma evolução sobre o modelo proposto pela especificação CORBA, pois além de ser acessível a aplicações Java, por meio do protocolo RMI-IIOP, é possível a aplicações não Java acessarem tais componentes utilizando o protocolo IIOP, pois é garantido pela especificação EJB que este modelo de componentes é compatível com CORBA.

Capítulo 4

Estudo de Caso: Sistema de Controle Imobiliário

4.1 Introdução

O estudo de caso, sob a forma de uma aplicação, apresentado neste capítulo, demonstra a utilização da plataforma Java 2, versão corporativa, e de suas tecnologias. Para esta aplicação foram utilizadas algumas das tecnologias que a plataforma disponibiliza para a criação de aplicações distribuídas tais como **Servlets**, **JavaServer Pages** e **Enterprise JavaBeans**.

A aplicação segue o Modelo 2 da especificação JSP, conforme apresentado na Seção 2.4. Desta forma, ela é dividida em três camadas distintas: **apresentação**, **lógica de negócios** e **acesso ao sistema de retaguarda (base de dados)**.

A modelagem apresentada neste capítulo trata apenas de uma parte das funcionalidades de um sistema para controle imobiliário, o que não prejudica a demonstração das tecnologias abordadas neste trabalho. É importante ressaltar também que a funcionalidade de autenticação de usuários na interface administrativa foi adicionada como um item extra para diferenciar entre as interfaces administrativas e de pública.

4.2 Descrição da Aplicação

A aplicação trata do controle imobiliário para empresas que administram, vendem e locam imóveis, residenciais e comerciais, e possui duas interfaces: a primeira, denominada **interface administrativa**, mostrada abaixo na Figura 4.1, é utilizada pela empresa e possui acesso restrito, permite a manutenção de proprietários e seus

respectivos imóveis, enquanto a segunda, denominada **interface pública**, mostrada na Figura 4.2, é utilizada por usuários em geral, fornece consultas aos imóveis previamente cadastrados. Estas interfaces são alcançadas por meio de diferentes endereços – URLs. Assim, usuários que querem consultar a base de dados de imóveis não são levados às páginas de acesso restrito do sistema.

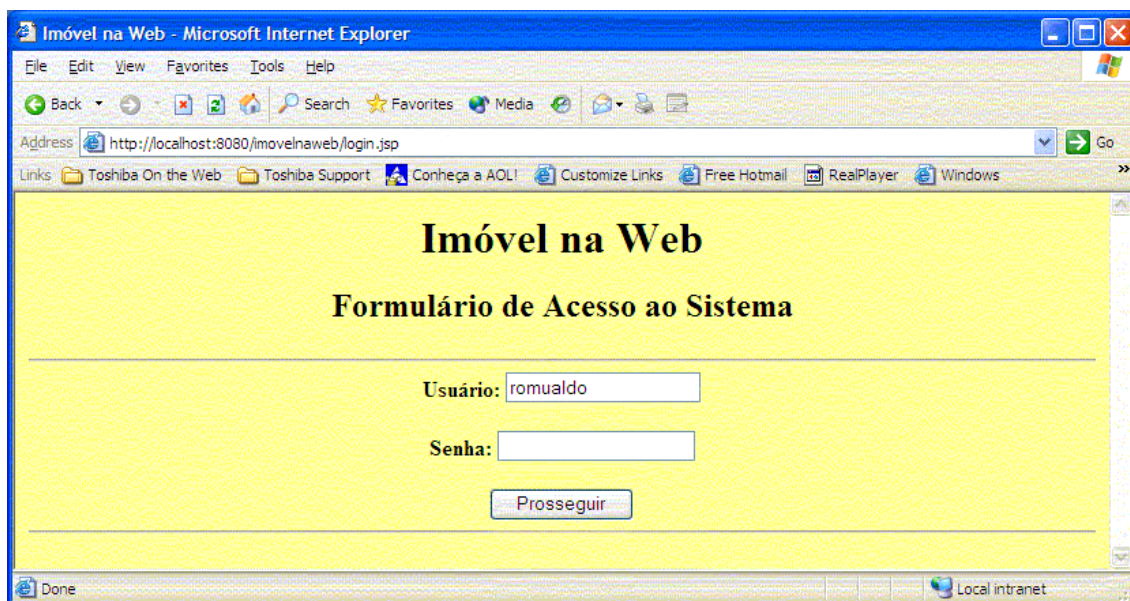


Figura 4.1: Interface administrativa do Sistema de Controle Imobiliário.

