Componentes para o Desenvolvimento de *Intranets*

Guilherme Rodrigues de Matos Junior

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Componentes para o Desenvolvimento de *Intranets*

Guilherme Rodrigues de Matos Junior [1]

Maio de 2001

Banca Examinadora

- Prof. Dr. Rogério Drummond (Orientador)
- Prof. Dr. Carlos Alberto Maziero[2]
- Prof. Dr. Ricardo Anido[3]
- Profa. Dra. Cecília Rubira[4]

[1] Projeto financiado pelo CNPq, processo Nº 131938/97-5, e pela FAPESP, processo Nº 97/11105-3
[2] Centro de Ciências Exatas e de Tecnologia, Programa de Pós-Graduação em Informática Aplicada (PPGIA), PUC-PR
[3] Instituto de Computação da UNICAMP
[4] Instituto de Computação da UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Matos Jr., Guilherme Rodrigues de

M428c Componentes para o desenvolvimento de intranets / Guilherme Rodrigues de Matos Junior -- Campinas, [S.P. :s.n.], 2001.

Orientador : Rogério Drummond

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Redes de computadores. 3. Internet (Rede de computadores). 4. Intranet (Redes de computação). I. Drummond, Rogério. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Componentes para o Desenvolvimento de *Intranets*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Guilherme Rodrigues de Matos Jr e aprovada pela Banca Examinadora.

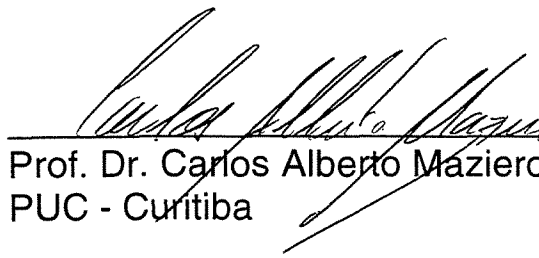
Campinas, 10 de Maio de 2001

Prof. Dr. Rogério Drummond (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 10 de maio de 2001, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Carlos Alberto Maziero
PUC - Curitiba

Prof. Dr. Ricardo de Oliveira Anido
IC – UNICAMP

Prof. Dr. Rogério Drummond Burnier Pessoa de Melo Filho
IC - UNICAMP

Agradecimentos

Aos companheiros do Instituto de Computação da Unicamp e do Laboratório A-HAND, especialmente ao Emerson Yoshio Takahashi, Alexandre Prado Teles, Carlos Alberto Furuti e Rodrigo Magalhães, que contribuiram diretamente para a execução deste trabalho.

Aos professores do Instituto de Computação da Unicamp, sobretudo ao meu orientador Rogério Drummond pelos conselhos, oportunidades e acima de tudo, pelo exemplo de obstinação ao trabalho.

Aos amigos da BCP pelo aprendizado, pelas oportunidades e pelo reconhecimento do meu trabalho.

Ao CNPq pelo financiamento do primeiro ano deste projeto e à FAPESP que financiou os equipamentos utilizados e os últimos 15 meses deste trabalho.

Por fim, e nem por isso menos importante, a fonte de inspiração pessoal e profissional que tem me acompanhado por todas as etapas deste trabalho: Ká, sou imensamente grato pelo seu apoio incondicional. Espero que este trabalho seja o primeiro de uma série em nossos projetos de vida !

Dedico este trabalho a minha família, que se a princípio não se identificou com a idéia da distância, pôde ao menos compartilhar a satisfação do dever cumprido. A todos meu muitíssimo obrigado pelo imenso apoio, dedicação e carinho, e como não poderia deixar de citar, da enorme saudade acumulada nestes últimos anos.

Resumo

A construção de sistemas baseados nas tecnologias Internet fez surgir um novo paradigma de desenvolvimento de *software*: o de sistemas *Web*. A demanda por sistemas *Web* foi impulsionada a partir da abertura da Internet ao uso comercial, quando o mercado passou a empregá-la como infra-estrutura para a distribuição de informações, aplicações e serviços aos seus funcionários (*intranets*), parceiros (*extranets*) e clientes (*sites* Internet).

Entretanto, como a Internet não foi concebida para estas finalidades, foram lançadas inúmeras iniciativas no sentido de resolver suas deficiências. A tentativa de suprir a demanda do mercado por novas soluções *Web* resultou no surgimento de inúmeras tecnologias, marcadas pela complexidade e imaturidade. Estes fatores contribuiram para dificultar a tarefa de se construir aplicações *Web* de forma correta, eficiente, rápida e barata.

Este trabalho analisa as dificuldades relacionadas ao desenvolvimento de *software Web*, e propõe alguns conceitos e ferramentas para facilitar as etapas de projeto e implementação. Com relação ao projeto de *intranets* apresentamos uma discussão sobre as oportunidades, as etapas e os recursos envolvidos. No contexto da implementação de sistemas *Web*, este trabalho contribui com uma discussão sobre as principais arquiteturas e tecnologias de implementação de *software Web*, com o desenvolvimento de quatro componentes de propósito geral, e um *framework* para o gerenciamento de documentos em *intranets*.

Abstract

The development of Internet based systems has introduced the paradigm of Web applications. Since Internet was opened to the market, the demand for Web systems has increased as a consequence of using Internet as an infrastructure to deploy information, applications and services to employees (intranets), partners (extranets) and clients (Internet sites).

On the other hand, since the Internet had not been conceived for these purposes, a lot of initiatives was launched to bypass its deficiences. Actually, while the market tried to supply the demand for new Web solutions, they did that in a manner that increased the number of technologies, as well as the complexity and immaturity of each one. These factors contributed to increase the difficulties in developing Web software in a correct, eficient, fast and cheap way.

This project focuses on difficulties related to the Web software development process, and proposes some concepts and tools to make the project and implementation tasks easier. At intranet planning, this work can be used to define the common steps, resources, oportunities, services and applications that an intranet must address. Concerning to implementing Web applications, this work offers a proof of concept through development of four general purpose components and a framework for intranet document management built on top of these components.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviações

ACL	Access Control List
API	Application Programming Interface
B2B, B2C	Business to Business / Consumer
CCM	CORBA Component Model
CFML	Cold Fusion Markup Language
CGI	Common Gateway Interface
CIDL, IDL	Component / Interface Definition Language
CORBA	Common Object Request Broker Architecture
DCOM, COM	Distributed / Component Object Model
DHTML, HTML	Dynamic / Hypertext Markup Language
DNS	Domain Name Service
EJB	Enterprise Java Beans
ERP	Enterprise Resource Planning
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IIOP	Internet Inter-Orb Protocol
INEX	Intranet Express
ISAPI	Internet Server API
ISO	International Standards Organization
JAR	Java Archive
JDBC	Java Database Connectivity
JFC, MFC	Java / Microsoft Foundation Classes
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface
JSP, ASP	Java / Active Server Pages
JTS	Java Transaction Service
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol

MTS	Microsoft Transaction Server
MVC	Model View Controller
NSAPI	Netscape Server API
OLE	Object Linking and Embedding
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
RMI	Remote Method Invocation
ROI	Return On Investment
SGBD	Sistema Gerenciador de Banco de Dados
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
SSE	Server Side Extensions
TCP/IP	Transport Control Protocol / Internet Protocol
TP Monitors	Transaction Processing Monitors
UML	Unified Modeling Language
URL, URI	Unified Resource Locator / Identifier
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WWW, Web	World Wide Web
XLST	Extensible Stylesheet Language Transformations
XML	Extended Markup Language

Capítulo 1

Introdução

A utilização da Internet em âmbito comercial gerou um processo de popularização e evolução sem precedentes das suas tecnologias. A idéia de se empregar a infra-estrutura de comunicação da Internet e dos seus meios de distribuição de informações efetivou-a como uma das formas mais eficientes de se implantar processos de negócio que dependem da publicação de informações e comunicação de usuários - no cenário comercial, a Internet permite divulgar e comercializar produtos e serviços a um público global e a baixo custo.

Recentemente, as organizações têm utilizado a experiência adquirida com a Internet para implantar *intranets*, que oferecem informações, serviços e aplicações institucionais aos seus funcionários de maneira similar à Internet. As *intranets* empregam as funcionalidades da Internet em âmbito institucional (privado) para promover a comunicação e a colaboração dos usuários (funcionários, parceiros, clientes, etc.) e automatizar processos administrativos.

Entretanto, como a Internet e as tecnologias que a suportam (TCP/IP e HTTP, por exemplo) não foram inicialmente projetadas para estas finalidades, temos acompanhado o surgimento de inúmeras soluções em termos de protocolos, linguagens e ferramentas.

Como as soluções baseadas na Internet têm sido empregadas tanto em âmbito público (*sites* Internet), quanto privado (*intranets*), a demanda por *software Web* tem aumentado em número e complexidade. A demanda por *software Web* cada vez mais sofisticado e as deficiências das tecnologias da Internet fez com que as tecnologias que suportam o desenvolvimento de *software Web* também evoluíssem rapidamente, tanto em número quanto em complexidade. Esta não corresponde à complexidade inerente da lógica de aplicação dos *softwares* envolvidos, mas às dificuldades que as tecnologias e a arquitetura *Web* impõem ao seu desenvolvimento, como podemos observar a seguir:

- **Imaturidade:** com a demanda pela utilização da Internet no âmbito comercial, surgiram inúmeras soluções, tanto abertas, definidas por órgãos de padronização (OMG, W3C), quanto proprietárias, desenvolvidas por fornecedores de ferramentas de desenvolvimento. Este cenário foi marcado pela pressão da demanda de mercado, o que contribuiu para o lançamento

de soluções imaturas. Na prática, a imaturidade das tecnologias se refletiu nas aplicações que as utilizaram, pois ao empregar soluções incipientes, muitos projetos de *software* ou o fizeram de forma incorreta, ou herdaram as deficiências inerentes às ferramentas.

- **Complexidade:** além de ser inerentemente distribuído, um *software Web* é construído a partir da combinação de uma série de tecnologias (linguagens, protocolos e ferramentas), geralmente imaturas e complexas. Não raro, um projeto de *software Web* lida com 3 ou mais linguagens de programação, que ainda podem em muitos casos, ser combinadas num único código fonte.
- **Exigência de mão-de-obra:** a dificuldade de se escolher e combinar as diversas tecnologias, a complexidade em utilizá-las e a necessidade de recursos altamente capacitados, representam a dinâmica do processo de desenvolvimento de *software Web*.
- **Falta de suporte a reutilização:** como as primeiras iniciativas de desenvolvimento *Web* eram voltadas a aplicações e serviços mais simples como a publicação de informações e controle de formulários HTML, o desenvolvimento das soluções *Web* não acompanhou a princípio a disciplina de reutilização de *software*; em outras palavras, grande parte das soluções não oferecem suporte a reutilização de *software*.

Estas deficiências foram levantadas, em grande parte, através da *extranet* do SBRC'97 (Simpósio Brasileiro de Redes de Computadores de 1997), um projeto desenvolvido pelo laboratório A-HAND (Unicamp), do qual fazem parte os autores deste trabalho. A *extranet* do SBRC'97 permitiu realizar todos os trabalhos de divulgação de notícias, submissão e controle das revisões e aceitação de artigos através da Internet.

Por ter se tornado referência, o projeto da *extranet* do SBRC poderia ser reutilizado em outros eventos, desde que suportasse algumas premissas como flexibilidade, extensibilidade e reutilização de código e de projeto - para permitir a sua configuração sob as particularidades de cada evento. Na prática, porém, a complexidade e a imaturidade das tecnologias empregadas no desenvolvimento da *extranet* do SBRC, a falta de suporte a reutilização de *software* e a alta exigência de mão-de-obra inviabilizaram a reutilização do seu projeto.

Neste trabalho, procuramos fornecer alternativas aos problemas levantados durante o projeto da *extranet* do SBRC. Para lidar com a complexidade e imaturidade das tecnologias, realizamos um levantamento sobre arquiteturas de *software Web* com o objetivo de identificar as melhores práticas na escolha e combinação de tecnologias de implementação. O problema de reutilização de código é abordado através da implementação de componentes de *software*. A idéia de se conceber soluções recorrentes (ou projetos reutilizáveis, como o da *extranet* do SBRC) é exemplificada através da implementação de um *framework* para gerenciamento de documentos em *intranets*.

Além dos aspectos técnicos que permeiam o desenvolvimento de aplicações *Web*, podemos destacar as dificuldades em se definir uma abordagem para explorar as soluções *Web*, como ocorre na implementação de *intranets*. Apesar das pesquisas e *benchmarks* da área apontarem uma série de vantagens em se empregar *intranets* [Cam97][Hil96][KM97][Usw97][Usw98], é imprescindível projetá-las sob um processo que aponte quais as oportunidades, as etapas e os recursos envolvidos em sua implementação.

O objetivo deste trabalho é oferecer soluções para o desenvolvimento de *intranets*, tanto nos aspectos técnicos relacionados ao desenvolvimento de *software Web*, quanto nos fatores organizacionais envolvidos na definição de uma *intranet*. Neste sentido, contribuímos com a elaboração de um *survey* sobre o processo de desenvolvimento de *intranets*, onde são

identificadas as questões pertinentes a cada etapa do desenvolvimento e as principais aplicações e serviços de uma *intranet,* e portanto, os recursos de *software* mais utilizados.

Para trabalhar os problemas acerca do desenvolvimento de *software Web*, investigamos as principais tecnologias e arquiteturas envolvidas e identificamos quando e como utilizar cada uma. O levantamento sobre arquiteturas de *software Web* revelou a tendência em se dividir as aplicações *Web* através de camadas lógicas (componentes e subsistemas) e físicas (sistemas distribuídos). Neste sentido, e nas tentativas de promover a reutilização de *software* e de diminuir os esforços no desenvolvimento de aplicações para a *Web*, adotamos a abordagem de componentes de *software* e procedemos com a implementação de quatro componentes a partir das funcionalidades recorrentes identificadas no *survey* sobre *intranets* (capítulo 2).

Por fim, como forma de realizar uma prova dos conceitos relacionados às arquiteturas e tecnologias *Web* e dos componentes de *software* desenvolvidos, implementamos parte de um projeto de *software* voltado ao gerenciamento de documentos em *intranets*, a INEX. Além de servir de base para as provas de conceito dos resultados deste trabalho, a INEX constitui um *framework* que pode ser estendido para implementar diversos serviços e aplicações de *intranets*, e deverá, assim como os componentes deste trabalho, promover a reutilização de *software* e diminuir os esforços despendidos na construção de aplicações para *intranets*.

Os componentes e parte do projeto da INEX desenvolvidos neste trabalho, assim como o *survey* sobre o processo de desenvolvimento de *intranets* e a discussão sobre os aspectos não funcionais relacionadas ao desenvolvimento de *software Web* são apresentados neste documento através da seguinte estrutura:

- O capítulo 2 apresenta o *survey* sobre o processo de desenvolvimento de *intranets*;
- O capítulo 3 aborda as principais tecnologias e arquiteturas de *software* utilizadas na construção de aplicações *Web* e apresenta um estudo comparativo entre as tecnologias de implementação;
- O capítulo 4 discute o processo de desenvolvimento de componentes de *software* e de aplicações baseadas em componentes, além de apresentar as principais ferramentas disponíveis atualmente para realizar a construção e execução (*application servers*) de componentes de *software* para a *Web*;
- Os capítulos 5 e 6 apresentam os trabalhos de implementação dos componentes e da INEX, respectivamente;
- O capítulo 7 discute as tecnologias e arquiteturas empregadas na implementação, propõe algumas extensões aos componentes e à INEX, e apresenta as conclusões.

Capítulo 2

Intranets

As *intranets* constituem um dos maiores exemplos de como as tecnologias Internet têm sido amplamente adotadas como infra-estrutura para construção dos sistemas de informação atuais. Da mesma forma que ocorreu com a Internet, as organizações decidiram implementar *intranets* de forma evolutiva, iniciando com a publicação de informações estáticas, e posteriormente empregando as tecnologias Internet para construir sistemas cliente/servidor *Web* mais sofisticados.

Ao contrário de *sites* Internet que são utilizados prioritariamente para a publicação de informações, as *intranets* geralmente incorporam inúmeros sistemas cliente/servidor *Web*. Como este trabalho enfoca o desenvolvimento de sistemas *Web*, optamos por trabalhar no domínio de aplicações sobre *intranets*.

Este capítulo apresenta as principais vantagens de se empregar *intranets* e destaca as principais etapas, recursos, decisões e conceitos acerca do processo de desenvolvimento de *intranets*. Como discute as soluções de *software* mais comuns em *intranets*, este capítulo serve de base para a definição das ferramentas implementadas neste trabalho.

2.1 Definição

Uma *intranet* é uma rede baseada nas tecnologias da *Internet* (protocolos TCP/IP e HTTP, navegadores e servidores WWW) que é utilizada na distribuição de informações, serviços e aplicações cliente/servidor privadas (institucionais). *Intranets* são implementadas sobre uma infra-estrutura de redes existentes e utilizam mecanismos de segurança para controlar o acesso somente aos usuários autorizados.

As *intranets* têm se tornado populares por oferecer um meio inovador e eficiente de distribuir informações corporativas, até então disponíveis através de sistemas de informação proprietários e de alto custo. O objetivo de uma *intranet* é oferecer uma infra-estrutura para a disseminação de informações, comunicação e colaboração de usuários, automatização de processos e integração

dos sistemas de informação sob um ambiente de acesso global, multiplataforma, privado e seguro.

Uma *intranet* pode oferecer serviços para comunicação e colaboração de usuários (*email* e *news*), publicação e gerenciamento de informações (serviço WWW) e aplicações cliente/servidor *Web* que implementam funcionalidades de *groupware, workflow*, acesso a banco de dados e integração com sistemas legados.

Quando uma organização disponibiliza alguns serviços e aplicações da sua *intranet* aos seus parceiros de negócio, diz-se que há uma *extranet.* Uma *extranet* é uma extensão da *intranet* utilizada para aprimorar as relações entre organizações que compartilham um objetivo comum. Na prática, uma *extranet* é uma interseção de *intranets* corporativas que cooperam para executar atividades de negócio distribuídas entre organizações. Numa empresa, por exemplo, uma *extranet* pode integrar os departamentos de produção e financeiro ao departamento de vendas dos seus fornecedores; ou os departamentos de produção e vendas ao departamento de compras dos seus clientes. Atualmente, tem-se empregado a terminologia de portais *business to business* (B2B) para designar *extranets* e *business to consumer* (B2C) para *sites* Internet; o propósito de portais B2B é automatizar processos corporativos inter-organizacionais para aumentar a eficiência na comunicação e diminuir custos.

A adoção das tecnologias Internet pela iniciativa privada tem ocorrido em três etapas. Inicialmente, as organizações utilizam a Internet para publicar informações sobre seus produtos e serviços, e obter retorno dos usuários. Após experimentar bons resultados com a Internet, as empresas tendem a aplicar sua experiência na implantação de *intranets,* publicando informações institucionais (WWW interna), instalando serviços de comunicação (*email*) e desenvolvendo aplicações cliente/servidor baseadas na *Web.* Por fim, há a evolução da *intranet* para a *extranet,* quando as empresas implementam os processos de negócio que envolvem a comunicação entre organizações sobre as *intranets* dos parceiros envolvidos, como é o caso das transações eletrônicas de compra/venda de matéria-prima, produtos e serviços.

2.2 Benefícios

Segundo [RM97], [Cam97], [Usw97] e [Usw98] projetos de *intranets* e *extranets* têm revelado tanto vantagens operacionais, como o aumento da satisfação e produtividade dos seus usuários, quanto econômicas, com o corte de gastos e aumento do lucro. As principais vantagens das *intranets* são decorrência do emprego dos padrões da Internet, como podemos observar a seguir:

- **Custo:** uma *intranet* é implantada sobre a infra-estrutura de *hardware* (clientes, servidores e equipamentos de rede) e *software* (bancos de dados, *data warehouses* e sistemas legados) existentes. Desta forma, o custo de implantação é em geral atraente, pois os investimentos já realizados são preservados.
- **Portabilidade:** construir *intranets* a partir de padrões abertos (TCP/IP, Java, JDBC, CORBA) garante a interoperabilidade de aplicações entre redes, plataformas, sistemas operacionais e bancos de dados heterogêneos, além de diminuir o custo total de propriedade[1] com informática.
- **Escalabilidade:** uma *intranet* pode ser projetada sobre uma arquitetura distribuída e descentralizada, com *sites* globais, regionais e locais, por exemplo. Neste cenário, é possível expandir a infra-estrutura de *hardware* à medida que a demanda por processamento aumenta,

[1] Tradução de *total cost of ownership*

sem interferir nos recursos de *software* existentes. Nas arquiteturas de *intranets*, há uma tendência de descentralização através do esquema matriz, filiais e departamentos; assim, o aumento de demanda sobre uma *intranet* pode ser resolvido através da expansão da sua infra-estrutura ou ainda através da sua descentralização em *sites* menores e específicos.

- **Interface universal:** a utilização de um navegador *Web* como interface padrão de acesso à *intranet* diminui os custos com treinamento, suporte e manutenção. A idéia é empregar a experiência dos usuários familiarizados com a Internet (uso de *browsers* e navegação através de páginas HTML) na operação de uma *intranet*, o que diminui as barreiras psicológicas à adoção de uma nova tecnologia, e facilita as atividades de treinamento de usuários. A utilização de um *browser* como interface padrão da *intranet* permite centralizar as informações e aplicações no servidor, diminuindo as atividades (e os custos) de gerenciamento e manutenção.
- **Acesso global:** como uma *intranet* emprega uma série de mecanismos de controle de acesso (filtros, *firewalls*, autenticação, etc.), pode ser utilizada a partir da Internet por usuários remotos, distribuídos geograficamente.
- **Mídia diversificada:** assim como a Internet, uma *intranet* pode distribuir informações em diversos formatos (texto, gráficos, vídeo, áudio, etc.).

Os benefícios associados às *intranets* não se limitam às oportunidades tecnológicas. O uso dos serviços de *email* e *news* permite aprimorar a comunicação dos usuários, enquanto que a publicação e a distribuição de aplicações através da *Web* interna facilitam o acesso à informação. Uma *intranet* é um meio ideal para manter e atualizar informações sobre negócios, pois dispensa a necessidade de produção, duplicação e distribuição de informações em papel, e as mantém ao alcance dos usuários, fator essencial para as atividades que envolvem tomada de decisões. Em outras palavras, as instituições que administrarem estas vantagens em benefício de seus processos de negócio podem aumentar a produtividade e a satisfação dos seus usuários (funcionários, parceiros, clientes, etc.).

Porém, há algumas desvantagens relacionadas às *intranets* tanto no que se refere a construção de aplicações *Web* quanto no planejamento do seu conteúdo (informações, serviços e aplicações). Quanto ao desenvolvimento de aplicações, o excesso e a complexidade das tecnologias atualmente disponíveis, além do esforço requerido na sua combinação, são apontados como os maiores problemas para a construção das aplicações de uma *intranet*. Quanto ao planejamento, [Tel97] destaca as dificuldades em se controlar quais informações podem ser publicadas, quem pode publicar o quê, quem pode acessar qual informação, e assim por diante.

2.3 Processo de Desenvolvimento

Normalmente, as empresas implementam *intranets* e *sites* Internet como parte de uma política de atualização tecnológica, ou seja, como uma forma de melhorar sua imagem junto aos seus funcionários, parceiros e clientes. Entretanto, uma iniciativa mais elaborada, que explore efetivamente as oportunidades destas tecnologias, deve ser baseada num plano de desenvolvimento que indique quais as etapas, as atividades, os cargos, as responsabilidades e as tecnologias envolvidas. Embora não exista uma receita padrão nem modelos bem definidos, há uma série de similaridades entre projetos bem sucedidos, como o processo de desenvolvimento composto pelas seguintes etapas [Net98]:

1. **Planejamento**, ou o estudo das oportunidades sob a ótica dos objetivos da organização.

2. **Projeto**, ou definição das informações, serviços e aplicações concebidas durante o planejamento, da infra-estrutura necessária, e de um plano de atividades para a execução do projeto.
3. **Implementação**, ou a criação das informações e a implementação das aplicações.
4. **Implantação** dos serviços, informações e aplicações para os usuários, através da política definida no planejamento.
5. **Gerenciamento**, ou manutenção, revisão e análise dos recursos implantados.

O processo de desenvolvimento de *intranets* é iterativo e evolutivo; em geral, os serviços e aplicações são implantados em etapas, segundo a prioridade da organização; durante a etapa de gerenciamento, os recursos podem ser reavaliados e novos serviços/aplicações podem ser concebidos, construídos e implantados.

2.3.1 Planejamento

O planejamento de uma *intranet* ou de um *site Web* sob a ótica organizacional consiste em definir metas relacionadas às necessidades da instituição que podem ser implantadas sobre estas tecnologias. Considere como exemplos uma instituição que atue na área de vendas e outra que empregue funcionários distribuídos geograficamente. Se os produtos da primeira puderem ser comercializados pela *Web,* pode-se implantar um *site* que inclua mecanismos de venda e suporte pós-venda; na segunda, se as atividades de seus funcionários exigirem cooperação, pode-se implantar uma *intranet* que ofereça serviços de *email* e publicação de informações. A idéia é empregar as vantagens das tecnologias como um diferencial que permita aprimorar a execução das atividades de negócio.

Considere o exemplo de especificação de uma *intranet:* "A *intranet* deverá implementar uma aplicação para o gerenciamento de memorandos, cartas, faxes e relatórios, com o objetivo de economizar 30% dos gastos atuais com impressão, distribuição e armazenamento de papel. Deverá incluir ainda um serviço de *email* para diminuir em 50% o número de ligações interurbanas para comunicação entre funcionários, clientes e parceiros." Neste caso, podemos notar os serviços concebidos, o contexto de sua utilização e os resultados esperados. A idéia é que, terminado o projeto de um *site Web* ou *intranet,* haja uma avaliação dos resultados concretos do projeto, sem se valer de expectativas subjetivas, que em geral são incomensuráveis.

2.3.1.1 Oportunidades

A definição das atividades da organização que devem ser incluídas numa *intranet* é um dos fatores determinantes do sucesso de sua implantação. Para que o planejamento identifique as atividades mais adequadas, deve ser realizado tanto pela equipe de profissionais técnicos, quanto pela equipe administrativa. Segundo [Usw98], as atividades organizacionais mais beneficiadas por uma *intranet* incluem:

- Produção, requisição, distribuição e atualização de informações dinâmicas, tradicionalmente publicadas em papel, como estatutos, procedimentos, manuais, catálogos e relatórios.
- Recuperação e processamento de informações de diversas origens. Considere como exemplo o atendimento de um cliente por um funcionário da área de suporte, que deve recuperar as informações sobre o cliente e o produto, além de pesquisar a base de dados de problemas e soluções, enquanto registra a ocorrência.

- Trabalho cooperativo, sobretudo entre usuários geograficamente dispersos. Cenários típicos desta natureza ocorrem durante o desenvolvimento de projetos por uma equipe distribuída e na prestação de contas de vendedores e representantes que trabalham fora da empresa, por exemplo.
- Aquelas já implementadas sobre uma arquitetura cliente/servidor (*mainframe*, por exemplo). Nestes casos, a implementação dos *front-ends* destes sistemas sobre uma *intranet*, ou a criação de aplicações integradas aos sistemas legados, permite preservar os investimentos já realizados, além de disponibilizá-los através da *Web*.

No contexto comercial, é possível implantar arquiteturas distribuídas de *intranets* que reflitam a estrutura das organizações, como é o caso da hierarquia matriz/filiais. O objetivo é disponibilizar os recursos da *intranet* em contextos locais, onde possam ser gerenciados pelos responsáveis diretos. Numa *intranet* distribuída entre matriz e filiais, o desenvolvimento e a manutenção são descentralizados, de forma que cada filial gerencie as informações sob sua responsabilidade. A mesma idéia tem sido empregada em *intranets* de empresas de maior porte, na descentralização do controle sobre o conteúdo de departamentos, projetos, produtos e eventos, por exemplo. Neste sentido, [Usw98] apresenta uma série de atividades de negócio, divididas por área funcional (departamentos), que podem se beneficiar com a implantação de uma *intranet*, conforme discutimos a seguir.

- **Recursos humanos**. A área de RH de uma organização é responsável pela publicação de uma variedade de informações sobre o relacionamento funcionário/empresa, além de coordenar contratações, demissões e treinamentos. Estas atividades envolvem a publicação de informações e formulários para requisições em papel, além de contatos telefônicos entre funcionários e departamentos. Uma *intranet* permite cortar custos com a distribuição de informações em papel, automatizar os processos de requisições através de aplicações e aprimorar a comunicação e a colaboração dos funcionários através do *email* e listas de discussão. Assim, um departamento de RH pode divulgar as informações (WWW) sobre o departamento, como os seus objetivos, procedimentos e métodos, endereços e responsáveis para contato, anúncios, calendários de atividades, além das informações específicas sobre RH, como as oportunidades de trabalho, formulários para requisição de férias e justificativa de faltas, descrições de cargos (responsabilidades, habilidades exigidas, benefícios), organogramas, informações para novos empregados, descrição dos benefícios (planos de saúde e previdência, planos de carreira e treinamentos), calendário de feriados, etc. De fato, os próprios sistemas de gestão empresarial, também conhecidos como sistemas ERP (*Enterprise Resource Planning*), já empregam soluções *Web* integradas às áreas de RH e finanças, por exemplo.
- ***Marketing* e Vendas**: Estes segmentos lidam com uma grande quantidade de informações sobre produtos/serviços tanto em papel, como apresentações e promoções de marketing, relatórios sobre resultados e previsões de vendas, quanto eletronicamente, através de sistemas legados de controle de estoque e pedidos, planejamento de produção, etc. Além disso, é comum haver interação constante entre equipes de vendas, *marketing*, suporte e produção. O emprego de uma *intranet* nestes segmentos permite aprimorar a comunicação e a colaboração entre as equipes encarregadas da divulgação e comercialização dos produtos e serviços, além de facilitar a elaboração de orçamentos, contratos e pedidos. Em geral, a seção de uma *intranet* que lida com *marketing* e vendas publica informações específicas do departamento, divulga (WWW) produtos/serviços (catálogos e especificações) e oferece grupos de discussão

(*email* e *news*) com clientes (para obtenção de *feedback*) e outros departamentos, como os de vendas, *marketing*, suporte e produção.

- **Suporte**: Nesta área é comum recuperar informações de diversas origens, como os dados cadastrais de clientes, as especificações de produtos/serviços e a base de dados sobre problemas e soluções. Mesmo que uma empresa já possua sistemas de informação para estes fins, a implementação destas aplicações sobre *intranets* permite disponibilizá-las diretamente aos clientes, parceiros, vendedores e revendedores através da Internet, desde que implantados os mecanismos de controle de acesso (autenticação). Além da resolução de problemas, o departamento de suporte pode prover informações sobre o *status* da entrega de produtos (*tracking*) e formulários para aquisição de dados (*feedback*). A seção de suporte de uma *intranet* pode incluir informações específicas sobre o departamento, divulgar (WWW) listas FAQ (*Frequently Asked Questions*) para funcionários e clientes na *intranet*/Internet, promoções, termos de garantia de produtos, formulários para pedidos de ajuda, *feedback* e atualização de dados cadastrais, serviços de *download* (correções, documentação), aplicações para consulta à base de dados de problemas e soluções, e de *workflow,* para o controle de requisições de clientes (*tracking*), e grupos de discussão (*email, news*) interdepartamentais entre **produção** e **pesquisa/desenvolvimento**.
- **Finanças**: Em geral, as informações da área financeira são manuseadas por sistemas de gestão proprietários. Essas informações são pesquisadas e processadas em relatórios de estatísticas de vendas, custos de produção, estoque e folha de pagamento, por exemplo, por vários usuários de diferentes departamentos. Desta forma, uma *intranet* pode ser integrada ao sistema de gestão existente para complementar suas funcionalidades e facilitar o acesso às suas informações. Além disso, é possível estender o acesso à *intranet* aos parceiros comerciais (fornecedores, revendedores, etc.) como iniciativa para o desenvolvimento de *extranets*. Assim, o segmento financeiro de uma *intranet* publica informações específicas sobre o departamento, divulga (WWW) relatórios, gráficos e indicadores financeiros, além de aplicações para controle de orçamentos e transações com parceiros (orçamentos, consultas, pedidos e pagamentos), que podem evoluir eventualmente para sistemas comerciais integrados do tipo *extranet*.
- **Pesquisa e desenvolvimento** de produtos/serviços: Este departamento lida com o gerenciamento de projetos, grupos de desenvolvimento interdepartamentais e recuperação de informações de diversas fontes (suporte e *marketing*). Desta forma, é fundamental oferecer recursos para a comunicação e colaboração entre as equipes de desenvolvimento e gerenciamento das informações relacionadas aos projetos, que são alteradas constantemente. Neste sentido, uma *intranet* pode publicar informações específicas sobre o departamento, além de divulgar (WWW) projetos (*status*, membros, agenda, atividades, relatórios, resultados), aplicações para gerenciamento de documentos (gerados pelos projetos) e grupos de discussão entre os departamentos de **produção**, **vendas**, **marketing**, **financeiro** e **suporte.**
- **Produção**: O departamento de produção controla aspectos relacionados à matéria-prima, sequenciamento de produção, estoque, controle de qualidade e armazenagem, além de colaborar com os departamentos financeiro (compra de matéria-prima) e de pesquisa e desenvolvimento (especificações, *feedback*). A colaboração entre uma empresa e seus parceiros pode ser aprimorada a partir de *extranets,* através de aplicações integradas com os fornecedores de matérias primas e os consumidores dos produtos (revendedores, clientes, etc.). Num ambiente de uma *extranet* pode-se implantar mecanismos mais elaborados de produção, controle de estoque e distribuição que diminuam os custos com armazenamento e

aumentem a capacidade de produção. Enfim, o setor de produção pode utilizar a *intranet* para divulgar (WWW) informações sobre peças, máquinas, sequenciamento de produção e controle de qualidade (especificações, estatísticas), aplicações para controle dos níveis de estoque, requisições de matéria e distribuição da produção, além de grupos de discussão entre os departamentos de **pesquisa e desenvolvimento**, **suporte** e **vendas**.

Como uma *intranet* pode conter vários serviços, aplicações e informações, é necessário definir a ordem de sua implantação, pois no processo de desenvolvimento de uma *intranet* os recursos são implantados de maneira independente, conforme a demanda da organização. Para determinar a ordem em que os recursos devem ser implantados, deve-se avaliar os seguintes critérios [Net98]: potencial de retorno do investimento[2], orçamento, recursos necessários, valor promocional, valor educacional, audiência, grau de complexidade e reusabilidade.

A idéia é empregar estes critérios na definição das aplicações e serviços prioritários, ou aqueles que oferecem uma combinação de alto retorno de investimento em curto prazo, orçamento reduzido, suporte as principais atividades de negócio e que sejam utilizados pelo maior número possível de usuários. Esta abordagem permite definir os recursos da *intranet* de acordo com as necessidades das organizações; projetos com orçamento reduzido, por exemplo, deverão implementar os serviços/aplicações que apresentarem a melhor opção de orçamento.

Embora a escolha adequada das atividades que serão incorporadas à *intranet* seja uma condição necessária para o sucesso do projeto, não é suficiente para garantir sua viabilidade econômica. Para tanto, é necessário calcular o retorno de investimento da *intranet*, que segundo [Cam97] deve ultrapassar os 20% para caracterizar um projeto economicamente viável.

2.3.1.2 Viabilidade Econômica

O cálculo do ROI pode ser realizado em duas circunstâncias: na etapa de planejamento da *intranet*, como forma de estimar o percentual de investimento a recuperar num determinado período e após a implantação, medindo os resultados obtidos e comparando-os às estimativas do planejamento, para que o projeto seja avaliado e reestruturado. A diferença entre os cálculos refere-se aos valores envolvidos, pois, enquanto o planejamento emprega estimativas, a avaliação dos resultados utiliza dados concretos.

O cálculo do ROI de *intranets* não é tarefa trivial, pois lida com uma série de fatores de difícil quantificação. Em primeiro lugar, deve-se conhecer as aplicações da *intranet* definidas na etapa de planejamento. Em seguida, faz-se um levantamento dos seguintes valores:

- A economia obtida com o uso da *intranet* sobre os processos e atividades convencionais, como corte de gastos com impressão, distribuição e armazenamento de informações em papel;
- O ganho de produtividade (lucro) atribuídos à *intranet*, como o fator aumento de produtividade;
- As despesas envolvidas no projeto, incluindo *hardware, software,* consultoria, treinamentos desenvolvimento de aplicações, gerenciamento do conteúdo, etc.

O corte de gastos é um dos fatores mais simples de se calcular (ou estimar) e refere-se aos custos envolvidos nas atividades substituídas por serviços e aplicações de uma *intranet*, como impressão, distribuição e atualização de documentos, manuais e formulários, comunicação entre

[2] Tradução de *return on investment* (ROI)

usuários, etc. A idéia é quantificar os custos destas atividades antes e depois da implantação da *intranet*; a diferença representa o corte de gastos.

Não raro, uma *intranet* oferece novas oportunidades para a organização, como é o caso da comunicação entre departamentos, escritórios e parceiros geograficamente dispersos. Nestes casos, deve-se estimar o custo da implantação destas novas funcionalidades sem o uso da *intranet* e computá-lo no montante de corte de gastos. Instituições multinacionais, por exemplo, podem se beneficiar devido às restrições impostas por fusos horários. Nestes casos, é preciso computar o aumento de produtividade correspondente às novas oportunidades.

Como uma *intranet* facilita o acesso à informação e melhora a comunicação e a colaboração entre seus usuários, é certo que há um aumento de produtividade. Alguns autores [Tel97], [Cam97], [Usw97], [Usw98], [Net98] apontam esta medida como a principal fonte dos benefícios de uma *intranet.* Entretanto, há também um consenso de que é a medida mais complexa de se calcular. A dificuldade reside em quantificar o aumento de produtividade de cada usuário, pois os benefícios estendem-se às várias atividades e processos de negócio incorporadas à *intranet.* Enquanto certas atividades são mensuráveis, como um *"aumento das vendas em 30%"*, outras dependem de um estudo mais detalhado, como a *"melhora na comunicação dos funcionários"*.

O cálculo do aumento de produtividade deve seguir algumas regras práticas:

- Deve-se realizar uma amostragem do tempo poupado por funcionários (para cada cargo) com o uso da *intranet.* Desta forma, pode-se calcular um valor homem/hora ou a produção excedente dos funcionários correspondente ao tempo ganho. Em geral, utilizam-se questionários e análises estatísticas da produção para esta finalidade.
- Faz-se os ajustes dos valores obtidos anteriormente, segundo o número de funcionários em cada setor, cargo ou departamento.

A última parcela envolvida no cálculo do ROI refere-se ao orçamento do projeto da *intranet.* Incluem-se neste cálculo os custos com *hardware, software,* treinamento, consultoria e manutenção. Embora uma *intranet* seja implantada sobre uma infra-estrutura existente, é necessária uma reavaliação dos recursos existentes (servidores, *links* de comunicação, serviços, protocolos, topologia da rede, etc.) e daqueles necessários à implantação dos serviços concebidos para o projeto.

Entre os recursos de *software* necessários à implantação de uma *intranet,* podemos destacar ferramentas de auxílio ao desenvolvimento de páginas (editores, gerenciadores de *sites*) e aplicações (compiladores), servidores (WWW, *email*, *news*, *proxy* e diretório) e navegadores (*browsers*). Quanto ao *hardware*, deve-se considerar a infra-estrutura de comunicação, como *links* Internet, servidores *Web*, servidores de bancos de dados para a *intranet*, e assim por diante. Como um dos principais serviços de uma *intranet* é a publicação de informações, há a necessidade de manter o seu conteúdo atualizado. Desta forma, deve-se considerar os custos com a conversão de documentos para o formato HTML, e manutenção das páginas existentes (conteúdo, *links,* local de publicação, *layout*, etc.) no orçamento de uma *intranet.* Por fim, podemos destacar a necessidade de pessoal especializado, que envolve o treinamento do pessoal interno, além da contratação de pessoal, como *webmasters* e consultorias, por exemplo.

Os componentes do cálculo do ROI (corte de gastos, ganho de produtividade e orçamento) são diretrizes suficientes para avaliar o resultado de uma *intranet* já implantada; basta utilizar questionários e planilhas de custos e receitas. Por outro lado, o cálculo realizado durante o planejamento da *intranet* para determinar a sua viabilidade econômica pode ser tão complexo

quanto o próprio projeto. Neste caso, sugestões [RM97] apontam um protótipo como forma de obter uma estimativa dos componentes do ROI.

Um protótipo de uma *intranet* é uma forma de obter retorno dos usuários, além de estimar os ganhos com o corte de gastos e aumento de produtividade, além dos custos com manutenção, gerenciamento, treinamento e suporte. A idéia é selecionar e incorporar alguns serviços e aplicações ao protótipo, como forma de estimar os esforços e os resultados envolvidos no projeto da *intranet*, necessários ao cálculo do ROI.

Outro componente envolvido no cálculo do ROI é o período de retorno[3], que corresponde ao tempo necessário para a recuperação dos gastos com o projeto (investimento). Segundo [Cam97], a planilha que representa os componentes do ROI é definida em termos de intervalos de tempo (mensal, trimestral, semestral, anual, etc.), e o período de retorno refere-se ao intervalo necessário à compensação do orçamento sobre os ganhos atribuídos ao projeto.

