UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Modelagem e Prototipação de um Repositório Extensível para Componentes de Software

por

ALEX MARRACCI SCHIROKY

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Profª. Drª. Ana Maria de Alencar Price
Orientadora

Porto Alegre, maio de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Schiroky, Alex Marracci

Modelagem e Prototipação de um Repositório Extensível para Componentes de Software / Alex Marracci Schiroky – Porto Alegre: Programa de Pós-Graduação em Computação, 2002.

114 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2002. Orientador: Ana Maria de Alencar Price.

1.Componentes de software. 2.Engenharia de Software 3.Reutilização de código 4.Gerenciamento de requisitos. I. Price, Ana Maria de Alencar. III. Título.

Agradecimentos

Muitas foram as pessoas que contribuíram para a realização deste trabalho, quer seja pelo suporte técnico e científico, realizando comentários, trazendo suas sugestões e compartilhando experiências, ou ainda pelo apoio e incentivo, sempre apresentados no momento correto.

A todas estas pessoas deixo os meus agradecimentos, num sincero muito obrigado. Entretanto algumas destas pessoas merecem um agradecimento especial:

- minha orientadora, Profa. Ana Maria de Alencar Price, pela oportunidade dada a mim de realizar este trabalho, pela confiança depositada, pela calma e orientação segura;
- ao Programa de Pós-Graduação em Computação da UFRGS pela valiosa oportunidade de realização do curso de mestrado;
- ao Prof. Adenauer Corrêa Yamin, pelo o apoio e incentivo dados, que foram muito além do esperado, pela sua dedicação e competência, sem contar na amizade;
- aos colegas Ricardo Miguens Nizoli, Peter Goulart e Luciano Pessman, que no decorrer deste curso tornaram-se amigos, dividindo seu tempo entre aulas, trabalhos, alegrias e tristezas;
- aos meus pais, Elisabete e Celso, por tudo que me ensinaram, pelo incentivo, e por todo o amor dedicado;
- a todos os colegas de trabalho da Intexnet, em especial seus sócios Denny Azzolin, Marco Lovato e Rafael Cunha, por depositarem em mim sua confiança, pelo profissionalismo, competência e amizade.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Várias ferramentas de desenvolvimento utilizam mecanismos para centralizar/gerenciar código compartilhado entre diferentes desenvolvedores ou equipes de desenvolvimento. Estes repositórios geralmente deixam muito a desejar em relação aos serviços que oferecem, não só aos desenvolvedores, mas também aos usuários de sistemas já desenvolvidos, que desejam ter atualizações e correções constantes em seus aplicativos. Surge então a necessidade de disponibilizar esses recursos em um ambiente de trabalho realmente cooperado.

Este trabalho propõe a modelagem de um repositório de componentes de software, formado por uma área de armazenamento centralizada comum aos desenvolvedores, e com capacidade de gerenciar componentes e agregar serviços extras, administrando seu ciclo de vida de modo integrado. Para tal, várias tecnologias baseadas em Java foram integradas, tais como as seguintes API's: Enterprise Java Beans (EJB), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Java Messaging Service (JMS), Java Transaction (JTA), Java Transaction Service (JTS) e Java Database Connection (JDBC).

Para que o modelo aqui proposto possua um nível de funcionalidade adequado, são fornecidos vários serviços pelo repositório, atendendo assim às expectativas tanto de desenvolvedores, como dos usuários finais. Dentre outros, são providos os seguintes serviços: *Serviço de Descrição*, *Serviço de Histórico*, *Serviço de Controle de Versões*, *Serviço de Teste OO*, *Serviço de Padrões de Projeto e Serviço de Invalidação de Componentes.*

Palavras-chave: componentes de software, engenharia de software, reutilização de código, gerenciamento de requisitos.

TITLE: “MODEL AND PROTOTYPE OF AN EXTENSIBLE REPOSITORY FOR COMPONENTS OF SOFTWARE”

Abstract

Software development tools commonly use mechanisms for storing/managing code shared among different developers or development teams. These repositories generally are very poor in relation to the services they provide to developers and also to the users of already developed systems which need to make constant updates and corrections on their applications. There is a need to supply supporting services to software integration through a really cooperated development environment.

This work considers the modeling of a software component repository comprising a common storage area to be accessed by developers, possessing capacity to manage components and to add extra services, and managing the development life cycle in an integrated mode. To accomplish this model, some technologies based on Java have been integrated, namely: Enterprise Java Beans (EJB), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Java Messaging Service (JMS), Java Transaction (JTA), Java Transaction Service (JTS) and Java Database Connection (JDBC).

For the model considered here to have an adjusted level of functionality, some repository services are supplied so assuring the expectations of developers and final users. Among the services provided, the following must be mentioned: *Description Service, History Service, Version Control Service, OO Test Service, Design Patterns Service and Components Invalidation Service.*

Keywords: software components, software engineering, code reuse, requirement management.

1 Introdução

Várias ferramentas de desenvolvimento utilizam mecanismos para centralizar e gerenciar código compartilhado entre vários desenvolvedores, realizando seu armazenamento em um servidor de rede local ou diretório compartilhado, constituindo um repositório central de informações.

Porém, estes repositórios geralmente deixam muito a desejar em relação aos serviços que oferecem, não só aos desenvolvedores, como também aos usuários dos sistemas desenvolvidos a partir dos recursos disponibilizados em um ambiente de trabalho realmente cooperado.

Segundo [MAR00], um repositório de componentes de software deve ser formado por uma área de armazenamento centralizada e comum aos desenvolvedores, e possui a capacidade de gerenciar componentes de software e agregar serviços extras, auxiliar o desenvolvimento de novos componentes e a evolução dos já existentes, administrando o ciclo de vida e desenvolvimento de modo integrado.

Como componentes de software são considerados neste trabalho todos os conjuntos classe/interfaces que possam ser definidos por desenvolvedores, tanto os que efetivamente utilizem o modelo de componentes de Java (JavaBeans), como os demais, desde que obrigatoriamente implementem a interface `Serializable`, necessária para que uma classe e suas instâncias tornem-se persistentes e possam ser armazenadas no repositório de componentes.

Na prática, para o desenvolvedor, a utilização de um repositório e seus serviços deve ocorrer de modo transparente e o mais amigável possível, como afirma Breslin em [BRE98]. Para isso, é necessária a modelagem da aplicação em várias camadas (multi-tiered aplication), onde as regras de negócio estão encapsuladas e definidas em classes armazenadas no repositório e acessadas por intermédio de suas interfaces. Assim o repositório deve ser considerado como um servidor de aplicações baseado em componentes.

Além disso, vários serviços devem ser agregados ao repositório para que o modelo aqui proposto possua um nível de funcionalidade que atenda às expectativas tanto de desenvolvedores como também dos usuários finais.

Dentre as ferramentas de repositório pesquisadas estão o Microsoft Repository, Platinum Repository, Rochade Repository e o Unisys Universal Repository, todos com características muito similares, trabalhando em plataformas distintas. Deste o sistema da Unisys, que é utilizado em ambiente de grande porte, até o Microsoft, disponível junto com serviços de meta dados para o ambiente Windows. Todos estes realizam o armazenamento genérico de objetos e seus meta dados, sem.tratar as características específicas de cada paradigma ou linguagem de desenvolvimento utilizada.

Todas estas ferramentas são utilizadas principalmente para Data warehouse, gerenciamento de meta dados, administração de dados, sistemas de ERP, desenvolvimento de aplicações e gerenciamento global de informações.

O objetivo principal deste trabalho é modelar um repositório de componentes de software que possua um gerenciamento gerenciável. Este gerenciamento será realizado a partir de diversos serviços executados de forma cooperativa e integrada, definindo um protocolo de gerenciamento de repositório.

Com base no modelo proposto e na metodologia apresentada a seguir, foi implementado um protótipo deste repositório, contendo componentes de software armazenados e exemplos de alguns dos serviços inicialmente propostos.

A metodologia de desenvolvimento deste trabalho deu-se em três etapas distintas. Na primeira etapa foram estudados e definidos cada um dos serviços propostos, com base em trabalhos que englobaram cada uma das áreas envolvidas, como descrição de componentes de software, controle de versões em banco de dados, teste de software, padrões de projeto, controle de históricos e mecanismos de invalidação de componentes. O resultado desta etapa constituiu a definição do protocolo de gerenciamento de repositório, base de todo o modelo proposto.

Na segunda etapa foi realizado um estudo sobre a linguagem Java e suas APIs, em especial, as que implementam persistência, serialização de objetos e invocação remota de métodos (RMI), procurando determinar que características da linguagem Java seriam exploradas pela ferramenta e como seria sua interação com os serviços propostos.

Como terceira e última etapa, foi prototipado um repositório para componentes de software, baseado nas tecnologias estudadas e no modelo obtido na primeira etapa, possibilitando assim a análise dos resultados obtidos e a exemplificação de cada um dos serviços implementados.

O presente trabalho está dividido da seguinte forma. No capítulo 2 são apresentados os principais conceitos sobre repositórios de componentes de software e a organização proposta e adotada.

No capítulo 3 são analisados e descritos os diversos serviços necessários para o gerenciamento do ciclo de vida de um componente de software armazenado no repositório. O capítulo 4 apresenta o modelo proposto para a implementação de um protótipo e o capítulo 5 faz uma revisão sobre as técnicas e conceitos de persistência de objetos, concluindo com isso a parte teórica deste trabalho.

O capítulo 6 inicia a parte prática, com a análise do modelo de persistência JDO. Em seguida, no capítulo 7, é apresentada a arquitetura J2EE, focando principalmente o desenvolvimento com JSP. O capítulo 8 é dedicado à apresentação da implementação do protótipo realizada e finalmente, no capítulo 9, são apresentadas as conclusões deste trabalho, pontos e temas que devem ser aprofundados em trabalhos futuros.

2 Repositórios de Componentes

A natureza e estrutura dos repositórios de componentes de software tem sido alvo de intensas pesquisas por alguns anos pela comunidade de reutilização de software. O autor [TRA88], possui a opinião de que a construção de repositórios estáticos de componentes é um verdadeiro exagero. Citando o caso das fábricas de software japonesas que praticamente não utilizam mecanismos de repositório de componentes em seus desenvolvimentos de software reutilizável.

Percebe-se também que muitos dos casos de reuso de componentes de software não envolvem a interação com um repositório de componentes. Entretanto, eles envolvem a modificação de componentes que ainda estão em processo de engenharia e projeto.

Um contra argumento a esta afirmação é apresentado por Frakes e Gandel [FRA90]. Eles apontam que o sucesso dos japoneses com mecanismos simples pode ser atribuído, talvez, à baixa rotatividade das equipes entre diferentes projetos. Com uma equipe coesa e estável, a importância de uma biblioteca detalhada, o suporte a pesquisas e buscas é reduzido, desde que todos os membros tenham a oportunidade de familiarizar-se com o conteúdo de uma biblioteca, por todo o tempo.

Frakes e Gandel citam também que, enquanto o tamanho reduzido de uma biblioteca minimiza a importância de mecanismos sofisticados de gerenciamento, o problema ainda se mantém. Se os componentes são complexos, eles irão precisar de um método de descrição mais poderoso, particularmente, se os mecanismos de busca estão automatizados, como em geradores de aplicações.

São apontados, também, os experimentos em *document retrieval* que indicam os diferentes tipos de mecanismos de busca necessários para obter-se itens diversos em uma biblioteca. Assim, eles alertam que a escolha de uma biblioteca e de um método de classificação pode fazer uma enorme diferença na ferramenta de suporte em um ambiente orientado à reutilização.

Mas existe um argumento que vem tornar-se decisivo e, sobretudo, traz à tona a absoluta necessidade da existência de um repositório de componentes de software centralizados: a Internet, e com ela o aumento na produção de componentes de software e comércio eletrônico. Sem levar em consideração a necessidade de repositórios para o desenvolvimento orientado à reutilização para uso interno das organizações, eles são absolutamente necessários para a centralização do tipo de acesso remoto que é imposto pelas redes globais e para a engenharia das equipes de desenvolvedores descentralizados suportados por estas.

Atualmente, a necessidade de tornar catálogos de desenvolvimento disponíveis para o mundo externo segue uma linha básica e integrada com a Web, utilizando para tal interfaces *World Wide Web* (WWW) baseadas em formulários.

Estas interfaces em conjunto com clientes gráficos ou navegadores como o Internet Explorer ou Netscape, tornaram-se um método barato e popular de prover um acesso amigável para sistemas de catálogo *online*.

As ferramentas necessárias para a publicação de informações na WWW são muito baratas e, em muitos casos, grátis, sendo geralmente de uso direto e fácil.

Os resultados obtidos são muito animadores, sendo uma tarefa consideravelmente fácil a produção de interfaces gráficas atrativas e com aparência similar em diferentes plataformas clientes. Nenhum sistema especializado é necessário residir no lado do cliente além do próprio navegador WWW, sendo que as facilidades de transferência de arquivos (FTP) e buscas simples (WAIS) estão realmente integradas no pacote de protocolos WWW.

Entretanto, existem certos obstáculos envolvidos com a utilização desta abordagem, já que os clientes WWW não possuem conhecimento sobre o domínio da aplicação em que estão operando.

Simplesmente, estes clientes recebem um fluxo de dados contendo as primitivas gráficas da interface com o usuário, como botões, caixas de texto e dados formatados para a resposta, gerados pelo servidor, e apresentam isto para o usuário.

Sistemas de informação, freqüentemente, suportam a noção de sessão, onde os resultados de uma consulta anterior podem ser reutilizados ou redefinidos. O protocolo *Hyper Text Transfer Protocol* (HTTP), que é o núcleo do sistema WWW, inerentemente não possui controle de estado ou o conceito de sessão. Vários problemas são levantados quando esta interface é adaptada para sistemas legados que possuem o conceito de manutenção de uma sessão contínua entre o cliente e o servidor.

2.1 Organização de um Repositório

A Figura 2.1 representa a infraestrutura necessária para a construção de um sistema de suporte ao armazenamento de componentes de software em ambientes heterogêneos. Segue uma breve descrição e relação de padrões que devem ser suportados sobre cada um dos itens presentes na representação sugerida.

Estes padrões estão enumerados e podem ser consultados através dos endereços [URL12] e [URL13], dentre todos estão destacados no presente trabalho apenas os documentos que possuem alguma relevância com o presente trabalho, principalmente os padrões relacionados com sistemas WEB e tecnologia Java.

Figura 2.1 – Estrutura de um repositório de componentes.

2.1.1 Serviços de Usuário e de Aplicação

Os serviços de usuário definem os métodos pelos quais as pessoas interagem com uma aplicação, enquanto que os serviços de aplicação fornecem as funções necessárias para a criação e manipulação, da exibição de imagens, definição e gerenciamento de objetos, suporte a diálogos, gerenciamento de janelas e especificação multimídia.

Atualmente, avanços significativos foram realizados na tecnologia de desenvolvimento de interfaces para torná-las de fácil utilização e reduzir o esforço necessário ao seu desenvolvimento.

Dependendo das capacidades exigidas pelos usuários e aplicações, os serviços podem incluir operações cliente/servidor.

2.1.2 Serviços de Programação

Os serviços de programação provém infra-estrutura para o desenvolvimento e a manutenção de software oferecendo características como: linguagens, ferramentas e metodologias, baseadas na abordagem de sistemas abertos, além de portabilidade, escalabilidade e interoperação de software.

ISO/IEC 16262:1998 - JavaScript: ECMA Script Language Specification

Define uma linguagem de script utilizada para a validação da entrada de dados no lado cliente de aplicações baseadas na Web.

Java™2 Platform, Standard Edition - Java™2 Platform, Standard Edition
Define a linguagem Java padrão voltada para a Web que permite o desenvolvimento de aplicações multi-plataforma.

W3C WAI-WEBCONTENT-19990505 - Web Content Accessibility Guidelines 1.0
Especifica um conjunto de diretrizes de acessibilidade que devem ser seguidas pelos desenvolvedores ao criarem novos conteúdos para a Web.

2.1.3 Serviços de Gerenciamento de Dados

Os serviços de gerenciamento de dados mantém um gerenciamento independente dos dados compartilhados por múltiplas aplicações. Estes serviços devem suportar definição, armazenamento e recuperação de elementos de dados. Este gerenciamento inclui serviços de dicionário/diretório de dados, sistemas de gerenciamento de banco de dados e distribuição de dados.

Alguns dos padrões que devem ser levados em consideração:

ANSI X3.135.10-1998 - SQL Object Language Bindings (SQL/OLB)
Define uma linguagem de consulta baseada na ligação de objetos, também denominados caminhos, serve como base para a especificação SQL/3.

CORBA 2.3 - Object Management Group's Common Object Request Broker Architecture
Padrão de comunicação entre objetos baseados em *broker* e definido pelo OMG.

FIPS PUB 127-2:1993 - Database Language Structured Query Language (SQL)
Especificação da linguagem de consulta estruturada a bases de dados.

JDBC - Java™ Database Connectivity
API de programação Java padrão utilizada para o acesso a bases de dados.

ODBC - Open Database Connectivity
Padrão aberto definido pela Microsoft para o acesso a bases de dados.

UML 1.4 - Unified Modeling Language
Linguagem para a modelagem de sistemas adotada como padrão pela indústria de software nos últimos anos.

2.1.4 Serviços de Troca de Dados

Os serviços de troca de dados provém suporte especializado para a troca de informações, incluindo o formato e a semântica das estruturas de dados entre as aplicações, de forma independente de plataforma. Existem vários níveis de complexidade para a troca de dados. Incluindo a definição da representação dos dados para serem trocados, o conteúdo dos dados, a representação dos objetos, o nível da linguagem e o nível da aplicação. Alguns dos padrões mais importantes são:

Adobe PostScript 3, 1999 – PostScript

É um padrão de fato, que define uma linguagem de programação, a qual descreve a aparência de uma página impressa por uma impressora (preto e branco ou colorida) ou por qualquer outro dispositivo de saída.

FIPS PUB 161-2 - Electronic Data Interchange (EDI)

Padrão utilizado para o intercâmbio de informações entre sistemas computacionais, representado documentos ou outros instrumentos monetários.

GIF Version 89a Non-Interlaced/GIF Version 89a Interlaced – GIF

Desenvolvida pelo CompuServe, é um padrão para a codificação de imagens baseado no algoritmo de compressão matemático LZW.

ISO 12639:1998 - Tagged Image File Format for Image Technology (TIFF/IT)

Tipo de arquivo utilizado para representar imagens de várias classes, tais como escalas de cinza, paleta de cores e RGB completo.

ISO 8879:1986 (Amendment 1, 1988) - Standard Generalized Markup Language (SGML)

É o padrão ISO que define um formato neutro capaz de descrever estruturas de dados consideravelmente complexas, HTML é uma aplicação de SGML.

ISO/IEC 10918-4:1999 - Joint Photographic Experts Group (JPEG)(Replaces IS 10918:1992)

Padrão de arquivo gráfico que fornece compressão digital e codificação de áreas com cores contínuas. Inclui 29 diferentes processos de codificação.

ISO/IEC 13818:1996 - General Coding of Moving Pictures and Associated Audio Information, Version 2 (MPEG) (Replaces ANSI/ISO IS11172:1992)

Define técnicas para a compressão de vídeo digital em fatores que variam entre 25:1 a 50:1.

PDF - Portable Document Format

Arquivos PDF são compactos e podem ser compartilhados, visualizados, navegados e impressos exatamente como gerados, utilizando-se o Adobe Acrobat Reader.

RTF - Rich Text Format (RTF)

É um padrão de fato, definido dela Microsoft, que permite a troca de arquivos texto entre diferentes processadores de texto e sistemas operacionais.

W3C REC-html40-19980424 - Hypertext Markup Language (HTML) 4.0

É a linguagem de publicação padrão na Web.

W3C REC-XML-19980210 - Extensible Markup Language (XML)

É um tipo de linguagem de marcação que contém meta dados e realiza a separação entre a estrutura e o formato de um documento.

2.1.5 Serviços de Rede

Os serviços de rede provém a conectividade e os serviços básicos para suportar a comunicação entre grupos de trabalho e sites. Estes serviços complementam a infra-estrutura de rede para suportar as capacidades e mecanismos para suportar o acesso a dados distribuídos e a interoperabilidade em ambientes heterogêneos. São definidos nos seguintes documentos:

ANSI/NISO Z39.50-1995 - Information Retrieval (Z39.50) Application Service Definition and Protocol Specification

Especifica um protocolo baseado em cliente/servidor para a recuperação de informações.

HTTP V1.1 - Hypertext Transfer Protocol (HTTP) Version 1.1

Protocolo unificado para para o acesso a Web, projetado para aumentar a velocidade na obtenção de páginas pelo navegador e reduzir o tráfego na Web.

IETF RFC 1777 - Lightweight Directory Access Protocol (LDAP), Version 3, 1997

É um protocolo utilizado para o acesso a serviços de diretório online, funciona diretamente sobre TCP e pode ser usado para acessar serviços de diretórios locais ou disponibilizados através de X.500.

IETF RFC 2045-2049 (11/96) - Multipurpose Internet Mail Extensions (MIME) Parts 1-5

Este protocolo é utilizado para suportar diversos tipos de arquivos de dados, sendo suportado pela grande maioria dos fornecedores de produtos de comunicação.

IETF RFC 821 (8/82) - Simple Mail Transfer Protocol (SMTP)

O objetivo deste protocolo é transferir e-mails de modo confiável e eficiente.

IETF STD-13/RFC 1034/RFC 1035 - Domain Names (Concepts and Facilities)/Domain Names (Implementation and Specification)

Fornece um mecanismo de nomeação de recursos, assim estes podem ser utilizados em diferentes servidores, redes, famílias de protocolos e organizações.

Internet Inter-ORB Protocol (IIOP) – IIOP

Define um protocolo orientado a objetos construído para tornar possível a comunicação entre programas distribuídos e escritos em linguagens de programação diferentes através da Internet.

TCP/IP RFC 793 (TCP)/RFC 801 (IP Version 4) - Transmission Control Protocol/Internet Protocol (TCP/IP)

É o protocolo utilizado para dar suporte a conectividade na Internet.

2.1.6 Serviços de Sistema Operacional

Os serviços do sistema operacional são os serviços essenciais necessários para operar e administrar a plataforma da aplicação e suportar uma interface entre o software da aplicação e a plataforma. A definição de padrões facilita o desenvolvimento de aplicações portáveis e fornece o suporte para outros serviços, maximizando os recursos computacionais e suas capacidades.

Alguns documentos que devem ser considerados:

IEEE 1003.10 - POSIX-Based Supercomputing Applications Environment Profile

Especifica um conjunto de padrões e requisitos necessários para portabilidade de aplicações em supercomputadores, usuários e administradores de sistemas.

IEEE 1003.2 (R1997) - Portable Operating System Interface (POSIX) Part 2: Shell and Utilities (Replaces FIPS PUB 189)

Define um interpretador de comandos e um conjunto de programas utilitários sobre a especificação POSIX, definindo com o usuário pode interagir com o sistema operacional.

IEEE/ANSI 1003.1:1996 - Portable Operating System Interface (POSIX) - System Application Program Interface (POSIX) - C Language (Replaces FIPS PUB 151-2)

Este padrão é utilizado por profissionais envolvidos com envolvidos com o desenvolvimento e implementação de sistemas e aplicações.

Single UNIX Specification, Version 2 - Single UNIX Specification, Version 2

Define o UNIX para fornecedores de sistemas operacionais e desenvolvedores de aplicações.

2.1.7 Serviços da Plataforma de Hardware

Os serviços da plataforma de hardware garantem uma infra-estrutura de tecnologia da informação onde escritórios, programas, recursos e sites podem realizar suas operações de modo transparente. Estes serviços tem seu foco principal em telecomunicações e processos de aquisição.

Algumas definições de plataformas de hardware são:

ANSI X3.131-1994 (R1999) - Small Computer Systems Interface - 2 (SCSI-2)

Define um conjunto de interfaces eletrônicas padrão que permitem a comunicação de computadores portáteis com diversos periféricos como discos, fitas, leitores de CD-ROM, impressoras e scanners.

Energy Star® - Energy Star®

Promove a conservação do meio ambiente através da conservação e racionalização do uso das fontes de energia.

ITU-T, Recommendation V.90, 1998 - Digital and Analog Modems Pair Used on the Public Switched Telephone Network

Especifica a operação de modems digitais e analógicos através do sistema público de telefonia, com taxas de transferência de até 56.000 bit/s.

2.1.8 Serviços de Segurança

Os serviços de segurança devem suportar um mecanismo de distribuição segura, mantendo a integridade da informação e protegendo a infra-estrutura computacional de acessos não autorizados. A interoperabilidade departamental, bem como a conectividade com servidores de aplicação, demandam uma arquitetura de segurança funcional, ainda que não intrusiva.

A segurança precisa ser projetada em todos os níveis dos elementos da arquitetura, tais como, sistema operacional, rede, troca de dados e gerenciamento de dados, equilibrando a acessibilidade e a facilidade de uso com os esquemas de proteção dos dados.

Uma arquitetura focada em sistemas abertos requer que os mecanismos de segurança colocados em prática sejam baseados em padrões aceitos no mercado, tais como:

FIPS PUB 113 - Computer Data Authentication
Especifica um algoritmo de autenticação de dados que, quando aplicado sobre os dados de um computador, automaticamente detecta modificações não autorizadas.

FIPS PUB 180-1 - Secure Hash Standard
Especifica um algoritmo seguro baseado em *Hash* que pode ser utilizado para gerar uma representação condensada dos dados de uma mensagem ou arquivo de dados.

FIPS PUB 196 - Entity Authentication Using Public Key Cryptography
Este padrão especifica dois protocolos de autenticação onde entidades em um sistema podem autenticar suas identidades em outro.

IETF RFC 1848 - MIME Object Security Services (MOSS)
Este protocolo define e aplica assinaturas digitais e serviços de criptografia em objetos MIME.

ISO/IEC 9796:1991 - Digital Signature Scheme Giving Message Recovery
Especifica um esquema de assinatura digital aplicado na recuperação de mensagens, para mensagens de tamanho limitado, utilizando uma chave pública do sistema.

Kerberos, DCE-SS 1.1 - Kerberos Network Authentication Service (V5) Generic Security Service API (GSSAPI)
Este protocolo utiliza criptografia forte para que clientes possam provar sua identidade em servidores e vice-versa sobre conexões de rede inseguras.

MISPC - Minimum Interoperability Specification for Public Key Infrastructure Components, Version 1
Fornece interoperabilidade entre a infraestrutura de componentes de chaves públicas de diferentes fornecedores.

SSL - Secure Sockets Layer (SSL) (Replaces S-HTTP)

É uma camada de software criada pela Netscape para gerenciar a segurança na transmissão de mensagens em uma rede.

2.1.9 Serviços de Gerenciamento

Os serviços de gerenciamento são parte essencial na operação de um ambiente integrado de sistemas. Eles devem ser integrados e suportar técnicas e mecanismos para monitorar e controlar as operações das aplicações, do banco de dados, dos sistemas, plataformas, redes e das interfaces, e interações do usuário com esses componentes em todos os níveis.

Alguns documentos que definem padrões de gerenciamento são:

IEEE 1387.2-1995 - Portable Operating System Interface (POSIX) System Administration - Part 2: Software Administration

Define um conjunto de informações mantidas sobre o software e um conjunto de programas utilitários para manipular estas informações.

IEEE 828-1998 - IEEE Standard for Software Configuration Management Plans (Updates ANSI 828-1990)

Estabelece os requisitos mínimos para um plano de gerenciamento e configuração de software, bem como as atividades específicas relacionadas.