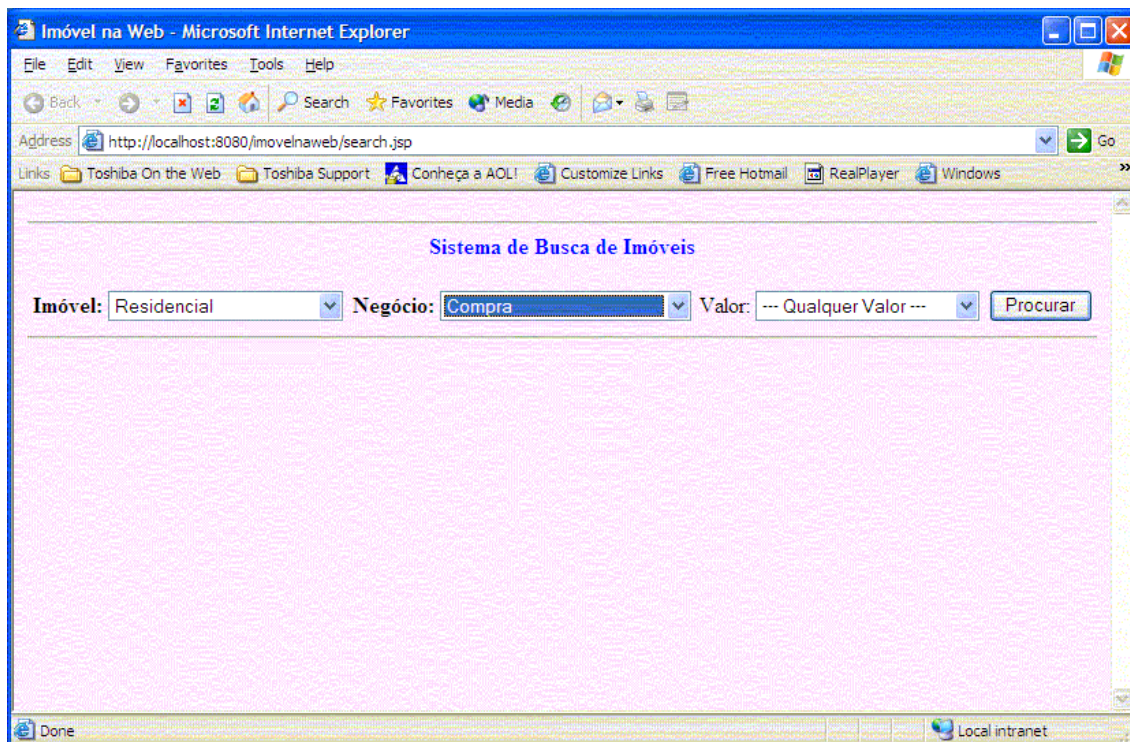


Figura 4.2: Interface pública do Sistema de Controle Imobiliário.

Para que usuários possam efetuar consultas aos imóveis cadastrados, eles devem acessar o sistema via *web*, com o auxílio de um navegador. Da mesma forma, a interface que permite a manutenção no sistema também é utilizada via *web*.

Na interface pública, a mais simples do ponto de vista funcional, o usuário está à procura de um imóvel, residencial ou comercial, para compra ou locação. Além destes dados, ele pode informar a faixa de valor desejada para a eventual realização do negócio. Como resultado, é retornada uma ou mais páginas com todos os imóveis, caso a pesquisa tenha obtido sucesso, que contemplem os dados fornecidos. Se houver uma resposta positiva, o usuário poderá agendar visita a um dado imóvel, bastando para isto, clicar no *link* “Agendar”, conforme a Figura 4.3.