Uma vez definidos os propósitos da *intranet* e os cálculos (estimativas) do período e do retorno de investimento, pode-se decidir pela continuidade e/ou reestruturação do seu projeto. A revisão das oportunidades e a análise da viabilidade econômica servem para reafirmar os propósitos da *intranet* e são utilizadas pelas demais fases do projeto. Após identificar os componentes do ROI, é possível avaliá-los e até modificá-los para refletir uma reestruturação do projeto. Pode-se, por exemplo, alocar mais recursos para a inclusão de um novo serviço ou aplicação, aumentando os valores dos custos e benefícios relativos à modificação; ou ainda eliminar algum serviço para avaliar o impacto de seu custo sobre o ROI. Desta forma, é possível planejar a implantação da *intranet*, escolhendo os serviços mais relevantes, determinando a ordem em que são implantados e o orçamento reservado a cada aplicação/serviço.

A etapa de planejamento é, portanto, um meio de identificar as vantagens de uma *intranet* sob a ótica dos objetivos da organização. Em geral, um planejamento bem elaborado deve justificar a implantação de uma *intranet* através de alguns dos seguintes cenários [RM97]:

- A adoção da *intranet* é um meio de melhorar a imagem e a reputação da organização frente ao mercado altamente competitivo. Ao aprimorar a infra-estrutura de seus sistemas de informação, a organização oferece melhores recursos aos seus funcionários, parceiros e clientes.
- A *intranet* representa apenas o primeiro passo de uma iniciativa (plano) de aprimoramento das atividades da organização, e é utilizada como infra-estrutura para a execução do plano de reestruturação dos negócios.
- As atividades de negócio da organização são muito bem adequadas à *intranet,* pois já existe demanda por um sistema de informações deste tipo.

2.3.2 Projeto

O resultado do planejamento é uma pré-especificação dos recursos previstos para a *intranet,* que não contempla aspectos relacionados à implementação. A fase de projeto, por sua vez, envolve a especificação detalhada das aplicações e serviços a ser implantados, a criação de um plano de atividades para o projeto, a designação da equipe de desenvolvimento (participantes, responsabilidades) e a definição da infra-estrutura de *hardware* e de *software.*

Dentre os serviços mais comuns em *intranets,* podemos destacar os seguintes:

---

[3] Tradução de *payback period*

- **Publicação de informações (WWW).** É um dos principais serviços de uma *intranet,* pois constitui um meio econômico e eficiente para a publicação de informações sobre a organização, como a sua história, objetivos, produtos e serviços, procedimentos e métodos, anúncios, *homepages* de funcionários, organogramas, bibliotecas, classificados e *links* para outros *sites* relacionados (departamentos, projetos, filiais, parceiros, competidores, etc.). Um dos aspectos a considerar na especificação deste serviço refere-se à conversão das informações existentes para o formato HTML.
- **Comunicação** (*email*) **e colaboração** (*news*). Assim como na Internet, são as ferramentas mais utilizadas na *intranet.* O *email* é uma alternativa aos meios de comunicação tradicionais, como os telefonemas, memorandos e faxes. O *news* permite criar grupos de discussão de interesse comum, como uma equipe que desenvolve um projeto, um departamento etc. Caso a organização já possua um sistema de *email* proprietário, deve-se considerar a utilização dos padrões da Internet, como o SMTP, ou a implantação de *gateways* para preservar os sistemas de *mail* legados. Outro aspecto do *email* refere-se ao número de usuários; para as organizações que possuam mais de 5000 usuários, recomenda-se a criação de subdomínios para cada filial, departamento ou projeto [Net98], dado que facilita a atribuição de *usernames* e o dimensionamento dos servidores.
- ***Search engines,* índices e mapas.** São serviços que complementam a publicação (WWW), pois facilitam o acesso às informações através de mecanismos de busca, classificação e organização, respectivamente. Um *search engine* pode ser configurado para visitar e registrar as páginas de vários servidores WWW, associando-as a palavras-chave, sobre as quais são realizadas buscas. Um índice é similar a um catálogo telefônico ou páginas amarelas e permite classificar páginas segundo o contexto das suas informações. Um mapa é uma página *Web* que apresenta *links* para as principais informações de um servidor WWW sob a forma de uma hierarquia de classificação.
- **Replicação**. Uma *intranet* pode implantar servidores *proxy* para diminuir o tráfego externo (Internet), melhorar o tempo de resposta e aprimorar a segurança da rede interna. Um *proxy* permite replicar as informações de interesse comum e disponibilizá-las na *intranet*, além de replicar porções da *intranet* para outros *sites* (departamentos, filiais, parceiros etc.). Desta forma, a requisição de páginas externas que estejam disponíveis no *proxy* é atendida pela própria *intranet.*
- **Segurança.** Como uma *intranet* requer controle de acesso aos seus recursos, deve-se considerar a implementação de segurança através de filtros, *firewalls* e mecanismos de autenticação que utilizem chave pública e certificados digitais.
- **Diretório**. Um diretório é um sistema que centraliza informações de forma hierárquica, similar ao serviço de busca de nomes de domínios da Internet, o DNS (*Domain Name System*). A iniciativa de diversas empresas (Netscape, Banyan, Novell, IBM, AT&T etc.) em adotar uma proposta de padronização para a tecnologia de diretórios, LDAP (*Lightweight Directory Access Protocol*), tem acentuado ainda mais a tendência de seu uso tanto em *intranets,* como na Internet. LDAP é uma versão compacta do padrão X.500 que permite organizar informações de forma hierárquica, representando limites geográficos, institucionais, ou qualquer outra classificação arbitrária. Numa determinada hierarquia (um país, por exemplo), pode-se incluir empresas, pessoas, equipamentos, documentos, ou qualquer outro recurso que necessite de registro para consulta. Como um diretório pode centralizar informações sobre usuários (*username,* senha, grupo, direitos de acesso, certificados digitais

etc.) e aplicações (configurações, controle de acesso etc.), é comum ser utilizado na implantação dos mecanismos de segurança de uma *intranet*.

Em geral, uma *intranet* possui aplicações de acesso a banco de dados, como aquelas que recuperam dados do servidor de banco de dados (SGBD) e os apresentam em páginas HTML, aplicações integradas aos sistemas legados, ou aquelas que se comunicam com aplicações existentes para disponibilizá-las através da *Web*, e outras específicas, como é o caso de aplicações de *groupware* e *workflow*. Como há uma diversidade muito grande de tecnologias para a implementação de aplicações para a *Web,* a escolha das soluções mais adequadas depende da definição dos requisitos das aplicações da *intranet,* como acesso a bancos de dados, diretórios, sistemas legados, etc. A especificação das aplicações de uma *intranet*, assim como todo o processo de desenvolvimento, segue os modelos convencionais de construção de *software*; cada aplicação pode ser considerada um projeto de *software* separado com seu próprio ciclo de vida, processo de desenvolvimento, tecnologias de implementação, e assim por diante.

Outro aspecto do projeto é a definição da infra-estrutura da *intranet*, ou os recursos de *hardware* e *software* necessários a sua implantação. O *design* da infra-estrutura é feito a partir da avaliação e reestruturação dos recursos existentes, de forma a satisfazer a demanda das novas aplicações, serviços e usuários da *intranet*. A implantação de uma *intranet* pode degradar o desempenho da rede e de outros recursos existentes, pois gera um aumento do número de usuários, do tráfego da rede, dos requisitos de qualidade de serviço (disponibilidade, *throughput*) e do processamento nos servidores. Além disso, uma *intranet* pode exigir mecanismos de segurança mais aprimorados (*firewalls*), a segmentação de redes em sub-redes (departamentos, filiais, parceiros etc.) e a integração das redes existentes (*switches,* roteadores, *links* Internet). Portanto, a definição da infra-estrutura de uma *intranet* deve contemplar os seguintes fatores [Net98]:

- Topologia de rede, ou como estão distribuídos os recursos de rede (servidores, *links,* roteadores, *bridges,* repetidores, *gateways, firewalls,* etc.);
- Capacidade de processamento, armazenamento e largura de banda[4] dos clientes e servidores;
- Estatísticas de uso dos recursos de rede e dos servidores;
- Plataformas das máquinas clientes e servidoras, sistemas operacionais e protocolos de rede utilizados;
- Tipos de aplicações concebidas para a *intranet* e os padrões de sua utilização (quais os seus usuários, qual a demanda por largura de banda de cada uma, etc.);
- Perfil dos usuários (número, distribuição geográfica, aplicações utilizadas, etc.).

A idéia é avaliar as estatísticas de uso e capacidade dos recursos existentes e estimar o impacto da implantação de uma *intranet* sobre esta infra-estrutura. Desta forma, é possível planejar a arquitetura da *intranet* a partir da reestruturação dos recursos disponíveis, adequando-os aos novos serviços, aplicações e padrões de utilização. Uma *intranet* descentralizada, por exemplo, deve considerar a utilização de sub-redes com seus próprios servidores, além de prover mecanismos de integração com os servidores do *backbone* da *intranet*, como ilustra a Figura 2.1; já o cenário de uma *intranet* distribuída deve dispor de recursos adicionais para a integração dos *sites.*

A seguir, apresentamos algumas considerações sobre a arquitetura de projetos de *intranets*:

---

[4] Tradução de *network bandwidth*

1. Aplicações de missão crítica, ou ainda aquelas com uma demanda de acesso muito grande, devem ser implantadas em servidores separados (dedicados), pois facilitam a implantação, a manutenção e a obtenção de estatísticas de uso. Uma tendência é implantar o servidor de banco de dados (que além de ser utilizado por diversas aplicações, consome muitos recursos de CPU e memória) como um servidor dedicado.
2. Um servidor WWW médio (Pentium III 500, 256 MB RAM) pode atender por volta de 1.000.000 de conexões por dia [Net98]. Desta forma, é recomendável criar *mirrors* de servidores muito requisitados para balancear a carga de processamento ou ainda utilizá-los como servidores redundantes para os casos em que se deseja aumentar a confiabilidade do serviço. Outra tendência é utilizar diversos servidores WWW para descentralizar o controle e a distribuição de informações, criando-se servidores para cada departamento, filial, projeto ou produto, por exemplo. Esta política também pode ser estendida a outros tipos de serviços, como o *email, news,* diretório e banco de dados.
3. Para as *intranets* distribuídas geograficamente ou que ofereçam acesso a Internet é fortemente recomendado o uso de servidores *proxy,* que permitem interceptar as requisições dos usuários e implementar controle de acesso e *cache* das informações externas à *intranet.* Além de diminuir o tempo de resposta às requisições de páginas externas (Internet, *intranets* remotas) devido ao uso de *caches,* um servidor *proxy* também diminui o tráfego no *link* de comunicação externo. Nestes casos, ainda é recomendável adotar medidas de segurança para preservar a autenticidade e a privacidade das informações trocadas entre as *intranets*; um recurso muito comum empregado atualmente é o de VPNs (*Virtual Private Networks*) que permite estabelecer um canal seguro através da abordagem de PKI[5], mesmo que se empregue um canal inseguro como a Internet.

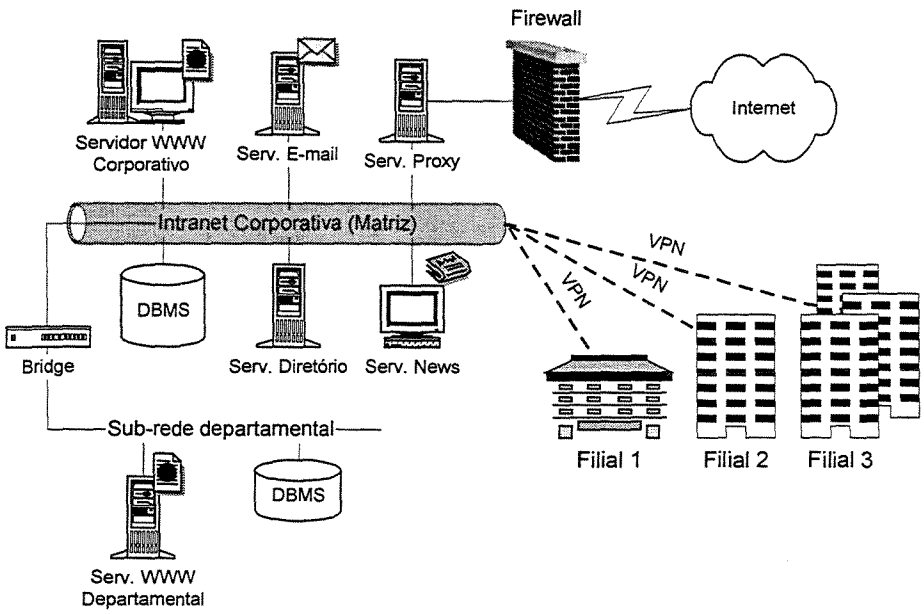


Figura 2.1 - Arquitetura de uma *intranet* distribuída

[5] *Public Key Infrastructure*, ou infra-estrutura de chave pública; abordagem que emprega um par de chaves pública e privada para a realização de criptografia.

Na *intranet* da Figura 2.1, por exemplo, podemos observar algumas das características discutidas anteriormente, como é o caso da descentralização do servidor WWW e da distribuição da *intranet* através do esquema matriz/filiais. Neste cenário, a *intranet* de cada filial é dimensionada de acordo com os requisitos locais da organização, como o número de usuários e as atividades envolvidas, da mesma forma que um projeto de uma *intranet* não distribuída; para uma filial pequena (até 50 usuários) pode-se implantar apenas os servidores *proxy* e WWW e integrá-la à *intranet* da matriz através de uma linha discada ou da própria Internet (Figura 2.3); uma filial de médio porte (50 a 500 usuários) por sua vez, deverá implantar mais recursos, como servidores de *email, news* e banco de dados; por fim, uma filial de grande porte (mais de 500 usuários) deverá implantar recursos similares à *intranet* representada pela matriz, como a descentralização através de sub-redes departamentais, uso de servidores redundantes, etc.

Da mesma forma que outros projetos relacionados à tecnologia da informação, a equipe de desenvolvimento de uma *intranet* envolve tanto papéis técnicos (administradores, analistas, programadores, *webmasters*), quanto não técnicos ou organizacionais (gerentes, autores, editores, publicadores). Ao contrário dos projetos de desenvolvimento de *software*, que em geral, dependem dos funcionários não técnicos somente na definição dos requisitos do sistema, uma *intranet* envolve papéis administrativos ou não técnicos durante todo o seu ciclo de vida. Em especial, vale ressaltar a importância de uma equipe centrada nas atividades de negócio da instituição durante as etapas de planejamento da *intranet*, investigando as oportunidades mais adequadas aos processos de negócio, e manutenção, contribuindo com a criação, revisão e reestruturação das informações distribuídas na *intranet*.

Segundo [Tel97], os papéis não técnicos envolvidos com a manutenção do conteúdo de uma *intranet* seguem a classificação hierárquica autor/editor/publicador, que permite sistematizar o ciclo de vida da informação na *intranet*. O autor é quem cria a informação e a disponibiliza de alguma forma; o editor gerencia as informações de um determinado assunto (ou área), determinando a relevância das informações de sua área de atuação para a publicação; os publicadores, por sua vez, são responsáveis por determinar quais as informações necessárias para os *sites* da *intranet* e da Internet e notificar esta demanda aos editores apropriados. Esta hierarquia representa os responsáveis pela concepção (publicador), organização (editor) e criação (autor) do conteúdo da *intranet*. Assim, um publicador do departamento de *marketing* poderá ser o próprio diretor (ou gerente) de *marketing*, e assim por diante.

Outro papel fundamental atribuído ao pessoal não técnico é a avaliação dos mecanismos já implantados na *intranet* e a identificação de novas atividades ou processos de negócio que podem ser implantados, pois é comum uma equipe técnica restringir-se apenas à manutenção dos recursos computacionais existentes.

Os papéis técnicos envolvidos no desenvolvimento de uma *intranet* incluem gerentes, que provêem direção ao projeto, administradores, que gerenciam *software* e *hardware*, e desenvolvedores de aplicações e informações (*webmasters*), que são responsáveis pela criação das páginas e aplicações. Um dos papéis mais importantes para uma instituição que mantém uma *intranet* e/ou um *site* Internet é o de *webmaster*, responsável pela manutenção das informações no âmbito técnico. Cabe ao *webmaster* facilitar a publicação de informações, liberando os usuários da *intranet* de atividades técnicas, como a conversão de documentos de formatos proprietários em HTML, a verificação da integridade das páginas e *links*, a aplicação de padrões de publicação (tipos, cores, menus, figuras) nas páginas publicadas, entre outras.

Além do *webmaster*, existem outros papéis difundidos pelas tecnologias Internet, como é o caso dos *webdesigners*. Atualmente, é comum visualizar páginas na Internet com aspecto gráfico

profissional, projetadas por *designers* gráficos e artistas plásticos e codificadas por *webmasters*. Assim como é comum empregar *designers* na criação de *sites* Internet, pois um dos parâmetros de audiência de um *site* é a qualidade de sua apresentação, deve ocorrer o mesmo com as *intranets,* onde a qualidade na diagramação das informações pode facilitar o uso e a disseminação da *intranet* entre seus usuários. Por ser uma área incipiente, é comum haver carência de mão-de-obra especializada interna. Nestes casos é fundamental que se proceda à contratação de pessoal especializado, consultorias e treinamentos.

2.3.3 Implementação

É na etapa de implementação que são desenvolvidas as aplicações e as informações de uma *intranet*. Para tanto, é necessário definir as tecnologias, as ferramentas de apoio e as estratégias empregadas na implementação.

A infra-estrutura para a criação e distribuição de informações oferecida pelo serviço WWW de uma *intranet* sugere um aumento na geração de informações, pois qualquer usuário pode contribuir como autor. Assim como ocorre com a Internet, o conteúdo de uma *intranet* pode experimentar uma quantidade excessiva de informações, e por conseguinte, apresentar dados imprecisos, inoportunos e irrelevantes, que comprometem a tarefa de recuperação de informações. Desta forma, é necessário sistematizar a publicação através de uma política de procedimentos que identifique a relevância e a precisão das informações, bem como o local de seu armazenamento e a forma de sua distribuição.

A política de procedimentos para publicação[6] deve abordar questões como os tipos de informações (relatórios, manuais, artigos, *homepages*) que podem ser incorporadas à *intranet,* quem pode contribuir com a criação de novas informações (todos os funcionários, somente os gerentes e diretores), quais os locais apropriados (área no *site, links*) para figuração das publicações, quais as formas de publicação (*push, pull,* formato e *layout*), quais as restrições de acesso aplicadas às informações, qual o procedimento padrão para a publicação, e assim por diante.

Há uma diversidade de informações que pode ser publicada numa *intranet,* de documentos oficiais (formais), como um relatório financeiro, a dados informais, como *homepages* de funcionários, notícias e artigos. Como um relatório financeiro é uma informação formal, deve ser publicado somente após a revisão e aprovação dos responsáveis pela área financeira, tomando-se as precauções necessárias com o controle de acesso. Dados informais, como notícias, artigos e *homepages* de funcionários, por sua vez, possuem um ciclo de vida diferenciado, menos restritivo que dados formais. Estes aspectos devem ser tratados pela política de procedimentos correspondente à publicação de informações na *intranet.*

Uma solução para o gerenciamento das informações de uma organização é apresentada em [Tel97], e consiste numa arquitetura lógica para a publicação de informações que emprega mapas, índices e *search engines* para a estruturação e a recuperação das informações (Figura 2.2). A divisão dos papéis envolvidos no ciclo de vida da informação é hierárquica, baseada na abstração autor/editor/publicador, apresentada na seção 2.3.2. Cada papel mantém um mapa das áreas em que atua com os *links* para o conteúdo correspondente. O mapa dos publicadores (Figura 2.2), por exemplo, representa cada área da organização com *links* para os seus respectivos conteúdos (áreas de negócio ou departamentos); o mapa dos editores contém informações ou *links* para informações sobre áreas específicas (projetos, produtos, serviços, etc.); o último nível

---

[6] Tradução de *policies and guidelines*

corresponde ao repositório de informações, onde os autores (gerenciados pelos editores) armazenam suas contribuições.

Esta arquitetura permite descentralizar o controle presente na política de procedimentos de publicação, delegando a responsabilidade do gerenciamento das informações em vários níveis (publicadores, editores e autores); cada nível, por sua vez, é segmentado entre as diversas áreas da organização, com publicadores, editores e autores para cada área (*marketing,* vendas, financeiro, recursos humanos, etc.). Assim, a hierarquia da área de *marketing,* por exemplo, fica incumbida de publicar e manter as informações sobre *marketing.* Um autor que queira contribuir com um documento de *marketing* deverá colocá-lo na área dos editores correspondentes; uma vez revisado e aprovado pelos editores, uma referência para o documento é colocada no mapa do publicador correspondente, que deverá incorporá-lo aos documentos da seção apropriada sobre *marketing*, disponível na *intranet.*

Na classificação hierárquica da arquitetura ilustrada na Figura 2.2 o índice e o catálogo são mecanismos que permitem simplificar e aprimorar a recuperação de informações. O mapa geral divide as informações em contextos bem definidos pela organização (filiais, departamentos, projetos, produtos, etc.); o catálogo agrupa as informações por assunto e o índice provê mecanismos de suporte a busca por palavras-chave através de *search engines.*

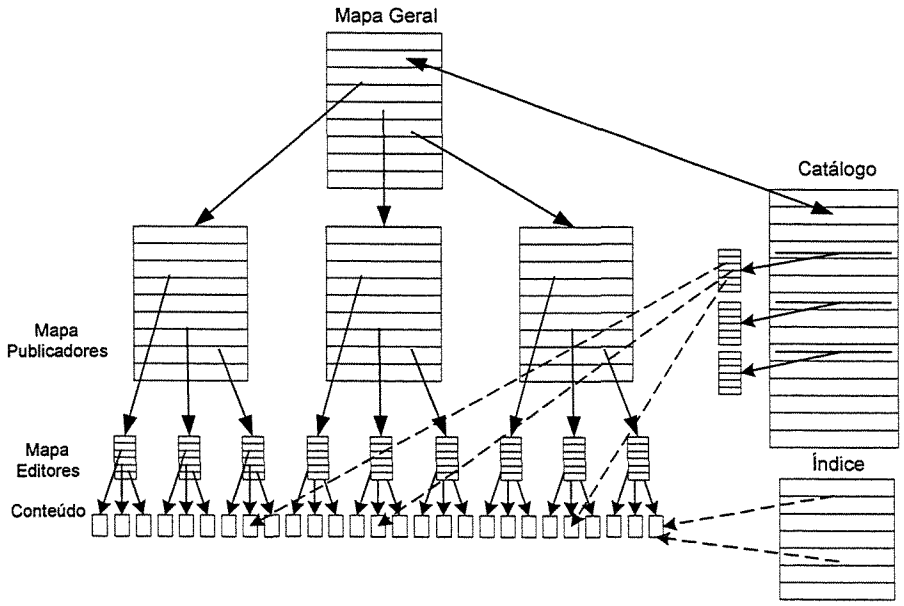


Figura 2.2 - Arquitetura lógica para a publicação de informações, adaptada de [Tel97]

Outro fator importante no processo de publicação é a definição da forma de distribuição: *push* ou *pull.* Em meios do tipo *push* a informação é endereçada ao usuário de forma passiva, ou seja, sem uma ação de requisição explícita (*email*, por exemplo); por outro lado, em meios do tipo *pull* o usuário requisita ativamente a informação desejada (WWW, por exemplo). A distinção destes meios é uma premissa para a construção de aplicações e serviços de publicação eficientes. Deve-se considerar meios do tipo *push* para a comunicação pessoal e informações com curto prazo de validade, e *pull* para aquelas sem restrição de tempo (informações recorrentes, publicações coletivas, etc.) [Tel97]. Uma prática comum é fazer uso de ambos os meios no processo de publicação, onde a informação é disponibilizada em um meio do tipo *pull* (WWW) e os usuários são alertados sobre a sua disponibilidade através de *push* (*email* com um *link* para a informação).

Um dos fatores preponderantes para a popularização das *intranets* é a padronização do formato das informações. Isto se deve aos elevados custos de manutenção de informações sob mídia eletrônica e formatos proprietários (Microsoft Word, por exemplo); a manutenção das informações de uma organização sob um formato padrão, independente do fornecedor do *software* para edição, diminui o custo de propriedade com informática.

A HTML (*HyperText Markup Language*), padrão para a representação das páginas de informação na WWW, oferece suporte a tabelas, imagens, *image maps, frames,* formulários e até estilos de formatação (*style sheets*). Além dos recursos do padrão HTML, os próprios fornecedores de navegadores têm incluído mecanismos que permitem explorar as funcionalidades do navegador em que a página é visualizada. É o caso das linguagens de *script,* por exemplo, que permitem executar programas embutidos em páginas HTML localmente (no próprio navegador), e do HTML dinâmico, que foi recentemente incorporado ao padrão HTML. Apesar dos *browsers* implementarem extensões do HTML não padronizadas pelo W3C, como foi o caso do DHTML, há esforços tanto no sentido de padronizar estes recursos, quanto em criar um novo padrão, a XML (*eXtended Markup Language*), que deverá suportar a incorporação de novos recursos. XML é apresentado em mais detalhes na seção 3.4.4 do capítulo 3.

Embora o uso de tecnologias proprietárias não seja recomendado na construção de páginas para *sites* Internet, pois a princípio, não se conhece os navegadores dos clientes, é comum utilizá-las nas páginas de uma *intranet.* O fato é que numa *intranet* pode-se estabelecer um navegador ou um conjunto de navegadores padrão, pois num contexto privado há um controle sobre a base de máquinas clientes instaladas, e por conseguinte, dos navegadores utilizados. Ainda assim, é recomendável racionar a utilização de tecnologias proprietárias, pois as incompatibilidades entre navegadores são comuns, como pode ser observado nos navegadores da Microsoft e da Netscape, que são os mais utilizados atualmente.

A necessidade crescente de distribuir informações em HTML fez surgir um novo segmento de ferramentas de *software*: a de gerenciamento de *sites*, que em geral, oferecem suporte à edição de páginas HTML, conversão de documentos de formatos proprietários para HTML, edição cooperativa, publicação em servidores WWW remotos, representação da hierarquia de páginas e *links,* entre outras funcionalidades.

Depois da publicação de informações institucionais, as aplicações cliente/servidor *Web* são os serviços mais comuns em *intranets*. A construção de aplicações cliente/servidor *Web* envolve desde a adoção de uma metodologia de suporte ao desenvolvimento e a escolha das tecnologias de implementação mais adequadas, até a definição de como as tecnologias são combinadas para formar as aplicações. Como o enfoque deste trabalho está centrado no desenvolvimento de sistemas cliente/servidor *Web*, apresentamos um estudo detalhado sobre estas questões nos capítulos 3 e 4.

2.3.4 Implantação e Gerenciamento

Durante a etapa de implantação, a infra-estrutura de *hardware* é instalada e/ou atualizada e as aplicações e serviços são instalados, configurados e testados. Como a etapa de implementação envolve testes das aplicações e informações desenvolvidas, a implantação dos serviços da *intranet* pode ocorrer em paralelo à implementação.

Na *intranet* da Figura 2.3, sugerida em [Net98] e apresentada na etapa de projeto, podemos notar uma arquitetura distribuída através da abordagem matriz/filiais; os detalhes da *intranet* da matriz foram omitidos, pois estão representados na Figura 2.1, enquanto que as filiais ilustradas

na Figura 2.3 correspondem aos detalhes das filiais da Figura 2.1. Neste cenário, é comum haver uma política de implantação que indique quais os recursos que devem ser instalados primeiro, quais as informações disponibilizadas entre as *intranets*, quais os mecanismos para a interconexão entre as *intranets*, e assim por diante.

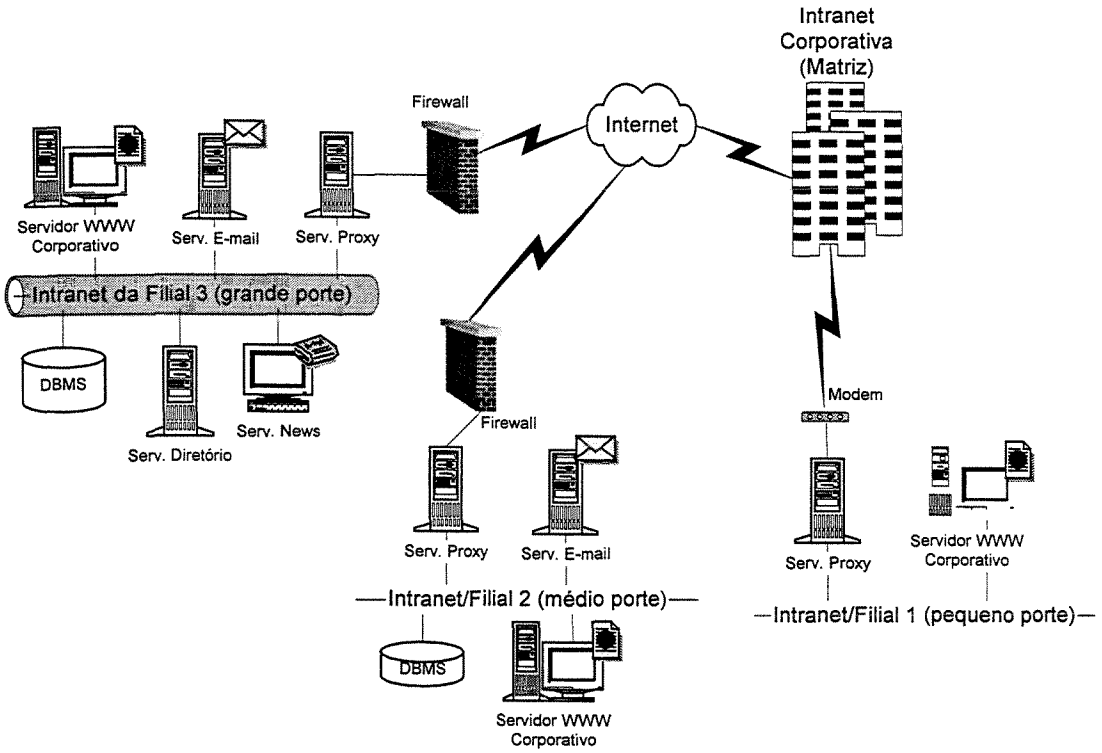


Figura 2.3 - Arquitetura detalhada de uma *intranet* distribuída

Segundo [Net98], pode-se adotar a seguinte estratégia para implantação de uma *intranet* distribuída, como a ilustrada na Figura 2.3:

1. Implantar a *intranet* da matriz, instalando primeiro as aplicações e serviços genéricos, e em seguida os mais específicos, como os servidores departamentais. Nesta etapa, são disponibilizadas as informações e as aplicações relativas à organização como um todo, no contexto da matriz.
2. Iniciar a implantação das *intranets* filiais. A princípio, a única exigência para tornar uma *intranet* remota operacional é fornecer uma infra-estrutura de conexão à *intranet* da matriz e servidores *proxy* em ambos os lados. Desta forma, pode-se utilizar os serviços e aplicações da *intranet* matriz a partir da *intranet* filial. Em seguida, disponibiliza-se as aplicações e informações da filial, reflexo do processo de descentralização do conteúdo da *intranet*. Neste ponto, os usuários da *intranet* filial possuem acesso ao conteúdo da filial e da matriz. Esta etapa é executada para todas as filiais.
3. Implanta-se os mecanismos de interconexão das *intranets*. Em alguns casos, como na *intranet* da filial 1, em que o tráfego não justifica o uso de uma linha dedicada para ligação com a *intranet* global, pode-se adotar linhas discadas ou o acesso periódico à Internet. Caso seja necessário compartilhar o conteúdo das *intranets*, deve-se adotar mecanismos de replicação de conteúdo e interconexão de *intranets*. A alternativa mais comum é replicar o conteúdo das *intranets* filiais, que devem estar disponíveis às demais *intranets*, na *intranet* da

matriz; neste caso, a interconexão é feita a partir da ligação de cada filial a matriz. Cada filial tem acesso ao conteúdo publicado pelas demais filiais, além do conteúdo da própria matriz. Esta abordagem pode ser implantada a partir da configuração adequada dos servidores *proxy* em cada *intranet.*

Após ser implantada, a *intranet* deve ser divulgada aos seus usuários, através de demonstrações, treinamentos, documentação e suporte adequados. Como o conteúdo da *intranet* pode incorporar informações e atividades de negócio inéditos, indisponíveis nos sistemas de informação existentes, é comum ocorrer um aumento do número dos usuários. O uso da *intranet* como ferramenta de trabalho constitui uma mudança nos hábitos dos funcionários e na cultura da organização, e deve, portanto, ser bem assistido. Desta forma, é imprescindível promover a *intranet* junto à comunidade de usuários [Tel97][Net98][Usw97][Usw98], demonstrando seu potencial, suas oportunidades e seus benefícios.

Por fim, há a etapa de gerenciamento da *intranet.* A natureza das *intranets* é inerentemente dinâmica, pois seu conteúdo é modificado frequentemente. Neste cenário, é imprescindível que se realize um gerenciamento contínuo dos recursos implantados. Além das atividades de gerenciamento de rede convencionais, como instalação, configuração, *backup*, manutenção e monitoração dos recursos de *hardware* e *software*, uma *intranet* depende de uma série de atividades de gerenciamento adicionais:

- Controle de acesso a diretórios, páginas, aplicações, servidores etc., para manter a privacidade do conteúdo da *intranet*;
- Gerenciamento das informações publicadas, para garantir a disponibilidade e a facilidade de acesso às informações, como é o caso da manutenção da integridade das páginas (*links,* por exemplo);
- Monitoramento dos serviços e aplicações implantados, avaliação das estatísticas de uso (requisições/*hits*) e reestruturação dos recursos disponibilizados. Nesta etapa pode-se detectar e eliminar os gargalos do sistema, por exemplo;
- Manutenção de *proxies,* diretórios, e outros servidores específicos;
- Suporte aos usuários.

2.4 Conclusões

As *intranets* têm despontado como plataforma ideal para a distribuição de informações e aplicações cliente/servidor institucionais. Na realidade, as organizações têm implantado *intranets* em vários níveis, da publicação de informações à implantação de aplicações cliente/servidor *Web* integradas a sistemas legados.

Como as *intranets* são construídas a partir das tecnologias Internet, há uma série de possibilidades de implementação a avaliar, e como envolve uma série de tecnologias incipientes, também há diversas oportunidades e consequências a considerar. Neste sentido, apresentamos um estudo sobre *intranets,* que destaca as etapas, as tecnologias, os recursos humanos e as decisões envolvidas no seu processo de desenvolvimento.

Capítulo 3

Arquitetura de *Software Web*

A popularização da Internet, sobretudo no meio comercial, contribuiu para o surgimento de diversas novas tecnologias de suporte ao desenvolvimento de aplicações *Web*[7]. Porém, podemos afirmar que o alto número de tecnologias disponíveis para implementação, a complexidade de cada uma, além do esforço despendido na sua combinação têm se tornado um problema na construção de aplicações.

Segundo [BRJ98b], o processo de desenvolvimento de *software*, *Web* ou não, é baseado em três paradigmas: *use-case driven, architecture-centric* e *iterative and incremental.* Em outras palavras, um sistema deve ser desenvolvido a partir da especificação dos seus casos de uso, deve ser implementado sobre uma arquitetura bem definida, além de seguir a abordagem iterativa e incremental, através da qual o sistema é desenvolvido em etapas. Enquanto os paradigmas de casos de uso e desenvolvimento iterativo e incremental lidam com os aspectos funcionais do sistema e com o processo de seu desenvolvimento, a arquitetura de *software* lida com os aspectos técnicos e não funcionais sobre os quais o sistema é desenvolvido.

A idéia do conceito de arquitetura de *software* é especificar os aspectos não funcionais de um sistema, como quais as tecnologias de implementação e integração cliente/servidor (*middleware*) mais adequadas, quais as funcionalidades e subsistemas que podem ser desenvolvidos ou adquiridos para promover a reutilização de *software*, quais as decisões de projeto que podem influenciar no desempenho, disponibilidade, escalabilidade, reutilização, portabilidade e integração da aplicação com sistemas legados, e assim por diante. Uma arquitetura funciona como um modelo, que em conjunto com as especificações de caso de uso, guia o desenvolvimento do sistema.

Como o foco deste trabalho situa-se no desenvolvimento de *software* para *Web*, procuramos analisar as arquiteturas de *software* sob a abordagem *Web*. Neste sentido, este capítulo apresenta

[7] Para efeito de simplicidade, denominamos *software* Web ou aplicação Web qualquer sistema que tenha sido projetado para funcionar sobre a infra-estrutura e tecnologias da Internet

um estudo comparativo das principais tecnologias para implementação de sistemas *Web*, enumera as arquiteturas de *software Web* mais comuns, e discute quando utilizar cada tecnologia e arquitetura.

3.1 Arquitetura de *Software*

Uma arquitetura de *software* é um artefato que contempla o desenvolvimento do sistema sob uma perspectiva técnica. Um arquiteto de *software* foca tanto os elementos estruturais do sistema, como os subsistemas, classes, componentes e infra-estrutura de implantação, quanto as colaborações que ocorrem entre estes elementos [BRJ98b]. A idéia sob a arquitetura é oferecer à comunidade de desenvolvimento (desenvolvedores, gerentes, testadores, clientes, usuários e parceiros) um modelo de especificação dos aspectos técnicos e não funcionais do sistema, que em conjunto com os aspectos funcionais (casos de uso, por exemplo), guiam o desenvolvimento do projeto.

O objetivo de desenvolver *software* sob o paradigma de arquitetura é promover o entendimento do sistema, organizar o seu desenvolvimento, aplicar políticas de reutilização e prepará-lo para a evolução e as mudanças futuras [BRJ98b]. Entender o sistema consiste em conhecer os diversos fatores que devem ser considerados no seu desenvolvimento; em sistemas *Web*, por exemplo, deve-se conhecer os ambientes de desenvolvimento e produção, as tecnologias sobre as quais são construídos, os esquemas de distribuição de processamento, os mecanismos de integração com sistemas legados (bancos de dados, sistemas de gestão, etc.), e assim por diante. A arquitetura de um sistema *Web* pode contribuir para o entendimento destas questões técnicas através da especificação dos mecanismos de distribuição, integração e implementação mais adequados no contexto das aplicações *Web*.

Em termos de organização do desenvolvimento, podemos destacar a comunicação entre os envolvidos num projeto. Segundo [BRJ98b], quanto maior e mais complexo o sistema, maior a dificuldade de comunicação, e portanto de organização do seu desenvolvimento. Como uma arquitetura funciona como um modelo para implementação, promove a comunicação e contribui para a organização do desenvolvimento. A construção de sistemas através de equipes dispersas, ou mesmo a partir da contratação de terceiros para o desenvolvimento de partes do sistema são alguns cenários que podem se beneficiar da utilização de uma arquitetura.

Outro aspecto que uma arquitetura deve considerar é o de reutilização de *software*. A idéia é definir a arquitetura sob a abordagem de componentes, de forma que o *design* do sistema considere as funcionalidades recorrentes e potencialmente reutilizáveis como componentes auto-contidos que podem ser reutilizados em outros projetos. Conceber sistemas através de uma arquitetura baseada em componentes facilita a divisão do seu desenvolvimento em subsistemas que podem tanto ser construídos quanto adquiridos. Como o conceito de componentes de *software* é largamente empregado neste trabalho, apresentamos um estudo detalhado sobre o tema no capítulo 4.

Por fim, a arquitetura de um sistema deve ser flexível o suficiente para suportar a evolução e as atualizações do sistema, de forma que seja possível executá-las sem modificar a estrutura da aplicação. Em outras palavras, um sistema construído sobre uma arquitetura flexível deve evoluir sem provocar impactos significativos no *design* e implementação atuais.

3.2 Padrões de Projeto e de Arquitetura

Um dos fatores mais importantes na definição de uma arquitetura de *software* refere-se às experiências anteriores e aos padrões já estabelecidos. Assim como ocorre na implementação de *software*, mecanismos bem conhecidos, testados e documentados tendem a ser reutilizados através de diferentes projetos. O conceito de reutilizar soluções bem sucedidas em problemas recorrentes foi introduzido por Christopher Alexander em projetos de arquitetura, e recentemente absorvida para a disciplina de engenharia de *software* como tratam [GHJ94] e [Pre94].

O conceito de padrões tem sido bastante desenvolvido, e atualmente estende-se desde as áreas de implementação, projeto e arquitetura até os aspectos organizacionais de gerenciamento de projetos. Os padrões relacionados à implementação, por exemplo, especificam normas de codificação e como estruturas de dados podem ser mapeadas em diferentes linguagens. Quanto aos padrões de projeto (ou *design patterns*), há uma série de catálogos que documentam decisões de projeto no sentido de construir sistemas flexíveis, configuráveis e extensíveis. Os padrões organizacionais documentam situações recorrentes no gerenciamento de projetos, como a administração de recursos para diminuir/eliminar os riscos em projetos de *software*. Por fim, existem alguns catálogos de padrões de arquitetura de *software* que identificam como dividir uma aplicação em subsistemas e camadas, além da forma como estes subsistemas e camadas podem ser distribuídos e integrados.

Neste projeto empregamos alguns padrões de implementação e de arquitetura, além de diversos padrões de projeto. Os padrões de projeto empregados neste trabalho são apresentados nos capítulos 5 e 6, e focam na maioria dos casos, nos aspectos de desacoplamento da lógica da aplicação dos detalhes de infra-estrutura de implementação. Os padrões de arquitetura mais comuns em sistemas *Web* são aqueles que tratam da divisão dos sistemas em camadas, da construção de sistemas distribuídos, e do desacoplamento entre as camadas de aplicação e de apresentação.

3.2 Arquiteturas Web

Algumas arquiteturas focam na distribuição física dos componentes de um sistema, como é o caso dos paradigmas cliente/servidor de duas camadas e cliente/servidor *Web*, que pode conter três ou mais camadas. Por outro lado, há arquiteturas que focam na distribuição lógica dos sistemas, como é o caso das arquiteturas de camadas, objetos distribuídos e MVC[8]. A arquitetura de sistemas *Web* emprega a combinação destas arquiteturas, pois deve especificar qual a estrutura de implantação dos componentes do sistema, ou em que parte (cliente, servidor) cada componente irá executar, além de indicar quais as camadas, subsistemas, componentes e serviços que compõem o sistema e como estes elementos interoperam durante a execução.