IETF RFC 2272 - Simple Network Management Protocol V.3 (SNMP)

Define um conjunto de variáveis que uma porta de comunicação deve armazenar e especifica operações de controle sobre esta porta.

ISO 9000 - Quality Management and Quality Assurance Standards - Guidelines for Selection and Use

Representa um consenso internacional sobre as características essenciais de um sistema de qualidade para garantir a efetiva operação de um negócio.

ISO 9001:2000 - Quality Management Systems – Requirements

Susbtitui as normas ISO 9001, 9002, 9003 e é baseada em um modelo de processo que utiliza oito princípios de gerenciamento de qualidade: escopo, prazo, pessoas, qualidade, comunicação, custo, integração e riscos. Estes princípios facilitam a evolução em direção a excelência do negócio e enfatiza a satisfação do cliente.

ISO 9004:2000 - Quality Management Systems - Guidance for Performance Improvement

Utiliza o mesmo modelo de processo e princípios de gerenciamento de qualidade da ISO 9001, mas sua ênfase está no encontro das necessidades de todas as partes interessadas através da manutenção da satisfação do cliente.

Todos estes documentos fornecem o embasamento teórico para a definição de uma infraestrutura global de serviços para a plataforma selecionada. Na abordagem proposta, que é baseada em J2EE, tem-se disponível a maioria destas tecnologias na forma de API´s disponíveis para os desenvolvedores.

3 Análise dos Serviços

Neste capítulo é apresentada uma descrição dos serviços do sistema considerados imprescindíveis ao escopo do repositório proposto. Esta descrição contém a finalidade, o escopo e a aplicação de cada um destes serviços.

3.1 Serviço de Descrição

A atividade de descrição é parte integrante da documentação de um sistema. Isto quer dizer que todas as características e problemas envolvidos nesta etapa do ciclo de vida de um sistema.

Geralmente a documentação é deixada em segundo plano devido a falta de tempo para a sua execução, prazos de projetos estourados ou analistas/programadores sobrecarregados.

Para que as características desejáveis como reaproveitamento e compartilhamento de código possam ser exploradas e como definido nos princípios básicos da engenharia de software, é fundamental que as informações sobre os componentes possam ser compartilhadas entre os membros de uma equipe e, também, com as demais equipes de desenvolvimento.

A implementação deste serviço é baseada na API de documentação padrão de Java, denominada *Javadoc*, com as seguintes características observadas:

- **Class/Interface** – ao descrever um componente, deve-se identificar se este consiste em uma classe ou interface. Com base nesta informação pode-se determinar quais itens são relevantes à descrição.

- **Extends** – identifica a existência de herança, que pode ser aplicada a classes ou interfaces. Um componente pode ou não estender outro, conforme sua implementação.

- **Implements** – se o componente é uma classe, esta pode implementar uma, nenhuma ou várias interfaces existentes, definidas por outros componentes, devendo obrigatoriamente implementar todos os métodos descritos nessas;

- **Constructor** – se o componente é uma classe, esta deve obrigatoriamente possuir um construtor para a criação e instanciação de objetos desta classe;

- **Methods** – os métodos implementados por um componente serão listados, junto com uma breve descrição funcional de seu objetivo principal.

Todas estas informações serão armazenadas em um banco de dados relacionando o componente com as suas informações de descrição, deste modo permitindo a realização de alguns tipos de pesquisa que também devem ser implementadas por este serviço.

As pesquisas disponíveis por este serviço referem-se à estrutura do componente descrito, sua hierarquia, interfaces e nomes de métodos que este possui ou implementa, sem detalhar a sua estrutura interna ou funcional.

3.2 Serviço de Histórico

O objetivo deste serviço é manter um histórico detalhado da evolução de um componente, registrando o seu ciclo de vida como um todo, a partir do momento em que este é registrado no ambiente do repositório.

A metáfora a ser empregada é a de *timeline*, onde as alterações realizadas na codificação do componente irão gerar registros no histórico do mesmo, permitindo assim a geração de medidas em relação à qualidade do desenvolvimento, tempo de resposta entre a identificação e a correção de *bugs*, entre outras informações gerenciais sobre a evolução dos desenvolvimentos.

Este serviço está intimamente ligado aos serviços de controle de versões, teste e invalidação de componentes. Assim sendo, qualquer evento gerado por estes serviços irá gerar um registro no controle de histórico.

3.3 Serviço de Controle de Versões

Através deste serviço o sistema de gerência do repositório permite a manutenção de versões independentes de cada um dos componentes armazenados. As operações básicas oferecidas definem como é possível adicionar, alterar e remover componentes, realizando o controle das versões, desde a fase de desenvolvimento, até a certificação do componente para a utilização por usuários/clientes em seus sistemas.

3.3.1 Registro de Classes

Para que uma classe possa ser manipulada e utilize toda a gama de serviços disponibilizados pelo repositório é necessário registrá-la no ambiente de desenvolvimento, de modo que esta passe a integrar a hierarquia de classes existente, ficando disponível tanto para o uso do próprio repositório, como dos seus usuários, desenvolvedores e clientes.

O registro pode ser feito através de uma das operações descritas abaixo, respeitando cada uma das situações previstas:

Add

Quando uma nova hierarquia de classes é iniciada, ou seja, uma nova classe é criada, sem herdar as características de nenhuma outra classe previamente registrada, deve-se utilizar a operação de *Add*. Para tal, as restrições são as seguintes:

a) o identificador da classe deve ser único, desta forma evitando possíveis conflitos entre os identificadores de classes preexistentes;

b) a classe não pode possuir a cláusula *extents*, devido a própria natureza da operação *Add*, que não permite herança;

c) caso a classe possua a clausula *implements*, deste modo implementando uma ou mais interfaces, estas precisam estar previamente registradas no repositório.

Copy

Quando uma nova hierarquia de classes é iniciada, ou seja, uma nova classe é criada, sem herdar, mas sim copiando as características de uma outra classe previamente registrada, deve-se utilizar a operação de *Copy*. As restrições são as seguintes:

a) o identificador da classe deve ser único, desta forma evitando possíveis conflitos entre os identificadores de classes preexistentes;

b) a nova classe somente pode, e deve obrigatoriamente, possuir a cláusula *extents*, se e somente se a classe copiada também fizer parte de uma hierarquia de classes;

c) caso a classe possua a clausula *implements*, deste modo implementando uma ou mais interfaces, estas precisam estar previamente registradas no repositório.

Inherit

Quando uma nova classe é registrada e herda as características de alguma outra classe previamente registrada, deve-se utilizar a operação de *Inherit*. As restrições são as seguintes:

a) o identificador da classe deve ser único, desta forma evitando possíveis conflitos entre os identificadores de classes preexistentes;

b) a classe obrigatoriamente precisa possuir a cláusula *extents*, devido a própria natureza da operação *Inherit*, que realiza a criação de uma hierarquia de classes através de herança;

c) caso a classe possua a clausula *implements*, deste modo implementando uma ou mais interfaces, estas precisam estar previamente registradas no repositório.

3.3.2 Exclusão de Classes

Para que uma classe possa ser excluída do repositório de classes é necessário que seu registro no ambiente de desenvolvimento seja removido. A operação de exclusão, porém, não é algo tão trivial, pois devemos sempre ter em mente a natureza cooperativa do ambiente de desenvolvimento que utiliza os serviços disponibilizados pelo repositório.

A exclusão deve ser feita, preferencialmente, através da seqüência de operações descritas abaixo, respeitando a sua ordem em cada uma das situações previstas:

Deprecate

Quando uma classe antiga está para ser abandonada, a operação de *Deprecate* deve ser utilizada, desta forma os desenvolvedores de aplicações, ao realizarem pesquisas no repositório de componentes, serão informados dos planos de futuramente abandonar o uso deste componente de software.

Opcionalmente, o desenvolvedor do componente pode relatar os seus motivos e qual a nova classe que irá substituir a atualmente usada. As restrições para tal são as seguintes:

a) caso alguma classe realize o *extends* da classe que está para ser abandonada, está será automaticamente marcada como *deprecated* também;

b) caso a classe que está para ser abandonada seja uma interface, todas as classes que realizem a implementação desta, através da cláusula *implements*, também serão automaticamente marcadas como *deprecated*.

Delete

Quando uma classe antiga é abandonada e está para ser eliminada definitivamente do registro de classes do repositório, a operação de *Delete* deve ser utilizada. Desta forma, os desenvolvedores de aplicações, ao realizarem pesquisas no repositório de componentes, não irão mais encontrar referências a esta classe, a não ser que procurem por classes já abandonadas.

Neste caso, a classe abandonada e suas informações continuam a existir dentro do repositório. As aplicações que ainda fazem uso desta classe podem continuar funcionando normalmente, porém os desenvolvedores de aplicações não poderão utilizar esta classe em novas implementações.

Opcionalmente, o desenvolvedor do componente pode indicar qual a implementação que irá substituir a classe abandonada. As restrições são as seguintes:

a) caso alguma classe seja sub-classe (realize o *extends*) da classe que está sendo abandonada, está será automaticamente marcada como *deleted* também;

b) caso a classe que está sendo abandonada seja uma interface, todas as classes que realizem a implementação desta, através da cláusula *implements*, também serão automaticamente marcadas como *deleted*.

Drop

Quando uma classe antiga é eliminada definitivamente do registro de classes do repositório, a operação de *Drop* deve ser utilizada. Desta forma, os desenvolvedores de aplicações, ao realizarem pesquisas no repositório de componentes, não irão mais encontrar referências a esta classe, nem mesmo se procurarem por classes já abandonadas.

Neste caso, a classe eliminada e suas informações deixam de existir dentro do repositório. As aplicações que ainda fazem uso desta classe não podem mais continuar funcionando. Neste ponto, os desenvolvedores de aplicações já deverão ter migrado seus sistemas, deixando de utilizar a classe eliminada e passando a utilizar as novas implementações indicadas. As restrições são as seguintes:

a) caso alguma classe realize o *extends* da classe que está sendo eliminada, esta será automaticamente eliminada também;

b) caso a classe que está sendo eliminada seja uma interface, todas as classes que realizem a implementação desta, através da cláusula *implements*, também serão automaticamente eliminadas.

3.3.3 Atualização de Classes

Quando uma classe registrada precisa ser atualizada, de modo algum devemos registrá-la novamente. Tal tentativa será prontamente recusada pelo repositório, retornando um erro e informando que a classe já existe no registro atual, fazendo-se necessário realizar uma operação de *Drop* (remoção) deste registro para tal. Esta operação, como descrita anteriormente, irá eliminar toda e qualquer referência à existência anterior de uma dita classe, verificando suas dependências.

Para realizar a atualização, deve-se utilizar um dos métodos descritos abaixo, que realizam a atualização de seu registro no sistema conforme as modificações efetuadas na antiga implementação da classe.

Patch

O método *Patch* aplica a atualização substituindo a implementação de alguns dos métodos de uma determinada classe, sem que seja gerada uma nova versão desta. Este tipo de atualização serve para corrigir *bugs* presentes na implementação interna de uma classe, porém não permite que sejam adicionados novos elementos à classe, como novos atributos ou propriedades, ou seja, sempre será respeitada a *interface* definida anteriormente.

Release

O método *Release* aplica a atualização substituindo a implementação de uma classe por uma outra, redefinindo todos os seus métodos obrigatoriamente, o que irá gerar o registro de uma nova versão desta classe. Não permite, porém, que sejam adicionados novos elementos à classe, como novos atributos ou propriedades, ou seja, sempre será respeitada a *interface* definida anteriormente.

Update

O método *Update* aplica a atualização substituindo totalmente a implementação de uma classe por outra, redefinido todos os seus métodos. Isso irá gerar o registro de uma nova versão desta classe e permitindo, desta forma, que sejam adicionados novos elementos à classe, como novos atributos ou propriedades, podendo redefinir sua *interface* desde que sejam observadas algumas restrições, tais como:

a) a nova *interface* pode substituir totalmente a antiga, desde que a versão anterior não seja mais utilizada, através das cláusulas *extents* ou *implements* em nenhuma outra já classe registrada;

b) caso exista alguma dependência, como as citadas anteriormente, será permitido somente o registro de uma nova *interface* para a classe a ser registrada, ou seja, a nova classe deve obrigatoriamente implementar todos os métodos da antiga *interface*, porém pode estender esta, agregando novos métodos e/ou atributos.

3.4 Serviço de Invalidação de Componentes

Um dos maiores problemas envolvidos com a manutenção de um ambiente de desenvolvimento descentralizado está na garantia de que todos os desenvolvedores estejam utilizando as últimas versões atualizadas dos componentes de software disponíveis para o projeto.

Os problemas detectados e corrigidos precisam ser propagados para todas as classes envolvidas direta ou indiretamente com a correção realizada. Também os desenvolvedores precisam estar cientes de que as correções realizadas em classes de mais alto nível da hierarquia podem afetar o comportamento esperado pela subclasse desenvolvida por estes, exigindo algum tipo de conferência ou realização de um novo roteiro de testes.

O serviço de invalidação integra-se ao repositório e aos demais serviços realizando a recompilação de classes registradas sempre que ocorrerem alterações na sua implementação, sinalizadas pelo serviço de controle de versões e registrando estas atividades no serviço de controle de históricos.

Assim sendo, sempre que uma operação for realizada pelo controle de versões, a data de compilação da classe será menor que a data da última alteração. O mecanismo de invalidação é então invocado: analisa as informações atuais da classe compilada, comparando-as com as novas informações armazenadas, e antes de realizar a recompilação da classe versionada, pode invalidar as demais classes dependentes desta, indicando que também devem ser recompiladas em algum momento, conforme a configuração adotada pelo administrador dos serviços do repositório.

O processo de invalidação é automático, sempre que uma classe precisa ser recompilada. As demais classes dependentes desta são marcadas como inválidas para a execução até a sua posterior recompilação, que pode ser realizada de dois modos distintos.

3.4.1 Recompilação Manual

Neste modo, o próprio desenvolvedor precisa solicitar a recompilação das classes invalidadas pelo mecanismo do serviço. É o único modo de operação caso ocorra algum erro na recompilação automática de uma classe, pois permite que o desenvolvedor possa acompanhar o processo de compilação, analisar e corrigir qualquer problema detectado.

3.4.2 Recompilação Automática

É o modo padrão de operação, onde as classes invalidadas são automaticamente recompiladas segundo o critério de validação definido pelo administrador dos serviços do repositório.

Caso ocorra algum erro durante a execução deste processo, é gerado um aviso para o administrador e para o desenvolvedor da classe que não pode ser compilada, o processo é então abortado, sendo necessária a intervenção do desenvolvedor, que através da ferramenta de recompilação manual, deve corrigir o problema sinalizado, recompilar sua classe e gerar um aviso para o administrador, informando que o problema de compilação está resolvido.

Depois de informado da resolução do problema, o administrador pode reiniciar o processo de compilação automática, ou se preferir, utilizar a recompilação manual para completar o processo de validação das classes restantes e ainda marcadas como inválidas.

Os critérios de validação que podem ser adotados pelo administrador dos serviços de repositório são os seguintes:

on Demand

Somente a classe versionada é recompilada. As demais marcadas como inválidas somente serão recompiladas quando alguma aplicação tentar realizar a sua carga.

Este critério possui as seguintes características quanto ao processo de compilação:

- é extremamente rápido;
- é pontual;
- possui um baixo *overhead*;
- a carga de classes fica mais lenta;
- erros somente serão detectados quando uma aplicação carregar uma classe; e

- erros em super classes somente serão detectados quando uma aplicação tentar carregar a classe dependente.

Statistic

Além da classe versionada, todas as classes mais utilizadas e as suas dependências são recompiladas. Para determinar o nível de utilização das classes, o serviço mantém associado a cada classe um contador que é incrementado toda a vez que esta classe é solicitada.

Baseado nesta informação estatística e num parâmetro numérico ou percentual informado pelo administrador dos serviços de repositório, serão recompiladas as *n* classes mais utilizadas ou o número percentual informado das classes registradas mais utilizadas, fazendo com que o serviço possa variar dinamicamente o número de classes a ser compiladas, conforme aumenta o número de classes registradas no repositório de componentes.

Este critério possui as seguintes características quando ao processo de compilação:

- é suficientemente rápido;
- é focalizado;
- erros são detectados na hierarquia das classes mais utilizadas;
- erros são detectados nas classes mais utilizadas;
- possui um *overhead* aceitável;
- a carga de classes precisa ser monitorada;
- mais informações precisam ser gerenciadas;
- a atividade de compilação pode ser demorada;
- a utilização do repositório em um novo projeto pode fazer com classes não mais utilizadas venham a ser recompiladas até que as informações estatísticas sejam atualizadas; e
- o administrador precisa monitorar constantemente o serviço, atualizando as suas configurações e as estatísticas geradas.

by Level

A classe versionada é recompilada, e também todas as sub classes da árvore hierárquica abaixo desta até o nível de profundidade especificado pelo administrador, ou até o fim da hierarquia caso esta possua um nível de profundidade menor que o definido.

Este critério possui as seguintes características quanto ao processo de compilação:

- tende a tornar-se lento;
- é pulverizado sobre uma parte da árvore;
- erros são detectados na hierarquia das classes até o nível especificado;
- possui um *overhead* considerável;
- a carga de classes não precisa ser monitorada;
- poucas informações precisam ser gerenciadas;
- a atividade de compilação tende a ser demorada;
- o administrador precisa monitorar constantemente o serviço, atualizando as suas configurações para que a compilação seja realizada de forma eficiente.

Full

A classe versionada é recompilada, e também todas as sub classes da árvore hierárquica abaixo até o fim da sua hierarquia. Caso seja especificado pelo administrador, a compilação pode ser realizada de forma completa, ou seja, todas as classes registradas no repositório serão validadas independentemente de estarem marcadas como inválidas ou não.

Este critério possui as seguintes características quando ao processo de compilação:

- é lento;
- é pulverizado sobre toda a árvore;
- erros são detectados em toda a hierarquia de classes;
- possui um grande *overhead*;
- a carga de classes não precisa ser monitorada;
- nenhuma informação precisa ser gerenciada;

- a atividade de compilação tende a ser muito demorada;
- o administrador precisa utilizar com bom senso este modo de validação, por ser o mais demorado e com um custo computacional elevado.

3.5 Serviço de Teste

O teste de software é a atividade pertencente ao ciclo de desenvolvimento de software, que tem como objetivo principal encontrar erros. "*A atividade de teste não pode mostrar a ausência de ´bugs´; ela só pode mostrar se defeitos de software estão presentes*" [MYE79] [PRE95].

Para tal a estratégia de teste deve ser flexível o bastante para permitir o uso da criatividade e a customização e, ao mesmo tempo, ser rígida para promover um razoável planejamento e rastreamento administrativo à medida em que o projeto progride.

Segundo Pressman [PRE95], o teste deve se iniciar no nível de módulos e prosseguir para fora, na direção da integração, terminando no teste de sistema. Refinando esta visão, e ainda de acordo com Jacobson [JAC92], o teste deve ser realizado em paralelo com o processo de desenvolvimento, assume-se que os testes devem ser realizados inicialmente sobre as classes, pois estas irão compor os módulos de um sistema construído em Java.

O resultado da execução de um caso de teste, segundo as definições da Norma IEEE 610.12-1990 [IEE90], pode ser:

- **Defeito (*fault*)** – processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto;
- **Engano (*mistake*)** – ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;
- **Erro (*error*)** – diferença entre o valor obtido e o valor esperado, podendo ser um erro computacional, que provoca uma computação incorreta, mas o caminho executado no programa é igual ao caminho esperado, ou um erro de domínio, onde, quando o caminho executado é diferente do caminho esperado; e
- **Falha (*failure*)** – produção de uma saída incorreta com relação à especificação.

3.5.1 Técnicas de Teste

A seguir são brevemente descritas algumas das técnicas de teste de software relevantes ao contexto do presente trabalho.

3.5.1.1 Teste Funcional de Classes

Também conhecido como teste de caixa preta (*black box*). As técnicas de teste funcional derivam os casos de teste a partir da análise da funcionalidade (dados de entrada/saída e especificação) da classe, sem levar em consideração a estrutura interna da mesma [MYE79] [PRE95].

A cobertura é expressa pela porcentagem de pós-condições, ou a porcentagem de transições na representação de estados cobertos por casos de teste selecionados. Cada pré-condição é utilizada para estabelecer o ambiente de teste apropriado para o objeto. Cada pós-condição é uma sentença lógica construída como uma seqüência de cláusula unida por conectores lógicos. Para cada cláusula “ou”, deve ser criado um caso de teste. Deve-se verificar se cada cláusula “and” foi testada.

3.5.1.2 Teste Estrutural de Classes

Também conhecido como teste caixa-branca (*white box*) ou teste caixa aberta. No teste estrutural, os casos de teste são derivados a partir da análise da estrutura interna da classe. O objetivo dos casos de teste é causar a execução de caminhos identificados no sistema, baseados no fluxo de controle e/ou no fluxo de dados. Como é impossível executar todos os caminhos possíveis, algum critério deve ser utilizado para limitar o número de caminhos a um número aceitável e confiável [RAP85].

Conforme [WEY80], um critério é válido se a execução de pelo menos um dos casos de teste detectar erros no programa, e é ideal se fornecer um conjunto de casos de teste que detecte todos os erros do programa que o critério se propõe a detectar.

Estes critérios podem ser baseados no fluxo de controle, onde a seleção de um conjunto C de elementos (arcos, laços, nodos ou subcaminhos) no Grafo de Fluxo de Controle (GFC) do programa em teste, para teste de unidade, ou no Diagrama de Chamadas (DC), para os testes de subsistema e de sistema.

Um outro tipo de critério de seleção é baseado no fluxo de dados, técnica utilizada tradicionalmente na implementação de compiladores, para a otimização de código. Este tipo de análise considera as relações entre definições e usos de variáveis, resolvendo algumas falhas encontradas nos critérios baseados no fluxo de controle [RAP85].

3.5.1.3 Teste Incremental

A estratégia de teste incremental de classes foi proposta por Harrold et al., em [HAR92]. O objetivo desta estratégia é testar classes isoladamente, de forma incremental, explorando a natureza hierárquica da relação de herança, através do reuso das informações e dos casos de teste sempre que possível, obtidas de uma superclasse para orientar os testes de uma subclasse.

Inicialmente é testada uma classe base que não possua herança, definindo um conjunto de casos de testes que realize o teste de cada função membro individualmente e, após, testar as interações entre estas funções.

Para projetar um conjunto de casos de teste para uma nova classe, o algoritmo atualiza de modo incremental o seu histórico baseado nas informações da sua superclasse, refletindo os atributos modificados, herdados e os definidos na subclasse.

Assim sendo, somente os atributos novos ou afetados serão testados, e os casos de teste aplicados a superclasse são reutilizados, se possível, para o teste. Os atributos herdados são retestados em seu novo contexto na subclasse, através das suas interações com os atributos definidos pela subclasse.

O histórico da subclasse direciona a execução dos casos de teste, pois indica que casos de teste devem ser executados para testar a subclasse.

3.5.2 Serviço de Apoio ao Teste

Com base nestas informações, o serviço de testes aqui proposto segue a idéia de realizar testes incrementais nas classes armazenadas no contexto do repositório de componentes, sendo que, para cada classe registrada será mantido um histórico dos casos de uso para testes.

O seu funcionamento está ligado ao serviço de controle de versões, tal que, quando uma operação *Add* for realizada, dando início a uma nova hierarquia de classes, o serviço de testes irá criar um histórico vazio para a classe que acabou de ser registrada e durante todo o seu ciclo de desenvolvimento, serão registrados e codificados novos casos de testes para esta classe que poderão depois ser executados.

Operações de *Copy* e *Inherit* realizadas pelo serviço de controle de versões, copiarão e herdarão, casos de testes, respectivamente, garantindo o direcionamento da execução dos testes.

No caso das operações de atualização de classes, *Patch*, *Release* e *Update* dos casos de testes relacionados, a identificação de possíveis erros encontrados nas classes poderá ser assinalada. Assim, estes deverão ser novamente executados de modo a exercitar os fluxos de controle e/ou dados que apresentaram *bugs* em versões anteriores da classe atualizada.

As operações de exclusão têm um tratamento um pouco diferenciado. Quando a operação *Deprecate* for realizada, o histórico de casos de testes permanece inalterado. A opção de remover o histórico de testes quando uma operação de *Delete* for realizada, ou postergar esta remoção até a execução da operação de *Drop* fica a cargo do administrador do repositório.

3.6 Serviço de Padrões de Projeto

Projetar software orientado a objetos é uma tarefa bastante árdua, mas projetar software reutilizável orientado a objetos é ainda pior. Deve-se encontrar objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes e as hierarquias de herança, estabelecendo as relações chaves entre eles.

O projeto deve ser específico para o problema a resolver, mas também genérico o suficiente para atender futuros problemas e requisitos, tentando também evitar o re-projeto, ou pelo menos minimizá-lo.

Os projetistas de software orientado a objetos mais experientes consideram um projeto reutilizável e flexível difícil, senão impossível, de ser obtido da primeira vez. Antes que um projeto esteja terminado, normalmente tentam reutilizá-lo várias vezes, modificando-o a cada vez.

Numa outra ótica, verifica-se que projetistas experientes realizam bons projetos, ao passo que novos projetistas são sobrecarregados pelas opções disponíveis, tendendo a recair em técnicas não-orientadas a objetos que já utilizavam antes. Leva um longo tempo para os novatos aprenderem o que é realmente um bom projeto orientado a objeto.

Obviamente os projetistas experientes sabem algo que os inexperientes não sabem. O que é?

Uma coisa que os projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero. Ao invés disso, eles reutilizam soluções que funcionaram no passado. Quando encontram uma boa solução, eles a utilizam repetidamente. Conseqüentemente, encontram-se padrões, de classes e de comunicação de objetos, que reaparecem freqüentemente em muitos sistemas orientados a objetos. Estes padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objeto mais flexíveis e, em última instância, reutilizáveis.

Os padrões de projeto ajudam os projetistas a reutilizar projetos bem-sucedidos ao basear os novos projetos na experiência anterior. Um projetista que está familiarizado com tais padrões pode aplicá-los imediatamente a problemas de projeto, sem a necessidade de redescobri-los.

É reconhecido o valor da experiência de projeto. Quantas vezes um projetista já não passou pela experiência do *déja vu* durante um projeto – aquele sentimento de que já resolveu um problema parecido antes, embora não sabendo exatamente onde e como? Se pudessem lembrar os detalhes do problema anterior e de que forma o resolveu, então poderia reutilizar a experiência em lugar de redescobri-la. Contudo, não fazemos um bom trabalho de registrar experiência em projeto de software para o uso de outros.

Em geral, um padrão de projeto possui quatro elementos essenciais, são eles:

- **Nome do padrão** – é uma referência que podemos usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras. Dar nome a um padrão aumenta imediatamente o vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite a conversa sobre eles entre os elementos de uma equipe, na documentação gerada e consigo mesmo. O nome torna mais fácil pensar sobre projetos e comunicá-los, bem como os custos e benefícios envolvidos a outras pessoas. Encontrar bons nomes é uma das partes mais duras no processo de catalogação de padrões;

- **Problema** – descreve quando aplicar o padrão, explica o problema e seu contexto. Pode descrever problemas de projeto específicos, tais como representar algoritmos como objetos. Pode descrever estruturas de classe ou objetos sintomáticas de um projeto inflexível. Na maioria das vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão;

- **Solução** – descreve os elementos que compõe o projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e de como um arranjo geral de elementos resolve o mesmo.