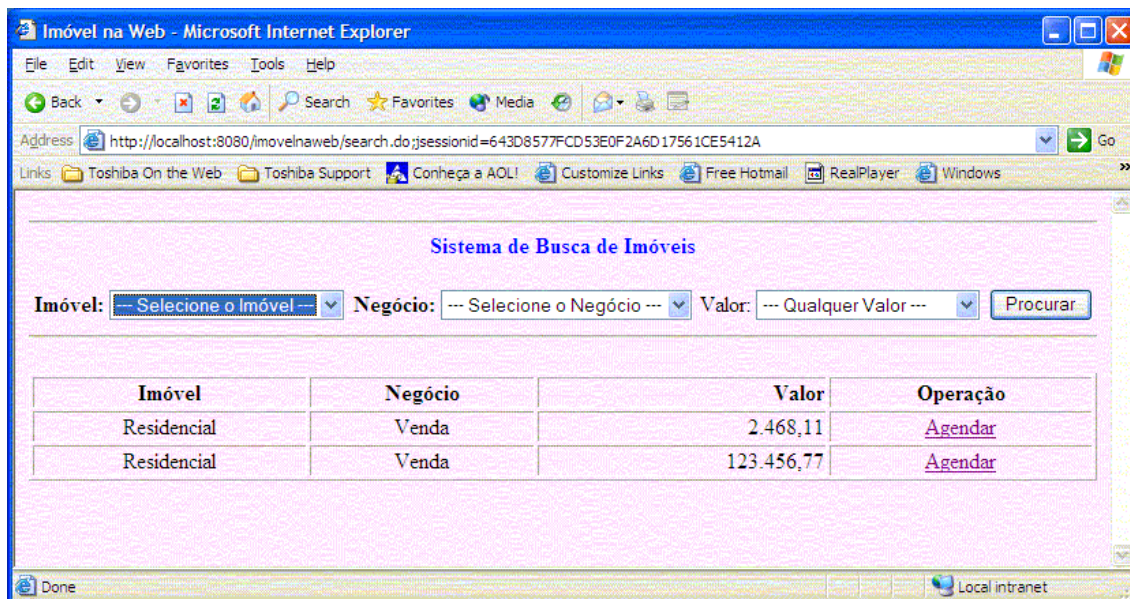


Figura 4.3: Resultado da pesquisa por imóveis na interface pública.

Já a interface administrativa apresenta uma maior complexidade, visto que é necessário, antes de tudo, que o usuário informe seu código e senha para que o sistema possa ser acessado. Uma vez que o usuário, neste caso, um funcionário da empresa imobiliária, tenha obtido permissão de acesso ao sistema, ele poderá prestar manutenção tanto em proprietários quanto em imóveis. É necessário entender que o termo **manutenção** designa as operações de inclusão, alteração, exclusão e consulta à base de dados da aplicação.

Durante o levantamento dos requisitos do sistema, são identificadas as entidades que compõem a aplicação e entidades, denominadas **externas**, que vão interagir com a aplicação de forma direta, entidades denominadas **atores**, e de forma indireta, por exemplo, subsistemas e outros sistemas.

4.3 Modelagem

Uma vez encontradas as entidades que compõem o sistema, a primeira etapa é a sua modelagem de tal forma que seja possível visualizar suas interações. A

modelagem da aplicação consiste em utilizar uma ferramenta de software que disponibilize elementos gráficos necessários à sua descrição.

O próximo passo consiste em descrever a interação dos atores com o sistema. Isto é feito através do diagrama de casos de uso. A Figura 4.4 apresenta o diagrama de casos de uso para o ator **funcionário**.

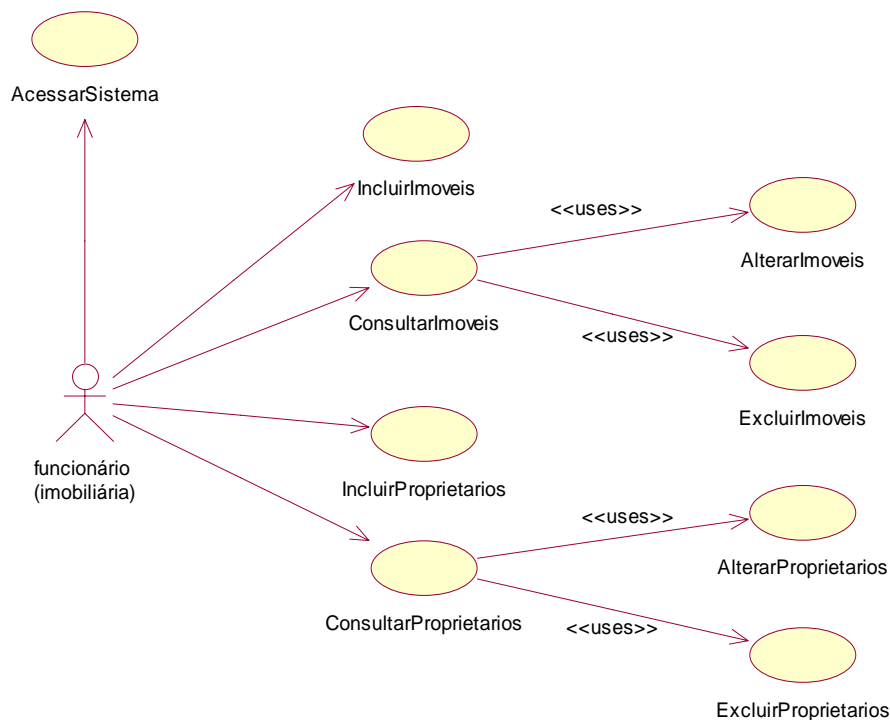


Figura 4.4: Diagrama de Casos de Uso para o ator funcionário.

A seguir é mostrado o caso de uso para a inclusão de imóveis. A interface do usuário para este caso de uso é apresentada na Figura 4.4.

Caso de Uso: IncluirImoveis

Modela o processo de inclusão de dados de um novo imóvel.

Curso Normal

1. O usuário informa os dados.
2. O usuário aciona o botão “Aceitar”.

Cursos Alternativos

- 1.1. Se o usuário não informou qualquer um dos valores pedidos o sistema recusa a inclusão.