3.2.1 Cliente/Servidor Web

O paradigma de aplicações *Web* refere-se às aplicações baseadas no modelo cliente/servidor, onde o código da porção cliente reside em servidores WWW e é acessado através de um navegador. Como a parte cliente reside no servidor é possível atualizá-la de forma transparente para os clientes, que não necessitam realizar nenhuma atividade de manutenção e atualização de versões. Outra grande vantagem de aplicações *Web* refere-se ao encapsulamento dos detalhes de implementação da porção servidor, que pode inclusive ser construída sobre os recursos já

[8] *Model, View, Controller*, conceito definido por Smalltalk

existentes (bancos de dados, sistemas de gestão, etc.) para realizar o processamento e apresentar os resultados sob os padrões da Internet.

Em geral, as aplicações baseadas no modelo cliente/servidor tradicional são construídas em duas camadas. Neste modelo, conhecido como *fat client,* o cliente embute tanto a lógica de apresentação (interface gráfica), quanto a lógica de processamento e comunicação com a segunda camada, que normalmente realiza a persistência das informações. No modelo cliente/servidor *Web*, conhecido como *thin client,* o cliente embute apenas a lógica de apresentação, enquanto o processamento fica a cargo da porção servidor, que pode ser implementada através de uma ou mais camadas. Segundo [OH98], o particionamento das aplicações em três ou mais camadas (Figura 3.1) oferece uma série de vantagens sobre o modelo cliente/servidor tradicional sob os aspectos de escalabilidade, confiabilidade, facilidade de manutenção, reusabilidade e flexibilidade, como apresentado na Tabela 3.1.

	Cliente/Servidor Web	**Cliente/Servidor tradicional**
Número de clientes por aplicação	Milhares	Centenas
Número de servidores por aplicação	Alto	Baixo
Geografia	Global	Local
Número de camadas da arquitetura	3 ou mais	2
Interações servidor/servidor	Sim	Não
Middleware	ORBs sobre a Internet	SQL e *stored procedures*
Arquitetura	Padrões abertos	Proprietária
Aplicações multiplataforma	Sim, interface universal	Não
Custo de desenvolvimento	Médio/baixo	Alto

Tabela 3.1 - Comparação entre sistemas cliente/servidor tradicionais e *Web* [OH98]

O diagrama da Figura 3.1 apresenta a estrutura da arquitetura cliente/servidor *Web*, na qual podemos destacar:

- A porção cliente que roda sob um *browser* que interpreta a linguagem de apresentação (XML, HTML, WML, *applets*, ActiveX) e é responsável pela apresentação da interface e controle da entrada dos dados dos usuários;
- A comunicação entre as partes cliente/servidor através dos protocolos padrão da Internet (HTTP, IIOP, RMI);
- Os componentes que implementam a lógica da aplicação (páginas dinâmicas JSP/ASP, documentos HTML/XML, *scripts* CGI/*servlets* e objetos/componentes distribuídos);
- Os servidores que oferecem infra-estrutura para a execução da parte servidor do sistema, como servidores de aplicação, ORBs e servidores *Web*;
- Os sistemas legados sobre os quais são construídas as aplicações (bancos de dados, sistemas de gestão, *mainframe*, etc.)

- A comunicação servidor/servidor através das tecnologias de objetos distribuídos, monitores transacionais e mensagens.

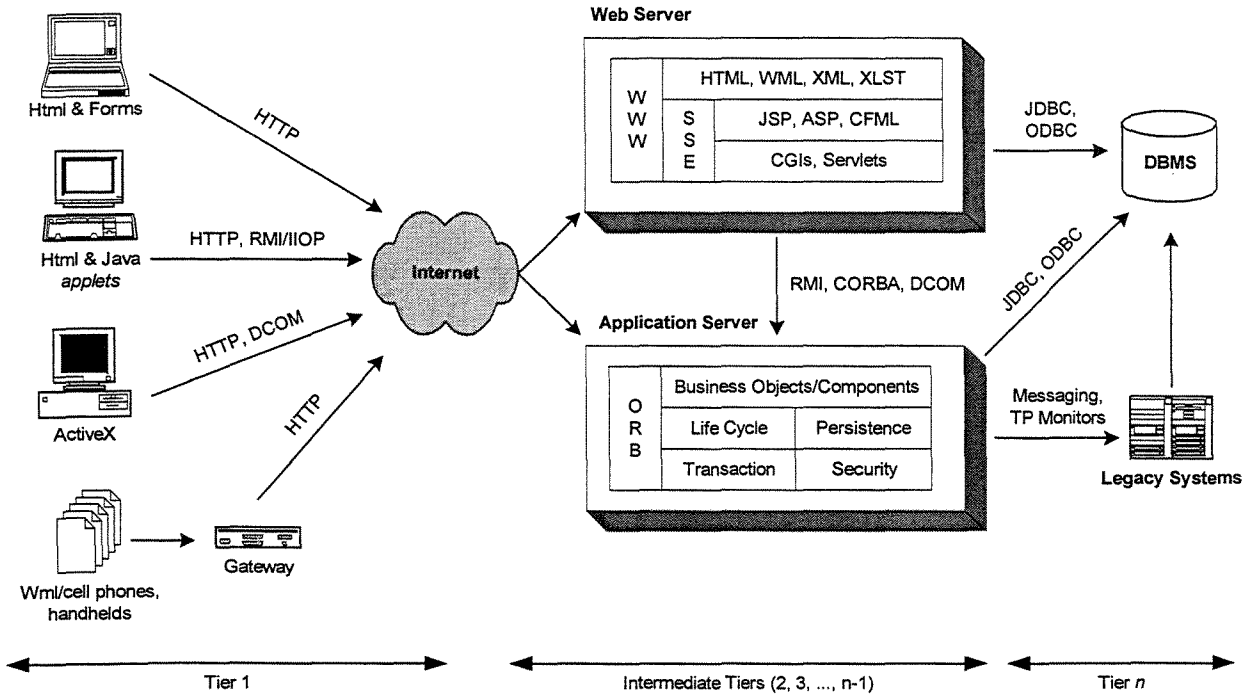


Figura 3.1 - Arquitetura cliente/servidor *Web*, adaptada de [OH98]

Vale ressaltar que as camadas ilustradas no modelo da Figura 3.1 representam serviços que podem rodar em diferentes máquinas, plataformas e redes. Apesar do exemplo ilustrar a utilização de um servidor *Web* separado do servidor de aplicação, podem ocorrer casos em que ambos compartilham a mesma máquina, ou ainda outros em que há vários servidores *Web* e de aplicação. Esta estrutura permite criar *clusters* de servidores para promover desempenho, escalabilidade, tolerância a falhas e reutilização de serviços (objetos e componentes de negócio).

3.2.2 Arquitetura baseada em Camadas

O padrão de arquitetura baseada em camadas permite organizar um sistema através de uma hierarquia de componentes, onde cada nível da hierarquia implementa um conjunto de funcionalidades da aplicação. O diagrama da Figura 3.2 apresenta a estrutura desta arquitetura, onde podemos verificar que cada camada da hierarquia presta serviços para a camada imediatamente superior. A divisão em subsistemas e o *design* de camadas que só podem utilizar os serviços das camadas imediatamente inferiores permitem diminuir a dependência entre as camadas e identificar funcionalidades recorrentes que podem ser construídas ou compradas sob a abordagem de componentes de *software*.

A abordagem da hierarquia de camadas faz com que os subsistemas das camadas inferiores sejam independentes dos demais componentes do sistema situados nas camadas superiores. Na prática, podemos afirmar que os componentes das camadas inferiores são mais estáveis, pois não acompanham o mesmo ritmo de evolução e atualização das camadas que implementam as rotinas de negócio da aplicação. Neste sentido, é comum implementar os subsistemas das camadas inferiores como componentes que podem ser reutilizados através de uma interface bem definida.

Como as modificações nestes componentes, e portanto, nas suas interfaces não são constantes, é possível isolar as modificações mais frequentes do sistema nas camadas superiores.

Na hierarquia de camadas da arquitetura da Figura 3.2, quanto menor o nível da camada, mais genéricos e independentes serão os subistemas que a compõem. Da mesma forma, quanto maior o nível da camada, mais específicos e dependentes da lógica da aplicação serão os subsistemas. Quanto mais genérico for um subsistema, maior a possibilidade de reutilizá-lo em outras aplicações.

A primeira camada da Figura 3.2 corresponde aos módulos que realizam os casos de uso do sistema; porém, como o sistema é construído sobre os serviços da segunda camada, estes módulos implementam apenas as rotinas específicas da aplicação. A segunda camada é composta por *frameworks*, componentes e bibliotecas de classes que são genéricas para o sistema, mas são dependentes do contexto da aplicação, isto é, podem ser reutilizados em outras aplicações do mesmo domínio. As camadas inferiores concentram os serviços de infra-estrutura, como suporte à distribuição de objetos, persistência de informações, serviços de transação e segurança, e serviços de base, como uma máquina virtual Java para rodar sistemas baseados em Java, servidores de bancos de dados, servidores de aplicações, servidores *Web*, e assim por diante.

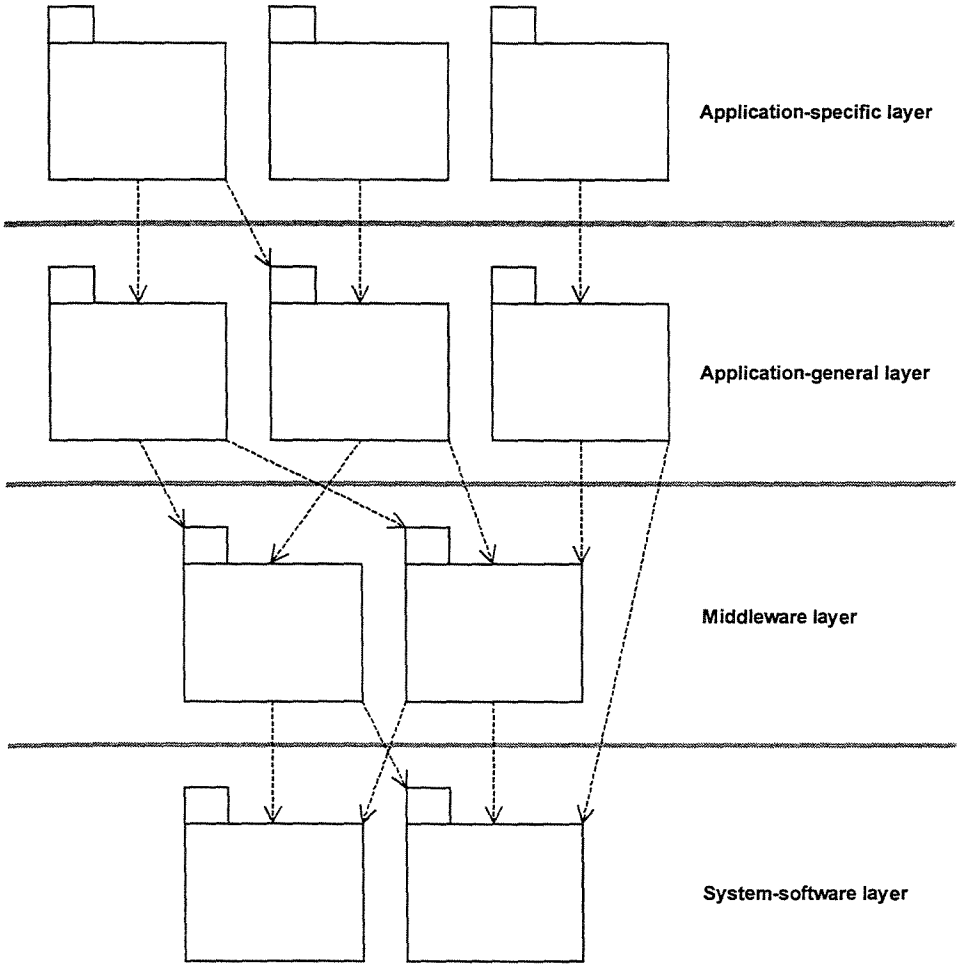


Figura 3.2 - Arquitetura baseada em camadas [BRJ98b]

A arquitetura de camadas pode ser combinada com outros padrões de arquitetura, como é o caso do exemplo apresentado na Figura 3.1, em que se emprega a abordagem de separação dos

componentes da lógica de apresentação (documentos HTML/XML, CGI/*servlets* e JSP/ASP), dos objetos/componentes de negócio e dos serviços de infra-estrutura (ORB, segurança, transação, persistência).

No cenário da Figura 3.1, por exemplo, é possível atualizar a lógica dos componentes e objetos de negócio sem prejudicar os componentes da camada de apresentação, desde que as interfaces dos componentes permaneçam inalteradas. Uma vez definidas as interfaces entre as camadas, é possível dividir o desenvolvimento entre equipes distintas - no exemplo da Figura 3.1, significa que enquanto *designers* e *webmasters* trabalham na codificação das páginas da camada de apresentação, engenheiros de *software* e programadores implementam os componentes da camada de aplicação.

3.2.3 Arquitetura de Apresentação (MVC)

O padrão de arquitetura MVC (*Model, View, Controller*) foi definido pela linguagem *Smalltalk* para guiar o desenvolvimento das interfaces gráficas das aplicações, e consiste em desacoplar a lógica de apresentação e de controle das informações manipuladas pelas interfaces destes sistemas. Como MVC especifica uma divisão entre informação e controle, é possível projetar aplicações que apresentam interfaces diferentes, embora utilizem os mesmos dados e a mesma lógica de controle. No contexto da *Web*, significa utilizar os mesmos dados e controle para gerar páginas de diferentes formatos, que podem por sua vez, ser utilizados por diferentes categorias de clientes.

Atualmente, há uma série de clientes com diferentes perfis de visão, que estendem-se de *desktops* com *browsers* HTML, até telefones celulares com *browsers* WAP e *handhelds* com *browsers* HTML/WAP. Não raro, há incompatibilidades entre *browsers* da mesma categoria, como se pode observar entre as implementações dos *browsers* para *desktop* Netscape Navigator e Microsoft Explorer, e para telefones celulares da Nokia e Phone.com. Além disso, a tendência aponta para um aumento no número de diferentes categorias de clientes, de *browsers* que as suportem, e por conseguinte, das incompatibilidades entre os *browsers*.

A separação dos sistemas em camadas como apresentado nos exemplos da Figura 3.1 e Figura 3.2 permite distribuir o desenvolvimento entre diferentes equipes e perfis. Enquanto os *designers* e *webmasters* implementam a camada de apresentação (visão e modelo da arquitetura MVC), os engenheiros de *software* e programadores podem implementar os componentes da camada de aplicação e as interfaces entre as camadas de apresentação e aplicação (controlador da arquitetura MVC) [Paw00]. Para que este cenário seja possível, é necessário desacoplar a visão e o modelo do controle da camada de apresentação.

Desta forma, podemos afirmar que MVC aplica-se muito bem à construção da camada de apresentação de sistemas cliente/servidor *Web* (Figura 3.1). O padrão de arquitetura MVC no contexto *Web* é apresentado na Figura 3.3 e pode ser descrito como segue:

- Modelo, que corresponde aos dados manipulados pela aplicação; podem ser representados em XML, HTML, WML ou classes de dados;
- Visão, que corresponde ao *browser* da porção cliente, é o módulo responsável por interpretar a linguagem de representação, formatar as informações e controlar a entrada de dados dos usuários;

- Controlador: corresponde tanto às páginas dinâmicas JSP[9], ASP[10] e CFML[11], quanto aos *scripts* que rodam no servidor, como *servlets* e CGIs; é responsável por manter as visões atualizadas sempre que ocorre alguma mudança no modelo, e por gerenciar as requisições dos usuários, através da utilização dos serviços da camada de aplicação e atualização do modelo e das visões correspondentes.

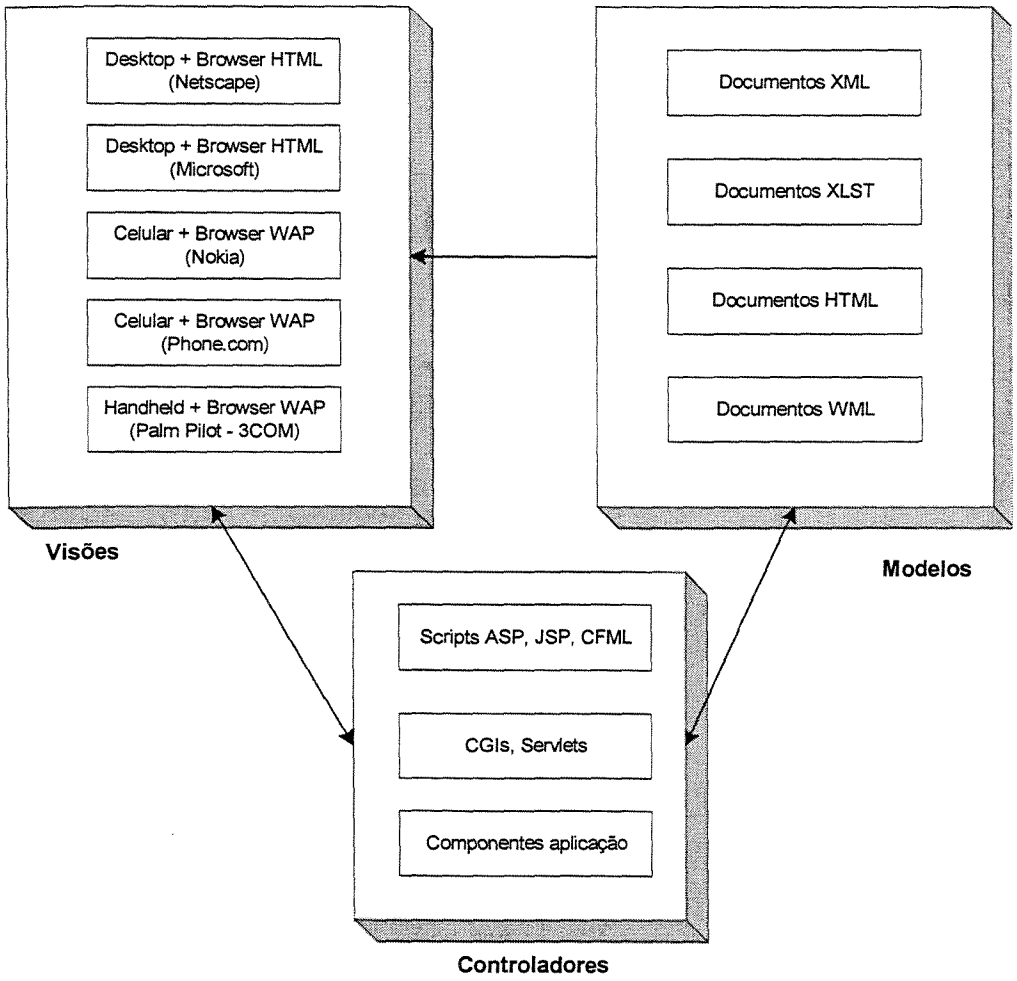


Figura 3.3 - Arquitetura MVC para *Web*

O diagrama da Figura 3.3 apresenta um exemplo da arquitetura MVC que permite empregar uma única camada de apresentação e aplicação para gerar interfaces compatíveis com diferentes categorias de clientes (*desktops*, telefones celulares e *handhelds*). Esta arquitetura tem sido largamente utilizada recentemente no porte das aplicações *Web* para WAP, que em resumo, consiste em trocar a linguagem de apresentação do formato padrão da Internet (HTML) para o formato padrão de WAP (WML).

A idéia da arquitetura MVC sob o contexto *Web* é permitir o tratamento das requisições dos clientes de forma transparente; se não empregarmos MVC num cenário como o ilustrado na Figura 3.1, deveríamos replicar a camada de apresentação, e possivelmente, parte da camada de

---

[9] *Java Server Pages*, solução da Javasoft (SUN)

[10] *Active Server Pages*, solução da Microsoft

[11] *Cold Fusion Markup Language*, solução da Allaire

aplicação para gerar o conteúdo num formato compatível com o cliente, comprometendo fatores como reutilização, facilidade de manutenção e escalabilidade.

3.4 Tecnologias de Implementação

A definição da arquitetura de *software Web* está fortemente vinculada à escolha da abordagem de implementação e das tecnologias que a suportam. Para tanto, é necessário examinar quais as abordagens mais comuns, quais as tecnologias relacionadas a cada uma, e quando utilizar cada abordagem e tecnologia. Esta seção apresenta as abordagens de *scripts* executados em servidores *Web* (CGI, *servlets*), páginas dinâmicas (JSP, ASP, CFML), objetos distribuídos e linguagens de apresentação (XML, XLST), com um estudo comparativo.

3.4.1 Common Gateway Interface (CGI)

CGI foi um dos primeiros avanços a ser incorporado às tecnologias Internet (HTTP) e permite embutir chamadas a *scripts*/aplicações remotas dentro de páginas HTML. O funcionamento é baseado na utilização de páginas HTML que contém formulários; após carregar uma página deste tipo e preencher as informações do formulário, por exemplo, o cliente pode submetê-lo ao servidor WWW; ao receber esta requisição, o servidor WWW identifica qual o *script* associado ao formulário, cria um processo para rodar o *script*/aplicação CGI e redireciona o pedido junto dos parâmetros referentes aos dados do formulário ao processo; ao ser executado, o *script*/aplicação lê os parâmetros da requisição (dados do formulário) através da entrada padrão *stdin* (*standard input file*) do sistema operacional, executa a lógica da aplicação, prepara os resultados do processamento em formato HTML, e redireciona o código HTML do resultado para a saída padrão *stdout* (*standard output file*); por fim, o servidor WWW recebe os resultados em HTML do *script*/aplicação e os repassa ao navegador do cliente como se fosse o conteúdo de um arquivo HTML estático. Um *script* é um programa escrito em C, Java, Perl ou qualquer outra linguagem que suporte a leitura de *input streams* e escrita de *output streams*, que são os mecanismos sobre os quais o protocolo CGI funciona.

Embora o funcionamento deste protocolo seja relativamente simples, há uma série de deficiências, como a falta de escalabilidade e o baixo desempenho, resultado da execução de um processo (aplicação/*script* CGI) por requisição. Além disso, aplicações/*scripts* CGI são complexas e ilegíveis, pois como seu código deve gerar uma página HTML com os resultados do processamento, há uma intercalação de código HTML com a linguagem de programação utilizada para desenvolver o CGI. Por fim, CGI não resolve a deficiência do protocolo HTTP referente ao estabelecimento de sessões, que é imprescindível para a construção de aplicações [OH98]. Apesar das desvantagens, grande parte das aplicações atuais ainda funcionam sobre este protocolo, pois além de ser um padrão da Internet, foi o primeiro mecanismo disponível para a construção de aplicações *Web*.

Devido às deficiências de escalabilidade e desempenho, não é recomendável empregar CGI em soluções de médio/grande porte. Como é onipresente, o custo de aprendizado é relativamente baixo; neste sentido, é recomendável utilizá-lo em soluções simples, sem requisitos de desempenho, escalabilidade e reusabilidade de *software*. Em termos de arquitetura, CGI se enquadra na camada de apresentação do paradigma cliente/servidor *Web* (Figura 3.1), embora seja possível utilizá-lo para implementar lógica de aplicação.

3.4.2 Extensões de Servidor WWW[12]

Na tentativa de criar mecanismos de integração entre servidores *Web* e serviços de infra-estrutura para a construção de aplicações *Web*, como é o caso de servidores de bancos de dados, a maioria dos fornecedores de servidores WWW definiram APIs (*Application Programming Interface*) de *callback* genéricas, como foi o caso do ISAPI (Microsoft), do NSAPI (Netscape) e dos *Apache Modules* (projeto Apache). Estas APIs funcionam de base para o desenvolvimento de *plug-ins*, que podem ser incorporados aos servidores *Web* para estender a sua capacidade de tratamento de requisições HTTP. Atualmente, os *plug-ins* mais comuns implementam:

- Integração com banco de dados, onde é possível utilizar *tags* HTML extras para expressar cláusulas SQL que são interpretadas e executadas pelos *plug-ins*;
- Integração com plataformas de desenvolvimento, como objetos Java e COM/DCOM; a idéia deste tipo de *plug-in* é permitir a integração entre o ambiente do servidor WWW e o ambiente operacional da aplicação; o objetivo é desacoplar a lógica da aplicação, que é implementada através de objetos e componentes disponíveis nas plataformas de desenvolvimento (JVM e Win32, por exemplo) de alto nível, da camada de apresentação, com acesso aos componentes/objetos de aplicação para montar as páginas HTML das interfaces do sistema.

Alguns exemplos de soluções baseadas em SSE (*server side extensions*) incluem *Active Server Pages* (ASP) da Microsoft, *Cold Fusion Markup Language* (CFML) da Allaire, *Java Server Pages* (JSP), PHP do projeto *Apache Software Foundation* e *servlets* da Javasoft. O funcionamento das SSE pode ser representado pelas seguintes etapas:

1. O desenvolvedor instala e configura o *plug-in* de suporte a linguagem que irá utilizar (ASP, JSP, CFML, *servlets*, etc.);
2. As páginas dinâmicas ou objetos de aplicação são instalados sob a área configurada na etapa 1;
3. O usuário final requisita a página dinâmica ao servidor através de um *browser*;
4. O servidor WWW identifica que a página corresponde a uma SSE;
5. O servidor WWW identifica qual o *plug-in* responsável pelo tratamento da página dinâmica através de uma consulta a sua configuração interna de *plug-ins*;
6. O servidor WWW redireciona a requisição ao *plug-in* identificado no passo 5;
7. O *plug-in* interpreta e executa as *tags* proprietárias (extra HTML) e devolve o resultado ao servidor WWW;
8. O servidor WWW intercala o resultado do *plug-in* com as *tags* HTML da página dinâmica, e repassa a página resultante ao *browser* do cliente.

Como podemos observar, a estrutura do mecanismo de SSE é similar à de CGI. Porém, os fornecedores de *plug-ins* têm desenvolvido diversos mecanismos para suprimir as deficiências de CGI, como é o caso do controle de sessões, *tags* que facilitam a execução e formatação de resultados de consultas a bancos de dados, suporte a *cookies*[13], etc.

Entre as vantagens do modelo SSE, podemos destacar:

- Suporte aos mecanismos de sessão e *cookies*;

---

[12] Tradução de *server side extensions* (SSE)

[13] Mecanismo incorporado ao protocolo HTTP que permite a um servidor WWW requisitar ao *browser* do cliente o armazenamento/recuperação de informações para controle de sessões HTTP.

- Definição de *tags* de alto nível que facilitam o trabalho de codificação;
- Suporte a implementação das arquiteturas de camadas e MVC; JSP e ASP, por exemplo, suportam a abordagem de anexar objetos de dados a sessões HTTP, e recuperá-los durante a montagem da página, que é realizada através da intercalação de código HTML e *tags* para acesso aos atributos do objeto de dados. Esta separação permite delegar a responsabilidade de criar o *layout* das páginas à comunidade de *designers*, e a definição das classes de dados, assim como os componentes que produzem os objetos de dados para a camada de apresentação, à comunidade de programadores e engenheiros de *software*. Além disso, é possível atualizar o *layout* das páginas sem interferir na camada de aplicação, e vice-versa.

Como podemos observar na discussão sobre o suporte às arquiteturas de camadas e MVC, SSEs aplicam-se muito bem à implementação da camada de apresentação de sistemas; porém, é comum encontrarmos atualmente sistemas de médio/grande porte totalmente implementados sobre páginas dinâmicas, que mesclam tanto lógica de apresentação, quanto lógica de aplicação, regras de negócio, e assim por diante. Quando submetido a este escopo de utilização, SSEs prejudicam a escalabilidade, a reutilização de *software*, a facilidade de manutenção, e até mesmo a portabilidade, dado que alguns *plug-ins* como ASP, por exemplo, não têm suporte a plataformas não-Windows. A produção de código que mescla as lógicas de aplicação e apresentação contribui para o aumento da complexidade do sistema, sobrecarga dos servidores WWW e queda de desempenho.

Na verdade, a idéia não é realizar uma recomendação sobre utilizar ou não SSE, mas ressaltar os cenários e a forma como podem ser empregadas. De fato, qualquer sistema *Web* que seja utilizado através de *browsers* (HTML, WML, etc.) deve empregar ou CGI ou uma das tecnologias de SSE. Por outro lado, é imprescindível discutir como utilizar SSE, para não incorrer nos riscos apresentados acima. Para sistemas de pequeno porte, é possível utilizar somente tecnologias de SSE sem maiores problemas. Para sistemas de médio/grande porte deve-se considerar a utilização da arquitetura de camadas e MVC e aplicar SSE apenas na camada de apresentação.

3.4.2.1 Servlets

Apesar de *servlets* serem um tipo de SSE que permite executar aplicações Java sob servidores *Web*, é discutida à parte por diferir na forma como são implementados das páginas dinâmicas ASP e JSP, por exemplo. Além disso, oferecem mecanismos de integração cliente/servidor indisponíveis em outras tecnologias, como é o caso da integração entre *applets/servlets* discutida adiante.

Páginas JSP e *servlets* são similares em tempo de execução, pois o *plug-in* responsável pelo tratamento de JSP e *servlets* converte cada página JSP em um objeto do tipo *servlet* em tempo de execução. Ao contrário de CGI e páginas ASP, que são interpretadas e executadas pelos respectivos *plug-ins* a cada requisição atendida, um *servlet* é carregado e interpretado uma única vez, e é mantido em memória onde é executado a cada requisição [HC98]. Quanto maior o número de requisições sobre um *servlet*, maior o número de *threads* Java correspondente ao *servlet* para o atendimento das requisições. Por ser implementado em Java, um *servlet* pode utilizar toda a infra-estrutura da plataforma de forma transparente; no caso da arquitetura ilustrada na Figura 3.1, por exemplo, os *servlets* da camada de apresentação podem interagir com os componentes da camada de aplicação através de CORBA ou RMI.

Enquanto páginas JSP são indicadas para representar a visão do modelo MVC, *servlets* são implementados como classes Java, e são portanto, mais indicados para implementar o controle do modelo MVC, e a integração entre as camadas de apresentação e aplicação do modelo de camadas. Em geral, *servlets* são utilizados na integração com a camada de aplicação, recuperando os objetos de dados dos componentes de aplicação, anexando-os à sessão HTTP e redirecionando o fluxo de execução a páginas JSP, que podem por sua vez, recuperar os objetos de dados da sessão e montar a página HTML que é enviada ao *browser* do cliente.

Como são implementados em Java, *servlets* oferecem a possibilidade de se empregar comunicação cliente/servidor através de serialização de objetos Java. Este cenário consiste em utilizar *applets* no cliente e *servlets* no servidor que trocam dados (objetos Java) através de serialização; esta abordagem é indicada para os casos em que a interface do sistema é muito complexa para ser representada por páginas HTML, e precisam portanto, utilizar *applets* sobre as APIs AWT e JFC[14] [Paw00], ou ainda para os casos em que é necessário trocar estruturas de dados complexas entre cliente/servidor - o pode ser feito de forma transparente através da serialização de objetos que representam as estruturas de dados.

3.4.3 Objetos Distribuídos (OD)

Uma das tecnologias mais desenvolvidas na área da computação cliente/servidor é a de objetos distribuídos, que reúne as vantagens da orientação a objetos à infra-estrutura de sistemas distribuídos; a aplicação desta tecnologia na construção de sistemas *Web* tem originado o paradigma de objetos *Web* [OH98]. Aplicações baseadas em objetos *Web* consistem de componentes que interoperam através de linguagens, redes, sistemas operacionais e plataformas. A infra-estrutura para a comunicação de componentes é conhecida como ORB, como é o caso de CORBA (*Common Object Request Broker Architecture*), RMI (*Remote Method Invocation*) e DCOM (*Distributed Component Object Model*); o objetivo de um ORB é oferecer um "barramento" através do qual objetos e componentes podem colaborar para formar aplicações.

A iniciativa de objetos distribuídos tem ganho a adesão dos maiores desenvolvedores de *software* da atualidade e está presente em *browsers*, linguagens de programação e protocolos. O uso de ORBs permite integrar aplicações escritas em diferentes linguagens e distribuídas em diversas plataformas e ambientes de rede. Por isso, tem despontado como uma infra-estrutura ideal para a construção de aplicações de missão crítica e de grande porte, sobretudo aquelas aplicações concebidas para *Web* que são construídas sobre sistemas legados, e que normalmente exigem a integração com sistemas construídos em diferentes linguagens e implantados em diferentes plataformas.

Em geral, uma aplicação construída sobre a tecnologia de objetos distribuídos é dividida em componentes e subsistemas, que por sua vez definem interfaces para representar os serviços que implementam. Para criar uma aplicação utilizam-se as interfaces dos componentes e subsistemas, que podem inclusive executar em outros servidores, redes e plataformas. Como permite dividir uma aplicação em subsistemas, que podem eventualmente ser distribuídos, OD são fortemente recomendáveis para a implementação das arquiteturas de camadas e cliente/servidor *Web*.

Da mesma forma que os ORBs definem e encapsulam os detalhes de infra-estrutura para a distribuição de objetos, também o fazem com uma série de outros serviços de base, que correspondem aos serviços das camadas mais inferiores da arquitetura de camadas (Figura 3.2); algumas destas facilidades incluem os serviços de suporte a persistência, transações, segurança,

---

[14] *Java Foundation Classes*, designação para a biblioteca `javax.swing` de Java.

controle de ciclo de vida, etc [OH98][Tho98][Cha96]. Como o modelo de objetos distribuídos é baseado na abordagem de divisão de sistemas em subsistemas com interfaces bem definidas, é natural que seja empregado como infra-estrutura para construção de aplicações baseadas em componentes de *software*. De fato, a combinação dos serviços dos ORBs com serviços de suporte ao desenvolvimento de componentes contribuíram para o surgimento dos *servidores de aplicação*. Como o desenvolvimento de componentes de *software* é um dos pontos focais deste trabalho, discutimos o tema em mais detalhes no capítulo 4, além de apresentar os servidores de aplicação baseados em CORBA, RMI e DCOM.

3.4.4 XML

XML, ou *Extended Markup Language*, também é padronizada pela OMG e é uma espécie de meta-linguagem, que permite definir outras linguagens. O estímulo para o desenvolvimento da XML surgiu com o SGML (*Standard Generalized Markup Language*) e seus derivados, como o HTML, mais precisamente com os problemas enfrentados devido às inúmeras versões e extensões que se seguiram após a criação do HTML pelo W3C (*World Wide Web Consortium*). Como uma meta-linguagem, XML é independente de sintaxe e formato, e pode ser utilizada para definir outras linguagens, como a própria HTML. Um exemplo recente de sua utilização é a definição das versões de WML (*Wireless Markup Language*), uma espécie de subconjunto do HTML para micro-*browsers* embutidos em telefones celulares e *handhelds*.

Além de definir outras linguagens, XML pode ser utilizada em outros contextos, como é o caso da representação de dados independente do formato e *layout* de apresentação. A família de linguagens XML também define um mecanismo para transformação da XML em formatos e *layouts* específicos, a linguagem XSLT (*Extensible Stylesheet Language Transformations*). Uma aplicação *Web*, por exemplo, cuja camada de apresentação seja construída sobre XML, pode trocar de *layout* (ou a visão do modelo MVC) sem modificar a estrutura dos documentos XML; a idéia é criar documentos XLST que contenham as informações sobre *layout* e posicionamento dos campos de dados; em outras palavras, cada documento XLST pode conter as instruções para converter as informações de um documento XML para uma linguagem de apresentação específica, como HTML ou WML.

Desta forma, podemos conceber um cenário onde é possível criar um único documento XML, um documento XLST para formatação dos dados em HTML e outro para formatação em WML, e fazer com que a mesma informação (representada pelo documento XML) seja apresentada em clientes de categorias diferentes. Como o XML é, em geral, montado pelo controlador (modelo MVC) do sistema, que se baseia na camada de aplicação para recuperar as informações e montar o documento XML, podemos afirmar que uma única camada de aplicação e um único modelo (documento XML) pode ser empregado por duas visões diferentes (modelo MVC) - representadas neste caso por um *browser* de um *desktop* e um micro-*browser* de um telefone celular ou *handheld*.

Ao oferecer suporte à separação de informações e formatos de representação (*layout*) e permitir a transformação das informações em diferentes *layouts*, XML destaca-se como a plataforma ideal para implementar a arquitetura MVC. Enquanto documentos XML correspondem ao modelo MVC, os documentos XLST referem-se a visão; para construir uma camada de apresentação *Web*, documentos XML e XLST devem ser combinados com páginas dinâmicas e *servlets* que correspondem ao controle de MVC.

3.4.5 Estudo Comparativo

Como podemos observar, as aplicações *Web* são implementadas a partir da combinação de uma série de arquiteturas e tecnologias, que não são mutuamente exclusivas, pois cada uma oferece recursos para a implementação de aplicações em contextos distintos. A escolha da tecnologia apropriada a cada aplicação não é tarefa trivial, pois há uma série de fatores que devem ser considerados. Para facilitar a escolha das tecnologias mais apropriadas durante o projeto da arquitetura de aplicações *Web*, apresentamos um estudo comparativo através da Tabela 3.2, que considera uma série de aspectos, como o grau de reutilização de *software*, o nível de abstração para construção de aplicações, o desempenho, a escalabilidade, a portabilidade, e o contexto em que cada tecnologia deve ser utilizada.

Reutilização é um fator desejável, pois permite cortar tempo e custos de desenvolvimento, além de aumentar a qualidade do sistema, uma vez que são reutilizados componentes testados e maduros. Neste aspecto, as soluções baseadas em objetos distribuídos oferecem a melhor abstração para a reutilização de *software*, pois são construídas sobre o paradigma de objetos e componentes de *software*. Ao contrário de objetos distribuídos, as soluções CGI e SSE possuem mecanismos rudimentares de reutilização, dado que são baseadas em *scripts* e *tags* extra-HTML.

O nível de abstração de cada tecnologia refere-se às facilidades oferecidas para a construção de aplicações *Web*. Segundo [OH98], quanto maior o nível de abstração, mais simples é o trabalho de desenvolvimento; para fazer uma analogia, se utilizamos *sockets* para construir aplicações *Web*, deveremos definir uma série de convenções para a requisição de serviços e recebimento dos resultados; se comparado ao desenvolvimento sobre objetos distribuídos, desenvolver sobre *sockets* é como utilizar linguagens de baixo nível, como *Assembly*. Os ORBs das tecnologias de objetos distribuídos apresentam o maior nível de abstração, pois encapsulam os detalhes de comunicação cliente/servidor e diversos serviços de infra-estrutura (transações, persistência, segurança, etc.). Segundo [OH98], CGIs e SSE situam-se num nível intermediário entre *sockets* e ORBs.

As notas sobre desempenho são baseadas nas medições de [OH98]. Embora SSE não tenha sido incluída na medição de [OH98], poderíamos classificar seu desempenho como médio, pois há um *overhead* no processamento das requisições a páginas dinâmicas, tanto para a interpretação das *tags* não-HTML, quanto para a comunicação entre o servidor WWW e os *plug-ins* correspondentes.

Em termos de escalabilidade, podemos afirmar que CGI é a solução menos escalável, ao passo que objetos distribuídos constitui a solução mais escalável. Como, em geral, um servidor WWW inicia um processo para atender cada requisição em CGI, o aumento no número de requisições inviabiliza o funcionamento do servidor, pois o próprio sistema operacional possui um número limitado de processos concorrentes. As soluções baseadas em SSE apresentam escalabilidade média, dado que os *plug-ins* incorporam mecanismos de compartilhamento de recursos, *multi-threading*, e *pooling* de objetos. Como nas aplicações distribuídas pode-se implantar seus componentes através de diversos servidores, além de configurar cada ORB/servidor de aplicação para realizar balanceamento de carga e compartilhamento de recursos, a solução baseada em objetos distribuídos é a mais escalável.

Por fim, classificamos a portabilidade de cada tecnologia, que diz respeito ao suporte às plataformas mais comuns em ambientes *Web*. As tecnologias mais portáveis são CORBA e CGI, que possuem implementações disponíveis na maioria das plataformas de clientes e servidores. Com são tecnologias baseadas em Java, RMI e *servlets* estão disponíveis em todas as plataformas

que suportam a JVM (RMI e *servlets*) e em todos os servidores que dispõem de *plug-in* (*servlets*). Como há JVM e *plug-ins* para a maioria das plataformas e servidores WWW, podemos considerar que RMI e *servlets* são portáveis. Por outro lado, DCOM, pela forte dependência da plataforma Windows, e *server-side extensions*, pela dependência do servidor WWW, são pouco portáveis. Um dos aspectos que influencia na portabilidade é a padronização das tecnologias: enquanto CORBA e CGI são padrões abertos, de direito, Java (RMI e *servlets*) e DCOM são tecnologias mantidas por empresas privadas (Javasoft e Microsoft) e submetidas à ISO (Java) e ao Open Group (COM) para certificação, o que as caracteriza como semi-abertas.

	CGI	**SSE**	**Servlets**	**RMI**	**CORBA**	**DCOM**
Componente de *Software*	Scripts em Perl, C ou Java	Tags extra-HTML	Objetos Java	Objetos Java	Objetos CORBA	Objetos COM
Nível de Abstração	Baixo	Baixo	Baixo	Alto	Alto	Alto
Desempenho	Baixo	Médio	Médio	Alto	Alto	Alto
Escalabilidade	Baixa	Média	Média	Alta	Alta	Alta
Portabilidade	Alta	Baixa	Média	Média	Alta	Baixa
Foco de Utilização	Aplicações simples, sem restrições de desempenho nem necessidade de escalabilidade.	Porte de CGIs, aplicações de pequeno/médio porte com requisitos de acesso a banco de dados e implementação da camada de apresentação de aplicações de médio ou grande porte.	Idem a SSE, mas para aplicações com camada de apresentação baseada em JSP e camada de aplicação baseada em CORBA e Java.	Integração entre os componentes da camada de aplicação (baseados em Java), bem como, entre *servlets* e a camada de aplicação.	Integração entre os componentes da camada de aplicação distribuídos através de diferentes linguagens, redes e plataformas, bem como, entre as camadas de apresentação e aplicação.	Integração entre os componentes da camada de aplicação (baseada em COM), bem como entre as camadas de apresentação (baseada em DLLs e ASP) e a camada de aplicação.