- **Conseqüências** – são os resultados e análises das vantagens e desvantagens da aplicação padrão. Embora as conseqüências sejam raramente mencionadas quando descrevemos decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. As conseqüências para o software freqüentemente envolvem compromissos de espaço e tempo. Elas também podem abordar aspectos sobre linguagens e implementação. Uma vez que a reutilização é freqüentemente um fator no projeto orientado a objetos, as conseqüências de um padrão de projeto incluem o seu impacto sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema. Relacionar estas conseqüências explicitamente ajuda a compreendê-las e avaliá-las.

Os padrões de projeto serão descritos usando um formato consistente, onde cada padrão é dividido em seções de acordo com o gabarito a seguir. O gabarito oferece uma estrutura de acesso uniforme as informações, tornando os padrões de projeto mais fáceis de aprender, comparar e usar.

Um padrão de projeto será descrito conforme os itens a seguir:

- **Nome e classificação do padrão** – expressa a sua própria essência de forma sucinta. Um bom nome é vital, principalmente porque ele se tornará parte do vocabulário do projeto em que será usado.

- **Intenção e objetivo** – é uma curta declaração que responde às seguintes questões: O que faz o padrão de projeto ? Quais os seus princípios e sua intenção? Que tópico ou problema particular de projeto ele trata?

- **Também conhecido como** – outros nomes bem conhecidos para o padrão, se existirem.

- **Motivação** – um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão soluciona o problema. O cenário ajudará a compreender as descrições mais abstratas do padrão que vem a seguir.

- **Aplicabilidade** – quais são as situações nas quais o padrão de projeto pode ser aplicado? Que exemplos de maus projetos pode tratar? Como você pode reconhecer essas situações?

- **Estrutura** – uma representação gráfica das classes do padrão usando uma notação baseada em *Object Modeling Technique* (OMT), bem como diagramas de interação para ilustrar seqüências de solicitações e colaborações entre objetos.

- **Participantes** – as classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.

- **Colaborações** – como os participantes colaboram para executar suas responsabilidades.

- **Conseqüências** – como o padrão suporta a realização de seus objetivos? Quais são os seus custos e benefícios, e os resultados da sua utilização? Que aspecto da estrutura do sistema ele permite variar independentemente?

- **Implementação** – que armadilhas, sugestões ou técnicas o desenvolvedor precisa conhecer quando da implementação do padrão? Existem considerações específicas de linguagem?

- **Exemplo de código** – fragmentos ou blocos de código que ilustram como implementar o padrão.

- **Usos conhecidos** – exemplos do padrão encontrados em sistemas reais. Incluir pelo menos dois exemplos de domínios diferentes.

- **Padrões relacionados** – que padrões de projeto estão intimamente relacionados com este? Quais são as diferenças importantes? Com quais outros padrões este deveria ser usado?

Os padrões de projeto variam na sua granularidade e no seu nível de abstração. Como existem muitos padrões de projeto, necessita-se de uma maneira de organizá-los. Assim sendo classificam-se os padrões de projeto de maneira que se possa nos referir a famílias de padrões relacionados. A classificação ajuda a conhecer os padrões mais rapidamente, bem como direcionar esforços na descoberta de novos.

Os padrões de projeto são, então, classificados por dois critérios, segundo Gamma [GAM00]. O primeiro critério, chamado **finalidade**, reflete o que um padrão faz. Os padrões podem ter finalidade de criação, estrutural ou comportamental. Os padrões de criação preocupam-se com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades.

O segundo critério, chamado **escopo**, especifica se o padrão se aplica primariamente a classes ou a objetos. Os padrões para as classes lidam com os relacionamentos entre classes e suas subclasses. Se estes relacionamentos são estabelecidos através do mecanismo de herança, assim eles são estáticos – fixados em tempo de compilação. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são mais dinâmicos. Quase todos os padrões para objetos utilizam a herança em certa medida, porém a sua maioria está no escopo de Objeto.

Os padrões de criação voltados para classes deixam alguma parte da criação de objetos para as subclasses, enquanto que os padrões de criação voltados para objetos postergam esse processo para outro objeto. Os padrões estruturais voltados para classes utilizam a herança para compor classes, enquanto que os padrões estruturais voltados para objetos descrevem maneiras de montar objetos. Os padrões comportamentais voltados para classes usam a herança para descrever algoritmos e fluxo de controle, enquanto que os voltados para objetos descrevem como um grupo de objetos coopera para executar uma tarefa que um único objeto não pode executar sozinho.

A tabela 1 a seguir descreve graficamente o modelo de classificação de padrões adotado.

Tabela 3.1 – Modelo para a classificação dos padrões de projeto.

		Propósito		
		Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**			
	Objeto			

Existem, claramente, muitas maneiras de organizar os padrões de projeto. Ter múltiplas maneiras de pensar a respeito deles aprofunda a percepção sobre o que fazem, como se comparam e quando aplicá-los.

Como um catálogo de padrões de projeto tende a crescer e evoluir, muito em breve pode ser difícil encontrar aquele que trata de um problema de projeto em particular, especialmente, se o catálogo é novo e estranho para o desenvolvedor. Assim sendo, as seguintes abordagens podem ser empregadas com a finalidade de encontrar o padrão de projeto correto para um determinado problema:

- Considerar como padrões de projeto solucionam problemas;
- Examinar a seção Intenção;
- Estudar como os padrões se inter-relacionam;

- Estudar padrões de finalidades semelhantes;
- Examinar uma causa de reformulação de projeto;
- Considerar o que deveria ser variável no projeto.

Depois de definido o padrão de projeto que será utilizado, os seguintes passos são sugeridos para a sua aplicação:

1. Ler o padrão por inteiro uma vez, obtendo uma visão geral;
2. Estudar as seções Estrutura, Participantes e Colaborações;
3. Olhar a seção exemplo de código;
4. Escolher nomes para os participantes do padrão que tenham sentido no contexto da aplicação;
5. Definir as classes;
6. Definir nomes específicos da aplicação para as operações do padrão;
7. Implementar as operações para suportar as responsabilidades e colaborações presentes no padrão;

Finalmente, os padrões de projeto não devem ser aplicados de forma indiscriminada. Freqüentemente, eles obtêm flexibilidade e variabilidade pela introdução de níveis adicionais de endereçamento indireto, e isso pode complicar um projeto e/ou custar algo em termos de desempenho. Um padrão de projeto deverá apenas ser aplicado quando a flexibilidade que ele oferece é realmente necessária. As seções Conseqüências são muito úteis quando avalia-se os custos e benefícios de um padrão.

4 Objetos Persistentes

Conceitualmente, um objeto persistente é aquele que sobrevive à execução do processo que o criou, enquanto que objeto transiente é aquele que deixa de existir quando o processo que o criou deixa de existir.

A implementação de persistência em orientação a objetos é, atualmente, uma área de intensa pesquisa, visto que o desenvolvimento de aplicações em linguagens orientadas a objetos possui uma grande aceitação prática. Entretanto, as linguagens de programação orientadas a objetos foram concebidas para manipular dados transientes.

4.1 Aplicações Convencionais

Em geral, as aplicações convencionais executam sobre uma arquitetura Cliente/Servidor, utilizando um banco de dados relacional (SGBDR) e possuem as seguintes características:

- **Uniformidade de instâncias –** um grande número de itens de dados, com estrutura semelhante e com tamanho idêntico (coleções de registros);
- **Orientação a registros –** um tipo básico de dados é o registro de estrutura e tamanho fixos;
- **Pequenos itens de dados –** campos são de pequeno tamanho (raramente passam de algumas centenas de bytes);
- **Campos atômicos –** não há estrutura dentro dos campos e os registros estão na primeira forma normal.

4.2 Aplicações Avançadas

Em uma aplicação orientada a objetos, as necessidades são:

- **Identificação de objetos –** nem sempre é aceitável a exigência de que cada objeto deve ter um atributo que o identifica de forma única para todo o sistema;
- **Objetos complexos ou compostos –** composição de documentos, composição de peças;
- **Definição de tipos de objetos –** um objeto por definição pertence a uma classe, que possui estados e comportamento definidos;

- **Armazenamento de procedimentos e encapsulamento –** características do próprio paradigma e inerentes ao modelo adotado;

- **Ordenação de conjuntos –** permitindo o acesso eficiente a qualquer objeto;

- **Esquemas de dados flexíveis e evolucionários –** permitindo que os mecanismos de extensão de classes e interfaces, tais como herança e composição sejam suportadas;

- **Integração com LPOO –** assim o desenvolvedor não tem a necessidade de aprender uma outra linguagem para realizar o acesso aos dados, sem precisar preocupar-se tipos de dados equivalentes ou mapeamentos;

4.3 Persistência de Objetos

A principal questão está em como prover persistência a uma aplicação, e em virtude disto como tratar os demais problemas relacionados a implementação de persistência, tais como:

- Como representar objetos que podem ser removidos e objetos que devem ser armazenados?

- Como acessar objetos eficientemente?

- Onde armazenar o código de uma aplicação?

- Como excluir objetos?

- Como realizar o casamento de impedâncias entre as diferentes formas com que SGBD´s e LPOO´s tratam os dados ?

As principais formas de implementar persistência são: persistência de sessão, de arquivo e ortogonal, sendo que em qualquer uma destas pode-se ter persistência explícita ou inferida, como descrito em detalhes a seguir.

4.3.1 Persistência de Sessão

É a forma mais primitiva de persistência, permitindo que um usuário salve toda área de trabalho do programa, como por exemplo a implementada em Smalltalk. Porém não atende a maioria das aplicações, principalmente, porque o usuário não tem controle sobre o que é armazenado e o que não é, os dados não são compartilhados (somente mono-usuário) e não há como executar consultas sobre os dados.

Em Java a persistência de sessão é conhecida por serialização, sendo um conjunto de API´s que são descritas na seção 0.

4.3.2 Persistência de Arquivo

Uma forma um pouco mais completa, onde ao final do programa, ou quando são executados "save/commit", os dados são transformados em alguma estrutura interna e escritos em arquivo ou armazenados em um SGBDR. Quando necessários "open/load", estes dados são trazidos do disco para a memória.

Os principais problemas residem no fato que o programa têm que incorporar a funcionalidade de persistência, com uma estimativa de 30% do custo de programação/manutenção, os dados podem ser acessados (e corrompidos) por outras aplicações, os dados são armazenados em disco separadamente de métodos e existem duas estruturas diferentes para dados, uma em memória e outra em disco.

A forma disponibilizada em Java para a realização da persistência de arquivo é através de JDBC, uma API para acesso a bancos de dados descrita neste capítulo, na seção 5.2.

4.3.3 Persistência Explícita

O programador explicitamente informa ao sistema que o objeto é persistente, permitindo que o programador obtenha maior desempenho, mas com o perigo de que um objeto referenciado possa ser excluído, viola integridade referencial, o BDOO tende a ser uma rede complexa de objetos e pode ser difícil determinar quem referencia um objeto.

4.3.4 Persistência Ortogonal

É o modelo ideal para a implementação de persistência. Este termo é usado para descrever sistemas em que os dados são estruturados no BD da mesma forma que na memória, mantendo a mesma hierarquia de classes, tanto na base de dados como na memória e onde qualquer instância, independente de seu tipo, pode tornar-se persistente.

As principais vantagens deste modelo estão em sua transparência, porque o código e dados são trazidos sob demanda, quando necessários, as referências entre objetos são avaliadas sob demanda e o programador trabalha com um único modelo de dados, sem distinção entre disco e memória.

4.3.5 Persistência Inferida

O sistema infere o fato de um objeto ser persistente, pelo fato de ele estar associado a um outro objeto persistente. É também chamada de persistência por alcançabilidade, ou seja, se um objeto é persistente, todo objeto que ele referencia também deve ser.

Este mecanismo inicia com um conjunto de raízes persistentes. Todos os objetos alcançáveis a partir das raízes são persistentes e assim sucessivamente, como ilustrado na Figura 4.1, a seguir:

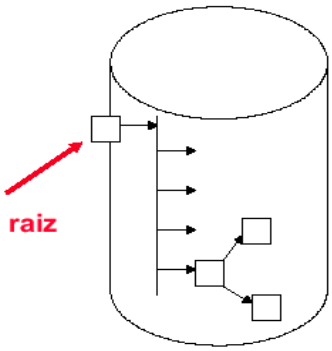

Figura 4.1 – Início de uma transação.

Novos objetos são instanciados em memória, utilizando os mecanismos para criação de objetos que a linguagem de programação utilizada fornece, geralmente um operador ***new***, como na Figura 4.2.

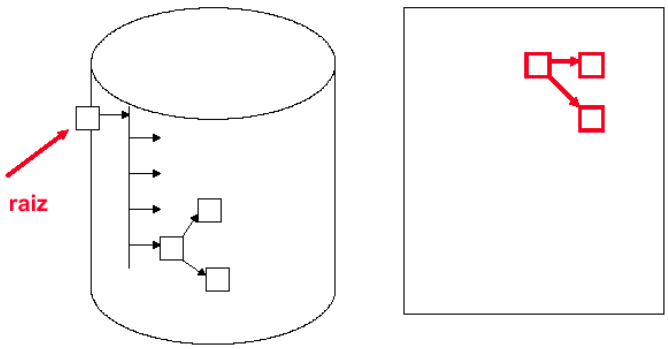

Figura 4.2 – Instâncias são criadas na memória.

Através de uma chamada ao mecanismo de persistência o sistema carrega para a memória um objeto armazenado na base de dados e cria uma referência entre este e um dos objetos transientes já instanciados, como ilustrado na Figura 4.3.

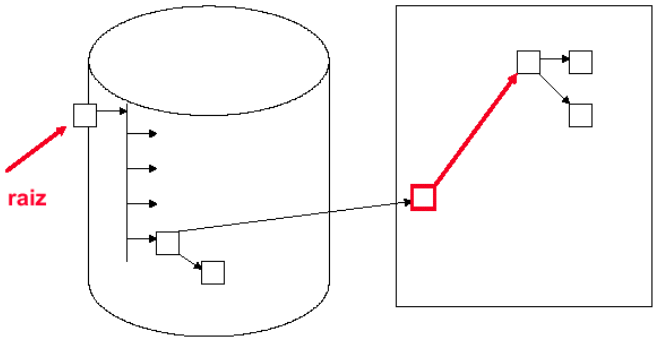

Figura 4.3 – Uma referência entre a base de dados e o novo objeto é criada.

Ao encerrar a transação, o mecanismo de persistência infere que os novos objetos, antes transientes, devem ser persistidos, pois estão sendo referenciados por um objeto persistente, assim sendo, toma a iniciativa de armazenar na base de dados todas as informações necessárias para restaurá-los, conforme representado pela Figura 4.4.

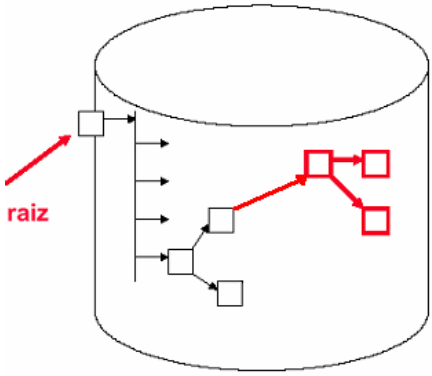

Figura 4.4 – Transação encerra.

Este tipo de mecanismo exige que o ambiente de execução seja gerenciado, como em um servidor de aplicações, ou então interpretado, onde temos um mecanismo de execução que trata e referencia todos as classes e objetos existentes nas aplicações em execução.

Além disso, o mecanismo de persistência insere uma sobre-carga no sistema como um todo, pois sempre que um objeto é criado, está para ser destruído ou é referenciado por outro, é necessária uma análise para determinar se este deve tornar-se persistente, continuar ou voltar a ser transiente.

Um modelo de persistência Ortogonal e Inferida está sendo proposto para integrar o pacote Java padrão, este mecanismo está descrito no capítulo seguinte na seção 5.3.

5 Persistência de Objetos em Java

Atualmente, existem dois padrões Java para o armazenamento de dados persistentes: serialização e JDBC, além de uma nova API de serviços de persistência denominada JDO. A serialização mantém as relações entre um grafo de objetos Java, mas não suporta o compartilhamento entre múltiplos usuários. JDBC requer que o usuário explicitamente gerencie os valores dos campos e faça o seu mapeamento em tabelas de um banco de dados relacional. Enquanto que a JDO promete ser a API de mais alto nível, tratando a persistência de forma transparente para os desenvolvedores. Os desenvolvedores podem assim ser mais produtivos, focando suas habilidades no desenvolvimento de classes Java que implementem a lógica do negócio, e utilizem classes Java nativas para representar os dados obtidos a partir das fontes de dados. O mapeamento entre as classes Java e as fontes de dados, se necessárias, pode ser feito por um especialista em EIS.

5.1 Serialização

Como citado na introdução deste trabalho, para ser considerado um componente de software e poder ser armazenado no repositório de componentes, uma classe definida por um desenvolvedor precisa obrigatoriamente implementar a interface `Serializable` para tornar-se persistente.

O conceito básico de serialização de objetos é a capacidade para ler e escrever objetos em seqüências de dados, tratados em Java como *streams*. Estes fluxos podem ser utilizados para salvar informações sobre o estado de uma sessão Servlet, para enviar parâmetros de chamada, através da invocação remota de métodos (RMI), para salvar as informações de estado sobre componentes produzidos pela tecnologia JavaBean e para enviar objetos através de uma rede, bem como para diversas outras tarefas.

Para a serialização de objetos deve-se utilizar as classes `ObjectOutputStream` para salvar objetos e `ObjectInputStream` para ler objetos serializados. O processo de serialização de um objeto trata de apresentar toda a árvore que representa o objeto na forma de tipos básicos que o compõe, incluindo todos os objetos que são referenciados. No momento de ler um objeto serializado, a árvore original que representa este objeto será recriada.

Para que um objeto suporte o processo de serialização, sua classe deve implementar a interface `Serializable`. Não existem métodos nesta interface, ela serve somente como uma marca para dizer que uma classe pode ser serializada. Portanto, adicionando `implements Serializable` na definição de uma classe não faz automaticamente com que esta classe seja serializável. Todas as variáveis de instancia da classe precisam ser serializáveis também. Se elas não o forem e tentar-se serializar a classe, uma exceção será sinalizada. Para marcar uma variável de instância como não serializável, deve-se adicionar a palavra chave `transient` em sua definição. Classes como `java.awt.Image` e `java.lang.Thread`, que contém informações de implementação específicas para uma determinada plataforma, não são serializáveis e devem ser marcadas como transientes.

```
transient Image image;
```

Quando uma classe for serializada, as únicas informações que serão salvas são os dados não estáticos e não transientes de uma instância. Definições de classe não são salvas e devem estar disponíveis quando uma operação de desserialização for tentada.

O processo básico de serialização de um objeto é como se segue:

```
ObjectOutputStream oos = new
ObjectOutputStream(anOutputStream);
Serializable serializableObject = new …
oos.writeObject(serializableObject);
```

Para ler um objeto serializado e realizar a sua desserialização, faz-se o contrário:

```
ObjectInputStream ois = new
ObjectInputStream(anInputStream);
Object serializableObject = ois.readObject();
```

Outro ponto muito importante no processo de serialização é sobrescrever os métodos `readObject` e `writeObject` para alterar o seu comportamento padrão. Sobrescrevendo estes métodos pode-se fornecer informação adicional para o fluxo de dados de saída, a qual pode ser lida e utilizada no momento da desserialização.

```
private void writeObject(ObjectOutputStream oos) throws
IOException {
     oos.defaultWriteObject();
     // campos adicionais
     oos.writeObject(new java.util.Date());
}
private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException {
     ois.defaultReadObject();
     aDate = (java.util.Date) ois.readObject();
}
```

Uma razão bastante comum para se sobrescrever os métodos `readObject` e `writeObject` é a serialização de dados de uma super classe que não é serializável por si só.

5.2 Java Database Connectivity (JDBC)

Java Database Connectivity (Conectividade em Banco de Dados em Java) consiste em uma API Java, composta por um conjunto de classes e interfaces escritas na linguagem Java e que são responsáveis pela execução das operações SQL (*Structured Query Language*). O JDBC provê uma forma simples de desenvolver aplicações com banco de dados utilizando a linguagem Java.

São três as suas funções:

- estabelecer a conexão com o gerenciador do banco de dados;
- enviar ao banco requisições SQL;
- processar os resultados das requisições.

Os programas desenvolvidos em Java com JDBC são independentes de plataforma e de banco de dados. Isso significa que um programa Java que acessa uma base de dados pode funcionar tanto em Windows como Unix, como também pode acessar qualquer banco de dados, como Oracle, Microsoft SQL Server e outros. Além da API Java, o JDBC contém um gerenciador de *driver* que se conecta a um *driver* proprietário do banco de dados. O que os desenvolvedores de banco de dados têm de fazer é fornecer um *driver* específico do seu banco de dados que se comunique com o gerenciador de *drivers* de Java. A figura abaixo mostra o cliente Java executando em diversas plataformas.

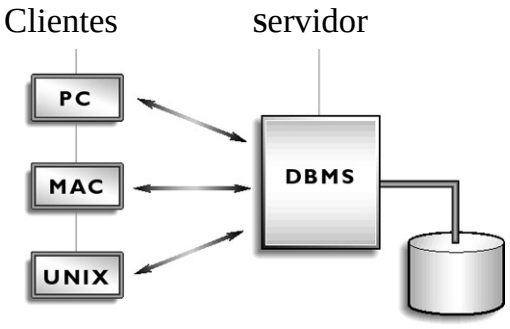

Figura 5.1 – Clientes Java rodando em diferentes plataformas.

A figura abaixo ilustra a tecnologia do JDBC. Os programadores Java devem conseguir acessar um banco de dados efetuando declarações SQL padrão, através de uma API JDBC. Estas declarações são enviadas ao gerenciador de *drivers* JDBC, que se encarrega de utilizar os *drivers* disponíveis para o banco de dados.

Caso não tenha um *driver* JDBC disponível, então pode ser utilizada uma ponte JDBC/ODBC, que envia as declarações SQL para o *driver* ODBC.

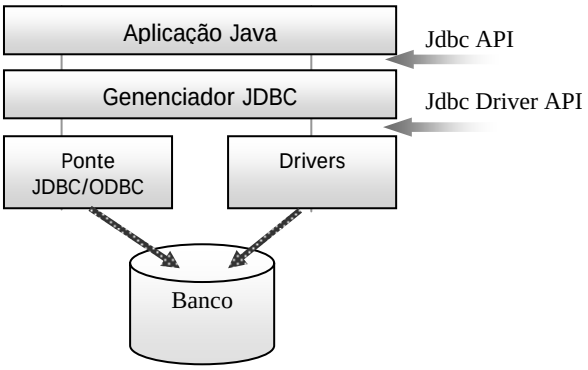

Figura 5.2 – Tecnologia JDBC

Para desenvolver uma aplicação com Java e bancos de dados relacionais, é preciso ter disponível:

- O pacote JDBC (padrão na distribuição da plataforma de desenvolvimento Java desde sua versão 1.1);
- Acesso ao servidor, o sistema gerenciador de banco de dados que entende SQL; e
- O *driver* JDBC adequado ao tipo de SGBD acessado.

Uma vez que esses recursos estejam disponíveis, a aplicação Java tem acesso ao banco de dados relacional através da execução dos seguintes passos:

1. Solicitar a conexão com o banco de dados;
   - Localizar o *driver* do banco de dados desejado;
   - Processar as chamadas de inicialização do JDBC;
2. Enviar as requisições SQL para o banco de dados;
3. Validar as chamadas de métodos JDBC;
4. Definir áreas e tipos de dados para armazenar os resultados das requisições SQL;
5. Requisitar os resultados;
6. Processar os erros;
7. Controlar transações de *commit* ou *rollback*;
8. Fechar a conexão.

É tarefa do *driver*, através das requisições da aplicação, enviar as consultas SQL para a base de dados e retornar os resultados para a aplicação.

O *driver* tenta:

- Estabelecer a conexão com a base de dados;
- Enviar as requisições ao banco de dados;
- Retornar os resultados das requisições a aplicação;
- Formatar erros no padrão do JDBC;
- Manipular cursores se necessário.