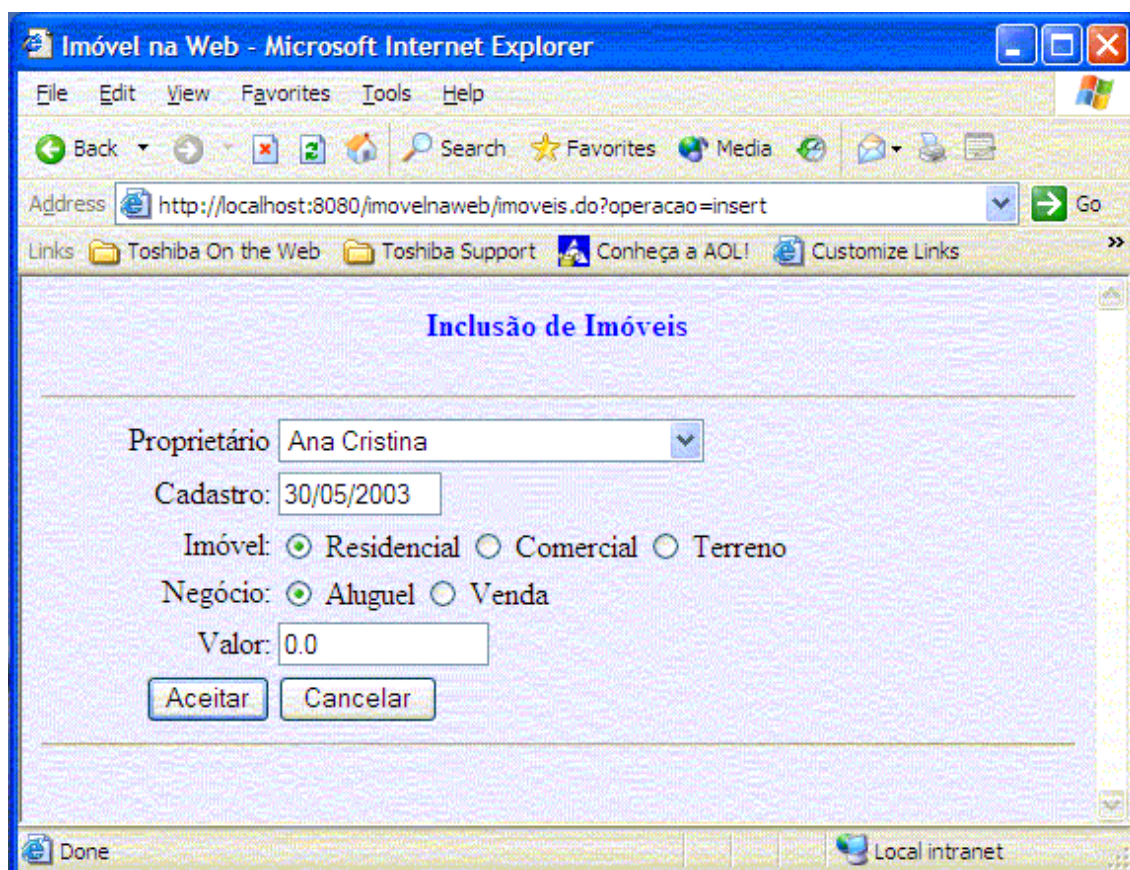
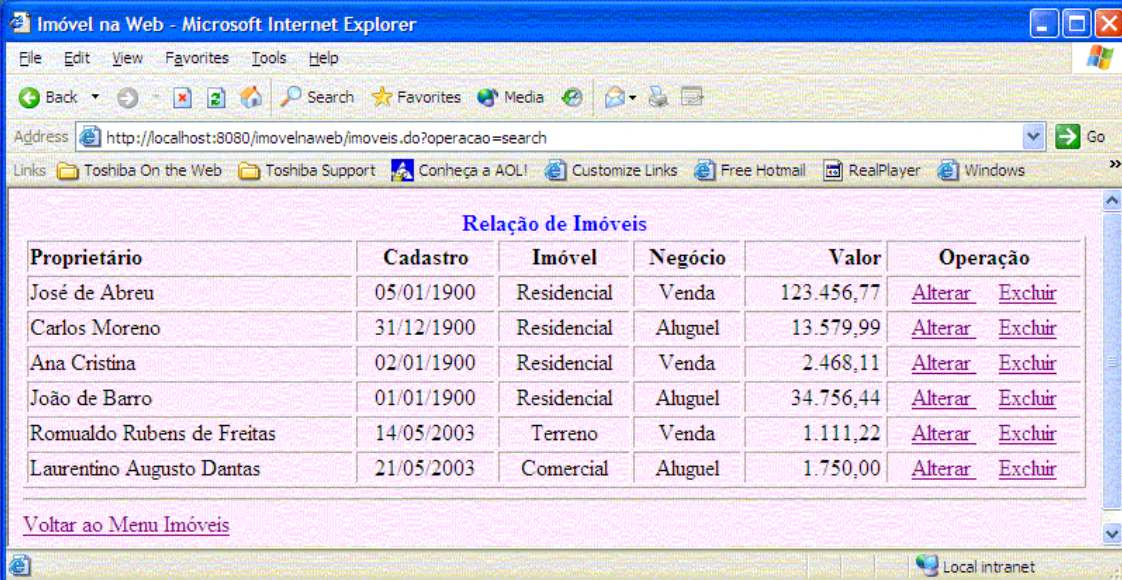