Tabela 3.2 - Comparação entre tecnologias de *middleware Web*

3.6 Conclusões

O desenvolvimento de *software* para *Web* pressupõe projetos de curta duração, envolve a escolha e a combinação de diversas tecnologias, além de uma série de requisitos, como escalabilidade, desempenho, portabilidade, múltiplas interfaces com usuário (HTML, WML, etc.), integração com sistemas legados, etc. Como podemos observar, a maioria dos requisitos de projetos deste porte são não funcionais, pois não se referem aos casos de uso do sistema. Neste sentido, é imprescindível que o desenvolvimento de um sistema *Web* seja suportado por uma arquitetura de *software* que enfoque os aspectos não funcionais do projeto.

O estudo sobre as arquiteturas mais comuns de sistemas *Web* deixa clara a tendência de se transportar a lógica dos sistemas para as camadas da parte servidora. Além disso, a análise das tecnologias de desenvolvimento *Web* aponta para a segmentação das aplicações em funções bem definidas, que é o princípio da arquitetura de camadas, da qual podemos destacar a camada de apresentação, onde atuam tecnologias como SSE; camada de aplicação, onde são empregados

servidores de aplicação (capítulo 4) para hospedar componentes de *software*; camada de comunicação, onde se situam os ORBs de objetos distribuídos, e serviços de infra-estrutura, que correspondem aos servidores de bancos de dados, servidores *Web*, etc.

Capítulo 4

Componentes de *Software*

O crescimento na utilização da Internet, sobretudo no meio comercial, contribuiu para o surgimento de diversas novas tecnologias de suporte ao desenvolvimento de aplicações *Web*. Porém, podemos afirmar que o alto número de tecnologias disponíveis para implementação, a complexidade de cada uma, além do esforço despendido na sua combinação para construir aplicações *Web* têm comprometido a reusabilidade, a escalabilidade e o tempo de desenvolvimento.

Neste capítulo apresentamos a abordagem de desenvolvimento através de componentes de *software* e discutimos como incorporá-la às arquiteturas de camadas e cliente/servidor *Web* (capítulo 3) para produzir aplicações reutilizáveis e escaláveis num curto espaço de tempo.

4.1 Origem

Apesar das pesquisas e desenvolvimentos na área de orientação a objetos nas últimas décadas, a promessa de reutilização de código em larga escala ainda não foi alcançada [Sch96a][Bar97], pois grande parte dos esforços envolvidos neste processo são despendidos na redescoberta e reinvenção de conceitos e componentes de *software*. Os mecanismos de reutilização oferecidos pela orientação a objetos são difíceis de se empregar e insuficientes para implantar uma política de reutilização de *software* em larga escala, devido à complexidade e à baixa granularidade oferecida por objetos. A falta de uma infra-estrutura adequada para reutilização, além do crescimento da heterogeneidade de *hardware*, arquiteturas de rede e sistemas operacionais (sobretudo em sistemas *Web* - Internet e *intranets*), têm tornado proibitivo o desenvolvimento de *software* correto, portável, eficiente e barato [FS97].

A tecnologia de componentes é uma evolução da orientação a objetos e permite representar entidades mais abstratas que objetos, com maior granularidade, e maior tendência à reutilização. Além disso, o emprego de interfaces constitui o diferencial em relação à orientação a objetos tradicional, que baseia-se principalmente em herança. Embora herança seja um mecanismo poderoso, pode criar fortes dependências entre objetos, aumentar a complexidade do código e

dificultar a reutilização [Bar97]. Por outro lado, as interfaces eliminam as dependências da aplicação com relação ao código interno do componente, que pode até ser modificado, desde que mantidas as suas interfaces e o seu comportamento original.

Componentes são blocos de *software* configuráveis que podem ser reutilizados e combinados entre si para formar aplicações completas [FS97][Tho98]. Um componente implementa funcionalidades recorrentes entre aplicações a partir de um ou mais objetos e as torna disponíveis através de suas interfaces. Para as aplicações que o utilizam, um componente é uma espécie de caixa-preta que dispõe de um conjunto coeso de operações que podem ser utilizadas a partir do conhecimento da sintaxe e da semântica de suas interfaces [FS97]. Controles ActiveX/OCX, Beans/Enterprise JavaBeans e componentes CORBA são alguns exemplos de componentes.

A metodologia de desenvolvimento de *software* baseado em componentes, na qual aplicações são construídas a partir da combinação e da integração de componentes, permite melhorar os níveis de modularidade e reutilização, além de aumentar a qualidade do *software* e reduzir o tempo e os custos do seu desenvolvimento. Um exemplo muito comum é o de componentes que implementam recursos de interface gráfica (GUI, *Graphic User Interface*), muito difundidos na plataforma Windows. Recentemente, o emprego de componentes estendeu-se tanto à construção de *software* para infra-estrutura, como protocolos de comunicação e mecanismos de acesso a bancos de dados, quanto às aplicações verticais que implementam atividades e processos de negócio em segmentos específicos, como é o caso de componentes para ERP (*Enterprise Resource Planning*), comércio eletrônico, etc.

Entre as características mais importantes de componentes podemos destacar:

- Independência de ambiente: componentes devem interoperar através de linguagens, sistemas operacionais, redes e ambientes de desenvolvimento;
- Interface bem definida: a interface é um contrato que especifica a funcionalidade de um componente e não deve expor dependências de implementação, plataforma ou ambiente;
- Extensão: componentes devem suportar configuração através de propriedades, herança e polimorfismo;
- Integração: um componente não é uma aplicação completa e deve ser capaz de se integrar com um ou mais subsistemas para formar uma aplicação ou um novo componente em combinações imprevistas. Apesar da sua generalidade, deve ser projetado para realizar um conjunto de tarefas limitadas dentro de um domínio de aplicação.

4.2 Tipos de Componentes

A iniciativa de reutilizar *software* não surgiu com componentes, pois há uma série de mecanismos que seguem esta abordagem, dos mais simples, como é o caso de bibliotecas de classes (orientação a objetos) e de funções (programação estruturada), aos mais sofisticados, como os *frameworks*. A diferença entre estes mecanismos corresponde aos níveis de abstração, reutilização e ganho em produtividade.

Bibliotecas de classes e de funções são muito difundidas em linguagens de programação e sistemas operacionais (APIs e *system calls*, por exemplo). Uma biblioteca é um meio de tornar disponíveis operações, serviços e estruturas de dados que podem ser (re)utilizadas na construção de aplicações. Segundo [Kar98], bibliotecas podem ser classificadas como componentes de grão

fino[15], pois representam abstrações de baixo nível e são muito dependentes do domínio de aplicação. Bibliotecas implementam funcionalidades específicas e limitadas e podem ser reutilizadas entre várias aplicações de diferentes domínios; porém, não oferecem um alto ganho de produtividade, devido ao seu baixo nível de abstração. Considere como exemplo, o mecanismo de *sockets*; qualquer aplicação com requisitos de comunicação entre processos pode reutilizar as bibliotecas que implementam os recursos de *sockets*, o que constitui um alto nível de reutilização; por outro lado, o ganho de produtividade é baixo, pois a lógica da aplicação que utiliza *sockets* pode envolver funcionalidades muito mais complexas que a comunicação entre processos.

A abordagem de componentes oferece diferentes níveis de abstração, reutilização e ganhos de produtividade, pois existem diferentes tipos de componentes implementados sobre diferentes domínios de aplicação. Componentes de baixa granularidade ou de pequena escala, como aqueles que implementam funcionalidades de interface gráfica (GUI) são os mais difundidos; embora ofereçam um alto potencial de reutilização, não proporcionam altos ganhos de produtividade devido ao baixo nível de abstração. Por outro lado, existem componentes de larga escala, de alta granularidade, como aqueles que implementam entidades de negócio (cliente, produto, etc.) e serviços de infra-estrutura (protocolos de comunicação, mecanismos de acesso a bancos de dados, etc.); estes componentes oferecem um alto nível de abstração, e embora sejam construídos sobre domínios de aplicação específicos (o que reduz o seu potencial de reutilização) são os que oferecem o maior ganho em produtividade [Kar98].

Outro mecanismo de reutilização em ampla ascensão é o de *frameworks*. Um *framework* é uma coleção de classes concretas e abstratas (e as interfaces que as descrevem) que define a arquitetura de um subsistema. Aplicações são construídas sobre um *framework* através da redefinição das suas classes concretas e implementação das suas classes abstratas. Desta forma, *frameworks* permitem reutilizar tanto código, definido pelas classes concretas, quanto projeto, definido pelo modelo de classes concretas e abstratas. *Frameworks* também podem ser vistos como a estrutura na qual componentes de *software* são combinados para formar uma aplicação. *Frameworks* diferem de componentes por oferecer maior nível de abstração: definem a arquitetura na qual os componentes são integrados, as interfaces para esta integração e o controle do fluxo de execução do sistema. A idéia de *frameworks* é prover o esqueleto de uma aplicação, um molde que pode ser reutilizado na construção de diversas aplicações do mesmo domínio que o *framework.*

O potencial de *frameworks* pode ser verificado em diversos domínios, desde a área de infra-estrutura, com os *frameworks* de interface gráfica MFC (*Microsoft Foundation Classes,* do Windows), MacAPP (do MacOS) e JFC *(Java Foundation Classes*, de Java) [Sch96a][Sch96b], *middleware*, com os *frameworks* de CORBA [FS97], até as áreas de negócio, como o projeto SanFrancisco (da IBM) [Ibm01]. As iniciativas de construção de *frameworks* têm se voltado à área de negócios por possuir um potencial de retorno de investimento bem mais atraente que outros tipos de *frameworks* [FS97]. A OMG, por exemplo, possui diversas propostas de *frameworks* verticais que implementam rotinas de negócio nas áreas de *workflow*, contabilidade, saúde, recursos humanos, bancos e transporte [OMG99]. Embora *frameworks* sejam pouco reutilizáveis, devido às dependências aos domínios de aplicação a que se referem, oferecem altos ganhos de produtividade, devido ao alto nível de abstração.

---

[15] Tradução de *fine-grained components*

Embora tenhamos comparado o nível de abstração dos mecanismos de reutilização, há uma série de fatores que influem no ganho de produtividade e nos níveis de reutilização. Componentes são mecanismos mais aprimorados que bibliotecas de classes, mas não somente pelo maior nível de abstração. A infra-estrutura sobre a qual componentes são construídos e utilizados para desenvolver aplicações oferece uma série de facilidades voltadas à reutilização. Alguns modelos de componentes permitem integrar componentes escritos em diferentes linguagens; outros, distribuir componentes a partir de redes em diferentes plataformas. Portanto, não podemos considerar modelos de componentes como uma metodologia apenas, mas como uma plataforma para a construção de *software* reutilizável.

Componentes de pequena escala para a construção de interfaces gráficas já são muito difundidos. Atualmente, o cenário de desenvolvimento de sistemas cliente/servidor multicamada, além da ênfase em *software* portável e multiplataforma para ambientes *Web,* têm contribuído para o crescimento da demanda por componentes de larga escala que implementem a porção servidor das aplicações multicamada. Além dos fatores modularidade e reutilização, componentes no lado servidor oferecem diversas oportunidades:

- Componentes podem encapsular sistemas legados e tornar disponíveis as suas funcionalidades através de interfaces, isolando detalhes como linguagem e plataforma de implementação. Dependendo do modelo de componentes empregado, pode-se dispor os serviços dos sistemas legados a aplicações locais ou remotas implementadas em diferentes linguagens e plataformas.
- Componentes podem implementar funcionalidades de infra-estrutura que são utilizadas por outros componentes e subsistemas de uma aplicação, como é o caso de acesso a banco de dados, acesso a diretórios, mecanismos de controle de acesso e segurança, entre outros. Como observamos no capítulo 3, componentes de infra-estrutura compõem a camada de base para a construção de aplicações, pois podem ser reutilizados através de diferentes sistemas, independentemente do domínio da aplicação. Por apresentar um dos maiores potenciais de reutilização, esta é a categoria de componentes implementada neste trabalho (capítulo 5).
- Sistemas multicamada baseados em componentes facilitam as atividades de manutenção, como a atualização de componentes para a correção de problemas. As alterações realizadas no código interno do componente não interferem nos demais componentes, e portanto, nas demais camadas do sistema. Esta abordagem também é empregada neste trabalho para a implementação da INEX, descrita em detalhes no capítulo 6.
- Os recursos dos servidores podem ser empregados para aprimorar o desempenho, escalabilidade, confiabilidade e facilidade de manutenção das aplicações. Como os modelos de componentes oferecem recursos para a integração de componentes distribuídos entre servidores, pode-se construir uma aplicação a partir de componentes distribuídos entre diferentes servidores, plataformas e redes.

4.3 Modelos de Componentes

Componentes são implementados segundo um modelo que define a infra-estrutura para a sua criação, integração, distribuição e utilização. Este modelo especifica como componentes devem ser criados e combinados para formar uma aplicação, além de serviços básicos como persistência, suporte a eventos, suporte a ferramentas de desenvolvimento integrado (IDE, *Integrated Development Enviroment*), entre outros.

O modelo define duas estruturas: componente, ou como é projetado, distribuído e utilizado, e *container,* ou como componentes são integrados, como provêem informações uns aos outros e ao *container,* como utilizam os serviços do *container*, etc.

Um componente é implementado através de alguma linguagem de programação e deve seguir o modelo de componentes para que possa ser (re)utilizado a partir de outras linguagens e plataformas. Um modelo de componentes define as linguagens disponíveis para implementação, a estrutura das interfaces, o formato de distribuição, além de uma série de serviços de infra-estrutura sobre os quais os componentes são implementados. Em tese, todos os componentes que seguem as normas de um determinado modelo podem ser combinados para formar aplicações.

Para que o formato de componentes seja independente de plataforma, linguagem e ambiente de desenvolvimento, deve seguir mecanismos padronizados de distribuição, geração de código executável e representação de metadados. **JavaBeans** (seção 3.3.1) por exemplo, é dependente de linguagem e independente de ambiente de desenvolvimento e plataforma. **COM** (seção 3.3.2), por outro lado, é independente de linguagem e dependente de plataforma, pois define um padrão de representação do código executável - componentes implementados em Java, C++ e/ou Visual Basic são traduzidos para um mesmo padrão de código binário.

Um *container* corresponde ao ambiente onde componentes são incorporados e executados [Wat98]; há *containers* de vários tipos, como *forms, shells,* páginas HTML, monitores de transação (TP *monitors*), servidores WWW, SGBDs, e até mesmo componentes. Durante a etapa de desenvolvimento de uma aplicação um IDE pode oferecer *containers* (um *form,* por exemplo) para a utilização de componentes; ao incorporar um componente, um *container* deve ser capaz de realizar introspecção para determinar metadados, informar as interfaces, os eventos e as propriedades disponíveis, permitir a modificação das propriedades, bem como a persistência destas informações, oferecer suporte a integração com outros componentes através da associação de eventos à execução de operações em interfaces, representar visualmente os componentes no ambiente de desenvolvimento, etc.

Para viabilizar a programação de aplicações através de componentes, um modelo de componentes ainda define uma série de serviços: introspecção, *layout*, suporte a IDEs, eventos, configuração, persistência e distribuição. Introspecção permite expor as funcionalidades de um componente à aplicação ou aos outros componentes que o utilizam. *Layout* define os aspectos da representação gráfica de um componente visual dentro de seu *container.* Suporte a IDEs corresponde aos mecanismos embutidos no modelo para que componentes possam ser incorporados e utilizados em IDEs. Um evento é algo que acontece no interior de um componente e que deve ser detectado e respondido pela aplicação e pelos demais componentes que o utilizam; o modelo define as formas como um componente expõe, detecta e reage a eventos. Por fim, o serviço de distribuição define a infra-estrutura para que componentes distribuídos possam ser integrados para formar uma aplicação.

Como é o modelo de componentes que define a forma como componentes são construídos e utilizados, é indispensável conhecê-lo para desenvolver *software* baseado em componentes, ou seja, tanto na construção de componentes reutilizáveis, quanto na (re)utilização destes componentes para formar aplicações. Os modelos de componentes mais difundidos atualmente são COM/ActiveX, JavaBeans/Enterprise JavaBeans e CCM/CORBA.

4.3.1 JavaBeans

JavaBeans é o modelo de componentes incluído na plataforma Java desde a sua versão 1.1. O modelo JavaBeans foi especificado por um consórcio de empresas (Javasoft, Netscape, Oracle, Microsoft, Symantec, Inprise, etc.), e consiste de um *framework* que permite definir classes Java como componentes JavaBeans. JavaBeans foi incorporado ao núcleo da plataforma Java (API `java.beans`), o que o torna disponível a todas as plataformas que dispõe de uma máquina virtual Java (JVM). Componentes JavaBeans, ou simplesmente Beans, são implementados em Java sobre a API `java.beans`, que define serviços de introspecção, eventos, configuração, *layout* e persistência, além de mecanismos para a distribuição de componentes (arquivos JAR) e um *container* (`java.beans.beancontext`). Por ser implementado sobre a plataforma Java, componentes JavaBeans herdam todas as suas funcionalidades, como interoperabilidade entre plataformas, infra-estrutura para distribuição de código (*applets*), suporte a objetos (ou componentes) distribuídos RMI e CORBA, bancos de dados (JDBC), diretórios (JNDI), transações (JTS), etc.

O objetivo de JavaBeans é permitir a criação de componentes que podem ser empregados em ambientes de desenvolvimento (IDEs). Como IDEs empregam um paradigma de programação visual, no qual Beans podem ser manipulados, integrados e configurados visualmente, desenvolver aplicações a partir de Beans é uma tarefa que requer menos esforços de programação. No entanto, como JavaBeans foi projetado para ser empregado em IDEs, é imprescindível que se utilize ferramentas deste tipo tanto na construção de aplicações, quanto na de componentes. Atualmente, existem uma série de ferramentas IDE com suporte a JavaBeans catalogadas na página[16] sobre JavaBeans no *site* da Javasoft.

Um Bean é um conjunto de classes Java compatível com o modelo de componentes especificado pela API de JavaBeans. Existem algumas regras de implementação para que IDEs possam fazer introspecção num Bean, além de uma classe adicional que representa os metadados (classe `BeanInfo`) do componente. Um Bean pode especificar um conjunto de propriedades, além de implementar *wizards*[17] e editores específicos para a configuração de propriedades pelo usuário/programador. Um Bean ainda pode incorporar eventos definidos pelo usuário e/ou programador para facilitar a sua integração a outros componentes. Em geral, um Bean é distribuído através de um arquivo JAR, um arquivo de formato ZIP que inclui o código compilado (arquivos `.class`), metadados (arquivo *manifest* ou `.mf`), além de eventuais arquivos utilizados pelo componente (imagens, arquivos de configuração, etc.).

4.3.2 COM/ActiveX

COM (*Component Object Model*) é o modelo de componentes da Microsoft, sobre o qual são construídos grande parte das aplicações e serviços do sistema operacional Windows. ActiveX é uma nova designação concebida para componentes COM, também conhecidos como controles OLE ou OCX; OLE (*Object Linking and Embbeding*) corresponde aos mecanismos de suporte a documentos compostos, *drag and drop* e automação da plataforma Windows implementados sobre COM.

COM define todos os mecanismos de um modelo de componentes, além de um padrão para representação binária de interfaces. Desta forma, componentes COM podem ser desenvolvidos

---

[16] `http://java.sun.com/beans/tools.html`

[17] Interface gráfica que apresenta uma sequência de janelas com questões para facilitar a configuração do componente.

em diferentes linguagens disponíveis no ambiente Windows (Visual Basic, Visual C++, Visual J++), pois o código gerado por IDEs compatíveis com COM segue um padrão definido pelo modelo de componentes. Como resultado, componentes escritos em Visual Basic, por exemplo, podem ser combinados com componentes escritos em Visual C++ e Visual J++ para formar aplicações ou outros componentes. Apesar de ser independente de linguagem de implementação, o IDE utilizado para construir componentes deve seguir o padrão binário especificado por COM. Além disso, como discutimos no capítulo 3.4.5 COM não é portável, pois é restrito à plataforma Windows.

COM, assim como JavaBeans e Enterprise JavaBeans, também define uma infra-estrutura para a interconexão de componentes distribuídos, conhecida como DCOM (*Distributed* COM). Um dos aspectos mais importantes de COM/ActiveX refere-se à disponibilidade: como são tecnologias consolidadas há tempo no mercado de desenvolvimento de *software*, há oferta predominante de componentes ActiveX para a construção de aplicações. Além disso, existem ambientes de desenvolvimento baseados em diferentes linguagens de programação com suporte a COM/ActiveX, e até mesmo um servidor de aplicações transacionais voltado à construção de componentes COM na parte servidor, o MTS (*Microsoft Transaction Server*). Além dos componentes disponíveis, os próprios produtos baseados no Office (Word, Excel, PowerPoint, Access) dispõem serviços internos através de interfaces COM, num mecanismo conhecido como automação OLE [Cha96]; este cenário contribui para o aumento do número de componentes (e funcionalidades) disponíveis ao usuário/programador ActiveX.

Apesar das restrições de portabilidade, a disponibilidade de ferramentas e componentes COM/ActiveX é maior que em outras tecnologias, o que faz deste modelo uma opção viável para a construção de componentes voltados ao ambiente Windows.

4.3.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) é um modelo de componentes voltado à construção e à distribuição de aplicações multicamada, transacionais e baseadas em objetos distribuídos [Tho98]. EJB define um *framework* (através da API `javax.ejb`) que estende o modelo de JavaBeans e especifica um conjunto de serviços imprescindíveis à construção e implantação de sistemas de grande porte, como o suporte a transações, compartilhamento de recursos (*threads,* conexões com bancos de dados), escalabilidade, portabilidade, etc.

A construção de aplicações cliente/servidor multicamada demanda o compartilhamento de componentes na parte servidor; no entanto, construir componentes compartilháveis não é tão simples quanto construir aplicações monolíticas [Tho98]. Neste cenário, os componentes devem compartilhar recursos escassos no sistema, como *threads,* processos, memória, conexões com bancos de dados e conexões de rede. JavaBeans pode ser utilizado para desenvolver tais componentes, desde que se implemente a lógica de compartilhamento de recursos como parte destes componentes [Tho98], que é uma tarefa difícil. Por outro lado, como EJB define mecanismos de compartilhamento de recursos, transações e gerenciamento como parte do seu modelo de componentes, é mais adequado à implementação de aplicações cliente/servidor *Web*. Uma das maiores vantagens de EJB é isolar os detalhes relacionados à infra-estrutura necessária à construção de aplicações de grande porte numa API (ou *framework*) padrão.

Uma aplicação baseada em componentes EJB é executada sobre um servidor EJB que oferece os serviços especificados pelo modelo de componentes. Um servidor EJB inclui um *container* que implementa alguns dos serviços previstos por EJB, tais como gerenciamento de ciclo de vida, coordenação de transações, persistência, serviço de nomes e segurança. Segundo [Tho98], um

container pode ser comparado a um monitor de transações (TP *monitor*), que controla recursos compartilhados e transações, a um SGBD, que coordena transações e *locks* de registros de vários usuários, ou a um servidor WWW, que pode executar processos adicionais (CGIs, *servlets, server-side extensions*) como resposta a requisições HTTP.

Diferente de um Bean, um EJB é sempre um componente não visual, além de não empregar eventos [Tho98]. Um EJB suporta transações implícitas, dispensando a necessidade de demarcar o código de implementação com *start, commit* ou *rollback*[18]. O próprio sistema de execução (servidor EJB) controla as transações associadas a um EJB a partir da definição da política de transações adotada pelo componente, que é especificada de forma descritiva através de um arquivo de configuração; este arquivo ainda pode incluir definições sobre ciclo de vida, ou como criar, ativar, desativar e destruir um EJB; persistência, indicando se o EJB realiza sua própria persistência ou se a mesma é delegada ao *container,* e por fim, segurança, especificando as regras de segurança aplicadas ao EJB. Desta forma, a configuração de um EJB é feita prioritariamente através do seu arquivo de configuração, também conhecido como *deployment descriptor.* Um componente EJB é composto por objetos `EJBObject` que implementam as interfaces do componente; como um `EJBObject` é um objeto distribuído, deve implementar suas interfaces através de RMI, que é o mecanismo padrão para a definição de objetos distribuídos em Java. Há duas opções distintas para implementação de componentes EJB: *session beans,* que não mantêm estado entre sessões e podem ser compartilhados entre vários clientes, e *entity beans,* que são objetos persistentes, únicos, que mantém o estado entre sessões [Tho98] e não podem ser compartilhados entre vários clientes.

Um aspecto importante do modelo de componentes EJB é que são especificados os serviços que um servidor/*container* deve oferecer, como os componentes são construídos, como interagem com o *container* e com outros componentes; o modelo não especifica detalhes dos serviços ou como devem ser implementados [Tho98]. Desta forma, fornecedores de servidores EJB possuem flexibilidade para implementar os serviços previstos no modelo de forma diferenciada; este cenário permite que ambientes de execução transacionais sejam adaptados para o modelo EJB. Segundo [Tho98] os seguintes ambientes podem ser considerados candidatos a servidor EJB: TP *monitors* (IBM TXSeries, BEA Tuxedo), servidores de aplicações transacionais (Sybase Jaguar CTS, Microsoft Transaction Server/MTS), ORBs CORBA (Inprise Visibroker/ITS, Iona Orbix/OTM), SGBDs (Oracle, Sybase, IBM DB2) e servidores WWW (Java *Web* Server, Netscape Enterprise Server, Oracle Application Server).

Enterprise JavaBeans também adota uma postura de integração ao padrão CORBA/IIOP. Apesar de empregar RMI como infra-estrutura para a distribuição das interfaces dos componentes, o mapeamento RMI/IDL e o desenvolvimento de RMI sobre IIOP deverão permitir a interoperabilidade entre clientes e servidores EJB e CORBA. Além disso, há uma tendência de aproveitar as especificações e as correspondentes implementações disponíveis de CORBAServices no desenvolvimento de servidores EJB, como é o caso do serviço de transações de EJB atual, compatível com CORBA OTS [OH98][Tho98].

Enterprise JavaBeans define uma infra-estrutura completa para a construção de aplicações multicamada, portáveis, escaláveis, flexíveis, transacionais e seguras. Assim como CORBA, EJB é apenas uma especificação e depende da adoção e desenvolvimento de terceiros.

---

[18] Operações utilizadas no controle de transações; *start* inicia uma transação; *commit* efetiva a transação, ou seja, todas as operações de modificação executadas entre *start* e *commit*; *rollback* cancela todas as operações executadas desde o último *start.*

4.3.4 CORBA

A demanda pela tecnologia de componentes fez a OMG estender CORBA para incorporar um modelo de componentes, o CCM ou CORBA *Component Model*. CORBA provê a infra-estrutura para a construção de aplicações baseadas em objetos distribuídos independente de linguagem, plataforma e redes. Apesar de uma das maiores contribuições de CORBA referir-se aos mecanismos para interconexão de objetos distribuídos, também define uma série de serviços e *frameworks* para a construção de aplicações distribuídas, o que constitui uma plataforma de desenvolvimento. No entanto, o excesso e a complexidade dos mecanismos disponíveis à construção de aplicações CORBA têm dificultado a sua utilização. Neste sentido, componentes são uma evolução dos objetos CORBA existentes para facilitar o processo de desenvolvimento sobre CORBA.

Componentes complementam a arquitetura de CORBA no sentido de prover maiores facilidades no desenvolvimento de aplicações. O *framework* CCM suporta a definição, a geração de código automática, a integração e a implantação de componentes CORBA. *Containers* CORBA são definidos a partir dos serviços de CORBA; na realidade, estes serviços são encapsulados pelo *container* e dispostos aos componentes através de interfaces simplificadas, definidas pelo CCM. Desta forma, a infra-estrutura definida por CORBA, como os serviços de um ORB (POA, por exemplo) e os adicionais previstos por CORBA (transações, persistência, serviço de nomes, etc.), pode ser incorporada aos componentes de forma transparente e simplificada.

CCM foi concebido como uma iniciativa de combinar o modelo de componentes JavaBeans e a arquitetura CORBA, para simplificar a programação de aplicações CORBA através do suporte a abstrações similares a Beans, além do emprego de ferramentas de desenvolvimento do tipo IDE. Como o padrão CORBA é voltado à construção de aplicações para a parte servidor, a OMG decidiu adotar o modelo Enterprise JavaBeans, que também é um modelo de componentes servidores [OMG99]. A especificação final de CCM é um superconjunto do modelo EJB [OMG99] que inclui inúmeras melhorias e alguns recursos adicionais, como o suporte a eventos (CORBA Notification Service); além disso, CCM define um mapeamento para o modelo EJB que garante a compatibilidade entre componentes CORBA e EJB. Há vários cenários de integração entre os modelos CCM e EJB: componentes CORBA executados sobre servidores CORBA podem se comunicar com componentes EJB executados em servidores EJB, ambos os componentes podem funcionar num mesmo servidor CORBA, e por fim, um componente CORBA pode funcionar sobre um servidor EJB [OMG99].

Por ser baseada no modelo EJB, o modelo de componentes de CORBA apresenta uma série de funcionalidades similaridades; porém, como CCM foi especificado como uma extensão da arquitetura CORBA, há uma série de diferenças na especificação das funcionalidades; em alguns casos, porque CCM se baseia na infra-estrutura da plataforma CORBA (serviços de transação, de nomes, de persistência, de eventos, etc.); em outros, para suplantar as deficiências inerentes a EJB. Um exemplo de serviço do núcleo de CORBA que é preservado no modelo CCM é a especificação de POA (*Portable Object Adapter*), que oferece mecanismos para o controle de ciclo de vida de objetos, ou como criá-los, ativá-los, desativá-los e destrui-los.

Um componente CORBA é especificado através de CIDL (*Component Implementation Definition Language*), uma extensão da tradicional linguagem de definição de interfaces de CORBA, a IDL [OMG99]. CIDL, assim como IDL, permite especificar as interfaces e outras características do componente através de uma linguagem universal, independente de plataforma e

linguagem de implementação. Um IDE com suporte a CCM deverá gerar código automaticamente, que inclua os *stubs, skeletons* e a estrutura do código do componente numa linguagem compatível com CORBA. O papel do desenvolvedor é o de codificar a lógica do componente a partir do código gerado pelo IDE. Diferente de EJB, cujas interfaces são definidas em RMI e implementadas em Java, um componente CORBA pode ser implementado sobre qualquer linguagem com suporte a CORBA e ao CCM. Componentes CORBA ainda podem empregar eventos através de uma interface simplificada do serviço de notificações padrão de CORBA, especificada pelo modelo CCM. Além dos tipos de componentes *session* e *entity* definidos por EJB, CCM define componentes do tipo *process*, que é uma categoria intermediária entre *session* e *entity beans,* e *empty,* que não emprega as facilidades providas pelo *container* CORBA, e sim os serviços padronizados pela plataforma CORBA [OMG99]. *Containers* CORBA também controlam o ciclo de vida de componentes CORBA (assim como *containers* EJB), além dos aspectos relacionados a transações, segurança, persistência e serviço de nomes. Da mesma forma que EJB, componentes CORBA são configurados através do seu *deployment descriptor,* que define aspectos relacionados ao suporte a transações, segurança e ciclo de vida de cada componente, e distribuídos juntamente com o código compilado em arquivos ZIP.

O modelo de componentes de CORBA é um dos mais completos da atualidade, pois combina as vantagens de EJB com as da plataforma CORBA. Em seu histórico, CORBA consta como mais um padrão bem definido, *de direito*, mas não *de facto*, devido às dificuldades envolvidas na sua aprendizagem e utilização. O modelo de componentes CORBA é uma extensão voltada à simplificação do processo de desenvolvimento, pois define uma abstração mais elaborada (componentes), encapsula os detalhes dos serviços previstos pela arquitetura CORBA através de interfaces simplificadas, e define mecanismos para que ferramentas IDE ofereçam suporte à geração de código e manipulação de componentes através de programação visual. Desta forma, ainda é necessária a adoção do CCM por parte da indústria e dos distribuidores de soluções CORBA, como os desenvolvedores de ORBs; para tanto, espera-se que os ORBs atuais sejam estendidos em servidores/*containers* CORBA.

4.3.5 Modelos de Componentes *Ad Hoc*

Apesar das especificações apresentadas anteriormente oferecerem uma excelente infra-estrutura para a construção de componentes, o emprego da abordagem de componentes não deve ser condicionado à disponibilidade de ferramentas compatíveis com os modelos apresentados anteriormente; além disso, pode haver ocasiões onde se deseja aplicar a metodologia de componentes, independente do modelo. Nos casos em que não há ferramentas disponíveis para suportar a adoção de um modelo de componentes, ou ainda nas ocasiões em que se deseja construir aplicações e componentes independentes de modelo, pode-se adotar modelos próprios, ou *ad hoc*.

Neste trabalho, por exemplo, decidimos implementar os componentes de forma independente, mesmo porque não havia disponível nenhuma implementação de servidor de aplicação compatível com os modelos de componentes de Java e CORBA.

4.4 Desenvolvimento de *Software* através de Componentes

Desenvolver *software* através de componentes significa construir aplicações a partir da combinação de componentes existentes; este processo reflete mudanças na tecnologia, no comportamento, na disciplina e na metodologia empregada no desenvolvimento. Para tanto, deve

ser assistido por um planejamento de reutilização de *software* que especifique uma política de desenvolvimento adequada. A organização que adota componentes como plataforma de reutilização de *software* deve estabelecer em sua política de desenvolvimento quando comprar componentes de terceiros, quando desenvolvê-los por meios próprios, quais *softwares* devem empregar a metodologia de componentes e qual a infra-estrutura adotada para o emprego de componentes.

Reutilização de *software* é um fator desejável, pois facilita o desenvolvimento de *software* correto, eficiente e barato. Na prática, as organizações têm revelado muitas dificuldades em adotar uma estratégia de reutilização de *software*, que não se limitam à infra-estrutura tecnológica, pois estendem-se a fatores econômicos, políticos e psicológicos. Embora haja uma série de tecnologias voltadas à reutilização de *software* disponíveis e em desenvolvimento, ainda são necessários muitos esforços para construir *software* reutilizável; em geral, a complexidade destas tecnologias dificulta a sua aprendizagem e utilização. Com relação aos fatores econômicos, há muitas dificuldades em estimar a recuperação dos investimentos referente à reutilização de *software*, um fator determinante para a alocação de recursos a projetos de desenvolvimento de *software* reutilizável. Dentre os fatores políticos, inclui-se o receio de compartilhar componentes de *software* com outras organizações, possivelmente concorrentes. Por fim, os aspectos psicológicos referem-se ao preconceito em (re)utilizar *software* de terceiros. Embora estes problemas sejam inerentes à disciplina de desenvolvimento de *software*, devem ser considerados na implantação de uma estratégia de reutilização de *software*.

O processo de desenvolvimento através de componentes pode ser classificado em duas etapas: o desenvolvimento de componentes reutilizáveis e a construção de aplicações a partir da reutilização de componentes existentes. Este processo permite dividir as responsabilidades do desenvolvimento de *software,* pois enquanto uma equipe de engenheiros de *software,* analistas, programadores e especialistas em determinados domínios de aplicação lidam com o projeto de componentes, outras equipes podem trabalhar na construção de aplicações a partir dos componentes existentes. O desenvolvimento de componentes não é tarefa trivial, pois envolve uma série de dificuldades para se conceber, projetar e testar componentes que atendam às premissas de reutilização, flexibilidade e estabilidade de suas interfaces. Desenvolver aplicações através de componentes ainda depende da disponibilidade dos componentes e de mecanismos para a sua descoberta, recuperação, aprendizagem e configuração.

4.4.1 Desenvolvimento de Componentes

O processo de desenvolvimento de componentes envolve as seguintes atividades: a concepção de funcionalidades recorrentes que podem ser encapsuladas num componente, a escolha das tecnologias para implementação (modelo de componentes, arquitetura de *software* e linguagem de implementação), a definição das partes flexíveis do componente, os testes para garantir seu funcionamento em configurações imprevistas e a manutenção das suas funcionalidades e interfaces.

Determinar as funcionalidades que um componente deve implementar ou definir se um conjunto de funcionalidades é compatível com o paradigma de componentes não são tarefas triviais. Em geral, há dois cenários em que componentes são especificados: ou a aplicação pode ser segmentada em componentes (com potencial de reutilização em outras aplicações) no sentido de promover a modularidade do sistema; ou a aplicação requer um conjunto de funcionalidades existentes sob a forma de componentes, ou tipicamente disponíveis através de componentes, como é o caso de componentes de interface gráfica, mecanismos de acesso a dados e algoritmos

para problemas bem conhecidos (ordenação, criptografia, etc.). Definir se uma solução é compatível com o paradigma de componentes depende do domínio da aplicação, do seu potencial de reutilização e das estimativas de retorno de investimento. Uma solução fortemente dependente do seu domínio de aplicação e sem perspectivas de reutilização constitui um contra-exemplo ao emprego de componentes; um algoritmo de criptografia confidencial e de uso limitado constitui um bom exemplo de aplicação não "componentizável". Componentes são recomendados para os casos em que há potencial de reutilização. No entanto, ainda é necessário definir bem o escopo das funcionalidades dos componentes para facilitar a sua (re)utilização; algoritmos de criptografia públicos são bons candidatos a componentes, por exemplo; neste caso, deve-se limitar o escopo das funcionalidades de cada componente: ao invés de definir um único componente que implemente todas as categorias de algoritmos de criptografia, é prudente especificar um componente que incorpore algoritmos de chave pública/privada, outro que suporte funções de *hash,* e assim por diante.

A escolha da tecnologia de infra-estrutura para a construção de componentes determina a forma como são projetados, implementados, distribuídos e utilizados. O modelo de componentes, por exemplo, define fatores como portabilidade, linguagens de implementação disponíveis, *middleware* para integração de componentes distribuídos, etc. Há duas categorias de modelos de componentes voltadas a contextos diferentes de utilização: clientes, como JavaBeans e COM/ActiveX, e servidores, como Enterprise JavaBeans e modelo de componentes CORBA. Para escolher a tecnologia adequada deve-se avaliar uma série de fatores, como a facilidade de aprendizagem e uso, as ferramentas de suporte ao desenvolvimento disponíveis, a portabilidade e o suporte a padrões.

Além das tecnologias de implementação, existem ferramentas que auxiliam a reutilização de componentes através de mecanismos de catalogação (repositórios) e documentação. Repositórios são ferramentas que oferecem suporte ao armazenamento e recuperação do código de componentes, além de mecanismos para o controle de versões. Segundo [Kar98], repositórios são úteis à medida que o número e a complexidade de componentes aumentam, pois permitem representar meta-dados que auxiliam na utilização e na manutenção de componentes. Documentação é essencial no desenvolvimento de aplicações através de componentes, pois informa o que um componente faz e como é possível configurá-lo e utilizá-lo. A documentação de componentes pode ser representada tanto em manuais que incluam diagramas UML (*Unified Modeling Language*), que oferece suporte à representação de componentes, quanto em metadados incorporados em repositórios. A qualidade da documentação é essencial para o desenvolvimento de aplicações através de componentes, pois garante que os componentes são (re)utilizados por programadores sem a intervenção dos seus projetistas.

Uma das questões suscitadas durante a concepção de componentes refere-se a sua flexibilidade ou às opções de configurações que devem oferecer. Como um componente deve ser reutilizado em ocasiões imprevistas, é necessário oferecer mecanismos de configuração para que o usuário/programador possa moldar o componente conforme suas necessidades. Um componente pode suportar configuração através de mecanismos de programação, como herança e polimorfismo, além de recursos integrados a ambientes de desenvolvimento, como propriedades e arquivos de configuração. No caso de configuração por programação, o usuário/programador redefine classes e métodos e implementa o comportamento adicional que o componente deverá incorporar. Quanto à configuração de propriedades e arquivos de configuração, é normalmente realizada através de IDEs com editores de propriedades e *wizards.* Em ambos os casos, o projetista implementa o componente sobre métodos, classes e propriedades de forma genérica; ao

configurar propriedades e redefinir métodos e classes, o componente funciona a partir das informações e comportamentos redefinidos pela configuração. Portanto, definir os pontos flexíveis de um componente, também conhecidos como *hot spots,* é muito importante, pois à medida que se aumenta a flexibilidade, aumenta-se o escopo de reutilização.

Além das atividades de projeto e implementação, construir componentes ainda envolve os processos de testes, distribuição e manutenção. Testar componentes é um ponto crucial deste processo de desenvolvimento, devido à flexibilidade embutida em componentes que permite utilizá-los em ocasiões imprevistas (inclusive através de plataformas e redes). Como a maioria dos componentes são configuráveis, é improvável testar todas as alternativas possíveis de configuração.

A abordagem de construir componentes de *software* ou aplicações através de componentes envolve decisões acerca da forma de sua distribuição. Componentes construídos para uso próprio, ou seja, para serem utilizados sob a mesma estrutura organizacional, são geralmente distribuídos com o código fonte. Por outro lado, pode haver casos em que o desenvolvedor do componente deseja comercializá-lo, cenário em que é comum distribui-lo sob formato binário, sem código fonte disponível. Componentes distribuídos com código fonte são conhecidos como *white-box*, e aqueles distribuídos sob formato binário, como *black-box*, em alusão à transparência do seu código fonte. A idéia dos componentes *black-box* é garantir ao desenvolvedor a propriedade sobre o código fonte, fator imprescindível para a comercialização de *software* em larga escala.