5.2.1 Interface e classes do pacote JDBC

O pacote JDBC é formado por interface e classes. Os *drivers* desenvolvidos para banco de dados específicos devem implementá-las. As principais interfaces são:

- **java.sql.Connection**: representa uma sessão com determinado banco de dados. As requisições SQL são executadas dentro do contexto de uma conexão. Por padrão o *autocommit* está habilitado, do contrário o usuário deve fazer o controle nas modificações do banco de dados;
- **java.sql.Statement**: associado a uma conexão, permite que requisições SQL sejam enviadas ao banco de dados;
- **java.sql.ResultSet**: permite acesso às linhas de resultado da requisição SQL na classe `Statement`;

A classe **java.sql.DriverManager** provê métodos para recuperar *drivers* e suportar a criação de conexões aos bancos de dados, usando métodos declarados na interface **java.sql.Driver**. Esta classe estabelece um conjunto básico de serviços para a manipulação de *drivers* JDBC. Como parte de sua inicialização, essa classe tentará obter o valor da propriedade jdbc.drivers de um arquivo de definição de propriedades e carregar os *drivers* especificados pelos nomes das classes. Alternativamente, um *driver* pode ser carregado explicitamente para a JVM. A forma usual para executar essa tarefa é através do método `forName()` da classe `Class`, tal como na linha abaixo, que exemplifica a utilização do *driver* JDBC/ODBC:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
```

A figura abaixo apresenta a relação de criação existente entre os objetos DriverManager, Connection, Statement e ResultSet.

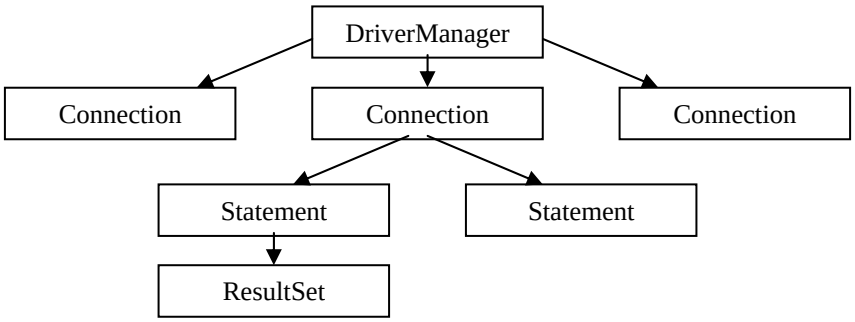

Figura 5.3 – Relação entre objetos do JDBC

O próximo passo é estabelecer a conexão com o banco de dados. Para tal é necessário carregar o *driver* de dados, e deve-se fornecer a URL do banco de dados. Esta URL tem o seguinte formato:

```
jdbc:nome_subprotocolo://url:porta
```

Na qual o subprotocolo depende do driver JDBC do banco de dados a ser utilizado.

Para fazer a inicialização da conexão, usa-se o comando:

```
Connection conn = DriverManager.getConnection(URL, user,
password);
```

Estabelecida a conexão ao banco de dados, é possível criar uma consulta e executá-la a partir da aplicação Java. Para representar uma consulta, tal como foi visto anteriormente, o JDBC utiliza um objeto de uma classe que implementa a interface **Statement**. Um objeto dessa classe pode ser obtido através do método **createStatement()** da classe **Connection**. Uma vez que um objeto `Statement` esteja disponível, é possível aplicar a ele o método **executeQuery()**, que recebe como argumento um *String* representando uma consulta SQL. O resultado da execução da consulta é disponibilizado através de um objeto **ResultSet**.

Cada uma destas classes confia em sua classe anterior para a criação de instâncias, por exemplo, uma instância de `Connection` não pode existir sem que tenha sido criada por um `DriverManager`.

Este mecanismo faz com que operações básicas, como a inserção de uma linha, sejam também realizadas de forma simples. Quando operações mais complicadas forem necessárias, como a passagem de parâmetros para funções armazenadas, o código JDBC será um pouco mais complexo.

Como JDBC pode funcionar com um grande número de *drivers*, existe uma classe chamada `DriverManager` responsável por carregar os *drivers* no ambiente.

Geralmente, o gerenciador de *drivers* somente será necessário uma vez no início, no método *main* ou *init* da aplicação ou *Applet*, respectivamente, como o exemplo:

```
try {
     Class.forName(driver);
     Connection connection =
DriverManager.getConnection(“jdbc:oracle://localhost:1521:d
b”, “username”, “password”);
} catch (ClassNotFoundException e) {

}
```

Existe na prática apenas um método importante na classe `DriverManager`, o método `getConnection`. Objetos de conexão são criados a partir da classe `DriverManager`. Estes objetos são orientados por URL´s que indicam a localização e os parâmetros necessários para a realização da conexão.

O `DriverManager` é politicamente correto sobre o mecanismo utilizado para encontrar o *driver* correto para a URL especificada em `getConnection`. Seu comportamento é flexível, operando através de delegação voluntária, onde o mecanismo JDBC visita cada um dos *drivers* registrados pelo DriverManager e pergunta: “Você pode se conectar a esta URL ?”, se a resposta for não, o próximo *driver* é questionado e o processo continua, até que um dos *drivers* aceitar a conexão com a URL especificada. O trecho de código a seguir demonstra a utilização deste mecanismo:

Neste exemplo, “jdbc” é o protocolo principal, “oracle” é o subprotocolo e é seguido pelo nome do servidor “localhost”, pelo número da porta “1521” e pela instância do banco de dados a ser conectado.

Como mencionado anteriormente, cada classe é responsável pela criação da próxima classe, como no caso dos objetos `PreparedStatement` que são criados a partir de uma instância de `Connection`, como demonstrado:

```
try {
     PreparedStatement ps =
connection.preparedStatement(“SELECT * FROM DUAL”);
} catch (SQLException e) {
     System.out.println(e.getMessage());
}
```

5.2.2 Controle de Transações

A grande maioria das interações com banco de dados envolve ações de uma etapa apenas, ou seja, uma instrução SQL deve ser executada sem interrupção. No entanto, freqüentemente uma única ação é na verdade composta de uma série de instruções SQL inter-relacionadas que devem ser bem sucedidas ou falhar juntas.

Por exemplo, transferir dinheiro entre duas contas é um processo composto de duas etapas. Primeiro deve-se debitar uma conta e em seguida creditar a outra. Como padrão, o banco de dados irá processar cada instrução imediatamente, numa ação irrevogável. Assim sendo, se a ação de crédito ocorrer mas a de débito não, o banco de dados perde sua integridade.

Os bancos de dados fornecem um mecanismo conhecido como transações, as quais ajudam a evitar tais problemas. Uma transação é um bloco de instruções SQL relacionadas, tratadas como uma ação única, e subseqüentemente chamada novamente no caso de qualquer uma das instruções individuais falhar ou encontrar resultados inesperados. É importante entender que para cada instrução na transação, o bando de dados mostrará qualquer mudança feita pelas instruções anteriores na mesma transação. Qualquer outra conexão acessando o banco de dados fora do escopo da transação não verá as mudanças até que a transação inteira tenha sido concluída, ou será proibido de utilizar estes dados até que a transação tenha terminado o que está fazendo.

O comportamento do banco de dados durante a transação é configurável, mas limitado às capacidades do banco de dados com as quais está se trabalhando. Esta habilidade de bloquear o acesso aos dados com os quais se está trabalhando permite que desenvolva-se transações compostas de uma complexa série de etapas sem ter que se preocupar em deixar o banco de dados em um estado inválido.

Depois que as instruções que compõem a transação estiverem completas, basta avisar o banco de dados para aceitar estas modificações como finais, através do método `commit` de seu objeto `Connection`. Do mesmo modo, para revogar qualquer mudança feita desde o ínicio da transação, simplesmente chama-se o método `rollback` do objeto `Connection` que retorna o banco de dados para o último estado no qual estava depois que a última transação foi submetida.

Como padrão, JDBC assume que quer-se tratar cada instrução SQL como sua própria transação. Este recurso é conhecido como *autocommit*, onde cada instrução é submetida automaticamente, tão logo é emitida. Para iniciar um bloco de instruções sobre controle de transação, deve-se desativar o recurso *autocommit*, como mostrado no exemplo que é apresentado a seguir:

```
connection.setAutoCommit(false);
try {
     Statement stmt = connection.createStatement();
     stmt.execute(“UPDATE CONTA SET SALDO = SALDO - 100
WHERE ID = 1”);
     stmt.execute(“UPDATE CONTA SET SALDO = SALDO + 100
WHERE ID = 2”);
} catch (SQLException e) {
     connection.rollback();
} finally {
```

```
    connection.setAutoCommit(true);
}
```

No exemplo, volta-se a transação se ocorrer algum problema. Existem diversas razões pelas quais um problema pode ocorrer. Por exemplo, as contas de ID 1 e 2 podem não existir, ou suas contas podem ser inacessíveis para este programa. A conta 1 pode não ter fundos suficientes para cobrir a transação, ou ainda poderia ocorrer algum problema de comunicação entre a aplicação e o banco de dados entre a primeira e a segunda instrução. Assim, coloca-las em uma transação garante que o processo inteiro seja concluído ou que o processo inteiro falhe, nada além disso.

5.2.3 Utilizando Metadados

Nas últimas sessões foram apresentados os mecanismos para preencher, consultar, atualizar e realizar transações sobre uma base de dados via JDBC. Entretanto, o JDBC pode ainda fornecer informações adicionais sobre a estrutura de um banco de dados e suas tabelas. Por exemplo, pode-se obter uma lista das tabelas de um banco de dados específico ou os nomes e tipos de colunas de uma determinada tabela.

Estas informações podem não ser tão úteis quando se está implementando uma aplicação para um banco de dados específico. Afinal, se existe o projeto do banco de dados, as tabelas e sua estrutura são conhecidas. Entretanto, as informações estruturais são extremamente úteis para que os programadores escrevem ferramentas para qualquer banco de dados.

Em SQL, os dados que descrevem o banco de dados ou uma de suas partes são chamados de *metadados*, principalmente para distingui-los dos dados reais que são armazenados no banco de dados. Pode-se obter dois tipos de *metadados*: sobre um banco de dados e sobre um conjunto de resultado (`ResultSet`).

Para descobrir informações sobre o banco de dados, precisa-se solicitar um objeto do tipo `DatabaseMetaData` a partir de um objeto de conexão `Connection`:

```
DatabaseMetaData md = connection.getMetaData();
```

Os bancos de dados são complexos e o padrão SQL deixa muito espaço para variação. Existe bem mais de cem métodos na classe `DatabaseMetaData` para obter informações sobre o banco de dados.

Muitos destes métodos se destinam a usuários avançados com necessidades especiais, em particular, aqueles que precisam escrever código altamente portável. Como exemplo tem-se o método `getTables` que permite listar todas as tabelas de um banco de dados:

```
ResultSet rs = md.getTables(null, null, null, new String[]
{“TABLE”});
```

Este método retorna um conjunto de resultados contendo informações sobre todas as tabelas do banco de dados, cada linha do conjunto de resultados contém informações sobre uma tabela, assim, para obter o nome de cada tabela basta solicitar a terceira entrada, como mostrado a seguir:

```
while (rs.next()) {
     String tableName = rs.getString(3);
}
rs.close();
```

Os *metadados* mais utilizados e também mais interessantes são aqueles a respeito de conjuntos de resultados. Quando tem-se um conjunto de resultados de uma consulta, pode-se perguntar sobre o número de colunas e nome, tipo e largura de cada coluna. O exemplo abaixo ilustra esta funcionalidade:

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM DUAL”);
ResultSetMetadata rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
     String columnName = rsmd.getColumnLabel(i);
     int columnWidth = rsmd.getColumnDisplaySize(i);
}
```

5.3 Java Data Objects – JDO

Java é uma linguagem que define um ambiente de execução onde classes definidas por usuários são executadas. Instâncias destas classes em geral representam dados do mundo real. Estes dados, os quais representam estados de objetos, talvez precisem ser armazenados em bancos de dados, sistemas de arquivos ou em algum sistema de processamento de transações de grande porte (*mainframe*). Estas fontes de dados são classificadas como “*Enterprise Information Systems*”, ou simplesmente EIS. Somados a isto, sistemas de menor porte freqüentemente exigem uma forma de gerenciar dados persistentes em armazenamento de dados locais.

As técnicas de acesso a dados são diferentes para cada tipo de fonte de dados. Realizar o acesso a dados representa um desafio para os desenvolvedores de aplicações, que geralmente precisam utilizar uma interface de programação de aplicações (API) para cada tipo de fonte de dados. Isto significa que os desenvolvedores de aplicações precisam aprender pelo menos duas linguagens diferentes a fim de desenvolver uma lógica de negócios para estas fontes de dados: no caso a linguagem de programação Java e uma linguagem específica para acesso aos dados necessários a partir da fonte de dados.

5.3.1 Arquitetura

Muitas implementações de JDO – possivelmente múltiplas implementações, conforme o tipo de EIS ou armazenamento local – são conectáveis em um servidor de aplicações ou utilizadas diretamente em duas camadas, ou ainda, em uma arquitetura embutida. Isto permite que os componentes da aplicação, desenvolvidos para uma camada intermediária do servidor de aplicações ou em uma camada cliente, acesse as fontes de dados básicas, utilizando uma interface padrão e uma visão centrada em Java dos dados. A implementação de JDO fornece o mapeamento necessário de objetos Java em tipos de dados especiais e o seu relacionamento com as fontes de dados fundamentais.

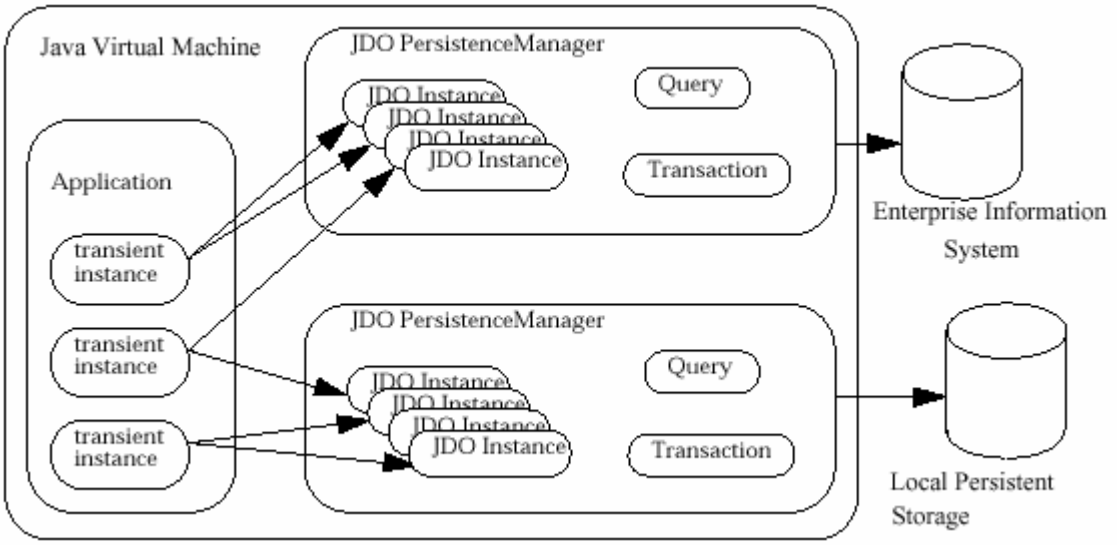

Figura 5.4 – Visão geral de uma arquitetura JDO não-gerenciada.

Em um ambiente não gerenciado, como o apresentado na Figura 5.4, a implementação JDO abstrai as características específicas do ambiente EIS, tais como, mapeamento de tipos de dados, mapeamento dos relacionamentos, mecanismo de recuperação de dados e armazenamento. Os componentes da aplicação possuem somente a visão Java, tendo seus dados organizados em classes com relacionamentos e coleções apresentados como construtores Java nativos.

Ambientes gerenciados adicionalmente fornecem transparência para que os componentes da aplicação utilizem mecanismos fornecidos no nível do sistema, por exemplo, transações distribuídas, segurança e gerenciamento de conexões, por ocultar os contratos/interfaces entre o servidor de aplicações e as implementações de JDO.

Com os ambientes gerenciáveis e não gerenciáveis, um desenvolvedor de componentes para aplicações pode manter seu foco no desenvolvimento do negócio e na lógica de apresentação para os componentes da aplicação, sem se envolver com os detalhes relacionados a conectividade com um EIS específico.

Utilização em duas camadas

Para uma utilização básica em duas camadas, JDO publica para os componentes da aplicação duas interfaces Java primárias: `javax.jdo.PersistenceManager`, a partir da qual os serviços são solicitados e `javax.jdo.PersistenceCapable`, que fornece a visão do tratamento das classes persistentes definidas pelo usuário.

A interface `PersistenceManager` fornece serviços tais como gerenciamento de consultas, gerenciamento de transações, e gerenciamento do ciclo de vida para as instâncias das classes que possuem persistência.

A interface `PersistenceCapable` fornece os serviços relacionados ao gerenciamento do estado do ciclo de vida para as instâncias das classes que possuem persistência.

Utilização em servidores de aplicação

Para a utilização em servidores de aplicação, a arquitetura JDO utiliza a arquitetura J2EE Connector que define um conjunto de interfaces em nível de sistema entre o servidor de aplicações e os conectores EIS. Estes contratos de nível do sistema são implementados em um adaptador de recursos do lado do EIS.

O gerenciador de persistência JDO é um gerenciador de cache como definido na arquitetura J2EE Connector, onde é possível utilizar o seu próprio adaptador de recursos nativo ou um adaptador de recursos desenvolvido por terceiros. Se o gerenciador de persistência do JDO possui o seu próprio gerenciador de recursos, portanto ele deve implementar os contratos de baixo nível do sistema especificados na arquitetura J2EE Connector. Estes contratos incluem as seguintes interfaces Java:

- `ResourceAdapter,`
- `ConnectionManager,`
- `ManagedConnectionFactory,`
- `ManagedConnection e`
- `LocalTransaction.`

As transações JDO precisam implementar a interface de sincronização `Synchronization` de forma que ao completar uma transação estes eventos possam causar a atualização do seu estado através do conector com o EIS.

Os componentes da aplicação não são capazes de distinguir entre implementações JDO que utilizam adaptadores de recursos nativos e implementações JDO que utilizam adaptadores de recursos de terceiros. De qualquer forma, o desenvolvedor precisa entender que existem dois componentes que devem ser configurados: o gerenciador de persistência do JDO e o seu respectivo adaptador de recursos de baixo nível.

Por conveniência, a classe `PersistenceManagerFactory` fornece a interface necessária para a configuração do adaptador de recursos de baixo nível.

Um adaptador de recursos provido pelo fornecedor JDO é chamado um adaptador de recursos nativo, e a sua interface é específica para aquele fornecedor JDO. É um *driver* de software no nível de sistema que é utilizado pelo servidor de aplicações ou por um cliente da aplicação para realizar a sua conexão com o gerenciador de recursos.

O adaptador de recursos é conectado em um container (fornecido pelo servidor de aplicações). Os componentes da aplicação armazenados neste container podem então utilizar a API cliente encapsulada por `javax.jdo.PersistenceManager` para acessar o gerenciador de persistência do JDO. A implementação do JDO por sua vez, utiliza a interface que do adaptador de recursos de baixo nível especifica para o armazenamento dos dados. O adaptador de recursos e o servidor de aplicações colaboram para fornecer os mecanismos de baixo nível, como transações, segurança e conjunto de conexões, para a conectividade com o EIS.

O adaptador de recursos é alocado dentro do mesmo espaço de endereçamento da implementação de JDO que vai usá-lo.

Exemplos de adaptadores de recursos JDO nativos são:

- Bancos de dados Objeto/Relacionais que utilizam seus próprios *drivers* nativos para a conexão com bases de dados objeto relacionais;

- Bancos de dados orientados a objeto que armazenam seus objetos Java diretamente em bases de dados orientadas a objetos.

Exemplos de adaptadores de recursos não nativos são:

- Mapeamento Objeto/Relacional que utilizam *drivers* JDBC para a conexão com bases de dados relacionais;

- Mapeamento hierárquico que usam ferramentas para a conectividade com *Mainframes* utilizando sistemas transacionais hierárquicos.

Conjunto de conexões (Pooling)

Existem dois níveis de *pooling* na arquitetura JDO. Os gerenciadores de persistência podem utilizar conjuntos de conexões, e as conexões de baixo nível com as fontes de dados também podem utilizar conjuntos de conexões independentes.

Os conjuntos de conexões são administrados pelos contratos da arquitetura de conexão. O *pooling* nos gerenciadores de persistência é uma característica opcional da implementação JDO, e sua utilização não é padronizada para aplicações em duas camadas. Para ambientes gerenciados, o *pooling* do gerenciador de persistência é exigido para manter a correção das transações associadas com os gerenciadores de persistência.

Por exemplo, uma instância de um gerenciador de persistência JDO pode ser utilizado por uma sessão executando uma transação otimista de longa duração. Esta instância não poderá ser utilizada por nenhum outro usuário durante toda a execução desta transação.

Durante a execução de um método de negócio associado com uma sessão, uma conexão pode ser necessária para obter dados de uma dada fonte de dados. O gerenciador de persistência irá solicitar uma conexão do conjunto de conexões para satisfazer a requisição. Ao término da execução do método de negócio, a conexão é retornada para o seu conjunto mas o gerenciador de persistência continua mantendo a sessão.

Após a finalização da transação otimista, a instância do gerenciador de persistência pode ser devolvida para o seu conjunto e reutilizada por uma próxima transação.

Contratos

A arquitetura JDO especifica os contratos em nível de aplicação entre os componentes da aplicação e o gerenciador de persistência JDO.

A arquitetura J2EE Connector especifica os padrões de contratos entre os servidores de aplicação e um conector EIS utilizado por uma implementação JDO. Estes contratos são necessários para que uma implementação JDO possa ser utilizada em um ambiente de execução em um servidor de aplicações. A arquitetura do conector define aspectos importantes da integração: gerenciamento de conexões, gerenciamento de transações e segurança.

O contrato do gerenciador de conexões é implementado por dois adaptadores de recursos: um JDO nativo e outro EIS de baixo nível.

O contrato do gerenciamento de transações é a camada entre o gerenciador de transações (separado logicamente do servidor de aplicações) e o gerenciador de conexões. Suportando distribuição através de vários servidores de aplicações e sistemas de gerenciamento de dados heterogêneos.

O contrato de segurança é necessário para garantir a segurança pela conexão JDO com o mecanismo de armazenagem de dados de baixo nível.

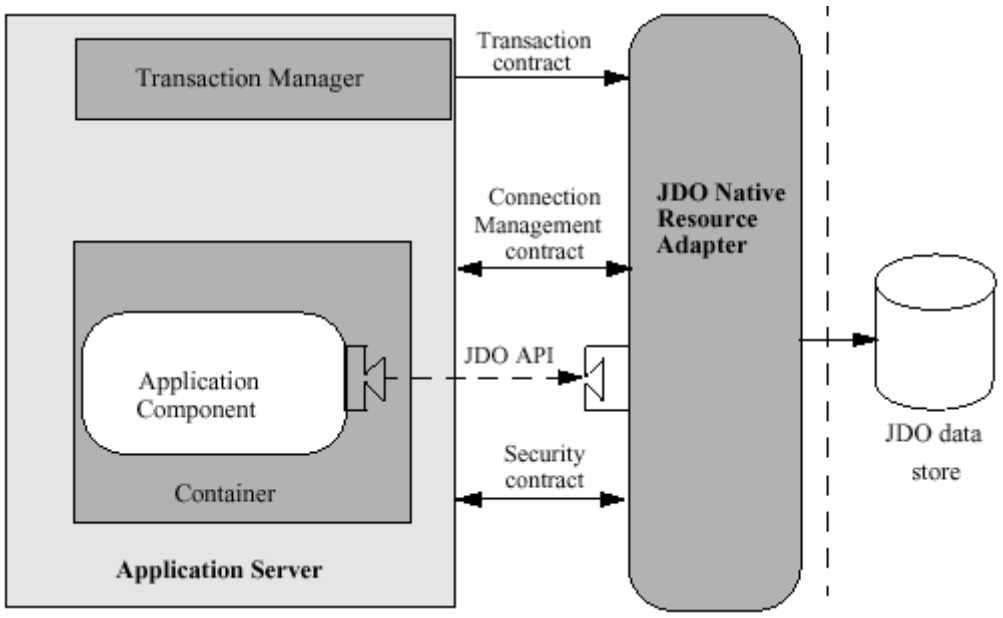

Figura 5.5 – Contratos entre servidor e adaptador de recursos JDO.

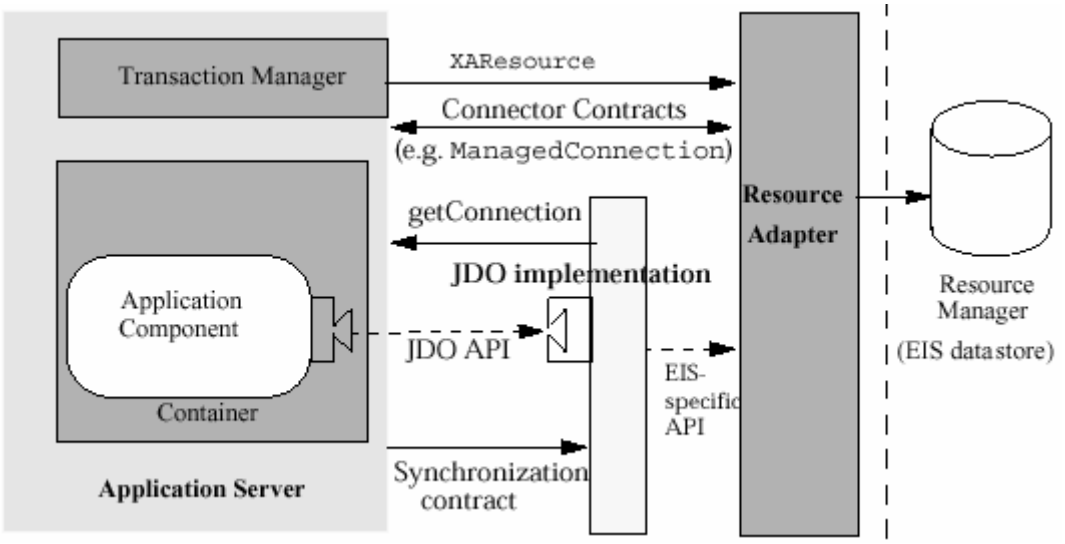

Figura 5.6 – Contratos entre o servidor de aplicações e as camadas do JDO.

A Figura 5.5 e a Figura 5.6 acima ilustram os relacionamentos entre uma implementação de JDO desenvolvido por um fornecedor terceiro e um adaptador de recursos de um EIS nativo.

5.3.2 Ciclo de Vida

As instâncias JDO podem ser transientes ou persistentes. Isto é, elas podem representar o estado persistente dos dados contidos em uma base de dados transacional. Se uma instância JDO é transiente, então não existe diferença entre o seu comportamento e o comportamento de uma instância não JDO de uma classe Java definida pelo usuário.

Se uma instância JDO é persistente, seu comportamento está ligado a base de dados transacional com a qual ela está associada. A implementação JDO automaticamente rastreia as modificações realizadas nos valores de uma instância, automaticamente valida esta instância com a sua representação na base de dados e salva os novos valores nesta operação, como exigido para a preservação da integridade transacional dos dados manipulados.

Durante a vida de uma instância JDO, esta transita entre vários estados, até que finalmente é destruída pelo coletor de lixo da máquina virtual Java, no processo denominado *garbage collection*. Durante sua vida, as transições de estado são administradas pelo seu comportamento agindo sobre si mesmo, através da execução de métodos de instância, bem como, pelo comportamento administrado pelo gerenciador de persistência do JDO, acionado pela própria aplicação ou pelo ambiente de execução, incluindo neste o gerenciador de transações.

Durante o seu ciclo de vida, por vezes uma instância pode estar inconsistente com a sua representação na base de dados, como no começo de uma transação. Se uma instância está inconsistente, diz-se que está “suja”, sendo a notação para este estado no ambiente JDO: *dirty*. As instâncias recentemente tornadas persistentes, apagadas ou alteradas em uma transação ficam “sujas”.

Por vezes, a implementação do ambiente JDO precisa armazenar o estado das instâncias persistentes na base de dados. Este processo é denominado *flushing* (esvaziamento), não afetando o estado *dirty* das instâncias.

Sob o controle da aplicação, as instâncias transientes devem observar os limites das transações, nas quais o estados das instâncias é também preservado (*commit*) ou restaurado (*rollback*).

As instâncias transientes que obedecem os limites das transações são denominadas instâncias transacionais transientes. O suporte a instâncias transacionais transientes é uma opção JDO, isto é, uma implementação JDO concordante não requer a implementação das API´s que tratam as transições de estado associadas com instâncias transacionais transientes.

Sob o controle da aplicação, as instâncias JDO persistentes não precisam observar os limites das transações. Estas instâncias são denominadas instâncias não-transacionais persistentes, e o ciclo de vida destas instâncias não é afetado pelos limites das transações. O suporte para instâncias não-transacionais é opcional na implementação do JDO.

Na implementação padrão, se uma instância JDO é persistente ou transacional, esta deve conter uma referência válida para uma instância do gerenciador de estados do JDO, sendo este o responsável por gerenciar as alterações de estado de uma instância JDO e pela sua interface com gerenciador de persistência do JDO.

O ciclo de vida de uma instância JDO possui os seguintes objetivos:

- O fato da persistência precisar ser transparente para o desenvolvedor de instâncias JDO, bem como para o desenvolvedor de componentes para as aplicações;

- As instâncias JDO devem possuir a capacidade de uso eficiente em uma variedade de ambientes, incluindo situações gerenciadas (servidores de aplicação) e não gerenciadas (ambientes de duas camadas);

- Vários gerenciadores de persistência precisam co-habitar e provavelmente compartilhar as mesmas classes que implementem persistência, ainda que qualquer instância JDO somente possa estar associada com zero ou um gerenciador de persistência de cada vez.

Instâncias JDO

Para as instâncias JDO transientes, não é necessária nenhuma infra-estrutura para suporte. Isto é, uma instância transiente nunca irá fazer chamadas para métodos fora do domínio da aplicação. Não há necessidade de instanciar objetos fora do domínio da aplicação. Na parte relacionada aos métodos adicionais disponíveis para a investigação de estado durante o ciclo de vida, não há diferença no comportamento entre instâncias transientes de classes aprimoradas, com suporte a JDO, e instâncias transientes de classes sem o respectivo suporte.

As instâncias persistentes JDO executam em um ambiente que contém uma instância do gerenciador de persistência JDO, responsável pelo seu comportamento persistente. Na implementação padrão, as instâncias JDO contém uma referência para uma instância do gerenciador de estados JDO, responsável pela transição de estados das instâncias, bem como pelo gerenciamento do conteúdo dos campos das instâncias. O gerenciador de persistência e o gerenciador de estados podem ser implementados pela mesma instância, mas as suas interface são distintas.

O contrato entre as classes que possuem a capacidade de persistência e os demais componentes da aplicação estende o contrato entre as classes não persistentes e os componentes da aplicação. Estas extensões nos contratos suportam o questionamento sobre o estado do ciclo de vida das instâncias e são destinadas ao uso pelos gerenciadores das partes do sistema.