Figura 4.5: Interface do Usuário para Inclusão de um Imóvel.

Uma vez que o funcionário tenha feito uma ou mais inclusões de dados acerca de imóveis, os mesmos podem ser alterados ou excluídos. Estes processos são intermediados pela consulta, que possibilita ao funcionário visualizar quais são os

imóveis cadastrados. Para cada registro listado, o funcionário pode alterar os dados existentes, clicando no *link* “Alterar” ou excluí-los, clicando no *link* “Excluir”.

Um exemplo da relação dos imóveis existentes pode ser visto na Figura 4.6.



The screenshot shows a web browser window titled "Imóvel na Web - Microsoft Internet Explorer". The address bar shows "http://localhost:8080/imovelnaweb/moveis.do?operacao=search". The main content area displays a table titled "Relação de Imóveis". The table has six columns: Proprietário, Cadastro, Imóvel, Negócio, Valor, and Operação. Below the table is a link "Voltar ao Menu Imóveis".

Proprietário	Cadastro	Imóvel	Negócio	Valor	Operação
José de Abreu	05/01/1900	Residencial	Venda	123.456,77	Alterar Excluir
Carlos Moreno	31/12/1900	Residencial	Aluguel	13.579,99	Alterar Excluir
Ana Cristina	02/01/1900	Residencial	Venda	2.468,11	Alterar Excluir
João de Barro	01/01/1900	Residencial	Aluguel	34.756,44	Alterar Excluir
Romualdo Rubens de Freitas	14/05/2003	Terreno	Venda	1.111,22	Alterar Excluir
Laurentino Augusto Dantas	21/05/2003	Comercial	Aluguel	1.750,00	Alterar Excluir

Figura 4.6: Relação dos imóveis existentes.

Como é possível observar pela Figura 4.6, a coluna “Operação” disponibiliza ao funcionário alterar os dados de um determinado imóvel ou excluir os dados de um imóvel desejado.

As operações de alteração ou exclusão são ilustradas nas figuras a seguir.

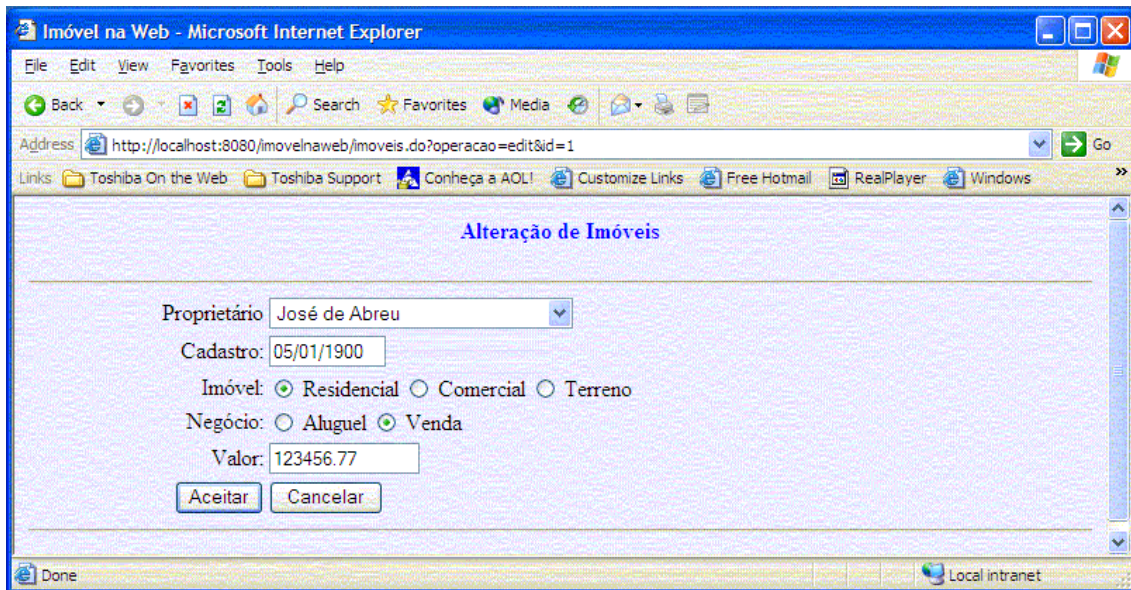


Figura 4.7: Operação de alteração dos dados de um determinado imóvel.