Por mais elaborado que seja, um componente pode apresentar falhas de execução ou incompatibilidades em determinados domínios de aplicação; nestes casos, é necessário corrigir os problemas e ajustar suas interfaces para que seja utilizado em outros domínios de aplicação. Em geral, a manutenção de componentes ocorre em várias frentes: adição e remoção de funcionalidades a partir da atualização das operações nas interfaces, generalização ou especialização de *hot spots*, identificação e atualização dos componentes afetados pela propagação de modificações e controle de versões. Vale ressaltar a importância de manter as interfaces de um componente estáveis para amenizar o impacto das modificações nos demais componentes da aplicação. A idéia é adicionar novas operações e manter as já existentes para garantir a compatibilidade do componente com as aplicações que o utilizam.

4.4.2 Desenvolvimento de Aplicações Baseadas em Componentes

Uma aplicação baseada em componentes é aquela construída sobre uma base de componentes existentes. Ao identificar a demanda por funcionalidades recorrentes, pode-se adquirir componentes existentes ou optar pelo seu desenvolvimento. O desenvolvimento de *software* a partir de componentes envolve tanto programação convencional, para suprir a lógica de integração de componentes, quanto a recuperação, aprendizagem e configuração de componentes.

A construção de aplicações através de componentes difere dos processos de desenvolvimento convencionais numa série de fatores. A equipe de desenvolvimento pode ser segmentada, pois enquanto engenheiros, analistas e especialistas do domínio da aplicação projetam componentes, programadores reutilizam os componentes para formar aplicações. Como discutido anteriormente, modelos de componentes oferecem diversos serviços sobre os quais são construídos componentes; desta forma, os projetistas podem se concentrar na lógica do componente, pois não há necessidade de implementar serviços básicos de infra-estrutura. No caso de modelos de componentes servidores, como CORBA e Enterprise JavaBeans, não é necessário implementar tratamento de transações e ciclo de vida, por exemplo. Os esforços de

programação consistem em prover a infra-estrutura para integração dos componentes que constitui a lógica da aplicação, além de configurar os componentes reutilizados a partir de programação, através de herança e polimorfismo, e edição de propriedades, através de arquivos de configuração.

Como a maioria dos modelos de componentes define suporte a ambientes de desenvolvimento, e como IDEs permitem construir aplicações com menos esforços de programação a partir de programação visual, é fundamental utilizá-los como ferramentas de apoio à construção de *software* baseado em componentes. Em geral, IDEs permitem incorporar novos componentes e torná-los disponíveis à construção de aplicações. Repositórios que permitem catalogar componentes, além de ferramentas para modelagem de aplicações que permitem gerar código e realizar engenharia reversa também são indicadas. Atualmente há iniciativas em todas as frentes: IDEs com suporte aos modelos de componentes mais difundidos, repositórios com suporte a metadados e integração com ferramentas de modelagem, UML para a modelagem de componentes, etc.

Para reutilizar um componente é necessário recuperá-lo, compreendê-lo e configurá-lo. Para tanto, deve-se empregar um repositório ou algum mecanismo de um IDE para a descoberta e inserção do componente no *container* (ou área) de trabalho. Para utilizar suas interfaces, é necessário conhecer a sintaxe e a semântica de cada operação a partir da documentação incorporada ao componente. Os *hot spots* do componente, bem como as formas de sua configuração também devem constar na documentação. Ou seja, para que um componente possa ser reutilizado, é imprescindível que haja mecanismos eficientes de recuperação e que a qualidade da documentação seja satisfatória. Vale ressaltar que nem todos os componentes utilizados numa aplicação são construídos pela própria equipe ou pela própria organização; podem existir componentes de terceiros, o código fonte pode não estar disponível (componentes binários), a equipe de desenvolvimento pode não estar acessível, e assim por diante.

As iniciativas de documentação de componentes têm incorporado informações textuais, diagramas referentes a modelagem e outros mecanismos que auxiliam a compreensão de modelos de classes e objetos como o de *design patterns* apresentados em [GHJ94] e *metapatterns*, propostos por [Pre94]. Neste trabalho, seguimos a abordagem de catalogar componentes como o são *patterns,* segmentando a documentação de um componente nas seguintes seções: *estímulo, funcionamento, flexibilidade, projeto, implementação, interface e prova de conceito. Estímulo* apresenta o contexto em que o componente foi concebido, ou os cenários que suscitaram o seu desenvolvimento; *funcionamento* descreve o esquema padrão de utilização do componente; *flexibilidade* descreve os *hot spots* do componente e como configurá-los; *projeto, implementação* e *interface* documentam as decisões de projeto e implementação, bem como as operações do componente, que são informações essenciais para a compreensão da semântica do componente e de como estendê-lo através de mecanismos de programação; por fim, *prova de conceito* demonstra alguns exemplos de configuração e utilização do componente.

Independente do meio de publicação (IDEs, repositórios, sistema operacional, etc.), para que componentes sejam reutilizados, o usuário/programador de aplicações deve conhecer o comportamento do componente, seu modelo, as operações definidas pelas suas interfaces, os eventos gerados e tratados, os *hot spots* disponíveis e as formas de configuração. Embora os IDEs atuais ofereçam suporte a eventos e configuração de propriedades, não há mecanismos de documentação das operações das interfaces, nem dos meios de configuração através de programação; a representação dos eventos manipulados por um componente também não é clara.

Desta forma, é imprescindível que haja formas complementares de documentação para que um componente seja efetivamente reutilizado.

A iniciativa de catalogar e documentar componentes, além do emprego de ferramentas do tipo repositório para o armazenamento e recuperação, torna-se essencial à medida que o número e a complexidade de componentes aumentam. É comum, atualmente, incorporar componentes diretamente aos IDEs; no entanto, se considerarmos a utilização de IDEs de vários fornecedores, ou um ambiente de desenvolvimento distribuído com múltiplos usuários/programadores, esta configuração torna-se inviável, porque centraliza as informações sobre componentes em ferramentas específicas sob formatos proprietários, inviabilizando o acesso aos componentes através dos demais IDEs. Ainda há casos, como ocorre com OLE/ActiveX na plataforma Windows, em que componentes são catalogados pelo sistema operacional sem transparência para o usuário, que não toma conhecimento da sua instalação e disponibilidade. Portanto, acreditamos que iniciativas como as de repositórios sejam o meio mais apropriado para a publicação de componentes; repositórios podem ser instalados em servidores acessíveis a toda a comunidade de desenvolvimento, além de incorporar componentes e metadados em formatos padronizados que podem ser integrados a IDEs e ferramentas de modelagem.

Um aspecto importante a considerar na construção de aplicações através de componentes refere-se ao desempenho, que pode ser inaceitável em alguns domínios. Como a flexibilidade dos componentes é alcançada a partir de mecanismos de indireção e polimorfismo (*dynamic binding*, por exemplo), há uma redução na eficiência da aplicação. A arquitetura de componentes e *frameworks* emprega classes abstratas para a representação genérica de funcionalidades que podem ser estendidas através de programação, além de estruturas de desacoplamento para facilitar a troca de comportamento em tempo de execução. Nestas arquiteturas, é comum haver uma propagação de mensagens (chamadas a métodos) através dos vários objetos que compõem um componente ou *framework*. Por outro lado, componentes servidores implementados sobre modelos de componentes como Enterprise JavaBeans e CORBA *Component Model* podem se beneficiar dos mecanismos de desempenho previstos pelos seus *containers*. Como um servidor EJB ou CORBA controla o ciclo de vida de componentes, compartilha recursos do sistema (*threads*, conexões com bancos de dados, conexões de rede, etc.) e implementa suporte ao balanceamento de carga, o desempenho é um fator favorável ao uso de componentes. Portanto, deve-se considerar o fator desempenho durante a definição do que pode se tornar componente, sobretudo nos casos em que o modelo de componentes não prevê recursos de escalabilidade.

No cenário de modelos de componentes como CORBA ou EJB um servidor controla o ciclo de vida de um componente iniciando um número predefinido de instâncias em memória, compartilhadas por um determinado número de usuários; conforme a variação da demanda, o servidor cria e descarta instâncias dos componentes para racionar o uso dos recursos do sistema (memória, processos, etc.). Compartilhar recursos permite aumentar a carga de processamento; uma conexão com um banco de dados, por exemplo, pode ser compartilhada sem perda de informações sobre transações, pois são controladas pelo próprio servidor. O balanceamento de carga permite criar instâncias de componentes em outros servidores e distribuir as requisições de serviços a componentes aos servidores disponíveis segundo alguma política de distribuição de carga.

4.5 Conclusões

O paradigma de componentes corresponde a uma evolução da orientação a objetos e provê um maior nível de abstração para os conceitos de modularidade e reutilização. Para que o potencial de componentes seja alcançado deve haver um planejamento que identifique uma política de reutilização de *software*, abordando questões como adquirir componentes existentes ou construir novos componentes internamente, que tipo de *software* pode ser tornado componente e quais os modelos de componentes adotados.

O processo de desenvolvimento de *software* através de componentes pode ser dividido em duas frentes: construção de componentes e de aplicações. Ao desenvolver componentes, deve-se escolher um modelo de componentes apropriado, gerar metadados que informem as operações, os eventos, os *hot spots* e as formas de configuração do componente, e se possível, incorporar o código e os metadados referentes ao componente em repositórios disponíveis aos IDEs utilizados na construção de aplicações. Ao desenvolver aplicações, deve-se empregar IDEs compatíveis com os modelos de implementação dos componentes e repositórios para a recuperação de código e metadados.

Capítulo 5

Implementação

O levantamento das aplicações *Web* mais comuns em *intranets* permitiu identificar uma série de funcionalidades recorrentes. O estudo sobre as arquiteturas de *software Web* apontam para a construção de sistemas distribuídos baseados em camadas e componentes. Este trabalho serviu de base para a concepção e implementação dos componentes de *software* implementados nesta tese, os quais apresentamos neste capítulo.

Apesar dos modelos de componentes definirem uma série de serviços de infra-estrutura, pode não haver servidores de aplicações disponíveis, ou um servidor não implementar os serviços necessários; nestes casos, procede-se com a adoção de modelos *ad hoc* de componentes. Neste trabalho adotamos um modelo *ad hoc* e a arquitetura de *software* baseada em camadas. Como um dos objetivos iniciais deste trabaho era o de construir aplicações para *intranets* baseadas em componentes, optamos pela implementação de componentes de infra-estrutura que formam a camada de base da aplicação desenvolvida para a prova de conceito, a INEX, apresentada no capítulo 6.

O levantamento sobre os tipos de aplicações e as necessidades mais comuns em *intranets* aponta o gerenciamento de informações, o controle de acesso e a personalização de conteúdo como funcionalidades imprescindíveis. Enquanto o gerenciamento de informações pode ser considerado um requisito de alto nível, e eventualmente implementado através de um *framework*, o controle de acesso e a persistência de informações são funcionalidades típicas de infra-estrutura, pois podem ser reutilizadas tanto na implementação de um *framework* para gerenciamento de informações, quanto em outras aplicações.

Neste sentido, dividimos os trabalhos de implementação deste projeto em duas etapas: na concepção e implementação de componentes de infra-estrutura que focassem na persistência e no controle de acesso de informações, cujos resultados são apresentados neste capítulo; no projeto de um sistema para gerenciamento de informações que pudesse ser implementado sobre os componentes de infra-estrutura, a INEX, apresentada no capítulo 6.

Como discutimos no capítulo 4, conceber componentes não é tarefa trivial; uma vez definido o domínio de aplicação do componente, ainda é necessário definir uma série de questões pertinentes ao modelo de componentes, arquitetura, forma de distribuição, interface, *hot spots*, etc. Neste sentido, iniciamos os trabalhos de implementação a partir de uma prova de conceito, realizada através do desenvolvimento de um componente simples que pudesse promover tanto o aprendizado do processo quanto a prática na construção de componentes de infra-estrutura. O componente *Log* foi desenvolvido como prova de conceito e serviu de base para o estabelecimento do processo de desenvolvimento dos demais componentes.

O componente *Log* implementa facilidades para a depuração de aplicações e pode ser configurado para persistir *logs* em diferentes meios de armazenamento, como bancos de dados, arquivos *flat* e consoles gráficos. Após algumas iterações no projeto dos componentes, identificamos a possibilidade de se empregar o componente de banco de dados para realizar a persistência de *logs* em banco de dados, e o componente *dispenser* para gerenciar o compartilhamento do serviço de *Log* entre clientes remotos, como sugere a dependência entre os componentes *Log*, BD e *Dispenser* na Figura 5.6.

O componente de acesso a banco de dados, ou componente BD, é um dos principais componentes implementados neste trabalho e oferece um conjunto de serviços para realizar a persistência de objetos Java em bancos de dados relacionais que ofereçam suporte a Java através da API JDBC. A idéia sob o componente BD é isolar os detalhes de infra-estrutura relacionados a persistência das informações de uma aplicação. Desta forma, qualquer aplicação Java com requisitos de persistência de objetos pode delegar ao componente BD todas as operações de inserção, remoção atualização e recuperação de objetos, sem que seja necessário codificar cláusulas SQL ou manipular conexões com bancos de dados.

Durante o projeto do componente BD identificamos uma série de funcionalidades relacionadas ao compartilhamento e controle do ciclo de vida de conexões com bancos de dados; a idéia era a de se empregar um mecanismo de pool de conexões para aumentar o desempenho e a flexibilidade do componente BD. Neste sentido, identificamos a possibilidade de reutilizar as facilidades de compartilhamento de recursos e os algoritmos de alocação em outros contextos e aplicações, o que permitiu desenvolver um novo componente: dispenser. A idéia do componente dispenser é promover o compartilhamento de recursos (objetos Java, conexões com bancos de dados, sockets, etc.), através da implementação de facilidades para a criação, manutenção e acesso a coleções de recursos. Após sua implementação, o componente dispenser foi integrado aos componentes BD, para suportar o mecanismo de pool de conexões com bancos de dados, e ao componente Log, como discutido acima.

O componente de segurança foi concebido a partir dos requisitos de controle de acesso às informações de uma *intranet* e envolve duas frentes: a autenticação de usuários e a autorização de acesso a recursos. A idéia do componente de segurança é isolar os detalhes de infra-estrutura relacionados a autenticação e autorização de acesso da lógica das aplicações. Como o componente de segurança requer persistência de informações relacionadas às listas de controle de acesso a recursos (ACLs e usuários), projetamos o componente sobre um banco de dados relacional e utilizamos o componente BD para executar a persistência das ACLs e usuários.

5.1 Componente *Log*

O componente *Log* oferece um conjunto de serviços para o registro de mensagens de *logging* geradas por aplicações. É possível classificar eventos de *log* segundo um conjunto de categorias

pré-definidas, além de registrá-las em diferentes meios de armazenamento, como em arquivos, bancos de dados ou num console gráfico da própria aplicação.

O emprego de mecanismos de *log* é fundamental na identificação e correção de problemas (depuração), na realização de métricas de utilização e no gerenciamento de falhas. O propósito do componente *Log* é isolar as aplicações dos detalhes de implementação dos mecanismos de *log*, e oferecer aos usuários/programadores uma interface uniforme e intuitiva para o registro de mensagens. Uma vez configurado e/ou customizado, o componente permite padronizar o formato das mensagens geradas pelas aplicações - independente da aplicação e dos desenvolvedores que a constróem.

5.1.1 Estímulo

Como as aplicações *Web* são inerentemente distribuídas, é necessário registrar eventos gerados tanto na porção cliente da aplicação, quanto na parte servidor. Desta forma, é possível determinar se uma falha, por exemplo, ocorre por problemas localizados no cliente ou no servidor. Um mecanismo de *log* é uma ferramenta que pode ser analisada sob diferentes circunstâncias para medir desempenho, detectar falhas, reestruturar as aplicações, ou simplesmente colher dados sobre o funcionamento dos serviços implantados.

Assim como os padrões de projeto definidos pela arquitetura de *software*, é desejável empregar um formato de *log* uniforme. Além de facilitar a interpretação dos *logs*, é possível programar ferramentas de gerenciamento de falhas, como o HP-Open View por exemplo, na monitoração das aplicações em ambiente de produção. Uma ferramenta deste tipo pode ser configurada para pesquisar os registros de *log* em busca de determinados padrões de *strings*, executar procedimentos em resposta a determinados eventos de *log,* como enviar *e-mails*, mensagens de texto em celulares e *pagers*, mostrar alertas em consoles de gerenciamento, etc.

A natureza multiplataforma das aplicações *Web* dificulta o registro de informações de maneira uniforme: por serem distribuídas, dificultam a coleta de informações sobre a execução nos clientes. Neste sentido, concebemos um componente *Log* multiplataforma, distribuído e configurável. Por ser implementado sobre Java, pode ser empregado em qualquer plataforma com suporte a JVM; como emprega uma interface RMI, pode ser utilizado remotamente, inclusive a partir da porção cliente das aplicações; por fim, pode ser configurado para refletir o padrão de *log* definido pelo usuário/programador.

5.1.2 Funcionamento

O funcionamento do componente *Log* é baseado numa interface Java RMI e numa classe de dados[19] que armazena as informações da mensagem de *log*. A sequência de utilização do componente pode ser descrita como segue:

1. Obtém-se uma referência ao serviço de *log*, representada pela interface do componente;
2. Define-se, opcionalmente, repositórios de armazenamento e filtros para roteamento de mensagens;
3. Cria-se um objeto para encapsular a mensagem de *log*;
4. Preenche-se a mensagem no objeto criado no passo 3
5. Chama-se o serviço de registro de *log* passando o objeto com a mensagem de *log* como parâmetro.

---

[19] Tradução de *entity class* - um estereótipo UML para definir uma classe que representa informações persistentes [BRJ98a].

6. O componente recebe a requisição de registro da mensagem de *log*, determina quais repositórios devem receber a mensagem, e efetiva o registro da mensagem nos repositórios compatíveis com a configuração do componente e com os filtros definidos pelo cliente.

Uma mensagem de *log*, e portanto, a classe de dados que a representa, possuem o seguinte formato:

- Data do evento
- Nome da máquina em que é executada a aplicação que gerou o evento
- Nome e número do processo que gerou o evento
- Usuário da aplicação que gerou o evento
- Descrição dos detalhes do evento
- Categoria do evento, segundo a classificação proposta por [AR97]:
  - *Debug:* corresponde às mensagens definidas pelo programador da aplicação para auxiliar o processo de depuração.
  - *Info:* indica que a mensagem de *log* é apenas uma informação registrada pela aplicação, como uma notificação de utilização de um serviço, ou uma mensagem de sucesso na execução de uma operação.
  - *Warning:* indica que uma operação foi executada com alguma falha.
  - *Error:* indica que uma operação não pôde ser executada, embora não tenha afetado o funcionamento da aplicação.
  - *Fatal:* indica uma falha na execução de uma operação que compromete o funcionamento da aplicação.

5.1.3 Flexibilidade

O componente *Log* oferece os seguintes mecanismos para configuração e extensão das suas funcionalidades:

1. É possível redefinir o formato da mensagem de *log*;
2. Ao definir os repositórios de *log*, pode-se optar pelos tipos de armazenamento de mensagens de *log default* (console, banco de dados e arquivo texto), ou estender o componente através da inclusão de novos tipos de armazenamento;
3. Uma aplicação pode definir vários repositórios de *log*, inclusive repositórios diferentes do mesmo tipo (dois arquivos texto, por exemplo);
4. Há suporte a definição de filtros, um mecanismo que permite rotear mensagens de *logs* para diferentes repositórios;
5. Como o componente *Log* oferece suporte a objetos distribuídos, é possível utilizá-lo remotamente.

Uma das principais flexibilidades do componente *Log* refere-se aos repositórios para registro de *logs*; inicialmente, foram implementados suporte a arquivos texto (*flat*), bancos de dados relacionais e janela de aplicação (console). Ao utilizar o componente, é possível definir um ou mais repositórios; ao classificar os eventos de *log* segundo as categorias *Info, Debug, Warning, Error e Fatal*, é possível definir filtros para rotear os *logs* para diferentes repositórios. Caso o usuário/programador julgue necessário, ainda é possível estender as funcionalidades do componente através da redefinição do formato da mensagem de *log* e da implementação de novos tipos de repositório.

5.1.4 Projeto

Em tese, um mecanismo de *log* pode ser utilizado tanto por aplicações locais, quanto remotas. Inicialmente, implementamos uma versão local do componente, restrita às aplicações residentes no mesmo local e contexto de execução do componente. Para suplantar esta limitação, realizamos o porte do componente para uma versão distribuída através de RMI.

Além da redefinir a interface do componente para uma versão remota, e adaptar o código do componente para tratar as novas funcionalidades de integração ao ORB RMI, registro no serviço de nomes e tratamento de exceções remotas, foi necessário atualizar o componente para implementar o tratamento à concorrência de acesso.

Um cliente do componente pode utilizar tanto os repositórios pré-definidos quanto definir novos; nestes casos, o componente mantém uma sessão para cada cliente, onde são armazenadas as informações de configuração dos repositórios e filtros criados pelo cliente. Quando um cliente encerra a utilização do componente, todas as informações da sessão do cliente são descartadas.

Como o enfoque deste trabalho é o de demonstrar a construção e (re)utilização de componentes de *software* independente da infra-estrutura e dos serviços dos modelos de componentes, empregamos uma arquitetura de componentes *ad hoc* (capítulo 4.3.5). A arquitetura de *software* escolhida para a implementação do componente *log* foi a de objetos distribuídos baseados em Java, ou seja, RMI. Para possibilitar a customização do componente através da criação de novos tipos de repositórios (subclasses de `LogTarget`, Figura 5.1) foram empregados os padrões de projeto *bridge* e *factory method* [GHJ94]. Para permitir a customização da classe de dados que representa a mensagem de *log* (`LogEntry`, Figura 5.1) foram empregados mecanismos de introspecção de Java, para a descoberta dos atributos e valores que compõem a mensagem de *log* em tempo de execução - uma vez determinados os pares de atributos/valores da classe de dados, as implementações dos tipos de repositório *default* (banco de dados, arquivo texto e console) se encarregam de registrar ou mostrar a mensagem de *log*.

5.1.5 Implementação

A Figura 5.1 ilustra o modelo de classes do componente *Log*, no qual podemos destacar:

- A interface do componente representada pela interface Java `RMILog`;
- O ponto de acesso ao componente *Log* representado pela classe `LogComponent`;
- O mecanismo de desacoplamento entre o componente e os repositórios de *log*, representado pelo padrão de projeto *bridge*, envolvendo as classes `LogComponent` e `LogTarget`;
- Os tipos de repositórios *default* fornecidos pelo componente que permitem registrar *logs* em tabelas de bancos de dados relacionais (classe `LogDB`), arquivos texto (classe `LogFile`) e num console gráfico (classe `LogWindow`);
- O mecanismo que permite lidar com repositórios e filtros para cada cliente do componente, representado pela classe de associação `TargetRef`.

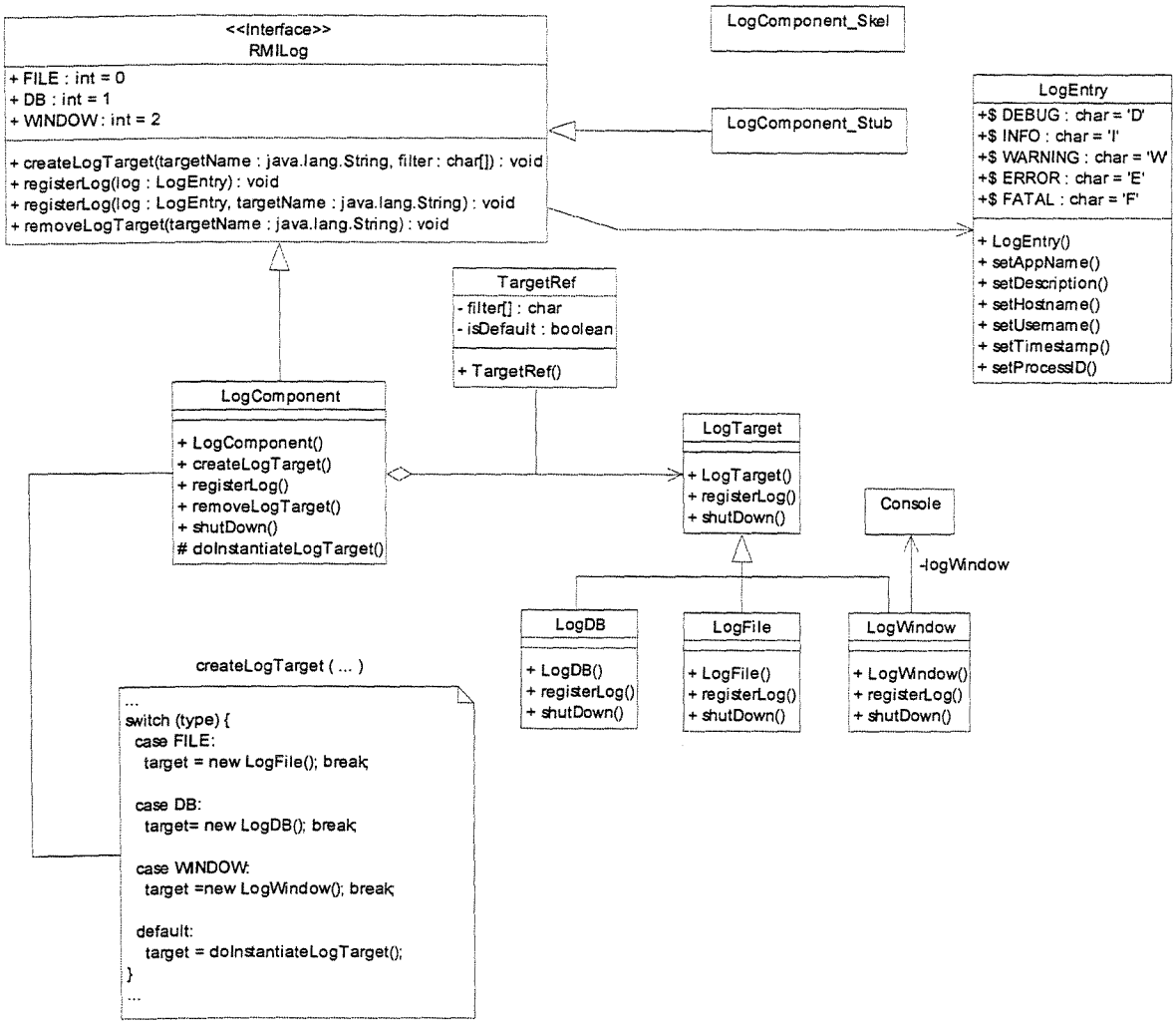


Figura 5.1 - Modelo de classes do componente *Log*

Embora não seja apresentado no modelo de classes da Figura 5.1, o suporte a novos tipos de repositórios também depende do padrão de projeto *factory method,* implementado sobre a classe **`LogComponent`**. A idéia é possibilitar a customização do componente através da definição de uma subclasse de **`LogComponent`** que sobrecarrega o método *hook* **`doInstantiateLogTarget()`** e implementa o código de tratamento dos novos tipos de repositórios, conforme discutido a seguir (Figura 5.1):

- A operação de criação/configuração de repositórios do componente *Log,* **`createLogTarget()`**, utiliza parâmetros para identificar o tipo de repositório solicitado pela aplicação cliente;
- Caso seja solicitado um dos tipos de repositório *default* (banco de dados, arquivo texto ou console), cria-se uma instância de alguma das subclasses pré-definidas de **`LogTarget`** (**`LogDB`**, **`LogFile`**, **`LogWindow`**);

- Caso seja um repositório criado a partir da customização do componente, delega-se a criação da instância do repositório ao método *hook* `doInstantiateLogTarget()`, que não contém nenhum tratamento na implementação original do componente; ao customizar o componente através de uma subclasse de `LogComponent`, sobrecarrega-se o método `doInstantiate LogTarget()`, incluindo o código para tratamento dos novos tipos de repositório.

A combinação dos padrões de projeto *bridge,* para desacoplar a lógica do registro de *log* do tipo de repositório utilizado, e *factory method*, para permitir a instanciação de novos tipos de repositório definidos em tempo de customização permite customizar o componente *Log*, incluindo novos tipos de repositório, sem modificar a lógica do componente.

5.1.6 Interface

A interface do componente *Log* é composta por 4 operações:

1. `createLogTarget(int type, String desc, String conf, char[] filter)`: habilita um repositório de *log* na sessão da aplicação cliente. É possível informar qual o tipo de repositório, o caminho do arquivo de configuração do repositório e o filtro de categorias de evento aplicadas sobre o repositório;
2. `removeLogTarget(String target)`: desabilita o repositório especificado da sessão do cliente; todas as operações de registro de *log* subsequentes desconsideram o repositório removido;
3. `registerLog(LogEntry entry)`: registra a mensagem de *log* em todos os repositórios ativos na sessão do cliente cujos filtros sejam compatíveis com a categoria do evento da mensagem de *log*;
4. `registerLog(LogEntry entry, String target)`: registra a mensagem de *log* somente no repositório especificado.

As constantes que identificam a categoria da mensagem de *log* (*Info, Debug, Warning, Error* e *Fatal*) são definidas na própria classe de dados `LogEntry`; já as constantes que indicam o tipo de repositório são definidas na interface `RMILog`.

Como o componente *Log* é distribuído, e como pode manter estado da sessão de diversos clientes, deve ser utilizado seguindo uma ordem de precedência de operações: a execução da operação 1 é pré-requisito para a execução das operações 2, 3 e 4; caso uma das operações 2, 3 ou 4 seja executada e o componente não possua nenhum repositório configurado, uma exceção é gerada em resposta à operação. Em outras palavras, deve-se configurar pelo menos um repositório para executar as operações de registro de *log* ou remoção de repositórios; caso todos os repositórios tenham sido removidos, as operações de registro de *log* levantarão exceções.

5.1.7 Prova de Conceito

Com o objetivo de demonstrar os mecanismos de extensão, além de exemplificar a utilização do componente, sugerimos executar a seguinte prova de conceito: "estender o componente *Log* para tratar um repositório do tipo SMS; toda mensagem de *log* enviada ao repositório SMS deverá ser enviada ao número do telefone celular indicado na mensagem de *log*". Neste exercício podemos identificar dois tipos de extensão: a criação de um novo tipo de repositório e a reformulação do *layout* da mensagem de *log*, que deverá incluir um número de telefone.

O roteiro a seguir ilustra os passos necessários à realização da prova de conceito:

- Estender a classe **LogTarget** através da subclasse **LogSMS** (Código 5.1), implementar as operações de inicialização e encerramento do repositório, além da operação de registro de *log* - todos os detalhes para tratamento de conexão e envio de mensagens SMS devem ser incluídos a partir da operação de registro de mensagens da classe **LogSMS**;

```
public class LogSMS extends LogTarget {
    ...
    public void shutDown() {...}
    public void sendSMS(String message, String phoneNumber) {...}
    public void registerLog(SMSLogEntry log) {
       String message = log.getAppName() + log.getDescription ...
       ...
       sendSMS(message, log.getPhoneNumber());
    }
}
```

Código 5.1 - Implementação do repositório SMS

- Estender a classe **LogComponent** através da subclasse **MyLogComponent** (Código 5.2), definir a constante do novo tipo de repositório (**SMS**, por exemplo), e redefinir o método *hook* **doInstantiateLogTarget()** para tratar a criação de instâncias do tipo **LogSMS**;

```
public class MyLogComponent extends LogComponent {
  public LogTarget doInstantiateLogTarget(int targetType) {
    switch(targetType) {
      case(SMSLogEntry.SMS):
        return new LogSMS();
      default:
        return null;
    }
  }
}
```

Código 5.2 - Implementação do método *hook* doInstantiateLogTarget

- Estender a classe **LogEntry** através da subclasse **SMSLogEntry** (Código 5.3) e definir os novos atributos da mensagem de *log*, como o número do telefone, por exemplo;

```
public class SMSLogEntry extends LogEntry {
  public static final int SMS = 3;
  public String phoneNumber;

  public String getPhoneNumber() { return phoneNumber; }
  public void setPhoneNumber(String pn) { phoneNumber = pn; }
}
```

Código 5.3 - Implementação da classe de dados SMSLogEntry

- Compilar as classes **LogSMS** e **MyLogComponent** e **SMSLogEntry** ;
- Criar uma aplicação cliente para utilizar o componente estendido, como a do Código 5.4. Um aplicação cliente do componente *Log* possui a seguinte estrutura:
- Obter uma referência para a interface remota do componente *Log*;
- Configurar os repositórios de *log* necessários;
- Criar um ou mais objetos do tipo **LogEntry** a partir das informações das mensagens de *log*.
- Registrar um ou mais objetos do tipo **LogEntry** através da interface remota do componente.

```
public class LogClient {
  public LogClient () throws java.rmi.RemoteException {
    RMILog log;              // interface do componente log
    SMSLogEntry logEntry;  // representação de uma mensagem de log
    char[] info = {logEntry.INFO,  logEntry.WARNING }; // filtros
    char[] error= {logEntry.ERROR, logEntry.FATAL };
    char[] debug= {logEntry.DEBUG };

    // Obtém uma referência ao serviço de log (stub)
    log= (RMILog)java.rmi.Naming.lookup("rmi://qwx.ahand.unicamp.br:LOG");
    log.createLogTarget(log.WINDOW, "myConsole", null, debug);
    log.createLogTarget(log.FILE, "myFile", null, info);
    log.createLogTarget(logEntry.SMS, "mySMS", null, error);

    logEntry= new SMSLogEntry("LogClient App", new Date());
    logEntry.setPhoneNumber("01192550436");
    logEntry.setLevel(LogLevel.INFO);
    logEntry.setDescription("Testando log de info");

    // Registra a mensagem de log em todos os repositórios,
    // cujos filtros sejam compatíveis com o nível INFO, i.e.,
    // no repositório do tipo arquivo definido por myFile
    log.registerLog(logEntry);

    // Registra a mensagem de log no repositório do tipo console
    // definido por myConsole
    logEntry.setLevel(LogLevel.DEBUG);
    logEntry.setDescription("Testando log de debug");
    log.registerLog(logEntry);

    // Registra a mensagem de log no repositório do tipo arquivo
    // definido por myFile
    logEntry.setLevel(LogLevel.WARNING);
    logEntry.setDescription("Testando log de warning");
    log.registerLog(logEntry);

    // Registra a mensagem de log no repositório do tipo SMS
    // definido por mySMS
    logEntry.setLevel(LogLevel.ERROR);
    logEntry.setDescription("Testando log de error");
    log.registerLog(logEntry);

    // Registra a mensagem de log no repositório do tipo SMS
    // definido por mySMS
    logEntry.setLevel(LogLevel.FATAL);
    logEntry.setDescription("Testando log fatal");
    log.registerLog(logEntry, "mySMS");

    log.removeLogTarget("myConsole");
    log.removeLogTarget("myFile");
    log.removeLogTarget("mySMS");
  }
}
```

Código 5.4 - Exemplo de utilização do componente *Log*

5.2 Componente de Acesso a Banco de Dados

O componente de acesso a banco de dados, ou simplesmente, componente BD, foi projetado para realizar persistência, pesquisa, alteração e remoção de objetos Java, independente da categoria da base de dados empregada (relacional, objeto-relacional, orientado a objetos, etc). Embora a arquitetura do componente suporte diferentes tipos de bancos de dados, a implementação realizada neste trabalho oferece suporte a bancos de dados relacionais compatíveis com a API JDBC.

O propósito do componente BD é oferecer suporte à implementação do modelo de arquitetura de *software* baseado em camadas (capítulo 3.2.2), onde é possível desacoplar as camadas que implementam a lógica da aplicação da camada que implementa persistência. Neste cenário o componente BD implementa a camada de infra-estrutura de persistência, sobre a qual são construídas as demais camadas das aplicações.

5.2.1 Estímulo

Em geral, as aplicações do tipo cliente/servidor *Web* utilizam os mesmos recursos de bancos de dados que os sistemas cliente/servidor tradicionais. É fato que as operações mais comuns nestes sistemas correspondem ao acesso a bancos de dados. Por outro lado, algumas arquiteturas de *software* como aquelas baseadas em server-side extensions são fortemente dependentes do sistema gerenciador de banco de dados. O componente BD foi concebido para atender tanto aos requisitos de acesso a banco de dados, quanto à necessidade de persistir informações independentemente dos servidores de banco de dados e da plataforma empregada.

5.2.2 Funcionamento

O funcionamento do componente BD é baseado numa interface Java e em algumas classes utilitárias para representação de critérios de pesquisa. Em linhas gerais, a sequência de utilização do componente pode ser descrita como segue:

1. Configura-se as fontes de dados sobre as quais o componente poderá atuar, indicando o nome do banco de dados, usuário e senha para conexão, nome do servidor, número de conexões que poderão ser utilizadas, etc;
2. Obtém-se uma referência ao componente, representada pela sua interface;
3. Cria-se uma transação indicando qual a fonte de dados a ser utilizada;
4. Executa-se uma das operações do componente: persistência, pesquisa, atualização ou remoção;
5. Encerra-se a transação se o resultado da operação 4 for satisfatório, ou cancela-se a transação, caso contrário.

5.2.3 Flexibilidade

Como o componente BD foi concebido para realizar persistência de objetos, foi projetado para suportar diferentes categorias de sistemas gerenciadores de bancos de dados. Embora a implementação *default* seja baseada em bancos de dados relacionais, é possível estender o componente para suportar bancos de dados objeto-relacionais, orientados a objetos, ou outros mecanismos de persistência compatíveis com a abordagem de inserção, remoção, atualização e pesquisa de informações.

Como é implementado em Java, a implementação sobre bancos de dados relacionais do componente BD é baseada em JDBC; desta forma, qualquer banco de dados que ofereça suporte à API JDBC, e portanto, um *driver* JDBC, pode ser utilizado em conjunto com o componente BD para persistir objetos Java. A implementação do suporte a bancos de dados relacionais do componente BD oferece flexibilidade em diversas frentes:

- A persistência de objetos Java pode ser realizada de forma transparente, dado que o componente emprega um mapeamento objeto/banco de dados relacional e utiliza reflexão para determinar a estrutura e os valores de objetos em tempo de execução; em outras palavras, é possível alterar a estrutura de uma classe de dados e a(s) tabela(s) correspondente(s) no banco de dados, sem a necessidade de modificar a lógica da aplicação;
- Há suporte à definição de diversas fontes de dados, além do conceito de reutilização e compartilhamento de conexões, também conhecido como *pool* de conexões;
- O componente BD suporta o conceito de arquivo de configurações para qualquer tipo de implementação que venha a ser incorporada; a implementação relacional utiliza um arquivo de configurações que permite definir os detalhes de cada fonte de dados (Código 5.6).
- É possível alterar a semântica da implementação relacional do componente através da redefinição de métodos *hook*. Alguns exemplos de métodos *hook* incluem aqueles responsáveis pela geração das cláusulas SQL correspondentes às operações do componente (inclusão, remoção, atualização e busca).

5.2.4 Projeto

A concepção da estrutura do componente DB foi realizada a partir da análise dos requisitos não funcionais mais comuns de aplicações *Web* que realizam acesso a bancos de dados, dos quais podemos destacar:

1. Implementação da lógica de acesso a banco de dados no lado servidor, que corresponde à utilização da arquitetura cliente/servidor *Web* com *thin clients* (capítulo 3.2.1);
2. Alta concorrência de acesso aos sistemas *Web*, e por conseguinte, aos sistemas de bancos de dados;
3. Desempenho e otimização de recursos em vista das altas taxas de concorrência de acesso;
4. Acesso a vários sistemas de bancos de dados, distribuídos em diversas redes e plataformas e baseados em diferentes fornecedores.

O requisito 1 foi abordado através da arquitetura em camadas e da forma como o componente trata (re)utilização, extensão e configuração. Como sugere a (re)utilização na camada de infra-estrutura das aplicações usuárias, pode ser incorporado à parte servidor das aplicações; a configuração é centralizada e baseada em arquivos de configuração que residem no servidor que hospeda o componente/aplicação usuária.

Com relação a concorrência, desempenho e otimização de recursos (requisitos 2 e 3), o componente oferece o mecanismo de *pool* de conexões, que permite implantar uma política de otimização de recursos e melhoria de desempenho através do compartilhamento de conexões. Como num *pool* as conexões podem ser compartilhadas, um cenário com n usuários concorrentes não implica na utilização de n conexões; de fato, o componente deve ser configurado de acordo com os requisitos de concorrência da aplicação que o (re)utiliza, que envolve entre outras tarefas, a definição do número máximo de usuários que compartilham uma conexão.

A questão do desempenho é abordada através do ciclo de vida das conexões de um *pool*: como a abertura e o encerramento de conexões impõe um *overhead* ao funcionamento das aplicações, o *pool* do componente BD mantém um conjunto de conexões abertas configuradas pelo usuário/programador da aplicação; ao executar uma operação sobre o componente, uma conexão do *pool* é utilizada, eliminando desta forma, perdas com relação ao estabelecimento de conexões. O mecanismo de compartilhamento de conexões é baseado na abordagem de *reference count,* e garante que ao executar uma operação é utilizada sempre a conexão com o menor número de usuários simultâneos; além disso, é aplicada a técnica de *round-robin*[20] para garantir a utilização de todas as conexões e evitar a ocorrência de *starvation*[21], que no caso de conexões pode ser traduzido na ocorrência de *timeout.*

Por fim, o componente BD permite configurar diversos *pools* de conexões, cada qual sobre um banco de dados. Como a arquitetura de JDBC contempla a conexão a servidores de bancos de dados remotos através de TCP/IP, pode-se configurar o componente para trabalhar com diferentes *pools* e bancos de dados (inclusive remotos) identificados por uma chave. Ao utilizar operações do componente BD, a aplicação usuária utiliza o identificador/chave para informar qual a fonte de dados sobre a qual as operações de persistência serão executadas (requisito 4).