Gerenciador de estados JDO

Na implementação padrão, as instâncias JDO não transientes contém uma referência para um gerenciador de estados JDO, para o qual todas as questões são delegadas. A instância do gerenciador de estados associada cuida das trocas de estado da instância JDO e realiza a interface com o gerenciador de persistência JDO, para que este gerencie os valores dos dados armazenados.

Identidade JDO

Java define dois conceitos para determinar de duas instâncias de um objeto são a mesma instância (*identity*), ou se representam os mesmos dados (*equality*). JDO estendem estes conceitos para determinar se duas instâncias representam o mesmo objeto de dados armazenado.

A identidade de um objeto Java é completamente gerenciada pela máquina virtual Java (JVM). As instâncias são idênticas se e somente se elas ocupam o mesmo espaço de armazenamento internamente a JVM.

A igualdade de um objeto Java é determinada pela sua classe. Instâncias distintas são iguais se elas representam os mesmos dados, como o mesmo valor para um inteiro, ou bits equivalentes em uma matriz.

A interação entre a identidade e igualdade de um objeto Java é um tópico importante para os desenvolvedores JDO. A igualdade entre objetos Java é um conceito específico da aplicação, e a implementação de JDO não deve alterar a implementação da igualdade na aplicação. Ainda, as implementações JDO precisam gerenciar a *cache* das instâncias JDO, do mesmo modo, que somente uma única instancia JDO pode estar associada com cada gerenciador de persistência JDO, representando o estado de cada objeto armazenado correspondente. Então, JDO define a identidade dos objetos de modo diferente da identidade de objetos da JVM e da igualdade na aplicação.

Note que as aplicações podem implementar igualdade para classes que possuam persistência de forma distinta da implementação padrão que simplesmente utiliza o identificador de objeto JVM. Isto porque o identificador de objeto JVM de uma instância persistente pode não garantir a igualdade entre os gerenciadores de persistência e através do espaço e tempo, exceto em casos muito específicos citados abaixo.

Adicionalmente, se as instâncias persistentes são armazenadas na base de dados e em seguida consultadas utilizando o operador de comparação ==, ou são referenciadas por uma coleção persistente que força a igualdade (*Set*, *Map*) então a implementação de *equals* precisa ser exatamente igual a implementação JDO para a igualdade, utilizando a chave primária ou o *ObjectId* como chave. Isto não é obrigatório, mas se não for corretamente implementado as semânticas das coleções padrão e das coleções JDO podem diferenciar-se.

Para evitar confusão entre a identidade de objetos Java e JDO, este trabalho irá se referenciar ao conceito JDO como identidade JDO.

JDO define três tipos de identidade JDO:

- Identidade JDO gerenciada pela aplicação e reforçada pela base de dados, freqüentemente chamada de chave primária;
- Identidade JDO gerenciada pela base de dados sem ser relacionada com qualquer valor de instância JDO;
- Identidade gerenciada pela implementação garantindo a exclusividade na JVM mas não na base de dados.

O tipo de identidade JDO utilizado é uma propriedade da classe JDO `PersistenceCapable` e é definido em tempo de análise.

A representação da identidade JDO na JVM é via um identificador de objeto JDO. Cada uma das instâncias persistentes (instâncias Java representando um objeto persistente) possui um identificador de objeto correspondente. Pode haver uma instância representado o identificados de objeto ou não. A instância do objeto que representa o identificador de objeto para uma instância persistente precisa ser adquirido pela aplicação em tempo de execução, salvo em um armazenamento estável, por serialização ou outra técnica, e utilizado depois para a obtenção de uma referência para o mesmo objeto de dados armazenado.

A classe que representa o identificador de objeto para uma classe não gerenciada pela aplicação é definida pela implementação JDO. A implementação pode escolher para ser utilizada qualquer classe que satisfaça as necessidades de um tipo específico de identidade JDO para uma classe. Pode escolher a mesma classe para várias classes JDO diferentes, ou talvez utilizar uma classe diferente para cada classe JDO.

A classe que representa o identificador de objeto para uma classe gerenciada pela aplicação é definida pela própria aplicação em meta dados, e pode ser fornecida pela aplicação ou por alguma ferramenta JDO de terceiros.

A representação visível da identidade JDO é uma instância que está completamente sobre o controle da aplicação. Os dois métodos da implementação JDO que podem ser utilizados pela instância são `getObjectId` e `getObjectById`. As instâncias utilizadas nesses métodos nunca serão salvas internamente, elas são cópias da representação interna ou utilizadas para encontrar instâncias da representação interna.

Então, o objeto retornado por qualquer chamada a `getObjectId` pode ser alterado pelo usuário, mas a modificação não afeta a identidade do objeto que foi originalmente referenciado. Isto é, a chamada a `getObjectId` retorna somente uma cópia do objeto identidade usado internamente pela implementação.

Isto é necessário para que a instância retornada pela chamada a `getObjectById(Object)` de instâncias diferentes do gerenciador de persistência, retornadas pelo mesmo *PersistenceManagerFactory* representem o mesmo objeto persistente, mas com identificadores de objeto Java diferentes.

Além disso, qualquer instância retornada por uma chamada a `getObjectById(Object)` com o mesmo identificador de objeto para a mesma instância do gerenciador de persistência precisa ser idêntico, assumindo que as instâncias não foram removidas pelo *garbage collector* entre as chamadas.

A identidade JDO para instâncias transientes não é definida. A tentativa de obter o identificados de objeto para uma instância transiente irá retornar *null*.

Unicidade

A identidade JDO de uma instância persistente é gerenciada pela implementação. Para uma identidade JDO gerenciada (base de dados ou chave primária), existe somente uma instância persistente associada com um objeto de dados armazenado específico por instância `PersistenceManager`, assim sendo a instância persistente é obtida:

- `PersistenceManager.getObjectById(Object oid, boolean validate);`
- Consultada via uma instância `Query` associada com a instância do `PersistenceManager`;
- Navegando a partir de uma instância persistente associada com a instância do `PersistenceManger`;
- `PersistenceManager.makePersistent(Object pc);`

Estados do ciclo de vida

Existem dez estados definidos na especificação JDO. Destes sete estados são obrigatórios e três estados são opcionais. Se uma implementação não suportar certas operações, então estes três estados não serão atingidos.

A Figura 5.7 a seguir, ilustra todos os estados possíveis e explica como um objeto persistente, gerenciado por uma instância JDO pode realizar a transição entre diferentes estados.

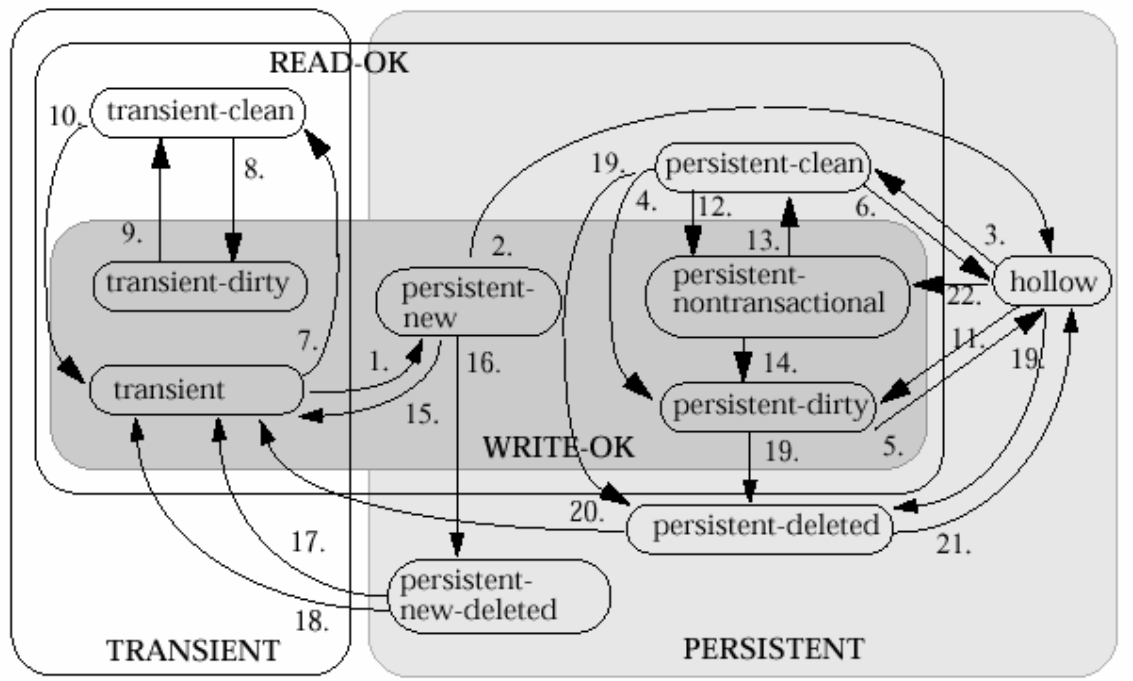

Figura 5.7 – Transições de estado de instancias JDO.

1. Uma instância transiente muda para *persistent-new* quando a instância for passada como parâmetro para o método `makePersistent`.

2. Uma instância *persistent-new* muda para *hollow* quando a transação na qual ela tornou-se persistente realizar o seu *commit.*

3. Uma instância *hollow* muda para *persistent-clean* quando uma campo é lido.

4. Uma instância *persistent-clean* muda para *persistent-dirty* quando um campo é escrito.

5. Uma instância *persistent-dirty* muda para *hollow* durante a execução de um *commit* ou *rollback.*

6. Uma instância *persistent-clean* muda para *hollow* durante a execução de um *commit* ou *rollback.*

7. Uma instância transiente muda para *transient-clean* quando for passada como parâmetro para o método `makeTransactinal`.

8. Uma instância *transient-clean* muda para *transient-dirty* quando um campo é escrito.

9. Uma instância *transient-dirty* muda para *transient-clean* durante a execução de um *commit* ou *rollback.*

10. Uma instância *transient-clean* muda para transiente quando é passada como parâmetro para o método `makeNontransactinal`.

11. Uma instância *hollow* muda para *persistent-dirty* quando um campo é escrito.

12. Uma instância *persistent-clean* muda para *persistent-nontransactional* quando é passada como parâmetro para o método `makeNontransactional`.

13. Uma instância *persistent-nontransactional* muda para *persistent-clean* quando é passada como parâmetro para o método `makeTransactional`.

14. Uma instância *persistent-nontransactional* muda para *persistent-dirty* quando um campo é escrito em uma transação.

15. Uma instância *persistent-new* muda para transiente durante a execução de um *rollback*.

16. Uma instância *persistent-new* muda para *persistent-new-deleted* quando é passada como parâmetro para o método `deletePersistent`.

17. Uma instância *persistent-new-deleted* muda para transiente durante a execução de uma *rollback*. Os valores dos campos são restaurados como na chamada do método `makePersistent`.

18. Uma instância *persistent-new-deleted* muda para transiente durante a execução de um *commit*. A instância perde todos os seus valores.

19. Uma instância *hollow*, *persistent-clean*, ou *persistent-dirty* muda para *persistent-deleted* quando passada como parâmetro para o método `deletePersistent`.

20. Uma instância *persistent-deleted* muda para transiente quando a transação na qual ela foi apagada realiza um *commit*.

21. Uma instância *persistent-deleted* muda para *hollow* quando a transação na qual ela foi apagada realiza um *rollback*.

22. Uma instância *hollow* muda para *persistent-nontransactional* quando a opção `NontransactionalRead` estiver ativada, um campo for lido e estiver ocorrendo uma transação otimista ou não houver uma transação em andamento.

6 Desenvolvimento de Interfaces para a Web

A apresentação de informações por um navegador Web, em sua forma mais simples, consiste na solicitação de um documento HTML para um servidor da Web, este servidor encontra o arquivo correspondente e o devolve para que o navegador realize a sua exibição. Se este documento HTML incluir qualquer imagem, o navegador por sua vez irá submeter novas solicitações para obter os documentos de imagem também.

Como descrito aqui, todas estas solicitações são para arquivos estáticos, ou seja, os documentos que são solicitados nunca mudam, nem dependendo que quem os solicitou e nem caso parâmetros adicionais sejam incluídos nesta solicitação. Em tais situações, o servidor Web precisa apenas localizar o arquivo correspondente ao documento solicitado e responder ao navegador da web com o conteúdo daquele arquivo.

No entanto, a maioria dos dados fornecidos através da web hoje tem uma natureza dinâmica. Os dados são dinâmicos por natureza, porque a informação está baseada em informações alteradas constantemente, ou ainda, devem ser personalizados para cada usuário individual, ou ambos.

Várias tecnologias se propõem a realizar a geração de conteúdo dinâmico para a Web, o presente capítulo apresenta a tecnologia proposta pela Sun como extensões para a linguagem Java neste domínio de problema.

6.1 Java Server Pages - JSP

JSP é uma tecnologia baseada em Java que simplifica o processo de desenvolvimento de sites da Web dinâmicos. Com JSP, os designers da Web e programadores podem rapidamente incorporar elementos dinâmicos em páginas da Web usando Java embutido e algumas tags de marcação simples.

A tendência atual é colocar mais processos de negócios na Web, centrados nos mercados *business-to-consumer* (B2C) ou *business-to-business* (B2B). Estas aplicações trazem a tona algumas das desvantagens do projeto convencional de Web sites, utilizando CGI´s por instância. Os principais problemas encontrados pelos desenvolvedores de aplicações Web são:

- **Escalabilidade –** um site de sucesso terá muito mais usuários do que a Intranet de uma grande corporação. E o número de usuários está aumentando todo o tempo, então sua solução precisa acompanhar esta demanda;

- **Integração entre *Backend Data* e *Business Logic* –** a Web é somente outra forma de conduzir negócios, então deve ser capaz de utilizar as mesmas camadas, intermediárias (*Middleware*) e código de acesso aos dados. A integração de requisições Web-based com o sistema *backend* pode ser um fator de grandes problemas no projeto de um sistema;

- **Administração –** *sites* tendem a aumentar de tamanho, e é necessário alguma forma de gerenciamento do seu conteúdo e da sua interação com os sistemas de negócio. As técnicas de programação orientadas a objeto tem provado em muito seu valor em ambientes cliente-servidor e podem ajudar no desenvolvimento de aplicações Web;

- **Personalização –** adicionar um toque pessoal para a página não é somente cosmética, é essencial para manter seu cliente. Conhecendo suas preferências, permitindo que ele configure as informações que quer visualizar, lembrando suas últimas compras ou termos freqüentemente procurados, são pontos importantes e fornecem um melhor *feedback* e interação do que uma comunicação unidirecional.

De todos estes pontos dependem o sucesso da arquitetura de uma aplicação, de outro modo temos uma batalha perdida. A arquitetura pode ser em muito auxiliada por um bom *framework* que forneça uma visão Web do negócio.

6.1.1 J2EE

O *framework* oficial para o desenvolvimento de aplicações de negócios é o Java 2 Enterprise Edition (J2EE). Existem outras soluções, como WebObjects da Apple, mas o J2EE tem o apoio da Sun, IBM, Oracle, BEA, Apache Group e um conjunto de fornecedores de servidores de aplicação.

O J2EE fornece:

- Uma plataforma para aplicações de negócio, com suporte completo de uma API para código de negócio e garantia de portabilidade entre diferentes implementações de servidores;

- Uma clara divisão entre código que manipula apresentação, lógica de negócios e dados.

A plataforma J2EE consiste em um conjunto de API´s, separadas em três camadas na seguinte ordem aproximada:

1ª Camada - Cliente

- JavaServer Pages

- Servlets

- Java support for XML

2ª Camada – Lógica da aplicação

- Enterprise JavaBeans (EJB´s)
- Java Messaging
- Java Transaction support

3ª Camada - Dados

- JavaMail
- Java Naming and Directory Interface (JNDI)
- JDBC
- Java support for CORBA

A Figura 6.1 apresenta esta visão em três camadas e o inter-relacionamento de cada tecnologia que forma o framework oficial para o desenvolvimento de aplicações de negócio J2EE e seu modelo de aplicação.

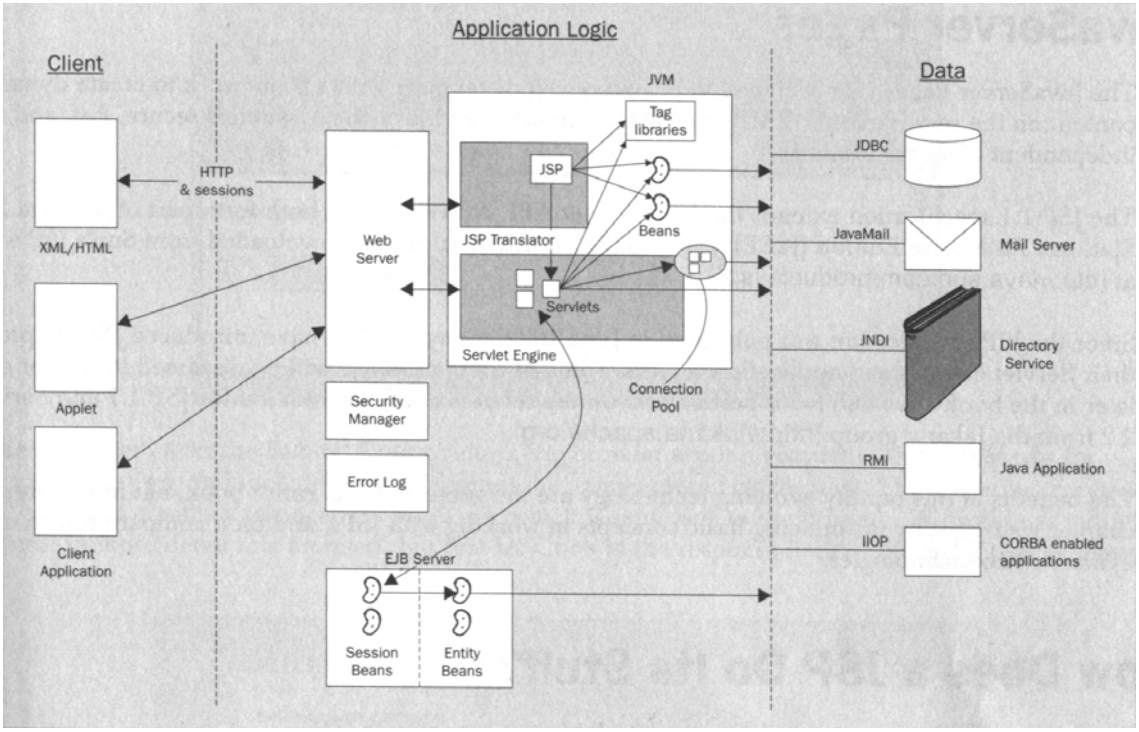

Figura 6.1 – Arquitetura J2EE

A especificação J2EE vem de encontro as necessidades das aplicações Web porque provê:

- Grande interação com o Web Server via Servlets, os quais fornecem objetos que representam os elementos de uma requisição HTTP e sua resposta;
- Suporte embutido para sessões, ambos em Servlets e EJB´s;
- uso de EJB´s para minimizar a interação do usuário com os dados, fornecendo um suporte automático para sessões e transações nas operações realizadas pelos EJB´s no servidor EJB;
- Entity EJB para representar dados como objetos;
- Suporte nativo (Seamless) com as API´s Java de acesso a dados;
- Saída flexível baseada em Templates, utilizando JSP e XML.

Esta família de API´s fornece o suporte para que a página Web final possa ser gerada a partir da requisição de um usuário, a qual será processada por um Servlet ou JSP e uma sessão EJB, utilizando os dados extraídos de um banco de dados e colocados em um *entity* EJB.

A especificação JavaServer Pages 1.1 disponibiliza aos desenvolvedores internet um *framework* para criar conteúdo dinâmico no servidor Web utilizando templates HTML e XML, e código Java, o qual é seguro, rápido e independente da plataforma servidora.

A especificação JSP 1.1 estende a API Java Servlet que, como mencionado anteriormente, também faz parte da Java 2 Plataform Enterprise Edition (J2EE). Desde que a especificação JSP foi publicada em Junho de 1999, vários fornecedores tem introduzido suporte a JSP em seus Servlets Engines e Application Servers. Informações específicas sobre JSP podem ser obtidas no web site da Sun sobre a tecnologia JSP em [URL09], para exemplificar o funcionamento deste mecanismo temos o seguinte formulário HTML:

```
<HTML>
<HEAD>
  <TITLE>JSP Example</TITLE>
</HEAD>

<BODY>
  <P>Quantas vezes ?</P>
  <FORM METHOD="GET" ACTION="Hello.jsp">
    <INPUT TYPE="TEXT" SIZE="2" NAME="num">
    <INPUT TYPE="SUBMIT">
  </FORM>
</BODY>
</HTML>
```

Hello.htm

```
<%@ page language=”java” %>

<HTML>
<HEAD>
  <TITLE>JSP Example</TITLE>
</HEAD>

<BODY>
<P>
<% int num = Integer.parseInt(request.getParameter(“num”));
   for(int i=0; i < num; i++) {
%>
Hello World!<BR>
<% } %>
</P>
</BODY>
</HTML>
```

Hello.jsp

Como podemos observar, em grande parte temos simplesmente HTML, intercalado com tags especiais JSP, destacadas abaixo:

- <%@ page language=”java” %> - É uma diretiva de página JSP, denotada pela tag <%@, que identifica os atributos para a página. Neste caso, indica que o código contido nesta página está escrito em Java;

- <% ... %> - Aparecem duas vezes como um par de *scriptlets* em cada um dos lados de uma linha de código HTML. A tag *scriptlet* confina o código Java específico de cada página.

O código HTML recebido pelo Browser é mostrado abaixo. Comparando este com o conteúdo de Hello.jsp, pode-se observar que as tags JSP foram interpretadas e substituídas na resposta.

```
<HTML>
<HEAD>
  <TITLE>JSP Example</TITLE>
</HEAD>

<BODY>
  <P>
    Hello World!<BR>
    Hello World!<BR>
    Hello World!<BR>
    Hello World!<BR>
    Hello World!<BR>
  </P>
</BODY>
</HTML>
```

Http Response

O ciclo de vida e o funcionamento de uma página JSP é ilustrado na Figura 6.2 e descrito a seguir:

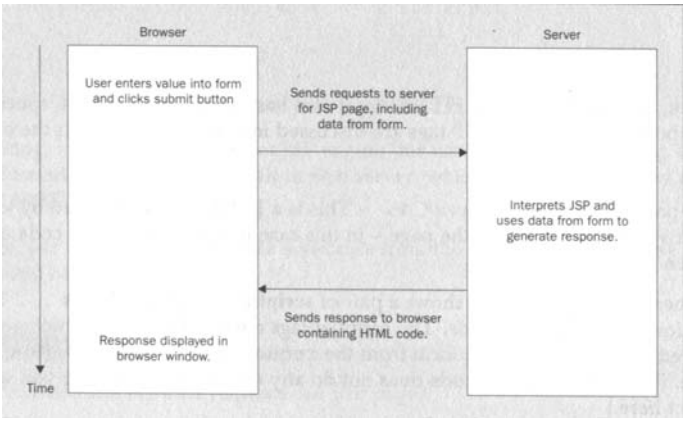

Figura 6.2 – Ciclo de vida

O que acontece quando o navegador solicita uma página JSP é o seguinte:

1. O navegador envia uma requisição para o servidor Web, solicitando uma URL qualquer, temos por exemplo: http://localhost:8080/Hello.jsp?num=5. Isto especifica um valor para a variável “num” obtido como um parâmetro da função GET;

2. O Servidor Web reconhece a extensão do arquivo .jsp na URL solicitada pelo Browser, indicando que o recurso solicitado é uma JavaServer Page, e que esta requisição precisa ser manipulada por um *engine* JSP;

3. O engine JSP traduz a página JSP em uma classe Java, que então é compilada como um Servlet. Esta fase de tradução e compilação ocorre somente quando a página é chamada pela primeira vez, ou toda a vez que esta for alterada. Pode-se perceber uma pequena demora na primeira vez que uma página JSP é executada;

4. O próprio *engine* JSP executa o Servlet, chamando o método `init()` quando o Servlet é carregado pela primeira vez na máquina virtual, para realizar qualquer inicialização global que seja necessária a cada requisição que o Servlet irá atender. Então, as requisiçõs individuais são enviadas para o método `service()`, onde as respostas podem ser processadas;

5. O Servlet cria uma nova *thread* para executar o método `service()` a cada requisição efetuada. A requisição originada pelo Browser é convertida em um objeto Java do tipo HttpServeltRequest, que é passado para o Servlet junto com um objeto HttpServeltResponse, utilizado para enviar a resposta de volta ao Browser.

Este processo também está ilustrado na Figura 6.3 a seguir:

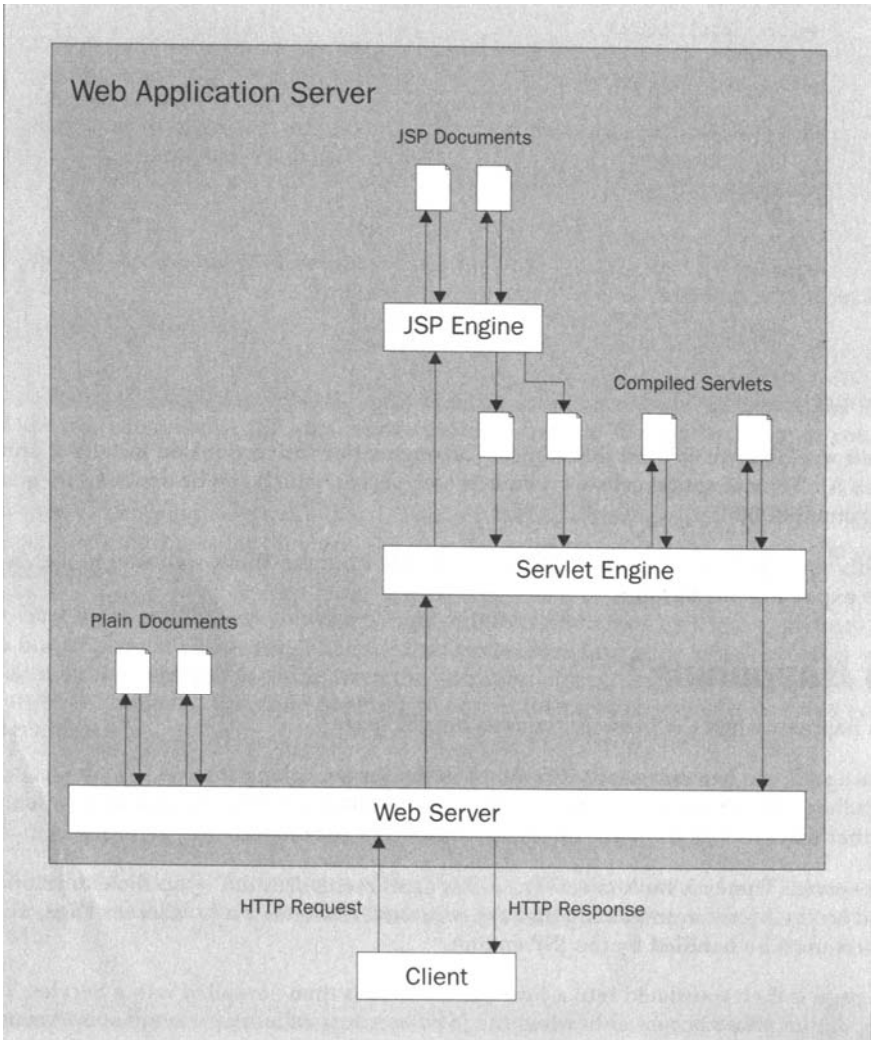

Figura 6.3 – JSP Framework

6.1.2 Elementos e Templates

As JavaServer Pages, em resumo, são arquivos texto que combinam HTML padrão e novas tags de scripting. JSP´s são muito similares com HTML, mas na verdade são compilados em Java Servlets na primeira vez que elas são invocadas. O Servlet resultante é uma combinação do HTML contido no arquivo JSP e do conteúdo dinâmico embutido especificado pelas novas tags.