Figura 4.8: Operação de exclusão dos dados de um determinado imóvel.

Conforme demonstrado em Figura 4.7 e Figura 4.8, o funcionário pode aceitar a operação em andamento ou cancelá-la. Para a operação de exclusão, o funcionário visualiza os dados do imóvel selecionado antes que seu registro, caso seja confirmada a operação, sejam excluídos.

4.4 Projeto

O caso de uso e as interfaces do usuário mostradas acima demonstram como o sistema foi idealizado e concebido. Seguindo o Modelo 2 da especificação JSP, conforme demonstrado pela Figura 4.5, a aplicação utiliza um servlet para o gerenciamento da aplicação e distribuição das requisições feitas pelo usuário – atuando como controlador, e *JavaServer Pages* e *Servlets*, juntamente com *JavaBeans*, para processamento e exibição das interfaces do sistema – atuando na camada de apresentação. A camada de retaguarda (banco de dados) é suportada pelo uso de um SGBD disponibilizado pelo servidor e no qual os dados, tanto de imóveis quanto de proprietários, controlados pelos EJBs, são armazenados.

Além dos *JavaBeans*, descritos na Seção 3.5, que servem como meio de comunicação de dados entre *Servlets* e *JavaServer Pages*, também são utilizados *Enterprise JavaBeans*, que permitem o acesso aos dados armazenados na camada de retaguarda e são executados no âmbito do servidor de aplicações.

Finalmente, esta aplicação demonstra o uso de uma importante tecnologia para desenvolvimento de sistemas distribuídos multicamadas, que é a tecnologia de *Enterprise JavaBeans*, vista na Seção 2.5. Para a aplicação demonstrada neste estudo de caso, foram criados quatro EJBs: um *Session Bean* para controlar a consulta realizada na interface pública, um *Entity Bean* que permite efetuar o agendamento de um imóvel listado na interface pública e dois *Entity Beans*, um para gerenciar as informações de imóveis e outro para gerenciar as informações de proprietários. Os *Entity Beans* citados possuem persistência gerenciada a *bean*, que significa que o *bean* é responsável pelo código utilizado no armazenamento e busca das informações das entidades representadas por estes *beans*.

4.5 Conclusão

Nos últimos anos a Internet tem deixado de ser um simples depósito de informações para tornar-se um meio rápido e relativamente fácil de empresas atingirem seu público alvo: os consumidores. Cada vez mais se vê a proliferação de sistemas de comércio eletrônico, os assim chamados *e-commerce*. Empresas dos mais variados ramos descobrem que este tipo de comércio tem seus atrativos, como, por exemplo, comodidade e individualidade. Assim, sistemas de comércio eletrônico, dos mais simples aos mais complexos, como, por exemplo, vendas de mapas astrológicos a comércio de passagens aéreas e transações imobiliárias, vêm se alastrando pela Internet.

Com esta crescente gama de aplicações, é necessário que o desenvolvedor tenha a sua disposição um ferramental adequado para a criação de sistemas que atendam requisitos tais como: escalabilidade, segurança, gerenciamento de transações, entre outros.

Um conjunto de ferramentas desta natureza deve, principalmente, fornecer suporte a uma arquitetura de componentes de software que possibilite ao desenvolvedor criar aplicações que, de forma transparente ao usuário, possam usufruir características existentes em ambientes distribuídos.

Dentre as arquiteturas de componentes de software apresentadas no Capítulo 3 Arquiteturas de Componentes de Software, bem como a arquitetura de componentes de software presente na Plataforma Java 2, Edição Corporativa, mostrada no Capítulo 2 A Plataforma Java 2 Edição Corporativa, optou-se pela última, pois, ao longo do estudo destas arquiteturas, a arquitetura de componentes *Enterprise JavaBeans* apresentou características relevantes e que devem ser consideradas quando da escolha deste tipo de artefato de software, pois componentes criados com EJBs, além da linguagem Java, possibilitam seu acesso por aplicações desenvolvidas em qualquer linguagem de programação desde que esta permita a execução de chamadas remotas a métodos com a utilização de CORBA.

Outro aspecto importante considerado na escolha desta arquitetura de componentes reside no fato de o desenvolvedor não ter que se preocupar com questões relativas à segurança, gerenciamento do ciclo de vida dos componentes, persistência, acesso remoto, concorrência, dentre outras. Tais questões são abordadas no âmbito do servidor de aplicações e tornam-se transparentes ao desenvolvedor que, por meio de arquivos de configuração de parâmetros de aplicações, pode configurá-las de acordo com sua necessidade.