Como o propósito da implementação do suporte a bancos de dados relacionais é o de demonstrar a incorporação de um mecanismo de persistência ao componente BD, decidimos implementar um subconjunto da disciplina de persistência de objetos sobre bancos de dados relacionais. O mapeamento objeto/banco de dados relacional utilizado é o descrito na Tabela 5.1; a conversão dos tipos de dados Java/SQL e SQL/Java utilizada segue a convenção definida pela API JDBC[22]. Para efeitos de simplificação, implementamos somente o suporte a relacionamentos 1:1 baseado em herança de classes; não contemplamos relacionamentos 1:N, nem o suporte a relacionamentos 1:1 baseados em agregação de objetos.

Classe		Banco de Dados
Objeto	<->	Registro
Nome/Tipo da classe	<->	Nome da Tabela
Nome do Atributo	<->	Nome do Campo
Tipo de Atributo	<->[23]	Tipo do Campo

Tabela 5.1 - Definição do mapeamento classes/bancos de dados relacionais

Seguindo a abordagem discutida no capítulo 4.3.5, empregamos uma arquitetura de componentes *ad hoc* na construção do componente BD. A arquitetura de *software* escolhida para a implementação do componente BD é baseada no modelo de camadas (capítulo 3.2.2), onde o componente pode ser (re)utilizado na implementação da camada de infra-estrutura das aplicações que utilizam persistência de informações. Para permitir a incorporação do suporte a novas

---

[20] Algoritmo comumente utilizado no escalonamento de processos em sistemas operacionais, onde garante a utilização de uma porção do processador a cada processo em execução [PS91]; no contexto do *pool* de conexões, garante que a cada requisição de acesso ao banco de dados uma conexão diferente é utilizada

[21] Processos em *starvation* são aqueles bloqueados à espera de recursos que não podem ser liberados [PS91]; no contexto do *pool* de conexões, uma conexão em *starvation* é aquela que nunca é utilizada, e pode portanto, ser fechada por *timeout.*

[22] Detalhes em http://java.sun.com/jdbc

[23] Baseada no mapeamento Java/SQL e SQL/Java definido pela API JDBC [HC99][HC00]

categorias de bancos de dados (subclasses de `DBTarget`, Figura 5.2) foram empregados os padrões de projeto *bridge* e *factory method* [GHJ94]. Além disso, o suporte à transparência na persistência de objetos é baseado nos mecanismos de reflexão de Java, que permite identificar a estrutura das classes de dados e os pares de atributos/valores dos objetos em tempo de execução (Código 5.7).

5.2.5 Implementação

A Figura 5.2 ilustra o modelo de classes do componente BD, no qual podemos destacar:

- A interface do componente representada pela interface Java `DBService`;
- O ponto de acesso ao componente representado pela classe `DBComponent`;
- O mecanismo de desacoplamento entre o componente e as categorias de bancos de dados disponíveis, representado pelos padrões de projeto *bridge* e *factory method*, envolvendo as classes `DBComponent` e `DBTarget`;
- A implementação do suporte à persistência em bancos de dados relacionais compatíveis com JDBC (classe `RelationalDB`);
- O mecanismo de gerenciamento de *pools* de conexões com bancos de dados representado pelas classes de extensão do componente *dispenser* `DBConnectionManager` e `DBPool`.
- Classes utilitárias de suporte, como a classe `Meta`, que implementa rotinas de reflexão para realizar introspecção nos objetos gravados pelo componente, as classes `ResultSet`, `Row` e `ColumnMetaData`, que representam uma alternativa ao mecanismo de *ResultSet*[24] definido pela API JDBC, e a classe `Transaction` que encapsula uma transação no contexto do componente. A estrutura que permite estender o componente BD para utilizar novos categorias de bancos de dados é similar ao mecanismo de definição de novos repositórios do componente *Log*, dado que utilizam os mesmos padrões de projeto *bridge* e *factory method.*

As funcionalidades relacionadas a reflexão são implementadas através da classe utilitária `Meta`; ao criar uma instância da classe `Meta` podemos informar um objeto, sobre o qual é realizada introspecção para determinar os seus atributos, tipos e valores, bem como o seu tipo (classe) e os tipos das suas superclasses (classes pai) até a raiz da hierarquia de objetos de Java, `java.lang.Object`. Quando aplicada a um objeto do tipo `List` (conforme ilustrada na Figura 5.3), a classe `Meta` identifica dois níveis de hierarquia, `List` e `Item`, além dos atributos e valores correntes do objeto correspondentes às classes `List` e a `Item`. Desta forma, todas as operações do componente que traduzem objetos em cláusulas SQL são implementadas sobre a classe `Meta`, seguindo os esquemas de mapeamento discutidos acima.

Ao realizar a persistência de um objeto do tipo `List`, por exemplo, são desconsiderados os atributos `keywords`, `permissions` e `children`, pois a implementação atual do componente não suporta *arrays*; para que o relacionamento 1:1 entre `Item` e `List` seja reconhecido o campo chave ID deve ser idêntico nos dois níveis da hierarquia do objeto; ao persistir o objeto em questão os atributos de `Item` são gravados num registro de uma tabela `Item`, assim como os registros de `List` são gravados num registro de uma tabela `List`. Além disso, ocorre o seguinte mapeamento de tipos Java em tipos SQL: `String` é traduzido em `VARCHAR` e `Timestamp` é traduzido em `Date`.

---

[24] Um *ResultSet* é a estrutura que permite recuperar o resultado de uma consulta em Java/JDBC [HC00]; também conhecido como *RecordSet* em outras linguagens.

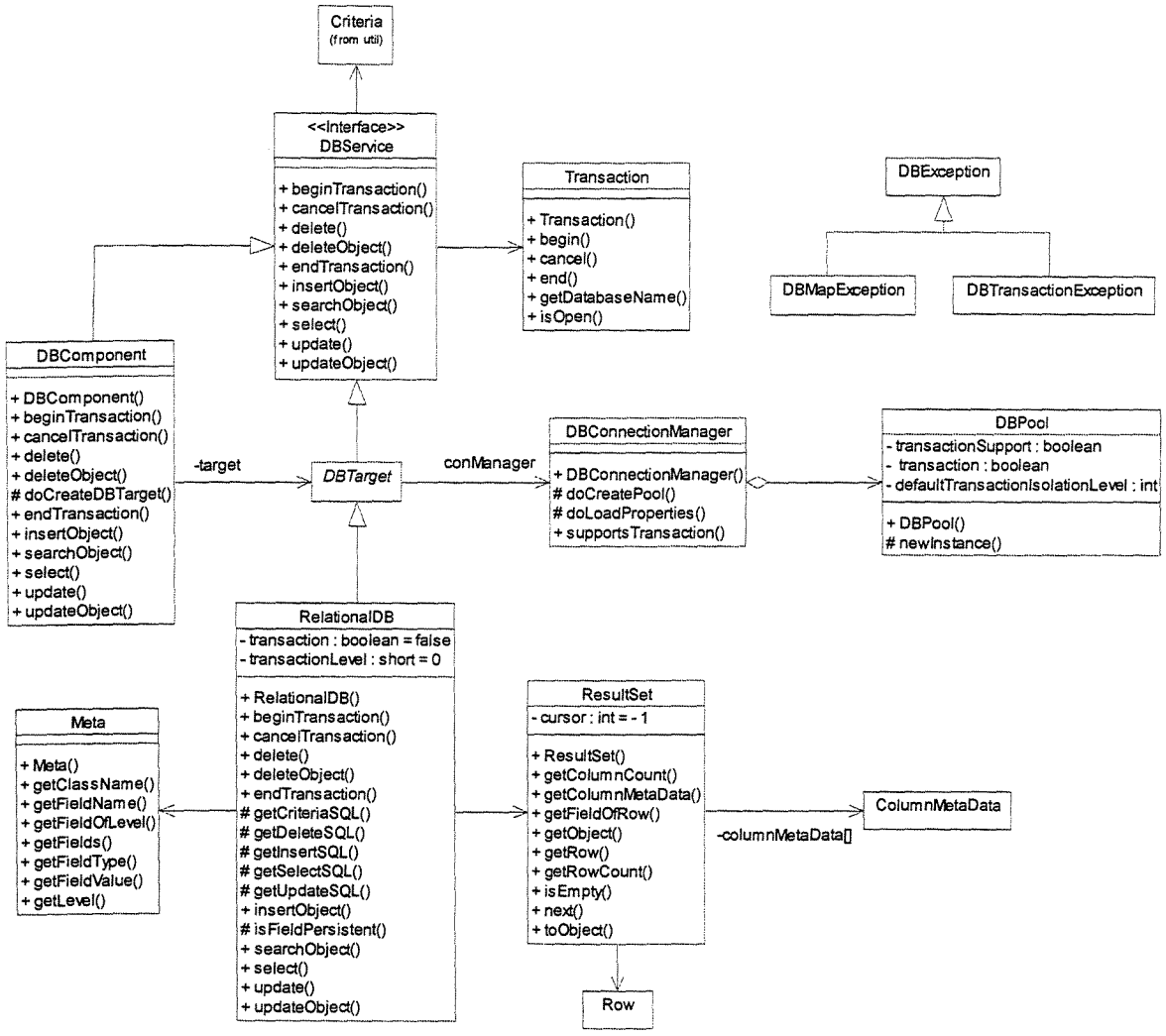


Figura 5.2 - Modelo de classes do componente BD

Outra classe utilitária do componente BD é a classe `Criteria` (Figura 5.2), que é uma abstração para a representação de critérios nas operações de pesquisa, atualização e remoção. Um objeto `Criteria` define o nome de um campo (uma `String` que representa o nome de um atributo, por exemplo), uma expressão de critério (pré-definida pelas constantes da classe `Operators`, como `=`, `>=`, `>`, `<=`, `<` e `*`), o valor associado ao critério e um operador lógico utilizado entre um critério e outro. O Código 5.5, por exemplo, utiliza as classes da Figura 5.3 e a classe `Criteria` para expressar o critério: *"todos os objetos do tipo List cujo proprietário tenha nome iniciado com Guilherme"*.

Como Java não possui uma estrutura transparente para representação de *ResultSets*, ou seja, independente da API JDBC, e portanto, do banco de dados utilizado, concebemos uma abstração própria para representar *ResultSets* no componente BD: a classe `ResultSet`, que é uma matriz 2x2 (ou *array* de duas dimensões) de objetos Java utilizada para armazenar resultados de buscas ou *ResultSets*.

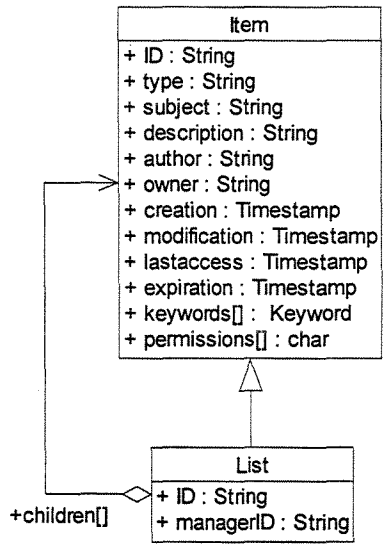


Figura 5.3 - Classes de dados gravadas pelo componente BD (seção 5.2.7)

Além de implementar as operações da interface `ResultSet` definida pela API JDBC, a classe `ResultSet` do componente BD implementa algumas operações complementares, como `isEmpty()`, para verificar a existência de resultados; `getRowCount()` para recuperar o número de registros do *ResultSet*; `getFieldOfRow(i,j)` para recuperar o `i`-ésimo campo da `j`-ésima linha; `toObject()` para recuperar uma matriz de objetos 2x2, onde o elemento `e[i][j]` ($0 < i < 3, 0 < j < 3$) representa o `i`-ésimo atributo do objeto correspondente à `j`-ésima linha do *ResultSet*).

```
Criteria[] c = new Criteria[] {
 new Criteria("type",  Operators.EQUAL, "list",       Operators.AND),
 new Criteria("owner", Operators.WILD,  "Guilherme*", null)
                                };
```

Código 5.5 - Exemplo de utilização da classe `Criteria`

O componente BD pressupõe a utilização um arquivo de configuração para cada categoria de banco de dados implementada. No caso da implementação relacional, o arquivo de configuração possui a estrutura apresentada no Código 5.6 e permite definir os seguintes parâmetros:

1. **Nome** do *pool*: como o arquivo pode conter a configuração de diversos *pools*, o componente interpreta como a configuração do *pool* `inex_read`, por exemplo, todas os parâmetros com prefixo `inex_read` (Código 5.6).
2. **Driver JDBC** para conexão ao banco de dados: campo `driver`, contém o nome da classe que implementa o driver JDBC para conexão ao banco de dados utilizado pelo *pool* corrente - Oracle, no exemplo do Código 5.6.
3. **URL** de conexão (campo `url`): todo banco de dados compatível com JDBC define um URI[25] para conexão.
4. **Username** e **Password** utilizados na conexão: correspondem ao usuário e senha para conexão.

[25] *Universal Resource Identifier* [HC00]

5. *Flag* para indicar se as conexões do *pool* podem ser **compartilhadas** (campo `shared`): um *pool* compartilhado. Vale ressaltar que no caso de banco de dados, um *pool* compartilhado deve ser utilizado somente para operações de leitura, pois a concorrência de operações de escrita e atualização impede o controle de transações por usuário/aplicação.
6. *Flag* para indicação de suporte a **transações** e **nível** de isolamento das transações: campos `transaction` e `transactionlevel`, indicam se haverá suporte a transações e qual o esquema de leitura dos dados durante operações simultâneas, conforme definido pela API JDBC.
7. Número **mínimo** e **máximo** de conexões do *pool*: campos `min` e `max`, informam ao gerenciador de *pools* quantas conexões serão criadas no início da execução do componente (`min`), e quantas conexões poderão ser criadas em tempo de execução caso o número máximo de usuários por conexão permitido seja alcançado e existam requisições de conexões pendentes.
8. Número máximo de usuários por conexão compartilhada: campo `load`.
9. Os campos `loader` e `class` são necessários para o funcionamento do componente *dispenser*, apresentado na seção 5.4, e permitem especificar a classe responsável pela inicialização dos objetos da coleção e o tipo dos objetos da coleção, respectivamente.

5.2.6 Interface

A interface do componente BD possui 10 operações:

1. `beginTransaction()`,
2. `endTransaction()`,
3. `cancelTransaction()`: oferecem suporte ao mecanismo de transações da base de dados corrente; a idéia é agrupar todas as operações de inserção/atualização de dados num bloco de código entre as chamadas às operações `beginTransaction()` e `endTransaction()` e executar a operação `cancelTransaction()` em caso de falha na execução de alguma operação do bloco (Código 5.7); desta forma, é possível garantir que ou todas as operações são executadas com sucesso ou nenhuma delas é efetivada.
4. `insertObject(Transaction transaction, Object obj)`: insere o objeto `obj` na base de dados representada pela transação `transaction`. Somente atributos simples (`String`, `boolean`, `char`, `int`, `long`, `Integer`, `Long`, `Character`, etc.) e não transientes[26] pré-definidos pela linguagem Java são utilizados na persistência do objeto. O mapeamento utilizado na persistência do objeto corresponde àquele apresentado na seção 5.2.5 no exemplo do objeto do tipo `List`.
5. `searchObject(String dataSource, Object obj, Criteria[] c)`: executa uma busca por objetos na base de dados identificada por `dataSource` (nome do *pool*, Código 5.7) que satisfaçam os critérios especificados em `c` e que sejam do mesmo tipo que `obj`. O componente realiza reflexão em `obj` e pesquisa a base de dados corrente pelos dados de `obj` que satisfaçam os critérios de `c`. Os resultados são recuperados num *array* de objetos do mesmo tipo de `obj`.

---

[26] Um atributo transiente em Java é definido através da cláusula *transient* e indica que o valor do atributo é descartado no momento da serialização do objeto; desta forma, atributos transientes são normalmente empregados na representação de variáveis temporárias de trabalho que dispensam serialização e/ou persistência.

6. `select(String dataSource, String source, String[] fields, Criteria[] c)`: realiza uma pesquisa simples na base de dados identificada por `dataSource` (nome do *pool*, Código 5.7), recuperando os dados num objeto do tipo `ResultSet`, definido pelo componente BD. Esta operação difere de `searchObject` por não realizar reflexão em objetos e não recuperar os resultados da busca num tipo específico de objeto definido pelo usuário. A operação `select` é utilizada em buscas simples, ou com critérios e resultados mais simples que os casos de `searchObject`. Seus argumentos indicam qual a fonte de dados para a busca (uma tabela, um objeto, um identificador de um objeto, etc.), os campos/atributos que deverão ser recuperados e os critérios utilizados na busca.
7. `updateObject(Transaction t, Object target, String[] fields, Criteria[] c)`: atualiza os objetos na base de dados representada pela transação `t` que satisfaçam os critérios especificados em `c` e que sejam do mesmo tipo que `target`. Somente os atributos indicados no argumento `fields` são atualizados - os valores utilizados na atualização são os próprios valores constantes nos atributos do objeto `target`.
8. `update(Transaction t, String source, String[] setFields, Criteria[] c)`: executa uma atualização simples na base de dados representada pela transação `t`. Esta operação difere de `updateObject()` por não atuar sobre um tipo específico de objeto indicado pelo usuário, e por empregar poucos (ou menos) critérios e campos para atualização quando comparada a `updateObject`. Seus argumentos indicam a fonte de dados para a atualização dos dados (uma tabela, um objeto, etc.) as expressões de atualização na forma `atributo = valor` e os critérios utilizados na atualização.
9. `deleteObject(Transaction transaction, Object target, Criteria[] c)`: remove todos os objetos na base de dados representada pela transação `transaction` que satisfaçam os critérios especificados em `c` e que sejam do mesmo tipo que o `target`.
10. `delete(Transaction transaction, String target, Criteria[] c)`: executa uma remoção simples na base de dados representada pela transação `transaction`. A operação `delete` é utilizada em remoções simples, ou que empreguem poucos (ou menos) critérios quando comparada a `deleteObject()`. Seus argumentos indicam os critérios e a fonte de dados para a remoção (uma tabela, um objeto, etc.).

Como o componente BD suporta a utilização de múltiplas fontes de dados, no momento em que se inicia uma transação é necessário indicar qual o nome da fonte de dados, ou *pool*, sobre a qual as operações subsequentes irão atuar (como o *pool* `inex_write` do exemplo ilustrado no Código 5.7). Ao iniciar uma transação, o componente devolve um objeto do tipo `Transaction`; todas as operações subsequentes devem utilizar o objeto `transaction` para que o componente controle a transação. A implementação atual do componente não permite executar operações de modificação (`insert`, `update` e `delete`) sem que se inicie um transação. Portanto a precedência de operações do componente BD segue o padrão: operação 1, operação 4, 5, 6, 7, 8, 9 ou 10, operação 2 ou 3; ou ainda: operação 5 ou 6.

5.2.7 Prova de Conceito

Como o componente BD utiliza os mesmos padrões de projeto (*bridge* e *factory method*) do componente *Log*, estendê-lo através da incorporação de novos mecanismos de persistência (bancos de dados orientado a objetos, por exemplo) segue os mesmos princípios discutidos na seção 5.1.7. O mesmo ocorre na redefinição das classes de dados: como o componente BD utiliza reflexão para executar a persistência de objetos, é possível estender classes de dados como

apresentado na seção 5.1.7 (Código 5.3, Código 5.4), sem modificar a estrutura do componente BD.

Neste sentido, procuramos realizar a prova de conceito do componente BD através da sua configuração e (re)utilização, como ilustrado nos exemplos do Código 5.6 e Código 5.7. Conforme apresentamos na seção sobre implementação, o componente suporta a utilização de um arquivo de configuração, que no caso do suporte a bancos de dados relacionais pode definir um ou mais *pools* de conexões. No exemplo do Código 5.6 podemos identificar dois *pools*: **inex_read** e **inex_write**; esta configuração segue o princípio de compartilhamento de conexões discutido acima, onde **inex_read** representa o *pool* de leitura que suporta o compartilhamento de uma conexão entre até 4 usuários (campo **load=4**), e **inex_write**, que representa o *pool* de escrita/atualização onde não é possível compartilhar conexões entre usuários simultâneos devido ao suporte a transações (campo **transaction=true**). Pela natureza do compartilhamento das conexões, podemos ressaltar que **inex_read** é um *pool* recomendado para operações de consulta, enquanto **inex_write** é recomendado para operações de atualização.

```
# definição do pool inex_read
inex_read.driver = oracle.jdbc.driver.OracleDriver
inex_read.url = jdbc:oracle:thin:@oraserver:1521:APP
inex_read.username = inex
inex_read.password = inex
inex_read.transaction = false
inex_read.transactionlevel = 1
inex_read.class = java.sql.Connection
inex_read.loader = component.db.DBPool
inex_read.min = 2
inex_read.max = 4
inex_read.load = 4
inex_read.shared = true

# definição do pool inex_write
inex_write.driver = oracle.jdbc.driver.OracleDriver
inex_write.url = jdbc:oracle:thin:@oraserver:1521:APP
inex_write.username = inex
inex_write.password = inex
inex_write.transaction = true
inex_write.transactionlevel=1
inex_write.class = java.sql.Connection
inex_write.loader = component.db.DBPool
inex_write.min = 2
inex_write.max = 2
inex_write.load = 1
inex_write.shared = false
```

Código 5.6 - Exemplo de configuração do componente BD

```
// Inicialização do componente
DBService db = new DBComponent(DBService.RDBMS, "rdbms.ini");
Transaction transaction;
Criteria[] criteria;
List list;
List[] lists;

List list = new List("1@inex.ahand.unicamp.br", "Guilherme");
((Item) list).setID("1@inex.ahand.unicamp.br");
list.setManagerID("Root");
list.setSubject("Mestrado");
list.setDescription("Material Mestrado-UNICAMP");
list.setCreation(new Date());

criteria = new Criteria[]{
  new Criteria("type",  Operators.EQUAL, Item.List,    Operators.AND),
  new Criteria("owner", Operators.WILD,  "Guilherme*", null) };

try {
  // inicia a transação na fonte de dados inex_write
  transaction = db.beginTransaction("inex_write");

  // insere o objeto list no banco de dados
  db.insertObject(transaction, list);

  // recupera todos os objetos list cujos owners iniciem com Guilherme
  list = new List();
  lists = db.searchObject(transaction.getDatabaseName(), list, criteria);

  // prepara o critério de busca:
  // ID = lists[0].ID OR ID = lists[1].ID OR ... OR ID = lists[n].ID
  criteria = new Criteria[lists.length];

  for (int i = 0; i < criteria.length; i++)
    if (i < criteria.length - 1)
      criteria[i] = new Criteria("ID", Operators.EQUAL,
                                 lists[i].getID(), Operators.OR);
    else
      criteria[i] = new Criteria("ID", Operators.EQUAL,
                                 lists[i].getID(), null);

  list = new List();
  list.setModification(new Date());
  list.setLastAccess(new Date());

  // atualiza a data/hora do último acesso e da última modificação
  // das listas recuperadas na pesquisa anterior
  db.updateObject(transaction, list,
                  new String[]{"lastaccess", "modification"}, criteria);

  // encerra a transação
  db.endTransaction(transaction);
} catch(DBException e) { if (transaction.isOpen())
                           db.cancelTransaction(transaction); }
```

Código 5.7 - Exemplo de utilização do componente BD

Com relação à demonstração de (re)utilização do componente, utilizamos a definição das classes `List` e `Item` da Figura 5.3, e apresentamos a seguinte prova de conceito: *"Gravar um objeto do tipo List, recuperar todos os objetos do tipo List cujo proprietário tenha nome iniciado com Guilherme e atualizar a data de último acesso e modificação dos objetos recuperados na pesquisa anterior sob uma transação atômica, ou seja, todas as operações devem ser executadas ou nenhuma deve ser efetivada."*. O exemplo do Código 5.7 ilustra a realização desta prova de conceito, do qual podemos destacar:

- A disposição das operações de transação para garantir a atomicidade da prova de conceito - qualquer falha na execução das operações do componente levanta uma exceção que é tratada através do cancelamento da transação;
- O mapeamento do relacionamento 1:1 através de herança nas duas atribuições de ID, tanto por parte da hierarquia da classe `List`, quanto `Item` - quando o componente realizar reflexão sobre o objeto do tipo `List`, determinará que `ID` é a chave do relacionamento entre `List` e `Item`, e que é possível portanto gravar os dados de `List` numa tabela `List`, e os dados de `Item` numa tabela `Item`;
- A utilização de um *pool* ou fonte de dados identificado por `inex_write`, que a exemplo do Código 5.6, deve ser definido como um *pool* não compartilhado e com suporte a transações;
- A utilização do objeto do tipo `Transaction` nas operações de atualização e gerenciamento de transações;
- A utilização da classe utilitária `Criteria` para definir os critérios de pesquisa e atualização dos objetos.

Como forma de testar a (re)utilização do componente sob diferentes bancos de dados e drivers JDBC, foram executados testes com os bancos de dados SQLServer/Microsoft, mSQL/Hughes Technologies e Oracle/Oracle. Os problemas detectados referem-se à utilização de classes de dados com atributos que são nomes reservados em bancos de dados, e que portanto não podem ser utilizados como nome de colunas em tabelas do banco de dados; *UID* por exemplo é válido em SQLServer, mas não é válido em Oracle; *user* é inválido tanto em Oracle, quanto em SQLServer, e assim por diante. Uma das possíveis soluções para este problema seria a incorporação de um esquema de mapeamento de nomes entre objetos/atributos e tabelas/campos ao componente BD, de forma que seria possível especificar o esquema de tradução de nomes para cada objeto manipulado pelo componente.

5.3 Componente para Controle de Acesso

O componente para controle de acesso, ou componente de segurança, é implementado sobre a abordagem de listas de controle de acesso de Java (`java.security.acl`) e oferece serviços para a manipulação de usuários, grupos e permissões de acesso. Assim como o componente BD, o componente de segurança pode ser (re)utilizado na camada de infra-estrutura de aplicações para prover mecanismos de segurança e controle de acesso. A arquitetura do componente de segurança foi concebida de forma a desacoplar as funcionalidades do componente do meio de persistência das informações sobre usuários, grupos, recursos e permissões; como a implementação atual foi realizada sobre o componente BD, a persistência de informações é realizada sobre bancos de dados relacionais.

Como a interface do componente permite configurar usuários, grupos e permissões de acesso, é possível utilizá-lo no gerenciamento de usuários e no controle de acesso aos recursos definidos

pelo usuário/programador da aplicação. Como é possível incorporar novos meios de persistência ao componente, pode-se implementar suporte a diretórios LDAP, por exemplo.

5.3.1 Estímulo

Uma das premissas de *sites intranet* e *extranet*, bem como os *sites* WWW que oferecem personalização de acesso por perfil de usuário, é o controle de acesso às informações. A idéia é que apesar destes *sites* contarem com uma série de informações, serviços e aplicações, nem todas estão disponíveis para todos os usuários. Para implementar controle de acesso, as aplicações e serviços *Web* devem incorporar um mecanismo que relacione usuários a recursos (arquivos, serviços, aplicações, seções de um *site*, etc.) através de permissões de acesso.

Vale ressaltar que os mecanismos de controle de acesso às aplicações e serviços *Web* são imprescindíveis, pois a infra-estrutura de segurança "física" composta por *firewalls*, por exemplo, não garante segurança "lógica" das aplicações. As *intranets* constituem bom exemplo da necessidade de se implantar segurança "lógica", ou seja, mecanismos de controle de acesso integrados às próprias aplicações/serviços, pois embora a maioria de seus usuários sejam locais, pode haver problemas de acesso não autorizado. Como o componente de segurança oferece serviços de controle de acesso, pode ser incorporado às aplicações como forma de isolar os detalhes de implementação da infra-estrutura de segurança.

5.3.2 Funcionamento

A API `java.security` de Java incorpora o conceito de *ACL*[27] para o controle de acesso aos recursos externos à linguagem/plataforma (*sockets,* arquivos, diretórios, propriedades da JVM, etc.). Uma *ACL* permite expressar a permissão que um usuário possui sobre um recurso. O conceito de usuário em *ACL*, também conhecido como *principal*, é genérico e permite expressar usuários, grupos de usuários ou qualquer outra abstração similar. Uma permissão em *ACL* pode ser positiva, para os casos em que é habilitada ao usuário, ou negativa, para os casos em que é revogada do usuário; em outras palavras, uma permissão positiva expressa os direitos do usuário sobre o recurso, enquanto uma permissão negativa expressa a proibição do usuário de utilizar o recurso. A plataforma Java não oferece implementação para a API de *ACLs,* apenas define as suas interfaces e a semântica de uma lista de controle de acesso que os implementadores devem seguir:

- Se não existir nenhuma permissão associada a um *principal* sobre um determinado recurso, o conjunto de permissões resultante é nulo; nestes casos o acesso é negado.
- As permissões positivas (direitos atribuídos) de grupo para o *principal* são as permissões de todos os grupos aos quais o *principal* pertence. Analogamente, as permissões negativas (direitos revogados) de grupo correspondem às permissões de todos os grupos (com permissões negativas) que o *principal* pertencer.
- Permissões individuais (positivas ou negativas) prevalecem sobre as permissões de grupo; assim um direito positivo individual prevalece sobre o mesmo direito negativo para o grupo, enquanto um direito negativo individual prevalece sobre o mesmo direito positivo para o grupo.

---

[27] Lista de controle de acesso ou *Access Control List* [HC99]

- O conjunto de permissões resultante de um *principal* sobre um recurso é representado pela expressão: `(i1 + (g1 - i2)) - (i2 + (g2 - i1))`, onde "+" representa a união de conjuntos e "-" a diferença, e:
  - `i1` é o conjunto de permissões positivas (atribuídas) individuais sobre um recurso;
  - `i2` é o conjunto de permissões negativas (revogadas) individuais sobre um recurso;
  - `g1` é o conjunto de permissões positivas (atribuídas) de um grupo sobre um recurso;
  - `g2` é o conjunto de permissões negativas (revogadas) de um grupo sobre um recurso;
  - A primeira parte da expressão corresponde aos direitos atribuídos, considerando que as permissões individuais negativas prevalecem sobre as permissões positivas de grupo; a segunda parte corresponde aos direitos revogados, onde as permissões individuais positivas prevalecem sobre as permissões negativas de grupo. O resultado das permissões atribuídas é a subtração, ou o conjunto disjunto, entre a primeira e a segunda parte da expressão.

Como é implementado sobre o conceito de *ACLs,* o funcionamento do componente de segurança depende da definição de *principals* e da atribuição de permissões (positivas/negativas) dos *principals* sobre os recursos definidos pelo usuário. Em linhas gerais, a sequência de utilização do componente pode ser descrita como segue:

1. Configura-se o meio de persistência sobre o qual o componente irá funcionar - na versão atual, implementada sobre o componente BD, consiste em configurar as fontes de dados do componente BD;
2. Obtém-se uma referência ao componente, representada pela sua interface;
3. Configura-se os *principals* através da criação, atualização ou remoção de usuários e grupos de usuários;
4. Verifica-se usuário/senha
5. Atribui-se ou revoga-se permissões a *principals* sobre recursos identificados pelo usuário;
6. Verifica-se se um *principal* possui uma determinada permissão sobre um recurso. Nesta etapa são recuperadas todas as listas de controle de acesso referentes ao *principal,* permissão e o recurso em questão e computa-se a regra discutida acima.

5.3.3 Flexibilidade

Como há iniciativas de registrar informações de controle de acesso e segurança tanto em bancos de dados como em diretórios (LDAP, por exemplo), o componente de segurança suporta a incorporação de diferentes meios de persistência. Como o componente BD oferece suporte à persistência de objetos Java em bancos de dados relacionais, optamos pela sua utilização no componente de segurança.

Assim como a API de *ACL*, o componente de segurança não define quais os tipos de permissão e recurso utilizados; desta forma, é possível definir em tempo de execução quais as permissões e os recursos utilizados. O componente oferece uma implementação do conceito de *principal* baseada em usuários e grupos de usuários, onde um grupo pode conter zero ou mais usuários, e um usuário pode pertencer a mais de um grupo. O modelo de relacionamento M:N entre usuários e grupos, além da possibilidade de se utilizar permissões e recursos genéricos permitem (re)utilizar o componente em diversos cenários.

5.3.4 Projeto

Em termos de projeto, o componente de segurança segue o mesmo modelo de arquitetura dos demais componentes, empregando um modelo de componentes *ad hoc,* uma arquitetura de *software* baseada no modelo de camadas, e os padrões de projeto *bridge* e *factory method* para permitir o desacoplamento entre a lógica do componente (`SecurityComponent`, Figura 5.4) e os meios de persistência das informações (`SecurityOnDatabase`, Figura 5.4).

Ao reutilizar o componente BD no projeto do componente de segurança ainda utilizamos um outro padrão de projeto, *adapter* [GHJ94], responsável por mapear as chamadas de persistência e recuperação de informações de segurança (permissões, usuários, recursos, etc.) entre o componente de segurança e o componente BD (`SecurityOnDatabase`, Figura 5.4).

5.3.5 Implementação

A Figura 5.4 ilustra o modelo de classes do componente de segurança, no qual podemos destacar:

- A interface do componente representada pela interface Java `SecurityService`;
- O ponto de acesso ao componente representado pela classe `SecurityComponent`;
- O mecanismo de desacoplamento entre o componente e os meios de persistência disponíveis, representado pelos padrões de projeto *bridge* e *factory method* [GHJ94], envolvendo as classes `SecurityComponent` e `SecurityTarget`;
- A utilização do componente BD para persistência das informações de segurança através do padrão de projeto *adapter,* envolvendo a classe `SecurityOnDatabase` e a interface `DBService`;
- A API de segurança de Java que define as interfaces para implementação da abordagem de *ACL*;
- A implementação da API *ACL* como parte do componente de segurança;
- Classes de dados para persistência das informações manipuladas pelo componente, como `ResourcePermissions`, que representa as permissões que um usuário/grupo possui sobre um recurso, `UserGroup`, que representa a relação M:N entre usuário e grupo, `User`, `GroupImpl` e `Permission`;
- A hierarquia de exceções que os serviços do componente podem gerar (subclasses de `SecException`).

Como o componente de segurança foi implementado sobre o componente BD, todas as informações sobre usuários, grupos e permissões de acesso são gravadas em banco de dados; ao verificar permissões por exemplo, o componente recupera as permissões individuais e de grupo relacionadas ao *principal* (que pode ser um usuário ou grupo) numa estrutura de `AclImpl` e `AclEntryImpl` e aplica a regra definida pela API de *ACL* (seção 5.3.2); um objeto do tipo `AclImpl` representa uma lista de controle de acesso que armazena as permissões de usuários/grupos sobre um determinado recurso; cada permissão é mapeada através da classe `AclEntryImpl`.

O componente também adota mecanismos de gerenciamento de transações para as operações que atualizam informações no meio de persistência; na implementação atual, as operações de transação são baseadas nos serviços de transação do componente BD.

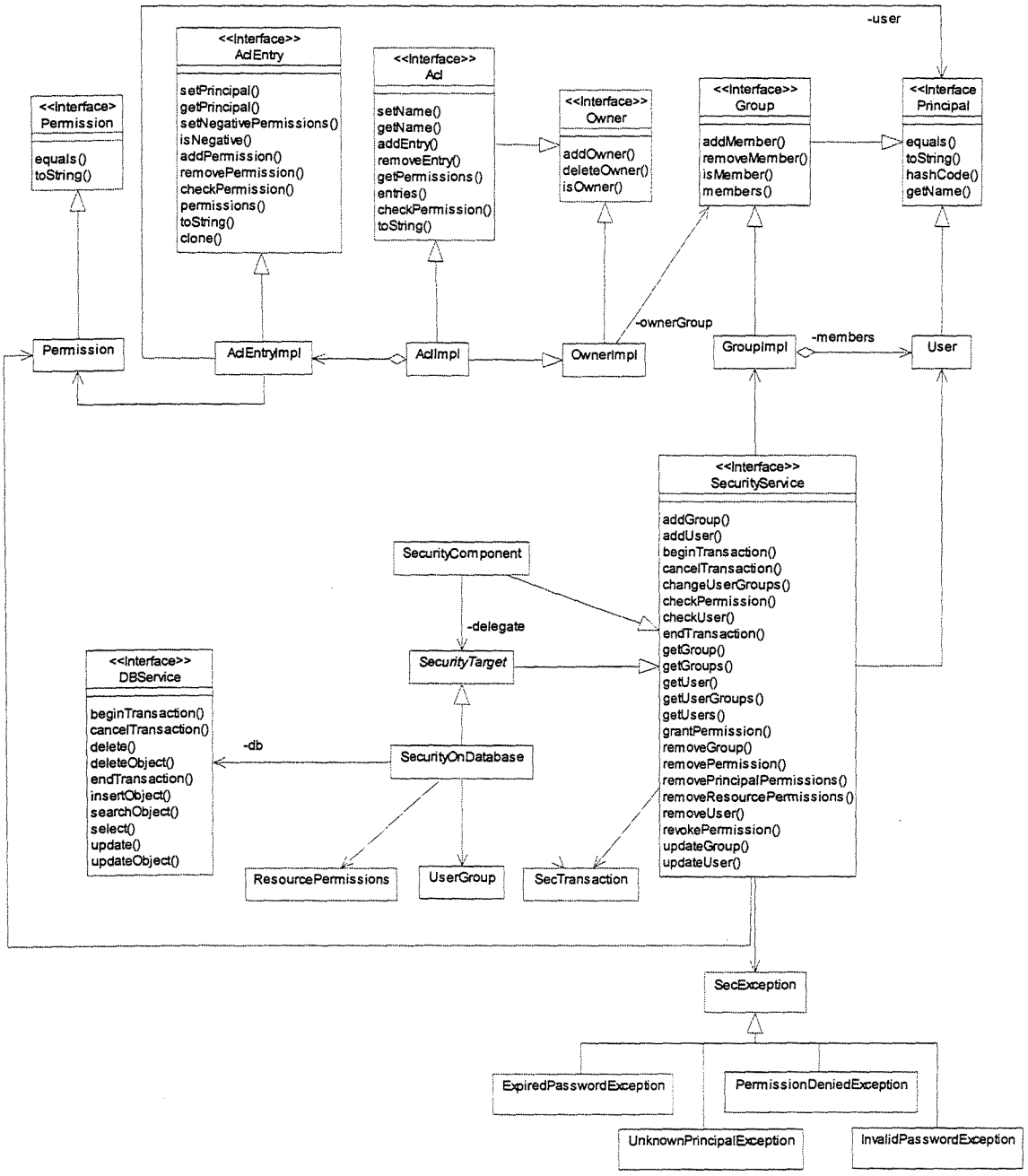


Figura 5.4 - Modelo de classes do componente de segurança

5.3.6 Interface

A interface do componente de segurança conta com 22 serviços que podem ser divididos em 3 categorias: tratamento de transações (3), gerenciamento de usuários/grupos (13) e manipulação de listas de controle de acesso (6):

1. `beginTransaction, endTransaction, cancelTransaction:` similares às operações de suporte a transações do componente BD, são utilizadas nas operações de atualização de usuários, grupos e permissões.
2. `addUser` e `addGroup` adicionam um usuário ou grupo ao meio de persistência corrente; `checkUser` verifica o *login* de um usuário, levantando uma exceção no caso de falha na checagem (senha expirada, *username* desconhecido, senha inválida, etc.); `getUser` recupera as informações de um usuário; `getUsers` recupera os nomes dos usuários registrados pelo componente; `getGroup` recupera as informações de um grupo; `getGroups` recupera os nomes dos grupos registrados pelo componente; `getUserGroups` recupera os grupos que contém o usuário; `removeUser` remove um usuário do meio de persistência corrente; `removeGroup` remove somente o grupo do meio de persistência corrente, preservando os usuários que pertencem ao grupo removido, dado que a semântica do componente suporta usuários órfãos de grupos, ou seja, usuários que não pertencem a nenhum grupo; `updateUser` atualiza as informações de um usuário; `updateGroup` atualiza as informações de um grupo; `changeUserGroups` atualiza os grupos aos quais o usuário pertence.
3. Operações de suporte a manipulação de listas de controle de acesso:
   3.1. `grantPermission(Principal p, Object res, Permission[] perms):` atribui as permissões representadas pelo *array* `perms` sobre o recurso `res` para o usuário ou grupo `p`; o componente utiliza o método `toString()` sobre o objeto `res` para recuperar o identificador do recurso - classes que representam recursos controlados pelo componente de segurança devem implementar `toString()` de forma a garantir a geração de identificadores únicos para seus objetos.
   3.2. `revokePermission(Principal p, Object res, Permission[] perms):` atribui as permissões negativas `perms` ao usuário ou grupo `p` sobre o recurso `res`. Este mecanismo oferece grande flexibilidade na manutenção de listas de controle de acesso, pois permite delegar direitos a um grupo, enquanto se revoga o mesmo direito de um indivíduo pertencente ao grupo, ou ainda revogar um direito de todos os usuários de um grupo, enquanto se delega o mesmo direito a determinados indivíduos do grupo.
   3.3. `checkPermission(Principal principal, Object resource, Permission perm):` verifica se o usuário ou grupo `principal` possui a permissão representada por `perm` sobre o recurso `resource`. Esta é a principal operação do componente, pois utiliza todas as informações das listas de controle de acesso relacionadas ao recurso pesquisado. Como não implementamos a semântica da hierarquia de grupos, ou seja, um grupo não pode conter outros grupos, a verificação é realizada da seguinte forma:
      3.3.1. Se `principal` é um grupo, determina-se o conjunto das permissões do grupo relacionadas ao recurso (`g1` e `g2`, seção 5.3.2) e verifica-se se a permissão `permission` pertence ao conjunto;
      3.3.2. Se `principal` é um usuário, determina-se as permissões individuais (`i1` e `i2`) e de grupo (`g1` e `g2`) e calcula-se o conjunto resultante, segundo a fórmula apresentada na seção 5.3.2, e verifica-se se a permissão `permission` pertence ao conjunto.
   3.4. `removePermission(Principal p, Object res, Permission[] perms):` remove as permissões (positivas ou negativas) representadas por `perms` do usuário ou grupo `p` sobre o recurso `res` do meio de persistência corrente. Esta operação é utilizada na remoção de

uma permissão específica sobre um determinado recurso e usuário/grupo; corresponde à remoção de um objeto do tipo `AclEntryImpl` (Figura 5.4).