Uma página JSP é executada por um JSP *engine* ou *container*, que precisa ser instalado em um Web Server ou em um Application Server. Quando um cliente solicita um recurso JSP, este *engine* intercepta a requisição, processa e a modifica, para incorporar a comunicação com o cliente. Depois prepara a resposta e a devolve ao cliente.

Os objetos que encapsulam e abstraem os elementos que compõem as transações do protocolo HTTP são, para as solicitações feitas a partir do navegador `javax.servlet.ServletRequest` e para as respostas devolvidas a ele `javax.servlet.ServletResponse`.

Todos os elementos que compõem uma página JSP podem ser divididos em algumas categorias, como abaixo:

6.1.3 Diretivas

As diretivas JSP servem como mensagens do container JSP para o JSP. Elas são utilizadas para definir valores globais, como declarações de classes, métodos a serem implementados, tipo de conteúdo, etc. Elas não produzem qualquer tipo de saída para o cliente. Todas as diretivas possuem como escopo o arquivo JSP a qual pertencem. Em outras palavras, uma diretiva afeta um arquivo JSP inteiro, e somente este arquivo. As diretivas são caracterizadas pelo caracter @ junto a *tag* e seu formato geral é:

```
<%@ directivename attribute=”value” attribute=”value” %>
```

As três diretivas são:

- include – esta diretiva informa ao *container* para incluir o conteúdo do recurso no JSP corrente *inline* na localização especificada.

- taglib – esta diretiva permite que a página utilize *tags* definidas pelo usuário. Também é utilizada para que o *engine* saiba qual biblioteca de *tags* este deve utilizar para realizar o *parse* nas páginas JSP.

6.1.4 Elementos do scripting

- declarações – é um bloco de código Java em uma página JSP, que é utilizado para definir variáveis de classe e métodos no arquivo de classe gerado. Estas declarações são inicializadas quando a página JSP é inicializada e possuem escopo de classe.

- *scriptlets* – é um bloco de código Java que é executado em tempo de processamento de requisições. Um *scriptlet* é expresso entre os símbolos <% e %> no JSP, podendo gerar saída para o cliente. Múltiplos *scriptlets* são combinados em uma classe compilada na mesma ordem em que aparecem na página JSP e podem modificar objetos internos com resultado de uma chamada de método.

- expressões – é uma notação simplificada para que um *scriptlet* possa produzir uma saída para o cliente. Quando a expressão é avaliada, o resultado é convertido para uma *string* e mostrado.

- ações padrão – são *tags* específicas que afetam o comportamento do *engine* JSP e a resposta enviada ao cliente. A especificação JSP define alguns tipos de ações que são padrão, e estas precisam ser suportadas por todos os *containers*, independentemente de implementação.

6.1.5 Objetos implícitos

JSP tenta simplificar o processo de autoria de páginas e provê alguns objetos implícitos que podem ser acessados por uma página JSP. Estes objetos não precisam ser declarados ou instanciados pelo desenvolvedor JSP, mas são suportados pelo *container* em sua classe de implementação.

Todos os objetos implícitos estão disponíveis somente para *scriptlets* ou expressões e não estão disponíveis em declarações. Isto é lógico, uma vez que, como mencionado anteriormente, as declarações são traduzidas como variáveis de classe. Como os objetos implícitos somente serão invocados pelo método `_jspService()` em tempo de execução, que é o método gerado para o Servlet a ser executado e, em Java, variáveis declaradas dentro de um método não estão disponíveis fora deste.

Os seguintes objetos implícitos estão disponíveis:

- request – este objeto existe no escopo de uma requisição, é uma instância de `javax.servlet.ServletRequest`. Encapsula a requisição vinda do cliente para ser processada pelo JSP, e é passado do container JSP como parâmetro do método `_jspService()`;

- response – este objeto existe no escopo de página, e é uma instância da classe `javax.servlet.ServletResponse`. Encapsula a resposta gerada pelo JSP, para ser enviada de volta para o cliente como resposta de uma requisição. Ele é gerado pelo container e passado para o JSP como um parâmetro do método `_jspService()`, onde é modificado pelo JSP;

- pageContext – este objeto possui escopo de página, sendo uma instância de `javax.servlet.jsp.PageContext`. Encapsula o contexto de página para um determinado JSP.

- session – este objeto possui escopo de sessão, e é uma instância da classe `javax.servlet.http.HttpSession`. Ele representa a sessão criada pela requisição do cliente e é válido somente para requisições HTTP. As sessões são criadas automaticamente, portanto, esta variável existe até mesmo que não haja nenhuma referência de sessão recebida. A única exceção ocorre se for usado o atributo de sessão de diretiva de página, para desativar as sessões. Neste caso, uma tentativa de acesso à variável de sessão irá causar um erro de compilação JSP.

- application – este objeto possui escopo de aplicação e é uma instância da classe `javax.servlet.ServletContext`. ele representa o contexto dentro da qual o JSP é executado.

- out – este objeto possui escopo de página e é uma instância da classe `javax.servlet.jsp.JspWriter`. Representa o fluxo de saída aberto para o retorno ao cliente. O método `PrintWriter` é usado para enviar a saída ao cliente. Entretanto, para tornar o objeto response utilizável, este utiliza uma versão bufferizada de `PrintWriter` chamada `JspWriter`. Note que pode-se ajustar o tamanho do buffer, ou mesmo desativá-lo utilizando o atributo buffer da página.

- config – este objeto possui escopo de página, e é uma instância da classe `javax.servlet.ServletConfig`, que representa a configuração do Servlet.

- page – este objeto possui escopo de página, e é uma instância da classe `java.lang.object`. Refere-se a instância da classe que implementa o JSP. Em outras palavras é o objeto em si, que pode ser acessado utilizando a palavra reservada `this`.

- exception – este objeto possui escopo de página, e é uma instância da classe `java.lang.throwable`. Ele refere-se às exceções em tempo de execução, que resultam em erros de invocação de página, e está disponível somente em uma página de tratamento de erro, que é uma página JSP, previamente, configurada através do atributo de página `isErrorPage=true`.

6.2 Comparação com outras tecnologias

Nesta seção é realizada uma comparação entre a tecnologia JSP e outras técnicas de geração de conteúdo para a Web dinâmico.

Essa comparação leva em consideração principalmente detalhes relevantes ao funcionamento de cada uma das abordagens apresentadas e realiza uma breve análise no fim de cada seção.

6.2.1 CGI

A Common Gateway Interface (CGI) é a mais simples e antiga forma de usar requisições HTTP para controlar a saída de uma aplicação *server-side*. Quando uma requisição é enviada de um Browser para um Web site, um *script* externo (muitas vezes escrito em PERL, C++ ou Python) é executado no servidor, e então a página pode ser gerada.

Nas implementações mais simples, cada requisição envolve a criação de um novo processo onde o script (e seu mecanismo de execução "runtime", se necessário) é carregado, executado e é descarregado uma vez que a resposta foi enviada. CGI´s podem então tomar significativos recursos do sistema. As implementações mais recentes utilizam *threads* para cada requisição e mantém o código em memória para melhorar a performance e a escalabilidade.

Existe mais um problema, pois o CGI somente exige que o servidor passe as informações da requisição para o *script* e esteja preparado para receber a saída que será enviada de volta ao cliente. Não há suporte para a construção de aplicações Web – por exemplo, é necessário implementar todo o suporte de controle de sessões para obter as informações sobre o estado de um cliente entre requisições, para uma sequência de páginas necessárias para completar uma tarefa.

Comparando-se CGI com JSP temos:

- JSP pode manter o estado no servidor entre requisições (usando uma sessão Servlet);

- Lança uma nova thread para cada request;

- Não é necessário ser carregado todo o tempo, uma vez inicializado;

- Executa em uma máquina virtual Java (JVM) já carregada que é uma extensão do Web Server.

6.2.2 Web Server API´s

Muitos Web Servers possuem sua própria API´s de desenvolvimento para a criação de conteúdo dinâmico. Por exemplo, tem-se os filtros ISAPI para o IIS e NSAPI para os Web Servers da Netscape. Estas API´s funcionam bem, mas abrem mão da portabilidade. Quando utilizada uma API do servidor, o código é escrito especificamente para aquele Web Server em uma plataforma proprietária. A falta de escolha de um Web Server é fonte de preocupação para muitos desenvolvedores Web.

6.2.3 Active Server Pages

A principal tecnologia que compete com JSP´s é Active Server Pages (ASP), da Microsoft. As tecnologias são muito similares no modo como suportam a criação de páginas Web dinâmicas, utilizando templates, código script e componentes para a lógica de negócio. Ambos também implementam interfaces para Enterprise Application Frameworks, J2EE e Microsoft DNA, repectivamente. A tabela abaixo apresenta uma comparação:

Tabela 6.1 – JSP versus ASP

	JSP	**ASP**
Plataformas	A maioria das plataformas Web	Somente Microsoft
Linguagem Básica	Java	Jscript ou VBScript
Componentes	JSP, JavaBeans e EJB´s	COM/DCOM
Interpretação de código	Somente uma vez	A cada instância

ASP é comumente usado em Web Servers Microsoft IIS ou PWS. Existem duas empresas que fornecem soluções que permitem a portabilidade de ASP para ser utilizado em outras plataformas. São elas Chili!Soft (www.chilisoft.com) e Halcyonsoft (www.halcyonsoft.com). O principal problema destas soluções está na portabilidade dos componentes COM para as novas plataformas. Geralmente eles convertem estes componentes para JavaBeans ou utilizam uma ponte JavaBeans-ActiveX.

Comparando-se ASP com JSP temos:

- JSP´s são interpretados e compilados somente uma vez para Java byte-code, e este processo repete-se apenas quando o arquivo é modificado;

- JSP´s executam em praticamente todos os principais Web Servers;

- JSP´s mantém uma melhor separação entre o código da página e os dados do *template*, através de JavaBeans, Enterprise JavaBeans e tag libraries.

6.2.4 Client-Side Scripting

Usando linguagens scripts do lado cliente como JavaScript, JScript e VBScript, a tarefa de interpretar e executar estes scripts fica a cargo do próprio Browser, o que pode ocassionar problemas devido a interpretações diferentes em Browser de diferentes fabricantes ou, até mesmo, diferenças entre versões do mesmo Browser. Além disso, o usuário pode decidir desabilitar a execução de *scripts* locais.

Como JSP executa do lado servidor, o Brwoser passa a não influenciar a execução, a menos, é claro, que o código JSP gere código HTML que contenha código script para ser interpretado do lado cliente.

6.2.5 Servlets

A especificação JSP é construída como uma implementação de mais alto nível da API Java Servlet. Servlets possuem a habilidade de gerar conteúdo dinâmico para *sites* Web utilizando a linguagem Java, e são suportados pelas principais plataformas Web Servers.

Por que então existe JSP se Java já possui a API para atender a requisições HTTP?

Apesar de que tudo que pode ser feito em JSP possa também ser feito através de um Servlet, JSP possui uma forma mais clara de separação entre a camada de apresentação e a lógica, além de ser mais fácil de escrever. Servlets e JSP trabalham muito bem juntos, integrando naturalmente suas API´s nativas.

6.3 O futuro desta plataforma

JSP e Sevlets estão ganhando uma rápida aceitação como tecnologia para suportar conteúdo dinâmico na Internet. Com acesso total a plataforma Java, executando em um servidor de modo seguro, as possibilidades de aplicação não possuem limites. Enquanto os JSP´s são utilizados em conjunto com a tecnologia de Enterprise JavaBeans, e-commerce e integrados aos recursos de banco de dados, que podem ser aprimorados, indo de encontro ao que uma corporação precisa para suas aplicações Web, promovendo transações seguras em uma plataforma aberta.

A tecnologia J2EE como um todo é facilmente assimilável, tanto para o desenvolvimento, como para a disponibilização e uso, se comparada às demais tecnologias similares, como CGI e ASP. Muitas ferramentas estão sendo construídas para permitir um desenvolvimento rápido de aplicações Web e também facilitar a migração das tecnologias server-side presentes atualmente para JSP e Servlets.

A tecnologia JSP continua em constante evolução, incluindo a todo o momento novos recursos. Já está a caminho a especificação Java Servlet 2.3 e JSP 1.2, cujas prinicipais novidades são:

- Suporte aprimorado para a localização de aplicações;
- Suporte adequado para a inclusão de páginas JSP, sem precisar forçar a limpeza de buffers;
- Suporte a aplicações orientadas a eventos;
- Um melhor mecanismo e ferramentas para depuração;
- Suporte melhorado para XML;
- Suporte a bibliotecas de tags padronizadas;
- Suporte à autoria JSP;
- Melhor suporte para a composição de componentes;
- Capacidade para atender requisições WebDAV e WAP.

Muitas empresas, como a IBM, estão distribuindo agressivamente a tecnologia JSP acompanhada com uma vasta gama de produtos, como o WebSphere Application Server e o WebSphere Studio. A empresa BEA Systems vem fazendo uma utilização pesada desta tecnologia em seu Portal Server, e possui suporte a JSP nos produtos e ferramentas desenvolvidos pelas suas subsidiárias.

O e-commerce também será muito beneficiado com as novas versões de JSP, com funcionalidades XML e escalabilidade. Como os sites de e-commerce estão neste momento muito visados por ataques contra a sua segurança, aparentemente o gerenciamento ativo de rede associado com implementações mais escaláveis precisam ser continuamente assegurar robustez as platafomas corporativas.

Por manter uma clara separação entre o conteúdo e a codificação, JSP está sendo uma ótima resposta aos anseios das empresas. Com acesso completo a API de Java, e a habilidade de operar com segurança em toadas as plataformas servidoras que dominam o mercado, JSP deve continuar a ser uma tecnologia muito atrativa para o desenvolvimento corporativo.

7 Modelagem do Projeto

Com base nas informações apresentadas sobre a estrutura de um repositório e da análise realizada sobre os serviços propostos, a modelagem do sistema é apresentada neste capítulo, utilizando a linguagem UML descrita em [RAT97a] [RAT97b].

São apresentados os diagramas de caso de uso identificados, bem com o detalhamento de cada um através de um diagrama de troca de mensagens associado.

7.1 Casos de Uso

O conceito de caso de uso (*use case*) foi introduzido em UML por Jacobson [JAC92], que o define como sendo uma interação típica entre um usuário e o sistema de software. Esta interação do usuário nem coincide com a realização do objetivo do usuário [FOW97].

Um diagrama de casos de uso representa os relacionamentos entre os atores e casos de uso dentro do sistema. Atores são entidades externas ao sistema, mas que interagem com ele, podendo representar pessoas, dispositivos de hardware ou outros sistemas de software. Já os casos de uso são funcionalidades gerais do sistema que ocorrem quando um estímulo proveniente de um ator os ativam.

O escopo de projeto contempla três atores, que são apresentados a seguir com os seus respectivos diagramas de casos de uso:

Administrador

Realiza todas as ações relativas a administração do repositório, bem como define a política de acesso e serviços relacionados com o desenvolvimento e a disponibilização de componentes.

Para tal, após validar o usuário e sua respectiva senha (*Login*), suas atividades dividem-se em:

- Gerenciar as contas e privilégios dos usuários do repositório de componentes (*User Manager*);

- Gerenciar as etapas, ordem e obrigatoriedade na execução dos serviços do repositório de componentes, definindo a política de registro dos componentes (*Deploy Manager*);

- Gerenciar e configurar cada serviço (*Service Manager*);

- Desenvolver novos serviços (*Service Develop*);
- Instalar novos serviços no repositório (*Service Deploy*).

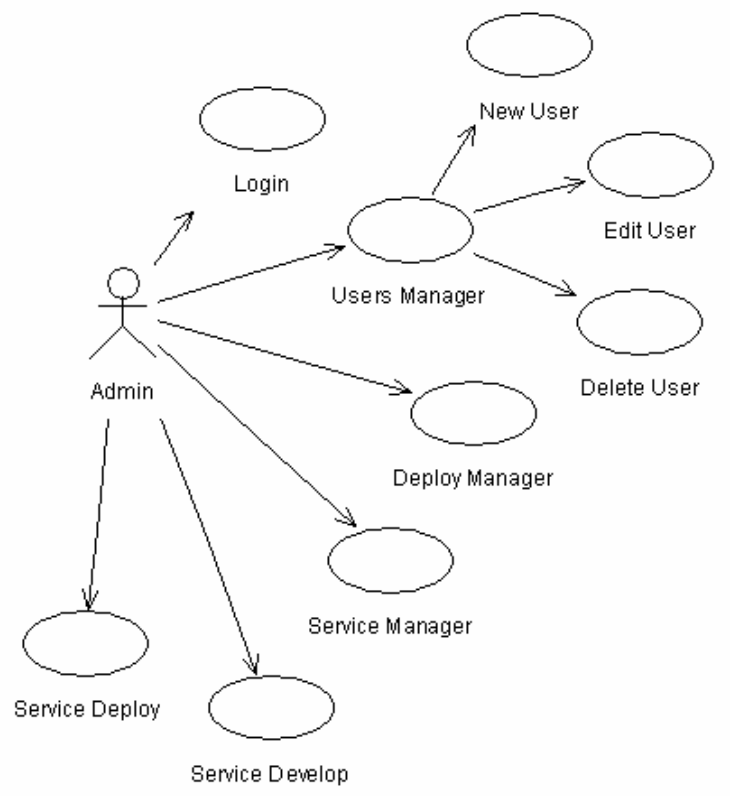

Figura 7.1 – Diagrama de casos se uso (*Admin*).

Desenvolvedor

Suas ações são relativas à população do repositório, desenvolvendo e evoluindo os componentes armazenados.

As atividades do desenvolvedor dividem-se em, após o *Login*, desenvolver novos componentes para popular o repositório (*Class Develop*) e disponibilizar estes componentes aos demais desenvolvedores, mantendo sua integridade conforme as regras definidas pelo administrador(*Class Deploy*).

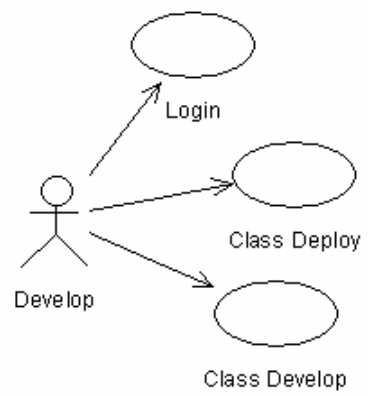

Figura 7.2 – Diagrama de casos de uso (*Develop*).

Cliente

Efetivamente utiliza os componentes e serviços do repositório, desenvolvidos e disponibilizados pelo administrador e seus desenvolvedores para construir as aplicações.

Basicamente, suas atividades dividem-se em utilizar os componentes existentes (*Class User*) e os serviços disponíveis (*Service User*) para realizar a identificação de qual componente vai melhor se adequar a uma determinada implementação.

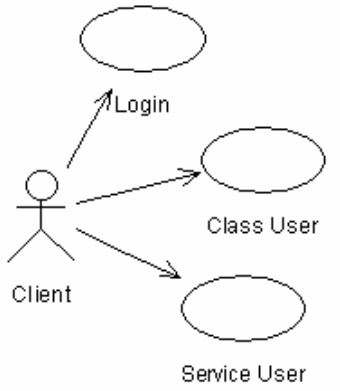

Figura 7.3 – Diagrama de casos de uso (*Client*).

8 Implementação do Protótipo

Para a validação do modelo apresentado, foi proposta a implementação de um protótipo para o repositório de componentes de software.

O projeto do sistema divide a aplicação em três formas básicas seguindo o modelo MVC que será descrito no presente capítulo. Deve-se ter em mente que esta divisão é conceitual e não necessariamente mapeada diretamente sobre detalhes da implementação como esquemas de banco de dados ou hierarquia de classes. Elas constituem um vocabulário para tratar o domínio do problema e possíveis estratégias de implementação e, portanto, não são estruturas implementadas por si só.

Assim, neste capítulo, são descritas as decisões e a seleção de tecnologias utilizadas no projeto para a implementação do protótipo.

8.1 Descrição

Inicialmente, depois de realizada uma avaliação do material apresentado no capítulo anterior sobre a modelagem do projeto, dada a necessidade de implementar-se um ambiente cooperado e, também, as exigências em relação a possível descentralização desejada no processamento dos diversos serviços, optou-se por utilizar o padrão de projeto *Model-View-Controler* (MVC).

Embora MVC possua suas raízes em Smalltalk, este modelo é realmente mais um modo de separar o projeto de software interativo do que um algoritmo ou uma linguagem de construção. O objetivo de MVC é separar a funcionalidade de uma aplicação em três categorias lógicas: o modelo (*Model*) que contém o funcionamento interno do programa e os algoritmos, a apresentação (*View*), que apresenta ao usuário o estado atual do modelo, e o controle (*Controller*), o qual define como o usuário altera o estado ou determina as entradas.

Mesmo para os dias de hoje, MVC é ainda um conceito que tem muita influência. De acordo com Erich Gamma em [GAM95], "*observando os padrões de projeto dentro de MVC deve ajuda-lo a perceber o real significado do termo padrão*". Da mesma forma Rosenberg and Scott em [ROS99] reconhecem o quanto *Model-View-Controller* relaciona-se com a abordagem *Boundary-Entity-Control* para a análise e projeto.

A seguir, tem-se o detalhamento da arquitetura proposta nesta metodologia de desenvolvimento, destacando, para cada uma das três camadas, suas necessidades e responsabilidades.

Model

Representa a estrutura de dados na aplicação, bem como, as operações específicas da aplicação sobre estes dados, ou seja, a funcionalidade do sistema.

É a camada de mais baixo nível, responsável por realizar as tarefas pertinentes ao processamento e acessos externos. Assim, os algoritmos definidos com base na especificação são implementados de forma concreta nesta camada.

View

Constitui-se da interface com o usuário do sistema e é a camada de mais alto nível, responsável pela apresentação dos dados dentro do contexto de uma função da aplicação como elementos de interface e das metáforas utilizadas para facilitar a utilização do sistema pelo usuário (*User Experience*).

Controler

Realiza a interação entre a interface do usuário (*View*), com as tarefas que devem ser executadas (*Model*), respondendo aos eventos gerados pelo usuário através de sua interface e traduzindo as ações do usuário (movimentos do mouse, acionamento de teclas, reconhecimento de voz, etc.), em chamadas de funções da aplicação no *Model*, selecionando a *View* mais apropriada, gerando eventos para a interface, baseado nas preferências do usuário, conforme as mudanças de estado detectadas no *Model*.

Essencialmente, um *Model* abstrai o estado da aplicação e sua funcionalidade, uma *View* abstrai a apresentação da aplicação e um *Controller* abstrai o comportamento da aplicação. O padrão de projeto MVC desta forma atinge o objetivo do projeto com uma decomposição racional do sistema.

É importante entender que *Model*, *View* e *Controller* não são geralmente representados por classes individuais, uma vez que estes são subdivisões conceituais da aplicação.

Existem muitos benefícios obtidos com a utilização de MVC em um projeto:

- A separação entre *Model* e *View*, isto é, separar a representação dos dados da sua apresentação, torna fácil adicionar múltiplas apresentações para os mesmos dados, facilita a adição de novos tipos de representação dos dados com o desenvolvimento da tecnologia. Os componentes *Model* e *View* podem ser modificados independentemente, exceto em sua interface, aumentando a manutenibilidade, extensibilidade e testabilidade.

- Separando *Controller* de *View*, isto é, o comportamento da aplicação da apresentação, permite a seleção em tempo de execução de *Views* baseada em fluxo de trabalho (*Workflow*), preferências do usuário, ou no estado atual de *Model*.

- Separando *Controller* de *Model*, isto é, comportamento da aplicação da representação dos seus dados, permite o mapeamento configurável das ações do usuário em *Controller* para as funções da aplicação em *Model*.

Dentre as vantagens obtidas com a adoção deste padrão de projeto na implementação proposta, a separação bem definida entre interface e funcionalidade de um sistema, neste projeto isto é muito desejável, pois se aspira a flexibilidade de desenvolver novos serviços, ou ainda, evoluir os já existentes, sem a necessidade de tratar questões relativas à interface com o usuário, por exemplo.

8.2 Implementação

A arquitetura MVC fornece flexibilidade, reusabilidade, testabilidade, extensibilidade e regras de projeto claras para os componentes da aplicação. O projeto em múltiplas camadas (*multitiered*) permite uma escolha flexível das tecnologias para a implementação, bem como escalabilidade e capacidade de evolução. O projeto modular decompõe as funções da aplicação em subsistemas intuitivos e com baixo acoplamento, que são individualmente analisáveis, testáveis e compreensíveis.

Separar uma aplicação em tantas partes conceituais torna complexo o projeto. A arquitetura do software é, em grande parte, a arte de atingir os requisitos do projeto na presença de várias exigências concorrentes: desenvolvimento vs. integração, generalização vs. especialização, estabilidade vs. maleabilidade, e assim por diante.

A divisão da aplicação em múltiplas partes conceituais abstratas permite que o arquiteto possa escolher a solução mais apropriada em um contexto particular. O desafio é projetar um sistema que capture a complexidade essencial do domínio do problema, enquanto minimiza a complexidade não-essencial dos detalhes da implementação. Ou, citando Albert Einstein, “Everything should be as simple as possible, and no simpler.".

Para implementar o protótipo foram estudadas várias das API’s de Java, tendo sido selecionadas para a presente implementação, as seguintes, divididas conforme o padrão de projeto adotado.

Model

Garantindo a portabilidade do sistema, principalmente de sua gerência e de seus serviços, estes algoritmos são implementados em Java, preservando assim os esforços de desenvolvimento e permitindo que o núcleo do projeto tenha sua implementação independente de plataforma.

Dentre, as tarefas realizadas por *Model* está o mecanismo de persistência adotado. Neste projeto selecionou-se como a API de persistência *Java Database Connection* (JDBC), que apesar de deixar o mapeamento objeto-relacional a cargo do programador é a API mais sólida, estando atualmente em sua segunda versão e possuindo uma nova implementação em processo de certificação.

Foi descartado o uso de *Java Data Objects* (JDO), pois apesar desta API possuir recursos avançados quanto ao gerenciamento da persistência, ainda encontra-se em processo de certificação, sendo que a documentação disponível resume-se ao *draft* versão 0.92 da especificação, e sua implementação de referência ainda possuir muitas pendências, o que torna o seu uso excessivamente trabalhoso em relação aos benefícios oferecidos.

View

Para permitir o acesso remoto, e por conseqüência a colaboração entre os usuários do repositório, selecionou-se a tecnologia de *Java Server Pages* (JSP), por ser mais flexível e de fácil implementação, tendo sido descartado as opções de utilizar-se *Java Sockets* ou *Java Remote Method Invocation* (RMI), devido a complexidade e requisitos exigidos por tais tecnologias.

Além disso, com JSP não há necessidade de se instalar software cliente para o acesso remoto ao sistema, utiliza-se um navegador Web, com suporte a HTML e JavaScript.

Controller

Como decorrência da escolha de JSP como a API para a implementação de *View*, temos *Servlets* como sendo a API mais indicada para a implementação de *Controler*, pois ambos utilizam a mesma interface, possuindo objetos *request* e *response*, já que uma página JSP é compilada para um *Servlet*.