Assim, optou-se pelo desenvolvimento de um estudo de caso, muito próximo a uma aplicação do mundo real, que fizesse uso, tanto dos conceitos de sistemas distribuídos, como também e principalmente da utilização da arquitetura de componentes de software *Enterprise JavaBeans*, demonstrando, assim, que esta arquitetura pode ser empregada com sucesso no desenvolvimento de aplicações distribuídas dos mais variados tipos.

Capítulo 5

Conclusão e Trabalhos Futuros

O modelo de componentes de software *Enterprise JavaBeans*, que foi utilizado no estudo de caso apresentado no Capítulo 4, é um dos modelos de componentes de software para desenvolvimento distribuído mais difundido e aceito na indústria de software. A sua especificação apresenta um contrato bem definido entre as várias partes que integram um EJB, uma convenção de programação clara a ser seguida pelo projetista de componentes e uma boa facilidade de uso e também possibilita acesso por qualquer aplicativo capaz de efetuar chamadas CORBA.

Com o uso de outra tecnologia da plataforma Java, JNDI, é possível localizar componentes EJB de forma simples, bastando, para isto, conhecer os nomes atribuídos aos componentes. Assim, uma aplicação pode fazer uso de componentes que estejam presentes em diferentes servidores de aplicações sem, no entanto, conhecer suas localizações, estando eles em uma mesma rede ou em diversas outras espalhadas ao redor do globo.

Com as interfaces `EJBLocalObject` e `EJBLocalHome`, um *bean* pode fazer chamadas a outros *beans* sem a necessidade de o projetista preocupar-se com o tráfego de rede, pois tais chamadas são realizadas no âmbito do servidor de aplicações. Esta facilidade permite que *beans* possam cooperar na realização de tarefas que, além de diminuir o tráfego de rede, como mencionado, possibilita redução de código na aplicação cliente.

Levando em consideração que este modelo de componentes de software permite acesso a partir de uma aplicação que faça uso de chamadas CORBA, é possível criar conjuntos de componentes que realizam tarefas específicas e agrupando estes conjuntos é possível criar soluções para os mais variados domínios de problemas, fazendo com que desenvolvedores possam utilizar tais grupos ou conjuntos na realização de aplicações cujas partes já estão prontas, testadas e validadas.

Outro aspecto importante à cerca deste modelo de componentes de software reside no fato de o desenvolvedor não ter que se preocupar com questões relativas a transações, gerenciamento de memória para instâncias de objetos criados a partir de componentes, persistência de dados, no caso de *Entity Beans*, controle de concorrência, entre outros, pois tais características são inerentes aos servidores de aplicações.

Diante do exposto nos parágrafos anteriores, as perspectivas para trabalhos futuros, que abordem modelos de componentes de software, são as melhores possíveis, visto que o desenvolvimento de aplicações distribuídas e escaláveis precisa ser realizado em menos tempo e com maior produtividade e a utilização de componentes de software tem papel fundamental nestes aspectos. Assim, pesquisas na área de componentes, bem como a concepção de novos componentes e novos modelos de componentes que se ajustem às necessidades do projetista de software precisam ser realizados. Da mesma forma, ferramentas de modelagem que permitam aos analistas descreverem o modelo de uma aplicação utilizando uma linguagem gráfica, como a UML, por exemplo, precisam contemplar a utilização destes artefatos de software.

Para dar continuidade a esta dissertação, algumas possibilidades de trabalhos futuros, são sugeridas. Como primeiro trabalho, propõe-se que seja seguida uma abordagem orientada a objetos para a manipulação das informações persistidas pelos servidores de aplicação utilizando a linguagem de SQL, apresentar a possibilidade de eliminação das interfaces *home* e remota e *local* e *local home*, diminuindo a quantidade de arquivos para cada EJB. Isto é importante porque, além dos arquivos para as interfaces mencionadas, é necessário o arquivo de realização do *bean*, criar novos padrões de projeto enfocando domínios de problema para aplicações de negócios, integrar de forma direta ferramentas de modelagem com a arquitetura EJB, permitindo que o projetista de software possa gerar código diretamente do modelo da aplicação.

A utilização de arquiteturas de componentes de software na criação de aplicações nos mais variados domínios torna o desenvolvimento ágil e possibilita um alto grau de reusabilidade não somente do código criado ou gerado por meio de

ferramentas, mas também permite que componentes utilizados em uma determinada aplicação possam ser reutilizados em outras aplicações afins.

Novos trabalhos que abordem arquiteturas de componentes de software são necessários devido a grande demanda por aplicações que possam ser utilizadas em ambientes distribuídos tais como a Internet e também que possam ser escaláveis, permitindo que o projetista possa disponibilizar funcionalidades mais rapidamente e a custo reduzido.