3.5. `removePrincipalPermissions(Principal principal)`: remove todas as permissões associadas ao usuário/grupo representado por `principal`. Em geral, é utilizada durante a remoção de um usuário/grupo do sistema; corresponde à remoção de todos os objetos do tipo `AclEntryImpl` que referenciam o usuário/grupo `principal`.

3.6. `removeResourcePermissions(Object resource)`: remove todas as permissões associadas ao recurso representado pelo objeto `resource`. Pode ser utilizada durante a remoção do recurso do sistema, por exemplo; corresponde à remoção de todos os objetos do tipo `AclImpl` identificados por `resource`.

5.3.7 Prova de Conceito

Assim como os componentes *Log* e BD, o componente de segurança utiliza a mesma estrutura de extensão baseada nos padrões de projeto *bridge* e *factory method*, aqui utilizada para garantir a incorporação de novos meios de persistência de informações de segurança. Neste sentido, estender o componente de segurança segue os mesmos princípios discutidos na seção 5.1.7.

Enquanto a prova de conceito do componente BD foca a persistência das informações representadas pelos objetos do tipo `List` e `Item` (Figura 5.3), o exemplo de utilização do componente de segurança foca a criação de listas de controle de acesso sobre estes mesmos objetos. Como a classe `List` é uma abstração similar a um diretório de um sistema de arquivos, que pode conter outros itens (agregação identificada por `children`, Figura 5.3), suponhamos que todo objeto `List` deve possuir uma *ACL* que identifica quais os grupos e usuários que podem ler, escrever e gerenciar a lista. Esta é a prova de conceito do componente de segurança apresentada no exemplo do Código 5.8, que demonstra as operações do componente relacionadas ao gerenciamento de transações e listas de controle de acesso.

Como uma lista (objeto da classe `List`) pode ser adicionada a outra lista, é necessário verificar se o usuário proprietário da lista adicionada (`newList`, Código 5.8) possui permissão de escrita sobre a lista origem (`rootList`, Código 5.8). Caso seja garantida a permissão de escrita sobre a lista origem, a lista é adicionada como no exemplo do Código 5.7, e uma *ACL* é criada para representar as permissões de acesso a nova lista; no caso do exemplo do Código 5.8 são acrescentadas as seguintes permissões de acesso:

- Permissão de leitura e escrita para o usuário proprietário da lista adicionada;
- Permissão de leitura para todos os grupos que contém o usuário proprietário da lista adicionada.

Vale ressaltar que assim como ocorre no componente de BD, as operações de atualização do componente de segurança são manipuladas entre os blocos de início e término/cancelamento de uma transação para garantir a integridade das informações de segurança no ambiente de persistência.

Por fim, deve-se considerar a forma de utilização do componente, de forma a controlar o acesso aos seus serviços. Caso o componente (e a base de dados que o suporta) seja compartilhado entre diversas aplicações, é necessário adotar um mecanismo de controle de acesso ao componente. Uma possível solução é adotar um conjunto de ACLs que indiquem qual aplicação ou grupo de aplicações podem empregar qual serviço(s) do componente.

```
public class SecurityLayer {
SecurityService sec;
SecTransaction transaction;
Group[] userGroups;
List rootList, newList;

rootList = new List("root@inex.ahand.unicamp.br");
newList = new List("1@inex.ahand.unicamp.br");
newList.setOwner("Guilherme");
newList.setSubject("Mestrado");

// obtém uma referência ao componente de segurança
sec = new SecurityComponent(SecurityService.RDBMS, "sec.ini");

try {
  // checa se o usuário Guilherme pode escrever na lista raiz (root)
  sec.checkPermission(newList.getOwner(), rootList.getID(), WRITE);
} catch (PermissionDeniedException) {
    // o usuário não pode escrever na lista root
   return;
  }

// o usuário pode escrever na lista root, executa a persistência
// do objeto newList, como demonstrado no exemplo do código 7.
...

try {
  // cria uma transação sobre o pool inex_write
  transaction = sec.beginTransaction("inex_write");

  // atribui as permissões de leitura e escrita ao proprietário da lista
  sec.grantPermission(transaction,
                      newList.getOwner(),
                      newList.getID(),
                      new Permission[] { READ, WRITE } );

  // atribui a permissão de leitura aos mesmos grupos que os do usuário
  userGroups = sec.getUserGroups(transaction.getDataSource(),
                                 newList.getOwner());

  if (userGroups!= null && userGroups.length > 0)
    for (int i = 0; i < userGroups.length; i++)
      sec.grantPermission(transaction, userGroups[i], newList.getID(),
                          new Permission[] { READ } );

  // encerra a transação
  sec.endTransaction(transaction);
} catch (SecException e) {
    // cancela a transação
    if (transaction.isOpen())
      sec.cancelTransaction(transaction);
  }
}
```

Código 5.8 - Exemplo de utilização do componente de segurança

5.4 Componente *Dispenser*

O componente *dispenser*[28] pode ser configurado para manter coleções de objetos de diferentes tipos, que podem ser alocadas por aplicações locais e remotas; o componente implementa mecanismos para instanciar e dealocar objetos, controlar o compartilhamento de objetos e adicionar ou remover coleções de objetos.

Quando utilizado através das suas interfaces remotas RMI, é possível empregar o componente *dispenser* como um servidor de nomes de objetos distribuídos Java, onde é possível requisitar objetos remotos RMI a partir do nome da coleção e deixar o mapeamento nome/objeto e o controle do ciclo de vida dos objetos remotos sob seu gerenciamento.

Ao utilizar o componente localmente, é possível incorporá-lo às aplicações que possuem requisitos de compartilhamento e controle do ciclo de vida de objetos, como é o caso do componente BD, onde o mecanismo de *pool* de conexões é implementado sobre o componente *dispenser*.

5.4.1 Estímulo

Como discutido na seção 5.2.4, as aplicações distribuídas como as do tipo cliente/servidor *Web* possuem requisitos de desempenho e otimização de recursos em vista das altas taxas de concorrência de acesso. No cenário de objetos distribuídos, por exemplo, é necessário realizar compartilhamento de recursos, ou seja, dos objetos remotos, entre diversos clientes. Ainda é preciso realizar o compartilhamento de forma equilibrada, para que o número de usuários concorrentes por objeto não degrade o desempenho do sistema.

Em outras palavras, é necessário prover mecanismos de compartilhamento de objetos como forma de otimizar recursos, e gerenciamento do ciclo de vida de objetos para não prejudicar o desempenho dos sistemas que empregam objetos compartilhados. Neste sentido, o componente *dispenser* implementa a abordagem de coleções de objetos compartilháveis com suporte ao gerenciamento do ciclo de vida.

5.4.2 Funcionamento

O funcionamento do componente *dispenser* é baseado em interfaces Java e em algumas classes utilitárias para tratamento das coleções e controle do ciclo de vida de objetos. A sequência de utilização do componente pode ser descrita como segue:

1. Configura-se as coleções de objetos sobre as quais o componente irá distribuir, indicando o nome da classe, o número mínimo e máximo de objetos instanciados por coleção, o número máximo de usuários por objeto, se a coleção é compartilhada, etc.
   1.1. Opcionalmente, pode-se estender o componente para suportar coleções de objetos que dependem de um controle de ciclo de vida específico, diferente do oferecido pela implementação *default* do componente;
2. Obtém-se uma referência ao componente, representada por uma de suas interfaces;
3. Se a interface utilizada for a de gerenciamento, é possível:
   3.1. Adicionar novas coleções em tempo de execução;
   3.2. Remover coleções existentes;
   3.3. Monitorar o *status* de utilização das coleções;

---

[28] Do verbo *dispense*, distribuir, dispensar

3.4. Encerrar a execução do componente;

4. Se a interface utilizada for a de suporte à coleção de objetos, é possível:
   4.1. Alocar um objeto de uma determinada coleção;
   4.2. Dealocar um objeto, devolvendo-o à coleção.

5.4.3 Flexibilidade

O componente *dispenser* pode ser (re)utilizado tanto localmente, através da sua incorporação à camada de infra-estrutura das aplicações, quanto de forma distribuída, através da utilização de suas interfaces remotas RMI. O componente define e implementa a abordagem de coleções de objetos, independente do tipo de objetos utilizados em tempo de execução. Para os casos em que é necessário definir coleções onde a instanciação de objetos difere da implementação *default* do componente, é possível estender o componente, redefinindo o comportamento de instanciação das classes utilizadas nas coleções.

O componente ainda oferece mecanismos de configuração em duas frentes: através de arquivos de configuração e via interface remota de gerenciamento RMI. Em ambos os casos é possível configurar uma ou mais coleções definidas em termos dos seguintes parâmetros: a classe que define o tipo de objeto utilizado na coleção, o número mínimo e máximo de objetos que o componente pode instanciar, se um objeto da coleção pode ser compartilhado, o número máximo de usuários concorrentes por objeto da coleção e a classe responsável por instanciar os objetos da coleção, para os casos em que o componente é estendido.

5.4.4 Projeto

Como discutido na seção 5.2.4, após a implementação do componente *dispenser,* os projetos dos componentes BD e *dispenser* foram integrados, de forma que o mecanismo de *pool* de conexões do primeiro fosse construído sobre os serviços do segundo.

Assim como o componente *Log*, o componente *dispenser* pode ser utilizado local ou remotamente; no primeiro caso, o componente é incorporado à camada de infra-estrutura da aplicação usuária que se baseia nos seus serviços para implementar uma política de coleções de objetos; no segundo caso, configura-se o componente para funcionar como um servidor de objetos distribuídos RMI, onde as aplicações usuárias utilizam o componente para alocar objetos distribuídos a partir do nome da coleção. Neste projeto, o componente *dispenser* foi empregado das duas formas: localmente para implementar o mecanismo de *pool* de conexões do componente BD, e remotamente, para permitir o compartilhamento dos objetos servidores do serviço de *log*, implementado pelo componente *Log*.

Desta forma, podemos afirmar que arquitetura de *software* do componente *dispenser* é híbrida, pois pode ser empregado tanto no modelo de camadas, quanto através de interfaces remotas RMI. Além disso, o componente emprega os padrões de projeto *bridge, factory method* e *template method*, para desacoplar o componente dos tipos de coleções e objetos suportados, e permitir a incorporação de novos tipos de coleções e novos mecanismos de controle de ciclo de vida de objetos.

5.4.5 Implementação

A Figura 5.5 ilustra o modelo de classes do componente *dispenser*, da qual podemos destacar:

- As interfaces RMI para acesso (`RMIDispenser`) e gerenciamento das coleções de objetos (`RMIDispenserManager`);

- O ponto de acesso ao componente (`DispenserComponent`), quando utilizado localmente;
- A estrutura dos padrões de projeto *bridge* e *factory method* envolvendo as classes `DispenserComponent` e `ServicePool`, além do padrão de projeto *template method* representado pelos métodos `newInstance()` e `doShutDown()` da classe `ServicePool`;
- A interface utilizada na notificação da alocação/dealocação de objetos (`Containee`);
- A classe que representa um objeto de uma coleção (`ServiceRef`);
- A extensão do componente *dispenser* para ser reutilizado no projeto do componente BD (`DBConnectionManager`, `DBPool`).

O controle do ciclo de vida de objetos no componente *dispenser* é implementado sobre a interface `Containee`, e dos métodos *hook* `newInstance()` e `doShutDown()` da classe `ServicePool`. Na implementação *default* do componente, se configurarmos uma coleção de objetos do tipo `String`, por exemplo, o componente criará os objetos da coleção através de chamadas sucessivas a `newInstance()`, cuja implementação é do tipo `Class.forName("String").newInstance()`. Esta abordagem funciona somente para as classes que possuem construtores públicos sem parâmetros, ou seja, um construtor do tipo `public String()`. Como nem sempre é possível garantir esta premissa, o componente define os métodos `newInstance()` e `doShutDown()` como métodos *hook,* que podem ser sobrecarregados em subclasses da classe `ServicePool` para redefinir a forma como as instâncias de objetos são criados. No caso do componente BD, por exemplo, em que um objeto corresponde a uma conexão, e uma coleção corresponde a um *pool,* não é possível utilizar o mecanismo *default* de ciclo de vida do componente *dispenser,* pois é necessário inicializar cada conexão através de uma chamada ao driver JDBC e passar uma série de parâmetros que identificam a URL, o usuário e a senha da conexão.

Quando uma classe que define os objetos de uma coleção implementa a interface `Containee`, toda requisição de reserva e descarte de objetos gera uma notificação ao objeto reservado/descartado; a idéia é utilizar as operações de notificação para executar as atividades de inicialização ou encerramento de cada objeto. Em outras palavras, se um objeto de uma coleção implementa a interface `Containee`, é notificado a cada operação de reserva, descarte ou encerramento.

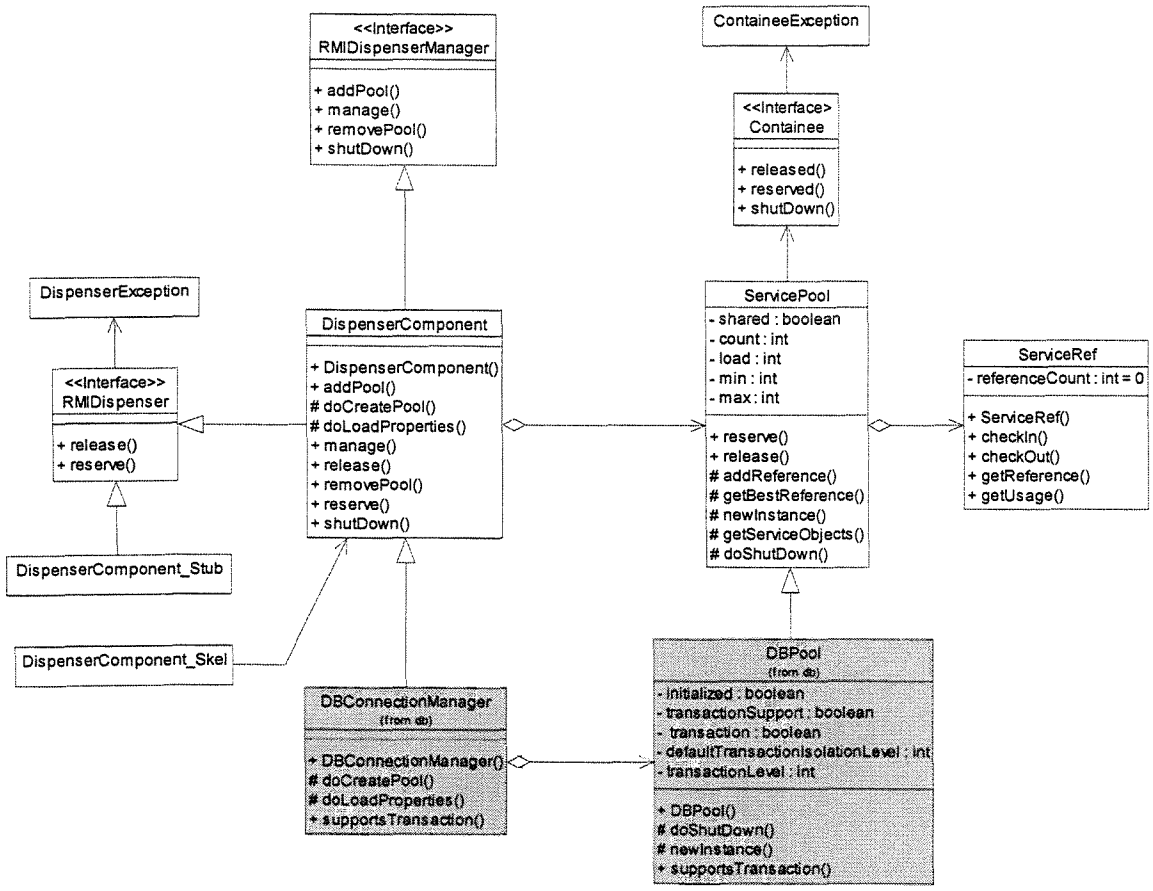


Figura 5.5 - Modelo de classes do componente *dispenser*

O componente *dispenser* oferece mecanismos de configuração através da interface remota de gerenciamento, para os casos em que se utiliza uma aplicação de administração com requisitos de gerenciamento *online*, em tempo de execução; também é possível configurá-lo através de um arquivo de configuração similar ao apresentado no exemplo do Código 5.6, que possui os seguintes parâmetros:

1. **Nome** da coleção/*pool*: como o arquivo pode conter a configuração de diversas coleções, o componente interpreta como a configuração da coleção `inex_read`, por exemplo, todas os parâmetros com prefixo `inex_read` (Código 5.6).
2. Flag para indicar se os objetos da coleção podem ser **compartilhadas** (campo `shared`): uma coleção compartilhada, é aquela que permite concorrência de acesso aos objetos, onde é possível haver mais de um usuário simultâneo utilizando um mesmo objeto.
3. Número **mínimo** e **máximo** de objetos da coleção: campos `min` e `max`, informam ao componente quantos objetos deverão ser instanciados no início da execução do componente (`min`), e quantos objetos poderão ser criados em tempo de execução caso o número máximo de usuários por objeto permitido seja alcançado e existam requisições de objetos pendentes.
4. Número **máximo de usuários por objeto** compartilhado: campo `load`.
5. **Classe** que define o tipo do objeto da coleção: campo `class`, representa o nome completo da classe, incluindo o nome do *package* (exemplo: `java.lang.String`);

6. **Classe** responsável pela inicialização (instanciação) dos objetos da coleção: campo `loader`, corresponde ao nome da classe que implementa a funcionalidade específica de tratamento do ciclo de vida de objetos da coleção (`DBPool`, por exemplo), ou `ServicePool`, se o componente não for estendido.

5.4.6 Interface

O componente *dispenser* define três interfaces e 9 operações:

1. Interface de acesso às coleções de objetos:
   1.1. `reserve(String poolName)`: recupera um objeto da coleção identificada por `poolName`; ao consultar a coleção na procura por objetos disponíveis, o componente verifica a possibilidade de criar novos objetos caso não exista nenhum objeto disponível;
   1.2. `release(String poolName, Object obj)`: devolve o objeto da coleção identificada por `poolName`; caso a coleção seja compartilhada, o componente apenas decrementa o contador de utilização do objeto; caso a coleção não seja compartilhada, o objeto é reinserido na coleção.
2. Interface de gerenciamento das coleções de objetos:
   2.1. `addPool(Properties properties)`: adiciona uma coleção em tempo de execução, especificando os parâmetros de inicialização (número mínimo e máximo de objetos, *loader,* etc.) através do objeto `properties`;
   2.2. `removePool(String poolName)`: remove a coleção identificada por `poolName` em tempo de execução, notificando e encerrando todos os objetos da coleção;
   2.3. `manage()`: pesquisa o status do componente com relação ao número de coleções configuradas, número de objetos por coleção, número de objetos alocados e disponíveis, nível de utilização por objeto, etc.;
   2.4. `shutDown()`: encerra o componente *dispenser*, notificando e encerrando todos as coleções e seus respectivos objetos;

5.4.7 Prova de Conceito

Como o componente BD foi construído sobre o componente *dispenser*, podemos destacar esta reutilização como prova de conceito. A Figura 5.5 ilustra a extensão do componente *dispenser* através da implementação dos padrões de projeto *bridge*, *factory method* e *template method*, que envolve a sobrecarga das classes `DBComponent` e `ServicePool` pelas classes `DBConnectionManager` e `DBPool`.

Neste caso, o componente *dispenser* foi estendido para incorporar a classe de suporte ao controle do ciclo de vida de conexões com bancos de dados (`DBPool`), que difere do controle *default* do componente. A configuração do componente *dispenser* é composta pelos parâmetros `class`, `loader`, `min`, `max`, `load` e `shared`, e corresponde a um subconjunto da configuração do componente BD (Código 5.6).

```
...
DBConnectionManager conManager = new DBConnectionManager("db.ini");

public Transaction beginTransaction(String poolName) {
  Transaction result = null;
  Connection con = null;

  try {
    if ((con = (Connection) conManager.reserve(poolName))!=null &&
        !con.isClosed()) {

      inTransaction= true;
      result = new Transaction(con, poolName);
      result.begin();
    }
  } catch (Exception e) { ... }

  return result;
}

public void endTransaction(Transaction transaction) {
  try {
    if (transaction!= null) {
      transaction.end();
      inTransaction = false;
      conManager.release(transaction.getDatabaseName(),
                         transaction.getConnection());
    }
  } catch (Exception e) { ... }
}
```

Código 5.9 - Exemplo de utilização do componente *dispenser*

O exemplo do Código 5.9 representa a utilização do componente *dispenser* pelo componente BD, onde é possível notar que as operações de reserva e descarte de conexões são delegadas ao componente *dispenser*; ainda é possível verificar que o componente BD utiliza a versão local do componente *dispenser*, pois nenhuma interface remota foi utilizada.

5.5 Conclusões

Como a arquitetura de *software* empregada neste projeto é baseada em camadas, decidimos pela implementação de componentes de *software* de infra-estrutura, que são os que oferecem o maior grau de reutilização, além de altos índices de ganho de produtividade; a camada de infra-estrutura, assim como os componentes que a compõem, formam a fundação sobre a qual são construídas as aplicações. O potencial de reutilização dos componentes pôde ser validado através do próprio projeto dos componentes, como podemos observar através da Figura 5.6.

Os componentes *Log* e Segurança empregam o componente BD para persistir informações em bancos de dados relacionais; o componente BD foi construído sobre o mecanismo de coleções de objetos compartilhados oferecido pelo componente *dispenser*; por fim, o acesso a versão distribuída do componente *Log* pode ser gerenciado através do componente *dispenser*, que neste caso, funciona como um servidor de nomes e gerenciador do ciclo de vida dos objetos servidores de *log*.

Os componentes deste projeto foram implementados de forma iterativa e evolutiva, sob a arquitetura de *software* baseada em camadas e objetos distribuídos. Neste processo, a construção de novos componentes baseados em outros já existentes contribuiu sobremaneira para a estabilização das funcionalidades e da estrutura das interfaces de cada componente.

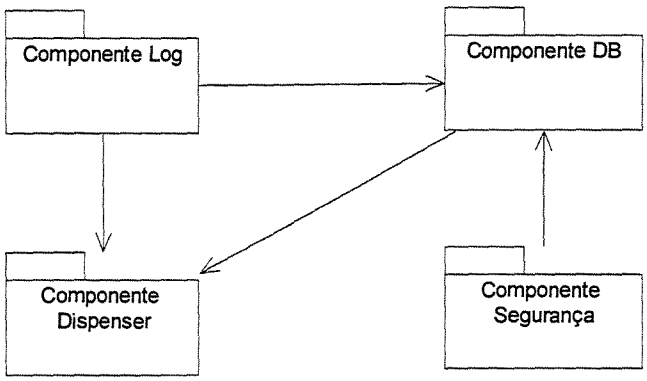


Figura 5.6 - Modelo de componentes do projeto

Capítulo 6

Projeto *Intranet Express* (INEX)

A INEX, ou *Intranet Express*, é um sistema concebido e implementado pela equipe de desenvolvimento do laboratório A-HAND, Unicamp, da qual faz parte o autor deste trabalho. Este capítulo apresenta o projeto de desenvolvimento da INEX, explica as razões da sua incorporação a este trabalho e descreve os resultados desta integração.

Um dos objetivos iniciais deste trabalho era realizar uma prova de conceito dos componentes de *software* apresentados no capítulo 5, reutilizando-os na construção de aplicações de um protótipo de uma *intranet*. Como alternativa a construir diversas aplicações de pequeno porte que constituíssem um protótipo de *intranet*, optamos por construir uma aplicação de médio/grande porte que pudesse ser reconfigurada para oferecer as funcionalidades mais comuns em *intranets*. A aplicação em questão é a INEX, um sistema gerenciador de informações para *intranets*, que teve parte do desenvolvimento incorporado a este trabalho de pesquisa. Como a INEX é um sistema cliente/servidor *Web*, optamos por realizar a prova de conceito dos componentes através da implementação da porção servidor deste sistema.

Para apresentar o projeto INEX e os resultados da integração com os componentes, optamos por empregar a seguinte estrutura: especificação da INEX, decisões de projeto e implementação, possibilidades de utilização e extensão da INEX e conclusões sobre a integração dos projetos INEX/componentes.

6.1 Especificação

A INEX é um sistema *Web* para gerenciamento de informações de *intranets*. Para tanto, deve oferecer mecanismos de armazenamento, recuperação, organização, compartilhamento e controle de acesso a documentos. Cada documento ou unidade de informação manipulada pela INEX é conhecida como um item. Inicialmente, a INEX deverá suportar os tipos de itens **arquivo**, **URL** e **texto livre** para representar informações, além dos tipos **lista**, ***link*** e **comentário** para facilitar a organização, recuperação e compartilhamento de itens. Além disso, o projeto da INEX deverá

considerar a possibilidade da incorporação de novos tipos de documentos, de forma a minimizar os impactos deste tipo de mudança.

Todo item da INEX deve possuir alguns campos básicos de informação, como um ID que o identifica unicamente, a sua descrição, as datas de criação, expiração e último acesso/modificação, o seu proprietário, a relação das permissões de acesso que indique quem pode lê-lo, atualizá-lo e gerenciá-lo, e um conjunto de palavras-chave que descrevem o seu conteúdo.

Um arquivo é um item submetido à INEX, e é similar aos arquivos submetidos pela Internet através do processo de *upload*. Um arquivo pode ser de formato HTML, DOC, XLS ou qualquer outro formato tratado pelos *browsers*, *helper applications* e *plug-ins*. Uma URL é um endereço de *email*, de um *site* WWW, de um servidor de *news*, ou de um servidor FTP, por exemplo. Um item do tipo texto livre é aquele utilizado na inclusão de documentos digitados pelo usuário.

As informações manipuladas pela INEX são organizadas através de hierarquias de informações, ou listas, conforme pode ser observado na Figura 6.1. Uma hierarquia é uma abstração que permite classificar as informações na INEX. Uma hierarquia intitulada *publicações*, por exemplo, representa uma classificação sob a qual poderíamos associar documentos ou outras subcategorias, como *procedimentos, relatórios* e *artigos* (Figura 6.1).

Um *link* é um atalho para um item qualquer da INEX e é empregado para organizar e compartilhar informações sem realizar replicação de dados. A INEX deverá permitir a criação de *links* que apontam para itens de informação, listas e comentários; ao criar *links* para listas, por exemplo, será possível acessar os itens das listas através de hierarquias distintas, o que permitirá personalizar a visão que cada usuário possui sobre as hierarquias de informação da INEX. O exemplo da Figura 6.1 ilustra dois *links*, um para a lista *relatórios* e outro para o documento *componentes de software*. *Links* na INEX são referências simbólicas que persistem enquanto o item apontado existir; ao remover um item na INEX, todos os *links* que o referenciam são automaticamente removidos.

Um comentário é uma informação que descreve as impressões de um usuário sobre um item. Comentários constituem um mecanismo eficiente para facilitar a busca por informações relevantes, pois oferece ao usuário a possibilidade de filtrar documentos através da leitura das meta-informações correspondentes. Para adicionar um comentário sobre um item, o usuário deverá visualizá-lo através da interface para navegação na hierarquia de listas e itens da INEX (Figura 6.1). Além dos campos básicos de informação de um item, um comentário deverá conter um título e o texto do seu conteúdo. Comentários só poderão ser apagados pelos respectivos proprietários, ou pelos usuários com privilégios de escrita/gerenciamento sobre o item correspondente; eventualmente, itens, listas e comentários serão removidos automaticamente a partir da data de expiração. Na Figura 6.1, podemos destacar um comentário relacionado ao item do tipo *link componentes de software*.

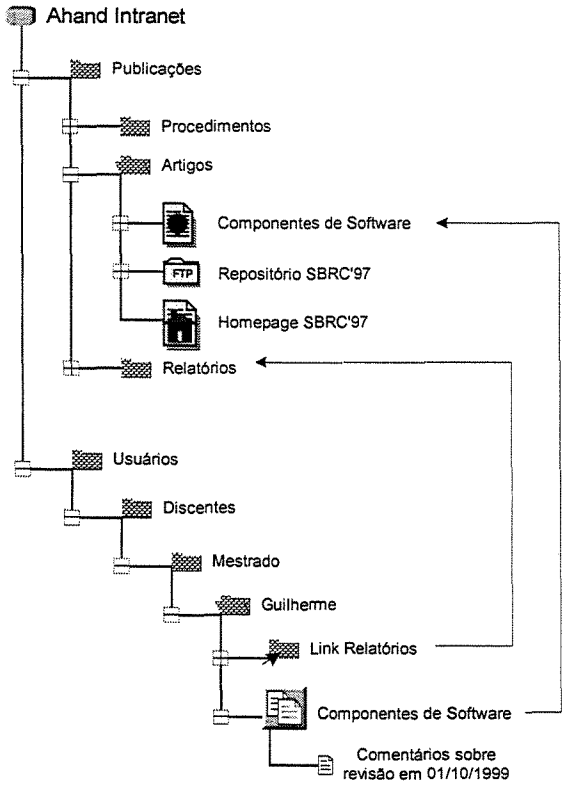


Figura 6.1 - Exemplo de uma hierarquia de itens da INEX

Como a INEX é um sistema inerentemente distribuído e multi-usuário, é imprescindível que implemente mecanismos de controle de acesso aos itens. Para tanto, a INEX deverá suportar o conceito de usuários, grupos e permissões de acesso sobre itens. Se o usuário ou grupo ao qual pertence o usuário tiver permissão de leitura sobre um item, será possível ler o conteúdo do item, visualizar o conteúdo da lista, no caso de itens do tipo lista, criar um *link* para o item numa lista particular ou ainda adicionar um comentário sobre o item em questão. Se o usuário tiver permissão de escrita, poderá atualizar ou remover o item, remanejá-lo para outras listas, adicionar itens à lista, no caso de itens do tipo lista, e remover comentários associados ao item, se houver.

Quanto às facilidades de recuperação de informações, a INEX deverá implementar um mecanismo de busca de itens através das palavras-chave e campos básicos discutidos acima (proprietário, assunto, descrição, etc.). A interface gráfica da INEX deverá incorporar um navegador como o apresentado no exemplo da Figura 6.1, onde cada tipo de item é representado sob um formato/ícone diferente; ao abrir uma lista, a INEX deverá mostrar um resumo das informações sobre os itens contidos na lista; o acesso aos comentários de um item deverá ser similar à abordagem de acesso aos itens de uma lista - ao expandir um item, a INEX deverá mostrar um resumo dos seus comentários; ao abrir um item do tipo *link,* a INEX deverá recuperar o item original de forma transparente, sem remeter o usuário à lista que contém o item original. O mecanismo de busca ainda poderá ser utilizado em conjunto com o navegador, nos cenários em que o usuário deseja realizar uma busca por itens a partir de uma determinada lista.

Por fim, o projeto da INEX deverá considerar alguns requisitos não funcionais, como a possibilidade de incorporar novos tipos de itens, além da possibilidade de extensão para incluir tratamentos específicos a determinados tipos de itens. Considere por exemplo, a definição dos novos tipos de itens *email* (um tipo de documento), *mailbox* (um tipo de lista). A INEX deverá oferecer flexibilidades de extensão, para que seja possível realizar tanto a persistência destes

novos tipos de itens, quanto a incorporação de tratamento específico para estes itens; em outras palavras, ao incluir um item do tipo *email* numa lista *mailbox* configurada como *outbox*[29], a INEX deveria armazenar o item e disparar um processo de envio da mensagem representada pelo item *email*.

6.2 Projeto

Os projetos da INEX e desta dissertação de mestrado surgiram concomitantemente. A idéia original era a de promover a sinergia entre os projetos, pois enquanto o estudo das arquiteturas de *software Web* e a implementação de componentes poderia facilitar a construção da INEX, a (re)utilização dos componentes e dos resultados dos estudos sobre arquitetura de *software* constituiria uma prova de conceito sobre os resultados deste trabalho.

O processo de desenvolvimento da INEX foi iniciado através da abordagem de protótipos. As primeiras versões do protótipo da INEX foram desenvolvidas através da arquitetura de *software* baseada em CGIs. A opção pelo uso de CGIs foi realizada apenas para construção dos protótipos, pois assim como ressaltamos no capítulo 3.4.1, poderia inviabilizar o desenvolvimento de uma arquitetura flexível, adaptável e extensível. Neste sentido, optamos pela arquitetura ilustrada na Figura 6.2, que em resumo é baseada em camadas, objetos distribuídos RMI e no modelo MVC (capítulo 3.2.3). Como a INEX é um projeto de médio/grande porte, optamos pela divisão da implementação entre pesquisadores, alunos de mestrado, iniciação científica e assim por diante. Desta forma, incorporamos a implementação da camada de controle da aplicação (Figura 6.2) da INEX como forma de realizar a prova de conceito sobre os componentes desenvolvidos (capítulo 5) e os estudos das arquiteturas de *software Web*.

A arquitetura apresentada na Figura 6.2 corresponde à arquitetura de camadas discutida no capítulo 3.2.2. A primeira camada concentra os serviços de base utilizados por toda a aplicação, como um servidor de banco de dados para realizar persistência de itens, sistema de arquivos para armazenar os itens do tipo arquivo, ORB RMI para permitir a integração entre as camadas de apresentação e aplicação, etc.

A camada de componentes de infra-estrutura corresponde aos componentes implementados como parte deste trabalho (capítulo 5). Apesar de constar na arquitetura da INEX, são potencialmente reutilizáveis em outras aplicações. O componente BD é utilizado na persistência de itens, o componente de segurança no controle de acesso aos itens, o componente *Log* no registro de informações sobre utilização e depuração e o componente *dispenser* pelos próprios componentes BD e *Log*.

A camada de controle da aplicação INEX é responsável por implementar a lógica da aplicação; como é construída sobre os componentes de infra-estrutura, todos as operações de infra-estrutura referentes a registro de *logs*, persistência de dados e controle de acesso são delegadas à camada inferior. Como podemos observar através da Figura 6.2, a camada de aplicação é composta por 6 blocos, ou subcamadas:

1. Camada de comunicação: concentra os detalhes de implementação da interface remota RMI da INEX, controle de sessões, tratamento de exceções, *caching* de informações no servidor, etc. A camada de comunicação recebe requisições de clientes através de RMI (*applet*, *servlet* ou uma aplicação *standalone*) e as repassa à subcamada de aplicação.

---

[29] Abordagem empregada pelas principais ferramentas de *email* que definem listas do tipo *inbox* (ou caixa de entrada) para recebimento de novas mensagens, e *outbox* (ou caixa de saída) para envio de mensagens.

2. Classes de dados: corresponde às informações gerenciadas pela INEX, como itens, listas, comentários, *links* etc (Figura 6.3).
3. Camada de controle da aplicação: é a camada que representa a semântica da INEX; requisitos funcionais como "só é possível adicionar um item a uma lista se o proprietário do item tiver permissão de escrita sobre a lista ..." , ou ainda "não é possível remover uma lista que contenha outros itens ..." correspondem a lógica implementada nesta camada. Como a arquitetura da INEX é subdividida entre várias camadas, a camada de controle ainda implementa a lógica de integração dos componentes que utiliza.
4. Camada de segurança: corresponde à implementação do padrão de projeto *adapter*, onde é realizado o mapeamento entre as interfaces da camada de segurança e do componente de segurança; todos os detalhes relacionados ao controle de acesso a itens da INEX são concentrados nesta camada.
5. Camada de acesso a dados: assim como a camada 4, concentra todos os detalhes de persistência e recuperação de itens.
6. Camadas de extensão: é uma camada que oferece a possibilidade de incorporar tratamentos específicos sobre itens, como o exemplo anterior sobre itens do tipo *email* e *mailbox*, e corresponde aos requisitos não funcionais de extensão da INEX discutidos na especificação. A estrutura da camada de extensão é discutida em detalhes na seção 6.3.

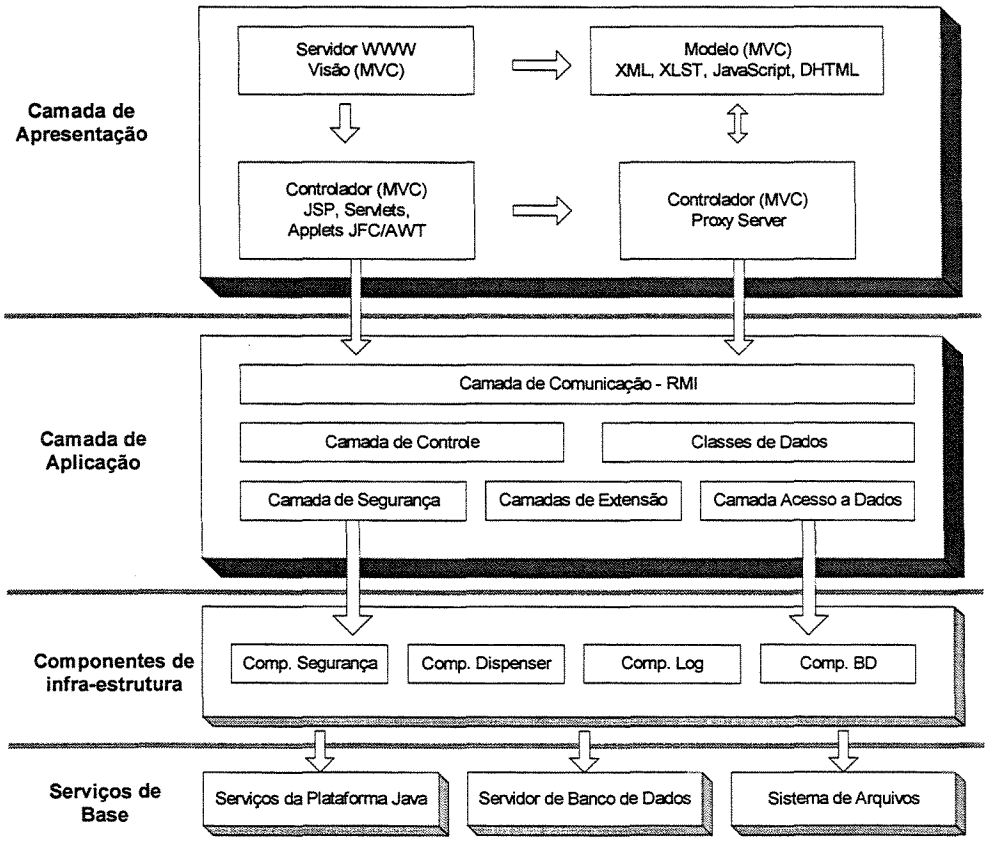


Figura 6.2 - Arquitetura de *software* do sistema INEX

A camada de apresentação da INEX corresponde a porção cliente do sistema, e envolve a implementação de interfaces gráficas, interação cliente/servidor para utilização dos serviços RMI da camada de comunicação, *caching* de informações, além de questões sobre compatibilidade entre *browsers* (*applet* x *servlet*), separação de dados e interface gráfica através do modelo MVC (*servlets,* XML, XLST), entre outros. Como o foco deste trabalho está centrado na implementação da camada de aplicação, não apresentaremos uma discussão detalhada sobre os componentes da camada de apresentação.

6.3 Implementação

Para facilitar a inclusão de novos tipos de itens à INEX, decidimos pela implementação das hierarquias de itens através do padrão de projeto *composite* (Figura 6.3), que permite representar hierarquias todo-parte. Desta forma, qualquer informação gerenciada pela INEX é uma subclasse de `Item`. As operações da interface da camada de aplicação da INEX que lidam com itens (listas, comentários, *links*, documentos, etc.) são descritas através da classe `Item`, o que torna a sua utilização transparente, independente do tipo de item manipulado em tempo de execução. Esta estrutura também oferece grande flexibilidade para extensão da INEX, sem que seja necessário alterar a interface ou os componentes da camada de aplicação.

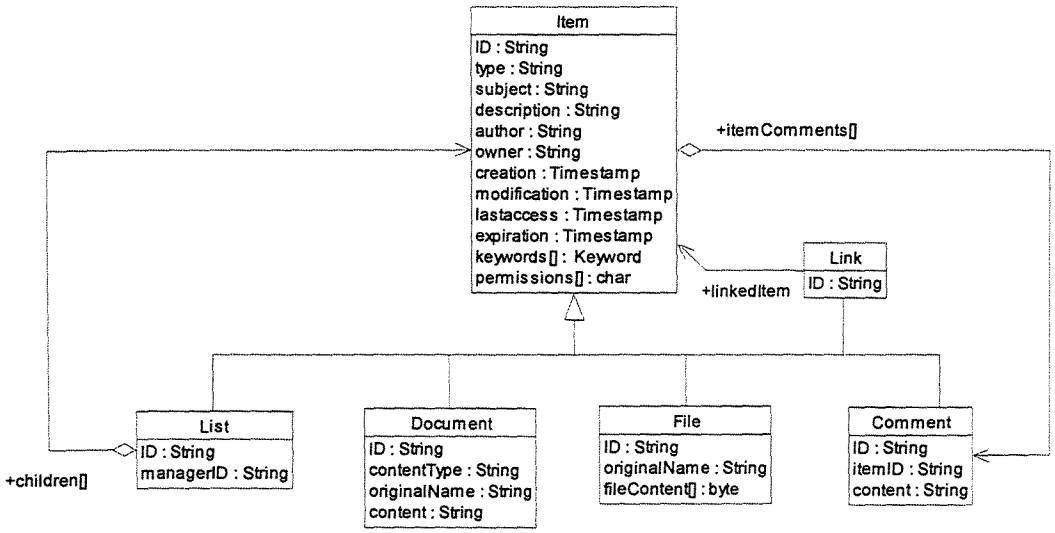


Figura 6.3 - Diagrama das classes de dados da INEX

A camada de comunicação, e portanto a parte servidor da INEX, dispõe os serviços implementados pela camada de aplicação à camada de apresentação através de uma interface RMI de 10 operações:

1. `public long login(String username, String password)`: esta é a operação inicial que todo cliente INEX deve executar antes de utilizar os demais serviços; ao executar o *login*, o usuário é validado com a senha e uma chave de 64 *bits* é gerada para representar o ID da sessão estabelecido com o servidor da INEX. Caso o usuário ou a senha sejam inválidos, uma exceção é gerada. Caso a aplicação cliente deixe de utilizar os serviços da INEX por mais de 15 minutos, a sessão é expirada, e o usuário deve refazer o *login*. É importante ressaltar que o ID da sessão deve ser repassado a todas as operações subsequentes ao *login* para que o servidor da INEX possa identificar unicamente o usuário que requisita uma

operação. Este mecanismo é imprescindível para a manutenção da segurança, pois garante que somente as aplicações clientes que efetuaram o *login* possam utilizar os serviços da INEX.