Servlets também podem acessar classes Java de forma mais elegante e eficiente que uma página JSP, podendo assim facilmente ter acesso às classes de *Model*, especialmente com classes que trabalham com *Threads* e *Sockets*, que não podem ser instanciados diretamente num container JSP, assim podemos resumidamente definir cada camada:

As *Views* no projeto proposto consistem de páginas JSP apresentadas em um navegador.

O *Model* é representado como um conjunto de classes Java.

O *Controller* consiste de um conjunto de Servlets que remetem as requisições do navegador para outros objetos de controle, e para as suas respectivas classes de suporte.

8.3 Diagramas de Classes

O diagrama de classes do sistemas foi dividido em três sub-diagramas, de modo a facilitar a apresentação e o seu entendimento. Na Figura 8.1 o conjunto de classes utilizadas para gerenciar os usuários do sistema, autenticar e realizar o seu controle de sessão.

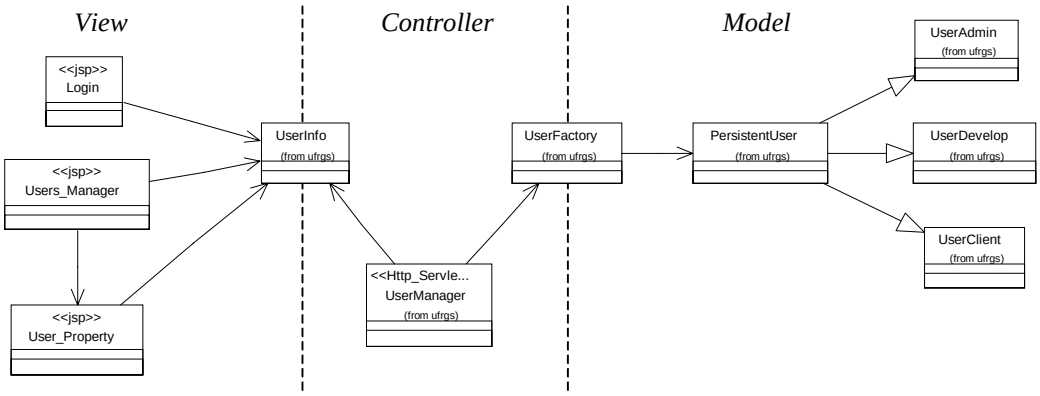

Figura 8.1 – Diagrama de classes dos usuários.

A Figura 8.2 apresenta o diagrama de classes dos serviços, o qual ilustra o relacionamento e as classes necessárias para o gerenciamento, e execução da publicação de componentes, configuração e execução dos serviços disponíveis no repositório, e definição das classes abstratas que devem ser estendidas durante o desenvolvimento de um novo serviço.

Todos os serviços do protótipo proposto devem, obrigatoriamente, estender a classe *AbstractService* para realizar as suas tarefas na camada *Model*, pois somente assim a classe *ServiceFactory* poderá realizar o registro, a carga e a execução do serviço desenvolvido e publicado no repositório de componentes.

Por outro lado, todas as interfaces públicas dos serviços devem estender a classe *Service.jsp* na camada *View*, pois somente assim a classe *ServiceManager* poderá realizar a apresentação da interface pública que um serviço venha a registrar durante a sua publicação.

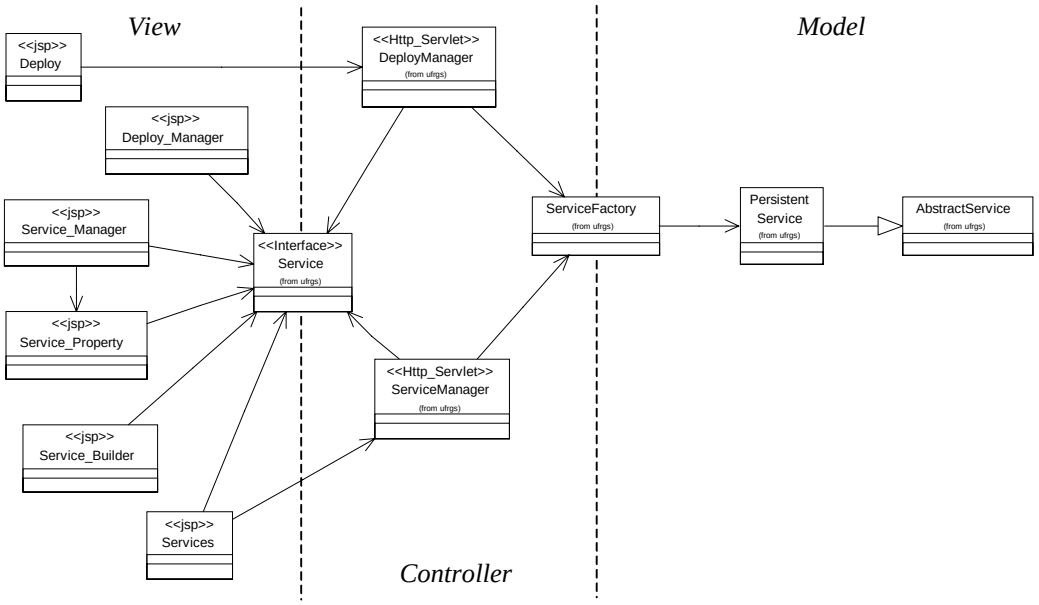

Figura 8.2 – Diagrama de classes dos serviços.

No diagrama apresentado na Figura 8.3 tem-se as classes envolvidas na descrição e construção de novos componentes de software que serão armazenados no protótipo de repositório proposto. Notar que a classe *ClassLoader* é de fato a própria classe que realiza a carga de classes na máquina virtual Java, pertencente ao pacote *java.lang* padrão.

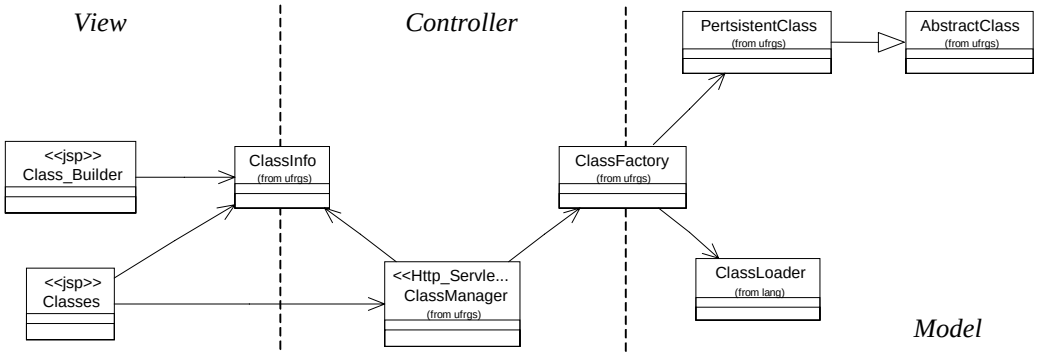

Figura 8.3 – Diagrama de classes dos componentes.

8.4 Diagramas de Troca de Mensagens

Os diagramas de troca de mensagens ou de seqüência apresentam as interações realizadas no sistema, enfatizando aspectos temporais, representando os objetos participando das interações através das trocas de mensagens entre estes, ordenadas na seqüência de tempo em que ocorrem.

Para cada caso de uso tem-se o respectivo detalhamento, com seu respectivo diagrama de troca de mensagens.

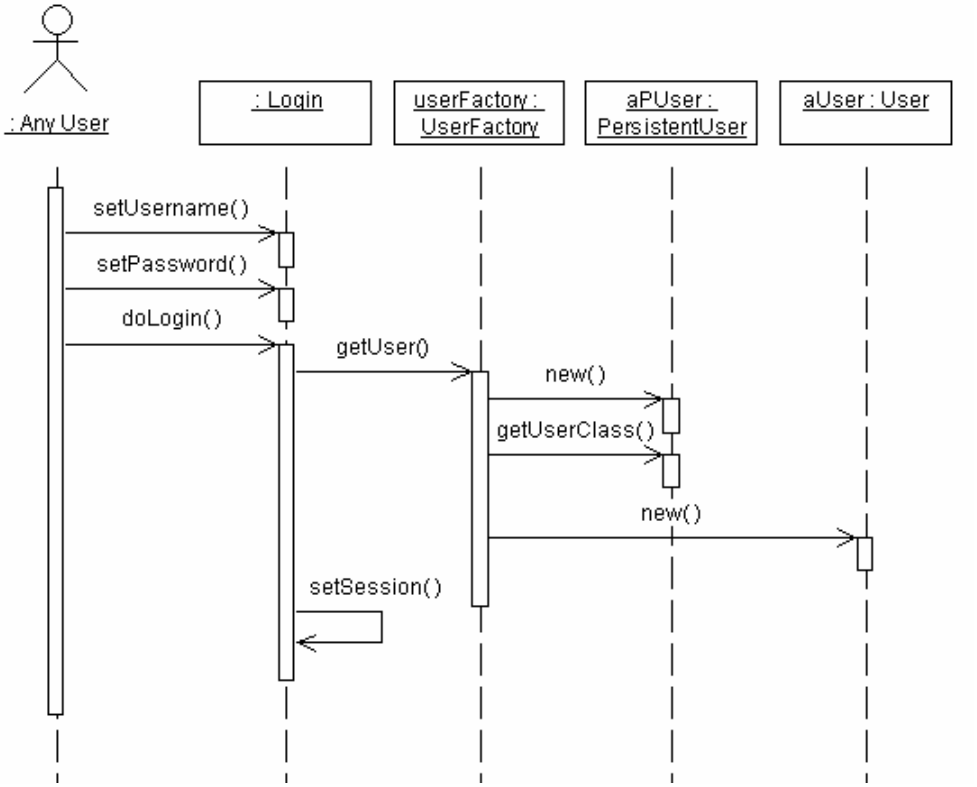

Figura 8.4 – Diagrama de troca de mensagens (*Login*).

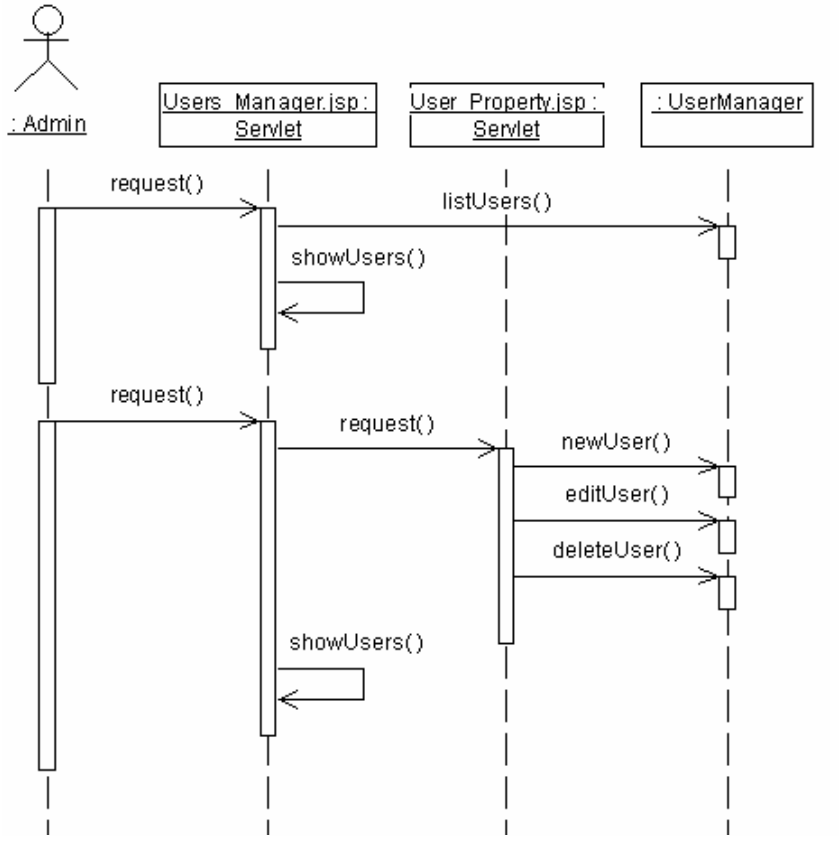

Figura 8.5 – Diagrama de troca de mensagens (*Users Manager*).

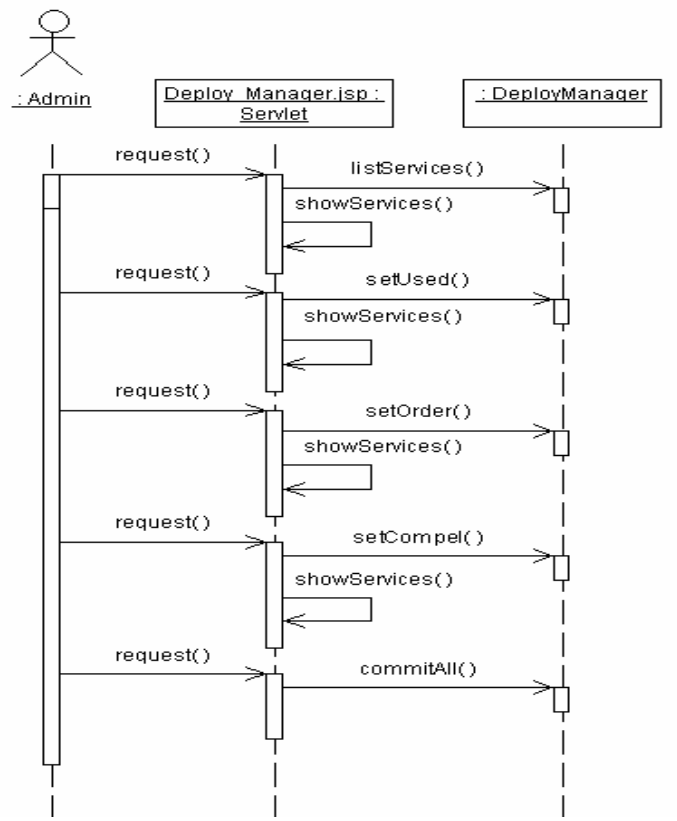

Figura 8.6 – Diagrama de troca de mensagens (*Deploy Manager*).

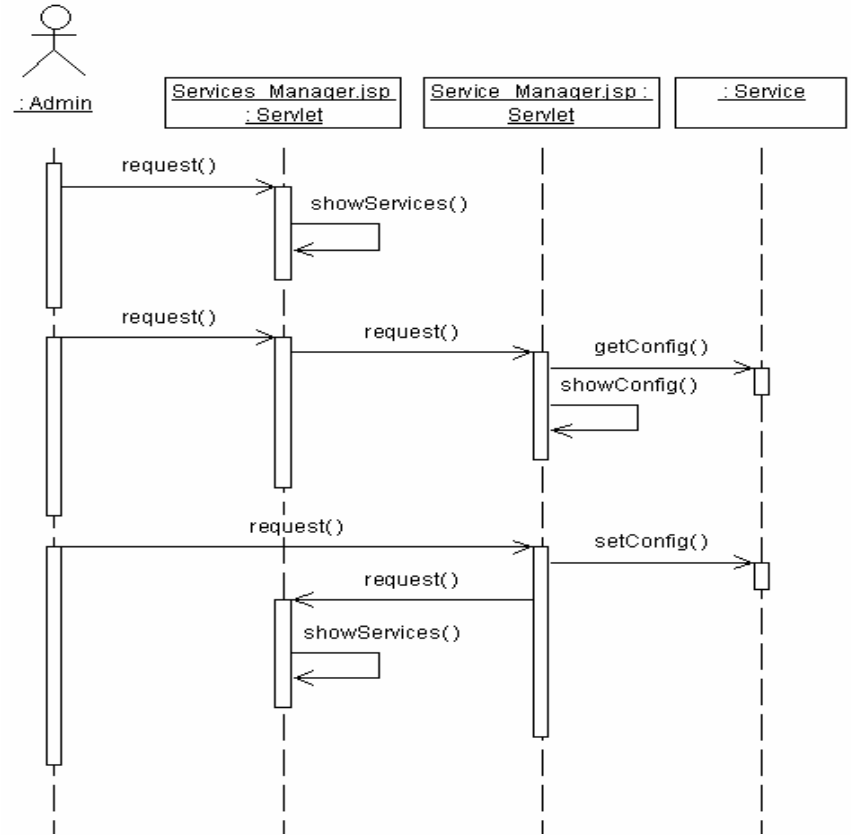

Figura 8.7 – Diagrama de troca de mensagens (*Service Manager*).

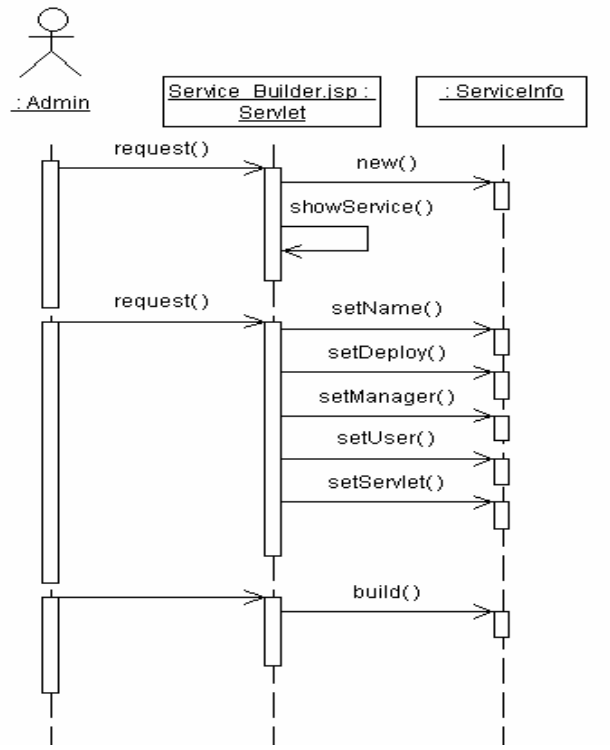

Figura 8.8 – Diagrama de troca de mensagens (*Service Develop*).

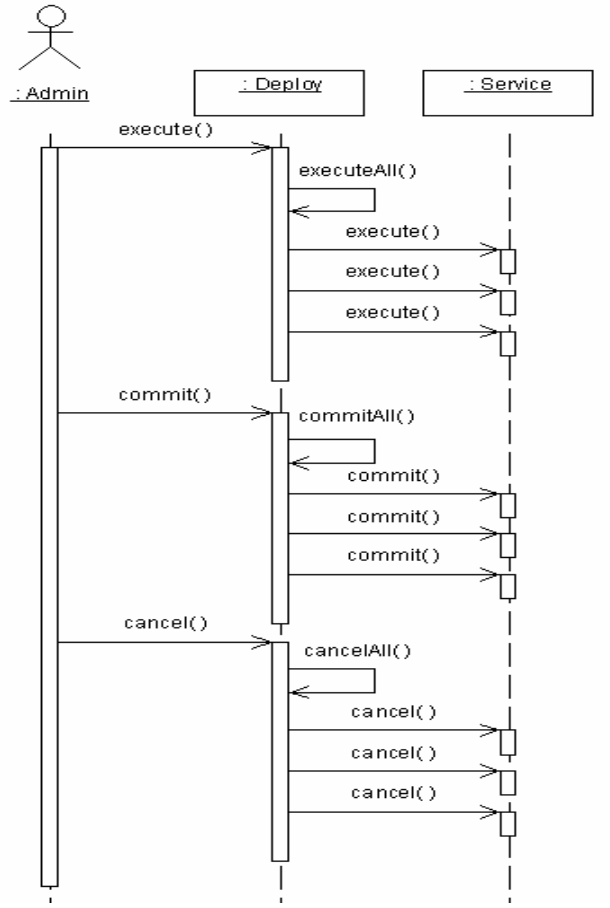

Figura 8.9 – Diagrama de troca de mensagens (*Service Deploy*).

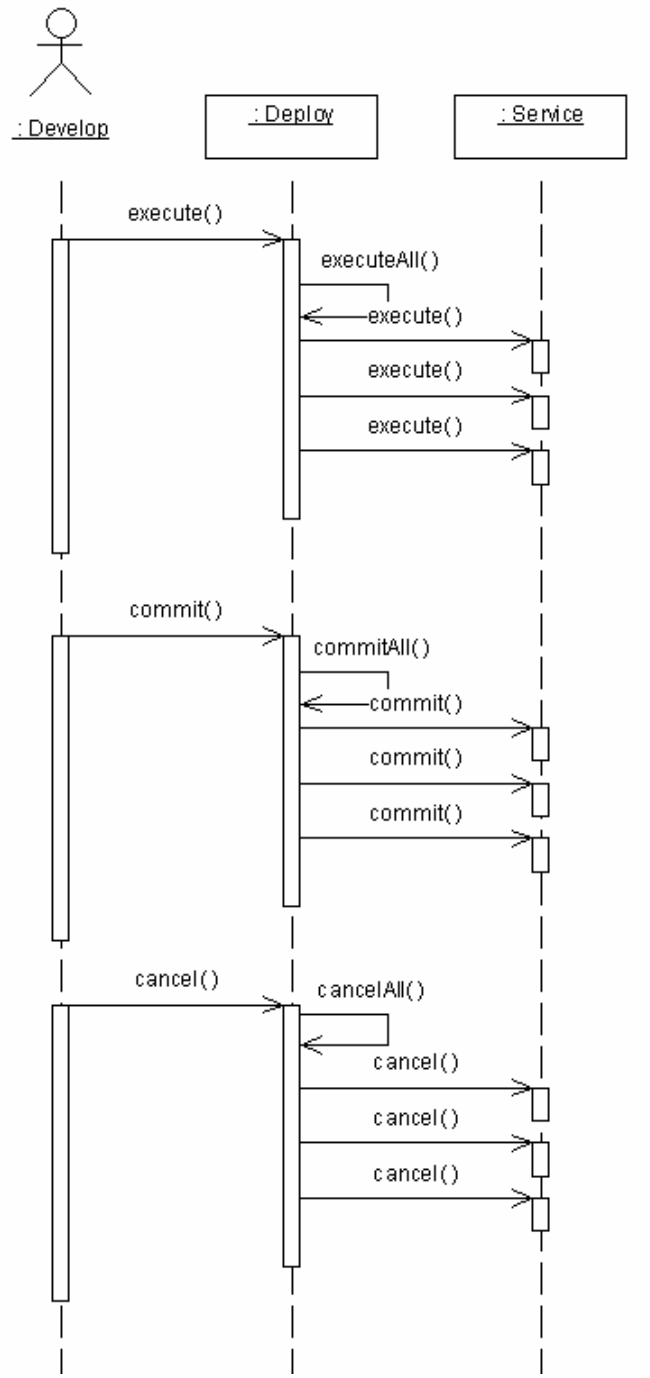

Figura 8.10 – Diagrama de troca de mensagens (*Class Deploy*).

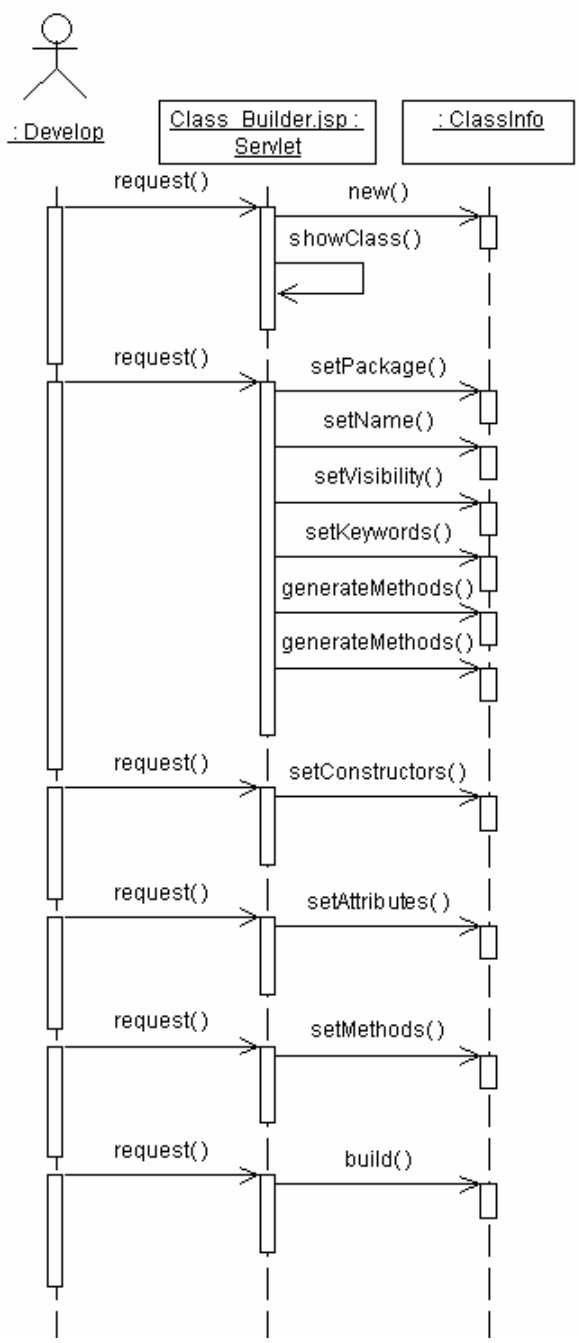

Figura 8.11 – Diagrama de troca de mensagens (*Class Develop*).

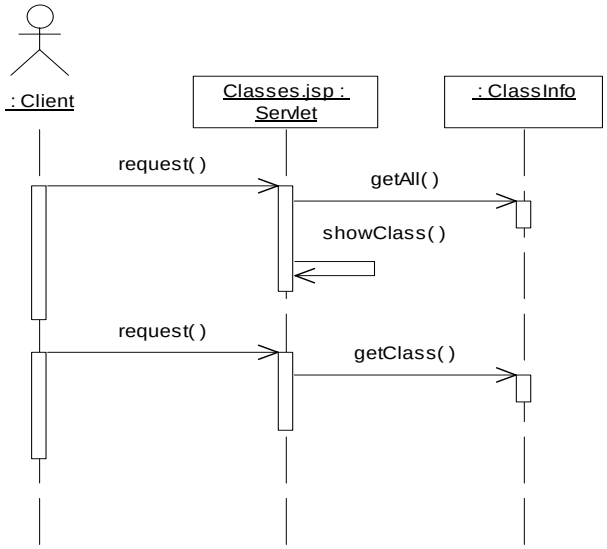

Figura 8.12 – Diagrama de troca de mensagens (*Class User*).