Referências

- [BO02] BODOFF, Stephanie, et. al.. *The J2EE Tutorial*. Sun Microsystems, Inc.. Janeiro, 2002.
- [BRO00] BROWN, Alan W.. *Large-scale, Component-based Development*. Prentice-Hall. New Jersey, 2000.
- [CG00] COMPONENT GROUP. *Component-based Development – An Overview*. URL: <http://www.componentgroup.com/whitepapers/overview.html> . Janeiro, 2003.
- [CK02] CAVANESS, Chuck, KEETON, Brian. *Special Edition Using Enterprise JavaBeans 2.0*. QUE, Indianapolis, 2002.
- [DD00] DEITEL, H. M., DEITEL, P. J.. *Java: Como Programar*. 3^a ed.. Bookman Cia Editora. São Paulo, 2000.
- [EMC02] THE E-BUSINESS MANAGEMENT COMPANY. *CORBA Primer*. URL: <http://www.cs.wustl.edu/~schmidt/corba.html> . Janeiro, 2003.
- [GOU00] GOULD, Steven. *Develop n-Tier Applications using J2EE*. JavaWorld. URL <http://www.javaworld.com/javaworld/jw-12-2000/jw-1201-weblogic.html> . Dezembro, 2000.
- [HAA02] HAASE, Kim. *Java Message Service Tutorial*. Sun Microsystems, Inc.. Setembro, 2002.
- [HAL02] HALL, Marty. *Core Servlets and JavaServer Pages*. Sun Microsystems Press. Versão on-line. Julho, 2002.
- [HDF03] HUSTED, Ted, DUMOULIN, Cedric, FRANCISCUS, George, WINTERFELDT, David. *Struts in Action: Building web application with leading Java framework*. Manning Publications, Co.. Greenwich, 2003.
- [HUG99] HUGHES, Merlin, et al.. *Java Network Programming*. Manning Publications Co., 2^a ed.. Greenwich, 1999.
- [JOH97] JOHNSON, Mark, A *Walking Tour of JavaBeans*. JavaWorld. URL: <http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans.html> . Novembro, 2002.
- [JOH98] JOHNSON, Mark. A *Beginner's Guide to Enterprise JavaBeans*. JavaWorld. URL <http://www.javaworld.com/javaworld/jw-10-1998/jw-10-beans.html> . Outubro, 1998.
- [JOH01] JOHNSON, Mark. A *Walking Tour of J2EE*. JavaWorld. URL <http://www.javaworld.com/javaworld/jw-07-2001/jw-0727-enterprisejava.html> . Julho, 2001.

- [LEE01] LEE, Rosanna. *The JNDI Tutorial: Building Directory-Enabled Java Applications*. Sun Microsystems, Inc.. Maio, 2001.
- [OMG02] OBJECT MANAGEMENT GROUP. *CORBA Components*. URL: <http://www.omg.org> . Dezembro, 2002.
- [RAJ02] ROMAN, Ed, AMBLER, Scott W., JEWELL, Tyler. *Mastering Enterprise JavaBens*. 2^a. ed.. Versão on-line. John Wiley & Sons. New York, 2002.
- [RE97] ROY, Mark, EWALD Alan. *Inside DCOM*. URL: <http://www.dbmsmag.com/9704d13.html> . Janeiro, 2003.
- [SCH02] SCHMIDT, Douglas C.. *Overview of CORBA*. URL <http://www.cs.wustl.edu/~schmidt/corba.html> . Janeiro, 2003.
- [SHA96] SHAH, Rawn. *Integrating Databases with Java via JDBC*. JavaWorld. URL <http://www.javaworld.com/javaworld/jw-05-1996/jw-05-shah.html> . Maio, 1996.
- [SHA02] SHACHOR, Gal, et al.. *JavaServer Pages – Biblioteca de Tags*. Ed. Ciência Moderna. Rio de Janeiro, 2002.
- [SM99] SUN MICROSYSTEMS, Inc.. *Simplified Guide to the Java 2 Platform, Enterprise Edition*. Sun Microsystems, Inc.. Setembro, 1999.
- [SM00] SUN MICROSYSTEMS, Inc.. *JavaServer Pages Fundamentals*. Sun Microsystems, Inc.. Setembro, 2000.
- [SM01] SUN MICROSYSTEMS, Inc.. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, Inc.. Agosto, 2001.
- [SM02] SUN MICROSYSTEMS, Inc.. *Java Remote Method Invocation – Distributed Computing for Java*. Julho, 2002.
- [ZMF99] ZHAO, Yong, MISHRA Punyashloke, FERDIG, Richard E.. *Duct tape and magic: Component architectures and web based learning environments*. URL: <http://punya.educ.msu.edu/PunyaWeb/pubs/print/component/index.html> . College Education. Michigan State University. Dezembro, 2002.