2. `public String addItem(long sessionID, String sourceID, Item item)`: adiciona o item representado pelo objeto `item` à lista identificada por `sourceID`, desde que o usuário da sessão `sessionID` possua permissão de escrita sobre a lista de destino. Todo item na INEX possui uma identificação única, o *UID* (*unique identifier*), associado no momento de sua inclusão no sistema. Se a operação for executada com sucesso, retorna-se o *UID* do item inserido, caso contrário, uma exceção é gerada. Como esta operação é definida em termos da classe `Item`, pode ser utilizada na inclusão de qualquer tipo de item, como um comentário, *link*, lista ou documento. O tratamento da persistência dos itens é realizado pela camada de acesso a dados, que delega ao componente BD a maioria das operações. Nos casos em que não há suporte do componente BD, como no mapeamento dos relacionamentos 1:N entre objetos da classe `List` e `Item`, por exemplo, a implementação é suprida pela camada de acesso a dados; a solução atual contorna este problema através do emprego de classes de dados que representam estes tipos de relacionamentos.
3. `public Item getItem(long sessionID, String itemID)`: recupera o item de *UID* `itemID` no formato em que foi inserido na INEX, desde que o usuário da sessão `sessionID` possua permissão de leitura sobre o item, ou seja, se o item for do tipo lista, um objeto do tipo `List` será recuperado, se for um documento, um objeto do tipo `Document`, e assim por diante. Vale ressaltar que esta operação recupera todos os campos de um item, e é indicada portanto quando for necessário visualizar os detalhes de um item.
4. `public Item[] getItemComments(long sessionID, String itemID)`: recupera o cabeçalho dos comentários do item de *UID* `itemID`, sob o formato de um *array* de objetos do tipo `Item`, desde que o usuário possua permissão de leitura sobre o item. Note que o termo *cabeçalho* indica que somente os campos básicos, como autor, data de criação e assunto são recuperados. Para visualizar os detalhes de um comentário, deve-se realizar uma chamada a `getItem`.
5. `public Item[] getItemParents(long sessionID, String itemID)`: recupera o cabeçalho dos itens do tipo `List` que possuem o item, ou um *link* para o item identificado por `itemID`, que sejam acessíveis pelo usuário corrente. Assim como ocorre na operação 4, as listas "pai" são recuperadas sob o formato de um *array* de objetos do tipo `Item` somente com os campos básicos.
6. `public Item[] getItemChildren(long sessionID, String fromListID)`: recupera o cabeçalho dos itens contidos na lista identificada por `fromListID` que sejam acessíveis pelo usuário corrente. Assim como ocorre nas operações 4 e 5, o resultado é expresso num *array* de objetos do tipo `Item` somente com os campos básicos. Este tipo de operação deve ser executada sempre que o usuário da INEX expandir uma lista para visualizar seu conteúdo, ocasião em que somente os cabeçalhos dos itens são mostrados.
7. `public void moveItem(long sessionID, String itemID, String sourceListID, String targetListID)`: remaneja o item identificado por `itemID` da lista de origem representada por `sourceListID` para a lista destino de *UID* `targetListID`, desde que o usuário corrente possua permissões de escrita sobre o item e a lista destino.

8. `public void removeItem(long sessionID, String itemID)`: remove o item identificado por `itemID`, desde que o usuário corrente possua permissão de escrita sobre o mesmo. Esta operação é recursiva, dado que todos os comentários e *links* associados ao item também são removidos para evitar referências inválidas. Na implementação atual da INEX não é possível remover uma lista não vazia.
9. `public Item[] searchItems(long sessionID, Item item, Criteria[] criteria, String fromListID)`: executa uma busca na hierarquia de itens da INEX, procurando por itens do mesmo tipo que o objeto `item`, a partir da lista identificada por `fromListID` e que satisfaçam os critérios especificados através do objeto `criteria`; porém, são retornados apenas os itens acessíveis ao usuário (permissão de leitura) corrente representado pela sessão `sessionID`. O argumento `criteria` é do mesmo tipo da classe `Criteria` discutida na seção 5.2.5 do capítulo 5; qualquer atributo de `item` que seja gravado em banco de dados pode ser expresso em termos de critérios num *array* de objetos do tipo `Criteria`. Para representar critérios baseados em palavras-chave, deve-se empregar o nome de atributo *keyword* e agrupá-los em sequência, conforme exemplo do Código 6.1. Da mesma forma que as operações 4, 5 e 6, somente os cabeçalhos dos itens são recuperados na forma de um *array* de objetos do tipo `Item`.
10. `public long updateItems(long sessionID, Item item, String[] fields, Criteria[] criteria, String fromListID)`: executa uma busca pelos itens do mesmo tipo que o objeto `item`, a partir da lista identificada por `fromListID` e que satisfaçam os critérios especificados em `criteria`; filtra os itens sobre os quais o usuário corrente tem acesso de escrita e os atualiza, modificando somente os atributos definidos em `fields` com os valores correntes do objeto `item` - se a operação for executada com sucesso, o número de itens atualizados é retornado.

```
Criteria[] c = new Criteria[] {
 new Criteria("creation", Operators.GREATER,
              new Date("01/01/1999"), Operators.AND),

 new Criteria("keyword",  Operators.WILD,
              "Componentes*", Operators.OR),

 new Criteria("keyword",  Operators.WILD,
              "Intranets*", Operators.OR),

 new Criteria("keyword",  Operators.WILD,
              "Arquitetura de Software*", null)
                                  };
```

Código 6.1 - Exemplo de utilização da classe `Criteria` na INEX

Em termos de flexibilidade, podemos constatar através da interface apresentada acima que há transparência no tratamento dos tipos de itens suportados pela INEX. Como a camada de acesso a dados da INEX é construída sobre o componente BD - que oferece transparência na persistência de objetos, e como um item é representado por uma hierarquia todo/parte - que permite expressar qualquer subtipo de item em termos da classe `Item`, podemos afirmar que a arquitetura atual suporta a incorporação de novos tipos de itens de forma transparente. Vale lembrar que a INEX também herda as flexibilidades de cada componente discutidas no capítulo 5; é possível configurá-la para funcionar sobre diferentes categorias e servidores de bancos de dados, por exemplo.

Além de suportar a persistência de novos tipos de itens, a INEX implementa uma camada de extensão que permite definir tratamentos específicos sobre estes novos itens. A idéia é oferecer flexibilidade suficiente para que a INEX seja (re)utilizada em situações imprevistas, como aquelas que exigem o gerenciamento de informações num formato diferente dos tipos de itens *default*, ou ainda nos casos em que é necessário redefinir o tratamento sobre os itens gerenciados. Em termos da disciplina de engenharia de *software*, a INEX é uma espécie de *framework* para gerenciamento de documentos, que permite incorporar componentes à camada de extensão para gerenciar novos tipos de documentos.

Os diagramas da Figura 6.4 e Figura 6.5 ilustram os detalhes do projeto da camada de extensão, onde podemos destacar:

- A implementação do padrão de projeto *publish/subscribe* (ou *observer*, Figura 6.5) [GHJ94][Pre94] envolvendo as classes **`ExtensionLayer`** e **`ExtensionHandler`** com o objetivo de desacoplar a camada de extensão dos componentes de extensão que podem ser incorporados à INEX;
- A interface para integração das camadas de aplicação e extensão representada por **`ExtensionContract`**, corresponde à interface que os componentes de extensão devem implementar para estender o comportamento da INEX;
- Um exemplo de extensão da INEX para o suporte/integração a *emails*, representado pelas classes de dados **`Mailbox`** e **`Email`** (Figura 6.4), e pelas classes de extensão **`MailboxHandler`** e **`EmailHandler`** (Figura 6.5);
- A integração entre as camadas de extensão e aplicação, dado que a implementação do tratamento dos novos tipos de itens pode requerer acesso aos serviços da camada de aplicação;

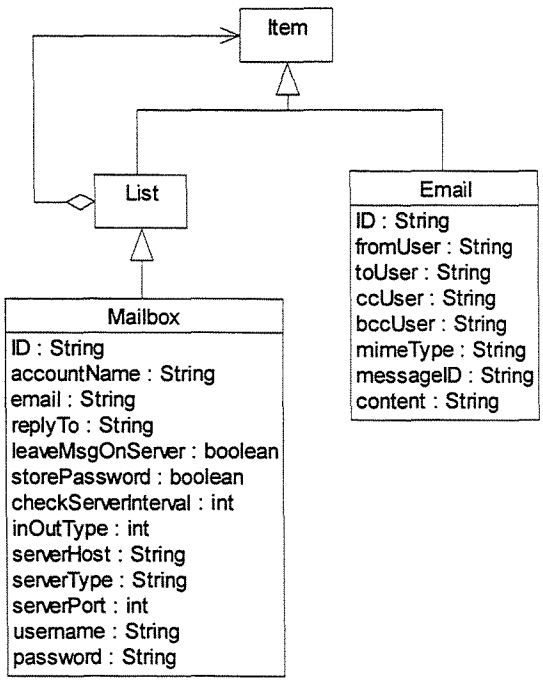


Figura 6.4 - Classes de dados para suporte a *email* na INEX

Como é possível observar através dos diagramas da Figura 6.4 e Figura 6.5, é possível estender a INEX através da definição dos novos tipos de itens e da sobrecarga e implementação dos métodos da classe **`ExtensionHandler`**; além disso, é necessário configurar o arquivo de inicialização da camada de extensão para informar quais componentes de extensão estarão ativos e quais operações da interface **`ExtensionContract`** cada componente tratará.

O exemplo do Código 6.2 ilustra o arquivo de configuração da camada de extensão e a definição dos componentes de extensão **`MailboxHandler`** e **`EmailHandler`**. Note que o prefixo utilizado na definição de um *handler* corresponde ao tipo de item que o *handler* gerencia; os parâmetros **`insert`**, **`update`**, **`remove`**, **`get`** e **`sons`** indicam quais os tipos de operação que a camada de extensão deverá interceptar e repassar ao *handler* para o tipo de item em questão. Para um item do tipo *mailbox,* por exemplo, a camada de extensão deverá notificar o *handler* **`MailboxHandler`** sempre que a camada de aplicação invocar as operações de inserção, remoção e recuperação dos subitens de um *mailbox*.

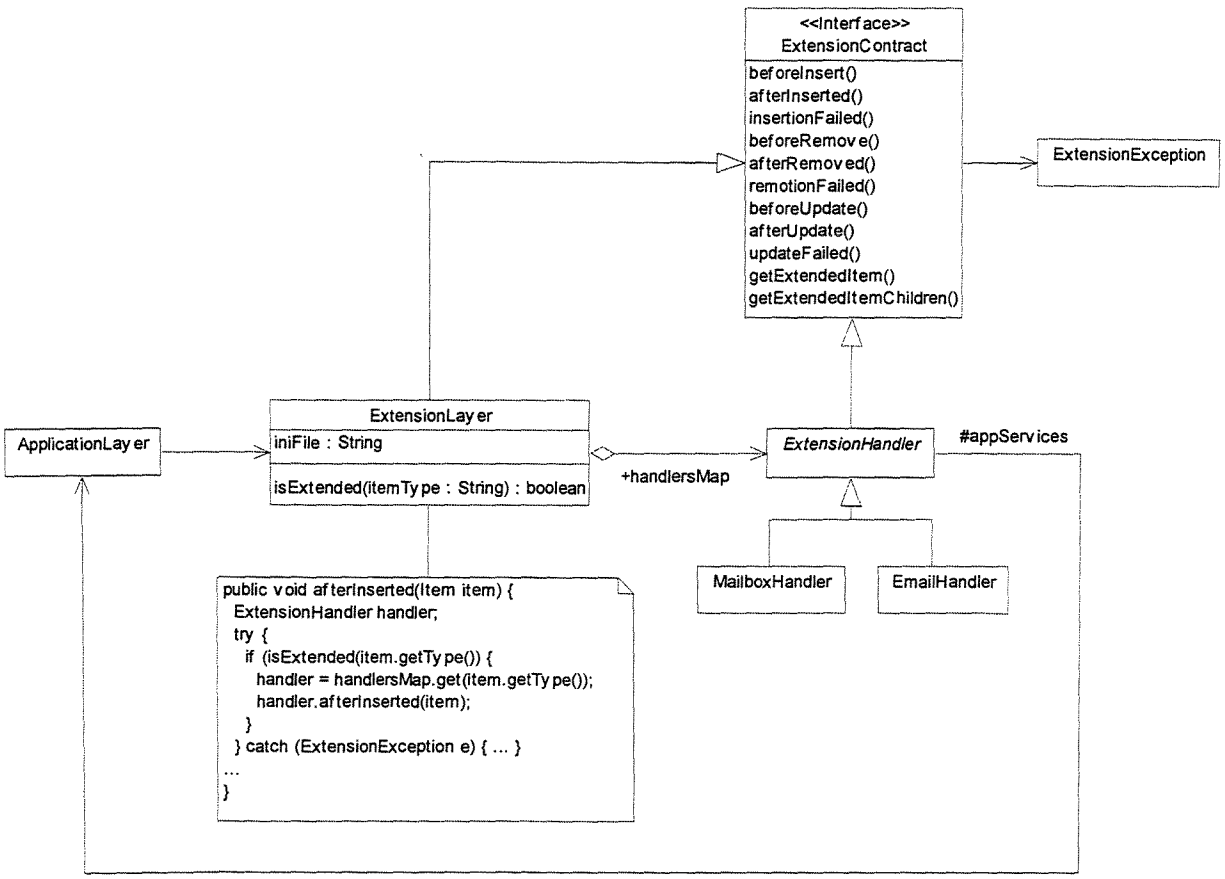


Figura 6.5 - Modelo de classes da camada de extensão da INEX

Para complementar o exemplo da extensão da INEX para suporte a *emails*, devemos ressaltar que as classes **`MailboxHandler`** e **`EmailHandler`** devem implementar mecanismos de suporte ao envio e recebimento de *emails*. A implementação do método **`afterInserted()`** da classe **`EmailHandler`**, por exemplo, deverá tratar o envio do *email* correspondente ao item, enquanto o

método **getExtendedItemSons()** da classe **MailboxHandler** deverá tratar a recuperação de novos *emails* junto ao servidor de *email*.

```
# definição da classe de extensão MailboxHandler
mailbox.class     = inex.common.Mailbox
mailbox.handler   = inex.server.MailboxHandler
mailbox.insert    = true
mailbox.update    = false
mailbox.remove    = true
mailbox.get       = false
mailbox.children  = true

# definição da classe de extensão EmailHandler
email.class     = inex.common.Email
email.handler   = inex.server.EmailHandler
email.insert    = true
email.update    = false
email.remove    = true
email.get       = false
email.children  = false
```

Código 6.2 - Exemplo de configuração da camada de extensão da INEX

A configuração da INEX é realizada através de arquivos de configuração e segue o formato apresentado no Código 6.3. A estrutura do arquivo de configuração da camada de extensão é a apresentada no Código 6.2, enquanto o formato dos arquivos de configuração das camadas de acesso a dados e segurança seguem o padrão dos componentes BD e segurança apresentados no capítulo 5.

```
# definições da camada de comunicação
rmi_layer.host    = qwx.ahand.unicamp.br
rmi_layer.port    = 5000
rmi_layer.servant = INEX_AHAND

# definições da camada de aplicação
app_layer.domain         = ahand.unicamp.br
app_layer.sessions       = 500
app_layer.sessiontimeout = 900000 # 15 minutos

# definições das camadas de suporte
sec_layer.inifile = inex_security.ini
app_layer.inifile = inex_database.ini
ext_layer.inifile = inex_extension.ini
```

Código 6.3 - Exemplo de configuração da INEX

6.4 Extensibilidade

Embora tenhamos explorado a customização da INEX na seção anterior, nos limitamos a focar os detalhes de implementação. Como observamos anteriormente, a INEX pode ser considerada um *framework* para o gerenciamento de informações de *intranets*, e portanto, ser (re)utilizada na implantação de diversos serviços. No capítulo 2, apresentamos uma série de aplicações e serviços comumente implantados em *intranets* corporativas. Nesta seção, exploramos algumas possibilidades de extensão da INEX no sentido de implantar algumas destas aplicações e serviços.

6.4.1 *Help Online*

Este caso de uso representa a possibilidade de tratar as próprias páginas de informações de ajuda do sistema INEX como uma hierarquia INEX. A idéia é permitir que o administrador do sistema possa personalizar a hierarquia de ajuda de acordo com a comunidade de usuários, além de acrescentar documentação referente aos novos mecanismos incorporados ao sistema através de customização. Este caso de uso ainda pode ser estendido através da manutenção de listas FAQ sobre sistemas, produtos e serviços, que podem eventualmente seguir a abordagem de documentação da INEX.

6.4.2 Organograma

Se estendermos a INEX através do mapeamento dos itens lista/departamento e item/empregado, poderemos empregar a INEX para representar organogramas através de hierarquias de itens do tipo departamento e empregado. A idéia é disponibilizar e facilitar o acesso às informações sobre os funcionários de uma organização; pode-se definir um item do tipo empregado que relacione local de trabalho, cargo ocupado, formação, telefones e *emails* de contato, e uma lista do tipo departamento que relacione os objetivos do departamento, os projetos em andamento, os seus integrantes (funcionários), e assim por diante.

Outro caso de uso possível é definir um empregado como uma lista que corresponde à área *home*[30] do sistema reservada para o funcionário. Neste cenário, o usuário/empregado poderia criar itens e listas privadas, *links* para serviços e aplicações da *intranet*, ou mesmo listas do tipo *mailbox* para acesso ao seus *emails*.

6.4.3 Linkoteca

Uma biblioteca de *links* para *URLs* organizada em listas que representam assuntos e nível técnico, por exemplo. Como é comum cada usuário manter um conjunto de *URLs* próprio, pode-se utilizar o mecanismo de *links* da INEX para publicar as *URLs* de cada usuário numa área pública. A idéia é promover a cooperação entre os usuários, a partir da publicação dos seus *links* favoritos na *linkoteca*, inclusão de comentários, etc.

6.4.4 Documentação de Projetos

É a hierarquia da INEX que representa o repositório de projetos de uma organização. Pode-se criar uma lista para cada projeto, uma sublista para cada etapa do projeto, adicionar documentos e arquivos que representam os artefatos do projeto, além de promover a discussão entre os participantes através da criação de comentários sobre os artefatos do projeto. Esta hierarquia da INEX ainda pode ser integrada a outras hierarquias como a de procedimentos operacionais (seção 6.4.6), que contém regras e modelos para o desenvolvimento de projetos, ou a de controle de atividades (seção 6.4.9), para tratar da atribuição das tarefas de execução do projeto. Por fim, deve-se utilizar os mecanismos de segurança da INEX para garantir que somente os usuários participantes de um projeto tenham acesso; vale considerar ainda a criação de grupos com diferentes perfis de acesso dentro da hierarquia de informações de um mesmo projeto, dado que nem todos os participantes têm acesso a todas as informações do projeto.

---

[30] *Home* corresponde a abordagem empregada em sistemas Unix, onde cada usuário possui um diretório do tipo `/home/username` de uso exclusivo e privado, onde pode colocar seus arquivos, criar subdiretórios, etc.

6.4.5 Procedimentos Operacionais

Representa o conjunto de informações institucionais de uma organização, também conhecido como políticas e procedimentos ou ainda *handbook*. A idéia é criar uma hierarquia de listas e documentos classificada por assuntos para facilitar e ampliar o acesso aos documentos institucionais de uma organização; algumas sugestões de informações para esta hierarquia incluem a história da instituição, descrição das atividades, departamentos, cargos e atribuições, produtos e serviços, política de segurança e privacidade de informações, manuais diversos, etc. Com a utilização de *links* é possível criar novas hierarquias baseadas nos documentos existentes para oferecer novas visões sobre as informações; o departamento de recursos humanos de uma organização, por exemplo, pode criar uma hierarquia com os documentos imprescindíveis para novos funcionários.

6.4.6 *Knowledge Base*

A implantação de uma política de gerenciamento do conhecimento deve ser suportada por uma ferramenta de gerenciamento de informações que ofereça suporte ao armazenamento e recuperação de informações de forma transparente, flexível e configurável. A INEX pode ser utilizada para representar os mais diversos tipos de documentos de uma base de conhecimento. Pode-se conceber hierarquias para o armazenamento de cursos, apresentações, catálogos de produtos, além de integrá-las a outras hierarquias como a de documentação de projetos e a de procedimentos operacionais.

6.4.7 Notícias

Um repositório de notícias organizado por data de publicação, onde cada categoria (notícias do dia, semana, mês, semestre, ano) é representada por uma lista, e as notícias são representadas por documentos, *URLs* e *links*. A idéia deste caso de uso é promover a INEX como um canal de comunicação institucional e a cultura de utilização da *intranet*.

6.4.8 *Workflow*

A INEX pode ser estendida através da criação de itens do tipo tarefa e repositório de tarefas, além da incorporação de classes de extensão que implementam a lógica de aprovação e roteamento de atividades. Neste cenário é possível criar listas do tipo repositório de tarefas (*to do list*) sob a lista que representa a área *home* de cada usuário, além de associar tarefas a cada *to do list*. Pode-se definir *to do lists* que contenham tarefas para aprovação, outras com tarefas para execução, outras para revisão, e assim por diante. Uma tarefa pode relacionar um responsável pelo tratamento da tarefa e um status que representa a sua situação (encerrada, aprovada, recusada, etc.). Esta hierarquia pode ser integrada com outras hierarquias como a de empregados, para comunicar novos funcionários sobre as atividades que devem executar, por exemplo. *To do lists* podem ser classificadas por assunto, projeto, prioridade de execução das tarefas que contém, status das tarefas que contém (pendentes, executadas), etc. Tarefas ainda podem ser representadas como um subtipo de lista, pois podem conter *links* para documentos, *URLs,* e outros itens necessários a documentação da tarefa. A manutenção dos processos de *workflow* pode ser realizada através do remanejamento de itens entre diferentes *to do lists* e criação de comentários sobre aprovação/reprovação.

6.5 Conclusões

Apesar do foco inicial deste trabalho não contemplar o desenvolvimento da INEX, acreditamos que sua incorporação promoveu a sinergia entre seu projeto, o desenvolvimento dos componentes e os estudos sobre *intranets* e arquiteturas de *software Web*. Como apresentamos neste capítulo, além de reutilizar os componentes desenvolvidos neste trabalho na construção da INEX, empregamos uma arquitetura baseada nos resultados dos estudos apresentados no capítulo 3. Em outras palavras, projetamos a parte servidor da INEX de forma modular, flexível e configurável, ao mesmo tempo em que executamos a prova de conceito sobre os componentes e a arquitetura baseada em camadas.

Desta forma, podemos concluir que a INEX é a prova de conceito que havíamos previsto no plano deste trabalho. Além disso, como a INEX é um projeto que pode ser reutilizado na implantação de diversos serviços e aplicações em *intranets*, concluímos que seu projeto também constitui uma contribuição à construção de *intranets*, que é o foco principal deste trabalho. Podemos afirmar ainda que o grau de reutilização proporcionado por um projeto do porte da INEX é muito superior ao alcançado pela reutilização de cada um dos componentes de infra-estrutura apresentados no capítulo 5.

Capítulo 7

Conclusão

Este capítulo apresenta algumas decisões de projeto referentes à escolha de tecnologias e métodos para a construção dos componentes de *software* e da INEX, discute os resultados da prova de conceito do trabalho através de uma avaliação dos projetos da INEX e dos componentes, sugere algumas propostas de extensão dos artefatos implementados e avalia os resultados do desenvolvimento deste trabalho.

7.1 Tecnologias e Métodos Utilizados

Todos os trabalhos de desenvolvimento realizados neste trabalho foram executados sob a premissa de reutilização em sistemas *Web*, preferencialmente em *intranets*. Desta forma, procuramos projetar os componentes e a INEX de forma flexível, portável, escalável, extensível e compreensível através da utilização de ferramentas de suporte a implementação, como Java, e de projeto, como *design patterns* [GHJ94][Pre94], UML [BRJ98a][HW97], arquitetura em camadas (seção 3.2.2) e o processo de desenvolvimento unificado [BRJ98b].

Os levantamentos realizados a respeito das arquiteturas e tecnologias de implementação de sistemas *Web*, bem como o estudo sobre a abordagem de desenvolvimento baseado em componentes foram realizados de forma imparcial, independente de tecnologia de implementação. Nestes estudos, apontamos os focos de utilização de cada tecnologia e ressaltamos a importância de se empregar padrões abertos ou semi-abertos e a tendência em se construir componentes de *software* distribuídos e multiplataforma. A idéia é que como os sistemas *Web* construídos para funcionar em *intranets* e *sites* Internet devem interoperar entre diversas plataformas e redes, além de se integrar com sistemas legados, devem ser suficientemente flexíveis para tanto. Neste cenário, decidimos pela escolha de Java como plataforma de implementação, dado que sua estrutura cobre a maioria das premissas apontadas anteriormente, como é o caso de portabilidade, escalabilidade, integração com sistemas legados (JDBC, JMS, JNDI), suporte a objetos distribuídos (RMI, CORBA, IIOP), modelo de componentes (EJB), entre outros.

Design patterns foram amplamente utilizados tanto nos projetos dos componentes quanto na implementação da INEX, principalmente para permitir o desacoplamento entre diferentes tipos de implementação de um mesmo domínio de aplicação, e para realizar *dynamic binding*. Na prática, o emprego de *design patterns* neste trabalho serviu de base para a criação dos pontos flexíveis de customização dos componentes (*hotspots*) e para documentar o projeto e os mecanismos de configuração e extensão de cada componente.

O desenvolvimento dos componentes e da INEX foi amparado pelo processo de desenvolvimento unificado e todos os artefatos gerados foram realizados através de UML, como podemos observar na documentação apresentada nos capítulos 5 e 6. Como ressaltamos no capítulo 4, a documentação de componentes é um fator crítico para a sua (re)utilização [Kar98], pois garante autonomia às equipes que constróem aplicações através da montagem de componentes, sem a necessidade da intervenção dos seus projetistas. Neste sentido, procuramos adotar um padrão de documentação que expressa desde a definição e o escopo do componente até os detalhes de sua interface e uma prova de conceito para exemplificar sua utilização. A proposta de documentar os componentes através de um padrão bem definido é baseada na abordagem dos catálogos de *design patterns*, que também adotam um padrão de documentação para representar os detalhes de cada *pattern*.

Segundo o levantamento sobre as arquiteturas de *software Web*, há uma tendência em se implementar sistemas através de uma hierarquia de camadas [BRJ98b] para aumentar o desacoplamento entre os componentes da aplicação e promover a reutilização de *software*. Seguindo esta tendência, projetamos a INEX através da arquitetura de camadas e os componentes de *software* para funcionar sob a hierarquia de serviços de base ou infra-estrutura. Na prática, os projetos da INEX e dos componentes ocorreram concomitantemente; enquanto o estudo sobre as aplicações mais comuns em *intranets* identificou as funcionalidades mais recorrentes, procuramos conceber componentes através destas definições que ainda fossem compatíveis com o projeto da INEX. Além de contribuir com a construção de componentes de propósito geral, esta abordagem permitiu a reutilização imediata dos componentes no projeto INEX.

Por fim, gostaríamos de ressaltar os motivos que nos levaram a implementar componentes no domínio de infra-estrutura, além de empregar um modelo de componentes *ad hoc*. Na época em que concebemos, projetamos e iniciamos a implementação dos componentes, não havia nenhum servidor de aplicação disponível no mercado; naquela ocasião havia apenas uma especificação sobre o modelo de componentes de Java, EJB, e nenhuma implementação, seja comercial ou de referência. Além disso, realizamos um levantamento sobre as funcionalidades previstas nas especificações [Tho98][OMG99][Rom99] e concluímos que apesar das facilidades propostas pelos *containers* EJB, tarefas como as de persistência e controle de acesso ainda devem frequentemente ser controladas pelo usuário/programador. O fato das especificações apontarem para o encapsulamento dos serviços de infra-estrutura pelos servidores de aplicação não implica na desobrigação do usuário de realizar tais atividades. Podemos citar como exemplo o suporte a persistência de objetos em bancos de dados relacionais, que até o momento, não conta com uma solução sofisticada e transparente, e é delegada na maioria dos casos ao próprio usuário/programador. Como não havia nenhum servidor de aplicação disponível na etapa de implementação, optamos por implementar os componentes através de um modelo *ad hoc* apresentado no capítulo 4.3.5. Apesar de não seguir a especificação de EJB, os componentes seguem a abordagem de se empregar uma interface (local ou remota) para acesso aos seus serviços, oferecem mecanismos de configuração e extensão, e assim por diante.

7.2 Resultados

Neste projeto, contribuímos com a implementação de *software* reutilizável para *intranets* em duas frentes: através da implementação de quatro componentes de infra-estrutura que podem ser reutilizados em aplicações de diferentes domínios, ou seja, em qualquer aplicação com requisitos de registro de *logs*, persistência de objetos, controle de acesso e compartilhamento de recursos; a partir do desenvolvimento de um *framework* para gerenciamento de documentos em *intranets*, a INEX, que pode tanto ser utilizada em seu formato padrão para gerenciar hierarquias de documentos, arquivos e URLs, quanto estendida para suportar novos tipos de documentos.

Também realizamos alguns estudos sobre o processo de desenvolvimento de *intranets*, e especificamente, sobre os aspectos não funcionais acerca do desenvolvimento de *software Web*, seja para *intranets* ou para *sites* Internet. No primeiro, identificamos as principais etapas, recursos e oportunidades que são comumente abordadas em projetos de *intranets*; este levantamento serviu de base tanto para a concepção dos componentes quanto para a especificação da INEX. No segundo, apontamos algumas arquiteturas de *software* voltadas ao desenvolvimento *Web*, as principais tecnologias de implementação, quais os contextos em que se deve empregar cada tecnologia e como podem ser combinadas através de uma arquitetura para formar uma aplicação; estes levantamentos, além da discussão sobre componentes de *software* apresentada no capítulo 4, serviram de base para as etapas de projeto e implementação dos componentes e da INEX.

Desta forma, podemos afirmar que abordamos a construção de *software Web* (como aquele voltado para *intranets*) tanto sob as perspectivas de planejamento, com o estudo sobre *intranets*, quanto as de projeto e implementação, com o estudo sobre arquiteturas de *software Web*, com os componentes de *software* e com o sistema INEX. Neste sentido, acreditamos ter alcançado o objetivo inicial de oferecer facilitadores para o desenvolvimento de *software* para *intranets*. Na verdade, os conceitos e os componentes apresentados neste trabalho podem ser utilizados em qualquer projeto de desenvolvimento de *software Web*, seja ele direcionado a *intranets* ou não.

Vale ressaltar ainda a importância do desenvolvimento de *software* sustentado por uma arquitetura, sobretudo em projetos *Web*, em que as tecnologias e a estrutura como são combinadas podem variar de acordo com os requisitos não funcionais do projeto. Muitos projetos *Web* falham ao focar a utilização de tecnologias de forma isolada, sem sustentação de uma arquitetura que estipule a forma adequada de se empregar cada solução. Por volta de 1997, havia poucas soluções de suporte ao desenvolvimento *Web*, além de ser em sua grande maioria, imaturas e ineficientes. Este quadro vem se revertendo, e já dispomos de uma série de soluções, consideravelmente sofisticadas. Porém, a utilização correta de cada solução, além da adoção de algumas disciplinas como a de reutilização de *software* através de componentes, constituem fatores decisivos para o sucesso de projetos *Web*.

Acreditamos que este trabalho seja referência sobre a abordagem discutida acima, pois focamos no desenvolvimento de *software Web* reutilizável na forma de componentes e realizamos o desenvolvimento sustentado através de arquiteturas bem definidas. Os resultados desta abordagem podem ser representados pelos aspectos de modularidade, flexibilidade, escalabilidade e reusabilidade alcançados nos projetos dos componentes e da INEX. Neste contexto, podemos destacar por exemplo os aspectos de reusabilidade e extensibilidade do projeto INEX, que deverá incorporar suporte ao envio e recebimento de *emails* como pré-requisito para a realização de outros trabalhos de pesquisa no laboratório A-HAND.

7.3 Propostas de Continuidade

O projeto dos componentes de *software* deste trabalho focou tanto a reutilização quanto a flexibilidade no que diz respeito à configuração e extensão dos componentes. Porém, procuramos implementar a lógica elementar de cada componente - no caso dos componentes BD e segurança, por exemplo, apesar de suportarem a incorporação de diferentes meios de persistência, implementamos somente o suporte a bancos de dados relacionais. Quanto ao projeto INEX, desenvolvemos um mecanismo de extensão e sugerimos uma série de possibilidades de utilização que dependem tanto de sua reconfiguração quanto da extensão através da incorporação de novos tipos de itens e de *plug-ins* para tratamento adequado destes novos itens. Esta seção discute algumas propostas de extensão da INEX e dos componentes, além de algumas sugestões de melhorias para dar continuidade a este projeto.

- **Extensão dos Componentes:** apesar de projetarmos os componentes de *software* deste trabalho sob uma estrutura flexível, não oferecemos uma diversidade de implementações para cada componente. No caso do componente BD, uma iniciativa desejável seria a de implementar suporte a bancos de dados orientados a objetos; analogamente, o componente de segurança poderia implementar suporte a persistência de informações sobre autenticação em diretórios LDAP. A idéia de estender estes componentes com a implementação de outras categorias de persistência serviria de base para a estabilização das suas interfaces, além de aumentar a flexibilidade das aplicações usuárias.
- **Integração dos Componentes a um Servidor de Aplicação:** como não havia nenhuma implementação disponível de servidores de aplicação Java e CORBA durante o período de desenvolvimento dos componentes, não vislumbramos a implementação sobre este tipo de ferramenta. Porém, como tem surgido uma série de servidores de aplicação, e como os serviços de infra-estrutura dos componentes podem ser reutilizados a partir de aplicações construídas sobre estes servidores, devemos considerar tanto a integração dos componentes deste trabalho aos serviços de infra-estrutura dos servidores de aplicação, quanto o seu porte aos modelos de componentes EJB ou CORBA.

  Como os componentes desenvolvidos neste trabalho são de infra-estrutura, pode-se adaptá-los para utilizar os serviços de transação, ciclo de vida e *pooling* de conexões dos próprios servidores de aplicação. A idéia é complementar os serviços de base dos servidores de aplicação com as funcionalidades de infra-estrutura dos componentes. Neste cenário, os componentes poderiam ser (re)utilizados por aplicações baseadas em componentes construídas sob um servidor de aplicação.
- **Extensão da INEX:** como o projeto da INEX prevê uma camada de extensão, é razoável sugerirmos a incorporação de novos tipos de itens, bem como tratamento diferenciado sobre estes novos tipos, como forma de realizar uma prova de conceito dos mecanismos de extensão da INEX. Em tese, todas as sugestões de utilização apresentadas no capítulo 6.4 devem ser suportadas, ou por configuração ou por extensão da INEX. Além disso, já existem algumas iniciativas de melhorias da INEX no laboratório A-HAND que visam a distribuição e o trabalho remoto; no primeiro caso, podemos imaginar que o *browser* da INEX permita navegar por hierarquias distribuídas em diferentes servidores INEX; outra possibilidade é a de criação de *links* para itens remotos; o segundo caso foca o trabalho remoto num cenário em que deverá haver uma versão *standalone* da INEX que implemente recursos de sincronização; nestes casos, será possível recuperar uma hierarquia de itens de um servidor para um

computador portátil, utilizá-los a partir da versão *standalone*, e sincronizar as alterações realizadas sobre a hierarquia local no servidor da INEX.

Referências Bibliográficas

[AR97] J. Ablan and E. Reiner. *Developing Intranet Aplications with Java*. SAMS, 1997

[Bar97] M. Barnes. *Component Road Map*. Hurwitz Group Balanced View Report, 1997

[BGK97] D. Baumer, G. Gryczan, R.Knoll, C. Lilienthal, D.Riehle and H. Zullighoven. *Framework Development for Large Systems*. CACM, vol. 40, no(10), October 1997.

[BRJ98a] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[BRJ98b] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Software Development Process*. Addison Wesley, 1998.

[Cam97] I. Campbell. *The Intranet: Slashing the Cost of Business*. IDC technical report, 1997.

[Cha96] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996

[CHS97] W. Codeni, K. Hondt, P. Steyaert and A. Vercammen. *Evolving Custom-Made Applications into Domain-Specific Frameworks*. CACM, vol. 40, no(10), October 1997.

[Dru97] R. Drummond. *Documento de Especificação da Intranet Express (INEX)*. Laboratório A-HAND, Unicamp - Universidade Estadual de Campinas, technical report, 1997.

[Fer94] A. Ferreira. Interconexão Dinâmica de Objetos Distribuídos Java. Master's thesis, IC - Instituto de Computação, Unicamp - Universidade Estadual de Campinas, 1998.

[FS97] M. Fayad and D. Schmidt. *Object-Oriented Application Frameworks*. CACM, vol. 40, n(10), October 1997.

[GHJ94] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Gon94] C. Gonçalves. *Objetos Distribuídos*. Master's thesis, IC - Instituto de Computação, Unicamp - Universidade Estadual de Campinas, 1994.

[HC00] C. S. Horstmann and G. Cornell. *Core Java 2 Volume II - Advanced Features*. Sun Microsystems Press, Java Series, 2000.

[HC98] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly, 1998.

[HC99] C. S. Horstmann and G. Cornell. *Core Java 2 Volume I - Fundamentals*. Sun Microsystems Press, Java Series, 1999.

[Hil96] M. Hills. *Intranet Business Strategies*. John Wiley & Sons, 1996.

[HW97] P. Harmon and M. Watson. *Understanding UML: The Developer's Guide with a Web-based Application in Java*. Morgan Kaufmann, 1997.

[Kar98] D. Kara. *The Repository's Role in Component Development*. Sterling Software technical report, 1998.

[Lea96] D. Lea. *Concurrent Programming in Java: design principles and patterns*. Addison Wesley, 1996.

[Net98] Netscape. *Intranet Deployment Guide*. Netscape technical report, 1998

[NM99] A. Nakhimovsky and T. Myers. *Professional Java XML Programming with Servlets and JSP*. Wrox Press, 1999.

[OH98] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, second edition, 1998.

[OHE96] R. Orfali, D. Harkey and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996

[OMG99] OMG. *CORBA Components*. OMG technical report, orbos/99-02-01.

[Paw00] M. Pawlan. *J2EE BluePrints Digest*. Java Developer Connection technical paper, http://java.sun.com/jdc, 2000.

[Pre94] L. Rising. *The Patterns Handbook*. SIGS, 1998.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1994.

[PS91] J. Peterson and A. Silberschatz. *Operating Systems Concepts*. Addison Wesley, 3rd edition, 1991.

[Rom99] E. Roman. *Mastering Enterprise JavaBeans and the Java 2 Plataform Enterprise Edition (J2EE)*. John Wiley & Sons, 1999.

[RM97] D. Ryan and K. Martin. *Intranet Business Value: Return on Investment Analysis*. META Group business report, 1997.

[Sch96a] D. Schmidt. *Using Design Patterns to Develop Reusable Object-Oriented Software*. ACM Computing Surveys, December 1996.

[Sch96b] D. Schmidt. *Software Components: Patterns and Frameworks*. ACM Group Report, June 1996.

[SM97] A. Schneider and G. McGrath. *Measuring Intranet Return on Investment*. Online white paper, http://www.intrack.com, 1997.

[Sun97] Sun Microsystems. *Component-Based Software with JavaBeans and ActiveX*. Sun Microsystems Technical report, 1997

[Tay99] A. Taylor. *JDBC Developer's Resource*. Second edition, Informix Press, 1999.

[Tel97] S. Telleen. *Intranet Organization: Strategies for managing change*. Online book, http://www.iorg.com/intranetorg, 1997.

[Tho98] A. Thomas. *Enterprise JavaBeans: Server Component Model for the Java Plataform*. Technical report, Sun Microsystems Inc, 1998

[Usw97] US*Web* Corp. *Justifying the Web for Your Business*. US*Web* white paper, 1997

[Usw98] US*Web* Corp. *50 Ways to Put an Intranet to Work*. US*Web* white paper, 1998.

[Wat98] M. Watson. *Creating Java Beans: Components for Distributed Applications*. Morgan Kaufmann, 1998.

[WH98] S. Wiener and K. Hanson. *Policies and Guidelines: Rules of the Road for na Intranet*. Intranet Journal technical article, http://www.intranetjournal.com, 1998.

[Ibm01] IBM. *San Francisco Project*. http://www.ibm.com/software/ad/sanfrancisco, 2001