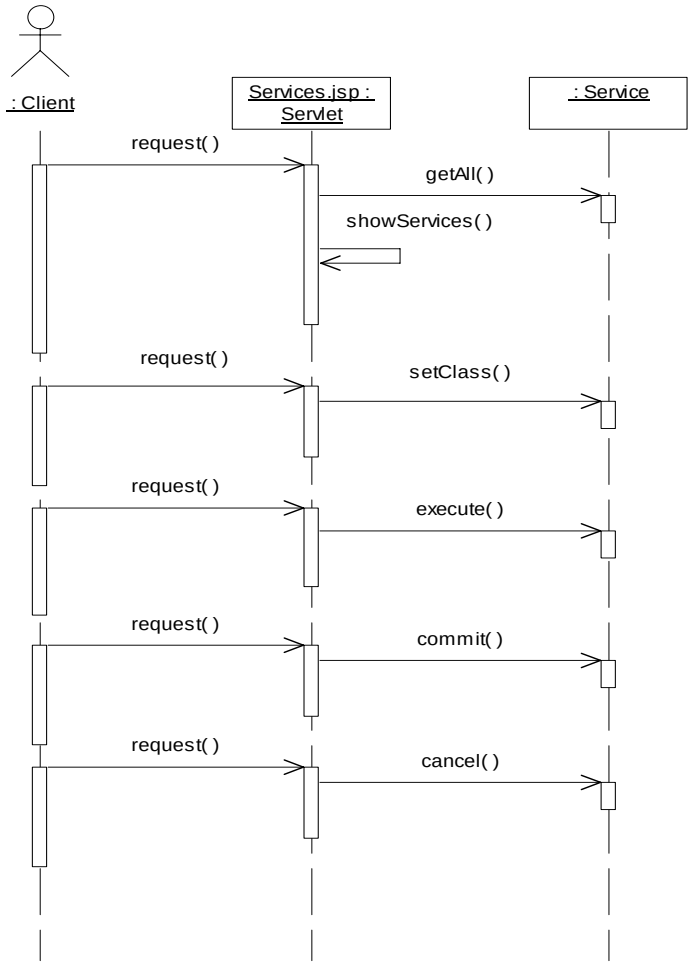

Figura 8.13 – Diagrama de troca de mensagens (*Service User*)

8.5 Modelo de Comunicação

A separação bem definida em camadas imposta pelo padrão de projeto adotado e as diversas tecnologias selecionadas, implicam na adoção de modelos de comunicação entre os elementos que compõem a implementação, descritos a seguir:

8.5.1 Browser → JSP

Comunicação baseada no protocolo HTTP. Para obter-se maiores informações sobre este protocolo acesse [URL14]. Como o protocolo HTTP não é orientado à conexão, utiliza-se o mecanismo de Cookies para o controle de sessão, mecanismo este definido pela Netscape e descrito em [URL15].

O protocolo HTTP trabalha com o conceito de requisições e respostas a estas, ou seja, para cada requisição realizada pelo navegador é esperada a respectiva resposta gerada pelo servidor Web.

Esta metáfora é encapsulada pela tecnologia de Java Server Pages, e é representada por objetos implícitos, a saber, *request* e *response*, já descritos em detalhe no capítulo 6, que apresenta esta tecnologia.

8.5.2 JSP → Servlet

Cada página JSP, depois de interpretada e compilada pelo ambiente de execução JSP, gera um *Servlet*, que traduz a representação dos objetos implícitos *resquest* e *response* para as interfaces que encapsulam e mapeiam o protocolo HTTP, a saber, `javax.servlet.ServletRequest` e `javax.servlet.ServletResponse`, respectivamente.

Como ambos, páginas JSP e *Servlets* utilizam as mesmas interfaces de comunicação, a comunicação torna-se transparente.

8.5.3 Servlet → Java

Como cada *Servlet* é na verdade uma classe Java compilada que realiza a extensão da classe `javax.servlet.HttpServlet`, e sobrescreve alguns de seus métodos, em geral, os métodos *init*, *doGet* e/ou *doPost*. O acesso a outras classes Java pode ser realizado de forma direta, importando o seu respectivo pacote (*import*) e realizando a instanciação e invocação de métodos.

Nada impede, também, que novos métodos possam ser adicionados, agregando novas funcionalidades à classe *Servlet*, o que torna a arquitetura proposta altamente flexível e dentro das especificações J2EE.

8.6 Modelo de Serviço

O modelo de serviço proposto para esta implementação do protótipo do repositório de componentes procura ser o mais simples possível, porém sem perder as características desejadas de escalabilidade e extensibilidade, bem como, as definições do padrão de projeto adotado.

Um serviço para o repositório de componentes é basicamente um arquivo no formato Java Archive (JAR). Detalhes sobre este formato de arquivo estão disponíveis em [URL16]. O arquivo JAR deve possuir o nome do novo serviço, como por exemplo **MyService.jar** e os seguintes arquivos, substituindo-se a palavra *AbstractService* por *MyService*.

AbstractService.jsp – define uma interface padrão para o usuário do repositório (*View*), a qual deve ser estendida, caso o serviço possua alguma funcionalidade que permita a inteiração com este;

AbstractServiceDeploy.jsp – define uma interface padrão para o usuário do repositório que deve ser utilizada durante a etapa de *Deployment* de uma classe (*View*), que deve ser estendida, caso o serviço possua alguma funcionalidade que necessite de integração com este;

AbstractServiceServlet.class – define um *Servlet* padrão, utilizando o padrão de projeto *Class Adapter*, sendo que `AbstractServiceServlet` deve ser estendida para a definição das operações que podem ser realizadas por este serviço (*Controller*);

AbstractService.class – define uma classe base, a qual deve ser utilizada como raiz na hierarquia de serviços (*Model*). A partir desta classe o desenvolvedor deve estender e adicionar as funcionalidades relativas ao novo serviço e realizar a integração com ferramentas de terceiros, caso necessário.

Como a manipulação de arquivos no formato JAR é suportada em Java pelo pacote `java.util.jar`, todas as rotinas necessárias já estão disponíveis, tornando assim a utilização deste padrão a escolha natural para a manipulação de conjuntos de arquivos, com as seguintes vantagens:

- **Segurança** – pode-se assinar digitalmente o conteúdo de um arquivo JAR. O usuário que identificar a assinatura pode opcionalmente conceder ao seu software privilégios de segurança que normalmente este não possui;

- **Menor tempo de download/upload** – Se um Applet é empacotado em um arquivo JAR, os arquivos de classes e os seus recursos associados podem ser transferidos em uma única transação HTTP, sem precisar abrir uma nova conexão para cada arquivo;

- **Compressão** – O formato JAR permite a compressão dos arquivos para um armazenamento eficiente;

- **Portabilidade** – O mecanismo para manipular arquivos JAR é uma parte padrão da plataforma Java.

A partir da versão Java 1.2, temos ainda:

- **Empacotamento para extensões** – O *framework* de extensões fornece um método pelo qual pode-se adicionar funcionalidades no núcleo da plataforma Java, este *framework* é baseado nas definições do formato de arquivo JAR;

- **Fechamento de pacotes** – Pacotes armazenados em arquivos JAR podem opcionalmente ser fechados ou selados, reforçando a consistência da versão. Selar um pacote em um arquivo JAR significa que todas as classes definidas naquele pacote devem ser encontradas no mesmo arquivo JAR;

- **Versionamento de pacotes** – Um arquivo JAR pode armazenar dados sobre os arquivos nele contidos, tais como informações de versão e fornecedor.

Conforme a especificação, os arquivos no formato JAR incluem também o caminho (*path*) onde o seu conteúdo deve ser disponibilizado. No próximo subitem é apresentada a estrutura de diretórios proposta para o protótipo, na qual será detalhada a localização de cada um dos arquivos citados.

8.7 Apresentação do Protótipo

A descrição desta implementação é realizada a partir da sua estrutura física, ou seja, localização de arquivos e estrutura de diretórios, e contém também todos elementos que compõem a interface do protótipo, sua utilização prática e como armazenar classes no repositório de componentes de software.

8.7.1 Estrutura de diretórios

Tendo como ponto de partida o diretório raiz (*root*), definido como sendo o diretório da aplicação Web, no caso do JRun rodando em um servidor Windows NT e servidor Web IIS, tem-se como diretório raíz: `./InetPub/Scripts` e a estrutura de diretórios adotada tal como descrita a seguir:

Estrutura de diretórios	*Descrição*
`./Repository`	Scripts de login
`./WEB-INF`	
`./classes`	Biblioteca de classes
`./Deploy`	
`./Class`	Assistente de instalação de classes
`./Service`	Assistente de instalação de serviços
`./Develop`	
`./Class`	Assistente de desenvolvimento de classes
`./Service`	Assistente de desenvolvimento de serviços
`./Images`	Biblioteca de imagens da aplicação
`./Manager`	
`./Deploy`	Gerenciamento da instalação de classes
`./Services`	Gerenciamento dos serviços
`./Users`	Gerenciamento dos usuários
`./Services`	Interfaces públicas dos serviços

8.7.2 Interface

A interface proposta para a implementação deste protótipo deve levar em consideração três fatores principais: usabilidade, navegabilidade e extensibilidade.

A usabilidade é definida de modo a estruturar e categorizar as informações existentes no repositório, servindo de base para a modelagem da navegação e tendo como principal objetivo a facilidade com a qual o usuário pode interagir com o sistema proposto.

A navegabilidade é proposta com base nos diagramas de casos de uso, criando premissas para o projeto da aplicação e levando-se em conta a facilidade com a qual o usuário encontra as ações ou informações que necessita.

A extensibilidade é caracterizada pela capacidade de acrescentar-se novas funcionalidades ao sistema do modo mais simples possível, porém mantendo a uniformidade com que as informações são apresentadas ao usuário, sem causar impactos significativos na usabilidade e também na navegabilidade.

Para tal, a interface proposta foi dividida em três elementos principais, conforme o layout definido a seguir (Figura 8.14).

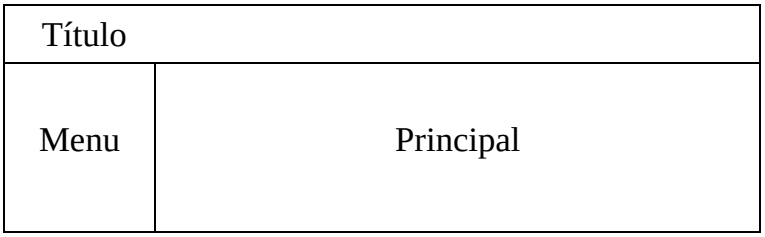

Figura 8.14 – Layout da interface.

8.7.2.1 Menu

Localizado a esquerda da página principal o menu do sistema procura apresentar as principais atividades que o usuário do repositório pode realizar sobre o sistema (Figura 8.15).

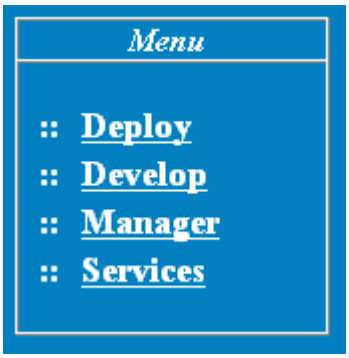

Figura 8.15 – Menu do sistema.

- **Principal**

No centro da tela está a área de trabalho no ambiente do protótipo proposto. Nela serão exibidos os elementos básicos e genéricos que compõem a interface com o usuário.

Estes elementos são classificados em *Window*, *Wizard* e *Service*. Os dois primeiros possuem características bem definidas e o último é apenas uma abstração, o que permite ao desenvolvedor trabalhar sua interface com o usuário do repositório, viabilizando sua extensão sem qualquer limitação de interface imposta pelo sistema.

Os possíveis elementos presentes na área principal serão descritos e apresentados a seguir:

- **Window**

Exibida no centro da página principal, uma *Window* sempre será criada após a seleção de um item de menu, apresentando as opções de navegação disponíveis e relacionadas com a atividade escolhida.

As opções da janela sempre serão apresentadas na forma de ícones com uma breve descrição. A seleção de um destes ícones irá iniciar um *Wizard*, disparar a execução de um serviço de manutenção, ou ainda, apresentar a interface pública (*Service*) para a utilização de um serviço específico.

As *Windows* são portanto elementos de interface que procuram centralizar a execução de tarefas correlatas para o usuário, como na Figura 8.16, que apresenta uma janela de gerenciamento, permitindo que usuários, publicações e serviços possam ser gerenciados a partir deste ponto.

Figura 8.16 – Janela com opções de gerenciamento.

- **Wizard**

Iniciados sempre a partir da seleção de um ícone, os *Wizard´s* procuram guiar o usuário durante a realização de uma tarefa específica da administração ou utilização do repositório.

Basicamente as tarefas mais corriqueiras, bem como, as mais extensas são apresentadas ao usuário sob a forma de um *Wizard*, que é caracterizado por possuir a metáfora *Next-Next-Apply*.

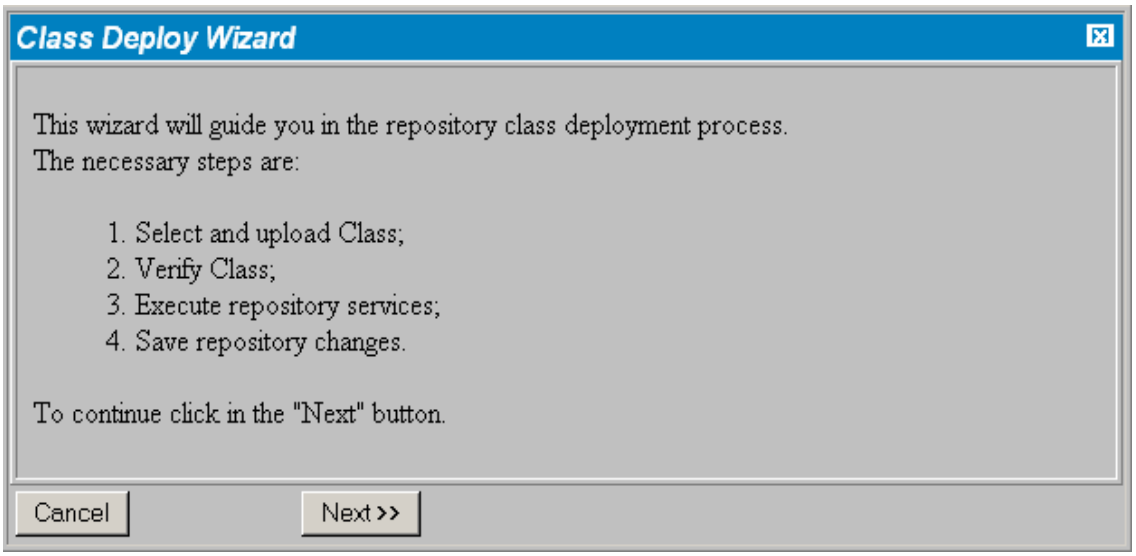

Figura 8.17 – Exemplo de um assistente.

Nesta metáfora, o usuário deve ler e/ou informar os dados necessários, clicar o botão *Next*, passando para o próximo painel e assim, sucessivamente, até que seja apresentado o botão *Apply*.

Neste momento, o painel irá apresentar sempre um resumo das atividades que serão realizadas pelo *Wizard* corrente e, ao ser clicado o botão *Apply*, o sistema executa e salva todas as alterações necessárias à efetiva conclusão da sua tarefa.

Para a implementação deste protótipo foram criados *Wizard´s* para as seguintes atividades:

- Class Deployment
- Service Deployment
- Manager Deployment

Um *Wizard* também pode ter sua execução cancelada a qualquer momento antes de ser executado o *Apply*. Para tal, basta clicar no botão *Cancel*. Como nenhuma alteração foi ainda realizada, o *Wizard* apenas descarta as informações até então coletadas e retorna para a *Window* que o iniciou.

- **Service**

Sempre que um serviço precisar apresentar sua interface pública para o usuário do repositório, este serviço irá fazê-lo pela extensão do elemento de interface *Service*, que será chamado a partir de um link na janela de serviços (*Services*).

Este elemento define apenas uma área de interface que pode ser utilizada pelo desenvolvedor do serviço como este entender melhor, o que possibilita a máxima flexibilidade na definição da interface pública e no desenvolvimento de um novo serviço que será adicionado ao repositório.

Para ilustrar esta flexibilidade, a Figura 8.18 mostra um exemplo de interface pública de um serviço, no caso, a interface proposta utiliza a metáfora de navegação em árvore para passar a idéia de hierarquia entre a super classe e as demais classes que a estendem.

Para efeito de comparação, tem-se na Figura 8.19, um outro exemplo de interface pública de um serviço, que utiliza um conceito similar a um gerenciador de arquivos, onde o usuário pode selecionar classes e realizar operações diretamente sobre estas.

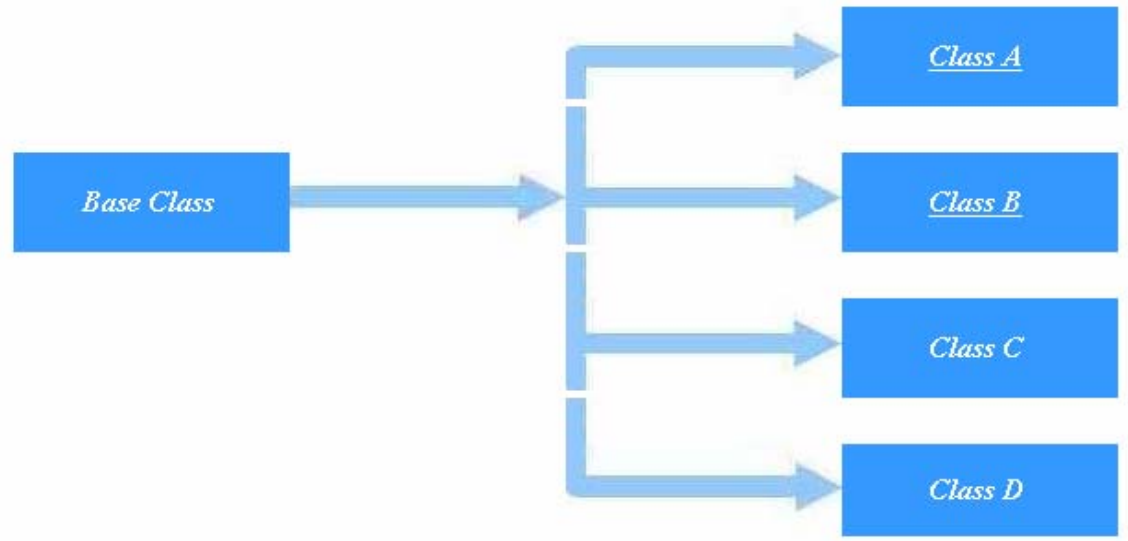

Figura 8.18 – Exemplo de um service.

HOME / com / cvslet

Documentation Download

File	Rev	Timestamp
doc/		
images/		
DefaultOutput.java	1.2	Sun Nov 11 01:09:51 2001
JCVSlet.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletConstants.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletEntry.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletLog.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletOutput.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletProject.java	1.2	Sun Nov 11 01:09:51 2001
JCVSletRequest.java	1.2	Sun Nov 11 01:09:51 2001
OutputHelper.java	1.2	Sun Nov 11 01:09:51 2001
ProjectDef.java	1.2	Sun Nov 11 01:09:51 2001
ProjectMgr.java	1.2	Sun Nov 11 01:09:51 2001
TaskThread.java	1.2	Sun Nov 11 01:09:51 2001
Utilities.java	1.2	Sun Nov 11 01:09:51 2001
Zipper.java	1.2	Sun Nov 11 01:09:51 2001
default.properties	1.2	Sun Nov 11 01:09:51 2001
rsrc.properties	1.2	Sun Nov 11 01:09:51 2001

Figura 8.19 – Exemplo de outro service.

Conclusão

Ao utilizar-se linguagens de programação orientada a objetos visa-se principalmente tirar vantagem de suas características inerentes ao paradigma, tais como: herança, polimorfismo, encapsulamento, entre outras. Tais características têm como objetivo o reuso de código já escrito e testado, a fim de aumentar a capacidade de desenvolvimento através da extensão de classes preexistentes.

Atualmente, porém, o padrão de fato empregado para o reuso de código e classes é baseado na utilização de componentes de software que agregam classes, métodos e eventos definindo sua funcionalidade. A medida que a indústria de software foi evoluindo, os componentes passaram a estar em evidência junto a todos os desenvolvedores. Entretanto, apesar de ter-se começado a tirar vantagem dos componentes no desenvolvimento de aplicativos, ainda não conseguiu-se alavancar os recursos de componentes disponíveis.

A Internet, e com ela o aumento na produção de componentes de software e comércio eletrônico, sem levar em consideração a demanda de repositórios para o desenvolvimento orientado à reutilização para uso interno das organizações, cria a necessidade de tornar catálogos de desenvolvimento disponíveis para o mundo externo segue uma linha básica e integrada com a Web, utilizando para tal interfaces *World Wide Web* (WWW) baseadas em formulários. Este argumento vem tornar-se decisivo e, sobretudo, traz à tona a absoluta necessidade da existência de repositórios de componentes de software centralizados. Os repositórios são absolutamente necessários para a centralização do tipo de acesso remoto que é imposto pelas redes globais e para a engenharia das equipes de desenvolvedores descentralizados suportados por estas.

Através da implementação de um modelo de repositório de componentes baseado na arquitetura J2EE, pode-se contar com um poderoso ambiente de desenvolvimento integrado, garantindo que componentes de software possam ser desenvolvidos uma única vez e distribuídos em qualquer ambiente operacional, provendo um ciclo de vida consistente e serviços de controle de transações, persistência, distribuição e segurança.

Esta é a característica mais importante na arquitetura J2EE, a qual mantém uma arquitetura totalmente independente de qualquer plataforma, protocolo ou infra-estrutura *middleware* específica. As aplicações desenvolvidas para uma plataforma específica, podem ser migradas para uma outra, podendo ser escaláveis a partir de um pequeno servidor monoprocessado para um grande multiprocessado ou ainda um *mainframe* sem qualquer modificação em seu código fonte.

Com todos os recursos de infra-estrutura e serviços necessários disponíveis de forma automática, tais como controle de transações, gerenciamento de *threads* e verificação de segurança, os desenvolvedores, tanto de componentes como de aplicações não precisam se preocupar em implementar funções complexas de gerência e podem deter-se somente no desenvolvimento da lógica de sua aplicação.

Muitos outros tipos de serviços podem e devem ser agregados pelos desenvolvedores ao modelo proposto, já que esta é uma especificação aberta e totalmente baseada em Java.

Durante a execução deste trabalho, foram identificados alguns aspectos que podem ser melhor explorados ou estendidos em trabalhos futuros:

- o mecanismo de persistência adotado é baseado na tecnologia JDBC, que apesar de deixar o mapeamento objeto-relacional a cargo do programador é a API mais sólida e estável até o presente momento. Ideal seria a utilização da API JDO, que fornece recursos mais avançados e características desejadas, tais como persistência ortogonal e inferida, porém é necessário que esta tecnologia seja consolidada para que todas estas vantagens possam ser exploradas de modo transparente;

- o serviço de controle de versões agregado ao repositório foi implementado através de uma camada de software que realiza a comunicação entre as classes Java pertencentes ao repositório com um sistema de controle de versões externo. Futuramente espera-se que seja desenvolvido um controle de versões completamente baseado em Java, tornando esta implementação totalmente portável;

- o serviço de descrição de componentes utiliza uma base de dados relacional para armazenar todas as informações pertinentes às classes registradas, porém um mecanismo de descrição baseado em XML seria mais flexível, permitindo a extensão do modelo de representação com um impacto menor sobre o sistema como um todo;

- uma exploração maior dos recursos de HTML, DHTML e JavaScript utilizados no protótipo proposto seria muito interessante, de forma a tornar a utilização do sistema e a definição de suas interfaces o mais amigável possível para os usuários, em todos os níveis de utilização.

Em um futuro próximo, as equipes de desenvolvimento consistirão de desenvolvedores de componentes e montadores de aplicações. Os desenvolvedores de componentes receberão especificações baseadas nos pedidos das equipes de montagem de aplicações, criarão novos componentes ou personalizarão componentes já existentes, avaliarão componentes e os tornarão disponíveis. A equipe de montagem de aplicações juntará os componentes, avaliará a função da aplicação e executará a aplicação integrada. Em muitos casos, a equipe de montagem da aplicação poderá sair dos departamentos de Sistemas de Informação e os desenvolvedores de componentes serão os fornecedores externos.

Este trabalho apresentou a modelagem de um repositório de componentes de software utilizando a tecnologia Java. O repositório é formado por uma área de armazenamento centralizada e comum aos desenvolvedores, possuindo capacidade de gerenciar componentes e agregar serviços extras, visando auxiliar o desenvolvimento de novos componentes e a evolução dos já existentes, administrando pois o ciclo de vida de modo integrado.

Referências Bibliográficas

[APP97] APPLETON, B. **Patterns and Software:** essential concepts and terminology. Disponível em: <www.enteract.com/~bradapp/docs/>. Acesso em: 14 julho 2001.

[BRE98] BRESLIN, J.; BAUMAN, B. E.; MCGANN, J. **The Business Knowledge Repository**. [S.l.]: Greenwood Publishing Group, 1998.

[COR97] CORBA services: common objetct services specification, OMG Document, 12/02/1997. Disponível em: <http://www.odmg.org>. Acesso em: 29 julho 2001.

[FOW97] FOWLER, M. **UML Distilled: a**pplying the standard object modeling language. Reading: Addison-Wesley, 1997.

[GAM95] GAMMA, E. **Design-Patterns:** elements of reusable object-oriented software. Reading, MA: Addison-Wesley, 1995.

[HAR92] HARROLD, M. J. et al. Incremental Testing of Object-Oriented Class Structure, In: INTERNACIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 14, 1992, Melbourne, AU. **Proceedings...** Los Alamitos: IEEE, 1992.

[HER99] HERBERT, J. S. **Teste Cooperativo de Software**. 1999. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IEE90] IEEE Standart Glossary of Software Engineering Terminology. Padrão 610.12. [S.l.], 1990.

[JAC92] JACOBSON, I. et al. **Object-Oriented Software Engineering**: a use case driven approach. Reading: Addison-Wesley, 1992.

[MAR2000] MARCO, D. **Building and Managing the Meta Data Repository:** a full lifecycle guide. Reading: John Wiley & Sons, 2000.

[MOR99] MORRISON, M. et al. **Como Programar em JavaBeans**. São Paulo: Makron Books, 1999.

[MYE79] MYERS, G. **The Art of Software Testing**. New York: John Willey & Sons, 1979.

[ODM96] ODMG. The Common Object Request Broker: Architecture e Specification, Revision 2.0, OMG Document 04/03/1996. Disponível em: <http://www.odmg.org>. Acesso em: 21 julho 2001.

[PRE95] PRESSMAN, R. **Engenharia de Software**. São Paulo: Makron Books. 1995.

[RAP85] RAPPS, S. Selecting Software Test Data Using Data Flow Information. **IEEE Transactions on Software**, Los Alamitos, v.11, n.4, Apr. 1985.

[RAT97a] RATIONAL SOFTWARE CORPORATION. **UML Notation Guide**: version 1.1. 1997. Disponível em: <www.rational.com/uml>. Acesso em: 23 novembro 2001.

[RAT97b] RATIONAL SOFTWARE CORPORATION. **UML Semantics**: version 1.1. 1997. Disponível em: <www.rational.com/uml>. Acesso em: 17 novembro 2001.

[ROS99] ROSENBERG, D.; SCOTT, K. **Use Case Driven Object Modeling with UML**. Reading: Addison-Wesley, 1999.

[TAN94] TANNENBAUM, A. **Implementing a Corporate Repository:** the models meet reality. Reading: John Wiley & Sons, 1994.

[WEY80] WEYUKER, E. J. Theories of Program Testing and the Application of Revealing Subdomains. **IEEE Transactions on Software**, Los Alamitos, v.6, n.3, May 1980.

[URL01] www.cetus-links.org

[URL02] www.sei.cmu.edu

[URL03] www.korzh.com

[URL04] www.javasoft.com/beans

[URL05] www.javasoft.com/beans/docs/spec.html

[URL06] www.javasoft.com

[URL07] www.sun.com

[URL08] http://java.sun.com/products/jsp.

[URL09] http://hillside.net/patterns

[URL10] http://www.normos.org

[URL11] http://c2.com/ppr/index.html

[URL12] http://cio.doe.gov/standards/repository_info.htm

[URL13] http://www-it.hr.doe.gov/standards/stdsdocs/prof1-5.htm

[URL14] http://www.w3.org/Protocols

[URL15] http://www.netscape.com/newsref/std/cookie_spec.html

[URL16] http://www.cetus-links.org/oo_repositories.html

[URL17] http://www.platinum.com/products/dataw/repos_ps.htm

[URL18] http://www.unisys.com/marketplace/urep/

[URL19] http://www.viasoft.com/products/product_details.asp?id=37

[URL20] http://www.gartner.com/reprints/asg/101382.html