CRISTINA VERÇOSA PÉREZ BARRIOS DE SOUZA

USO DE REFLEXÃO COMPUTACIONAL EM APLICAÇÕES DA PLATAFORMA J2EE™

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná, como parte dos requisitos para obtenção do título de Mestre em Ciências.

Área de Concentração:
Sistemas Distribuídos

Orientador:
Prof. Carlos Alberto Maziero, Dr.

CURITIBA

2001

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
Pró-Reitoria de Pós-Graduação, Pesquisa e Extensão

ATA DA SESSÃO PÚBLICA DE EXAME DE DISSERTAÇÃO DO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA APLICADA DA PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ.

Exame de dissertação nº 029

Aos 12 dias do mês de março de 2001, realizou-se a sessão pública de defesa de dissertação "**USO DA REFLEXÃO COMPUTACIONAL EM APLICAÇÕES DA PLATAFORMA J2EE**", apresentada por Cristina Verçosa Pérez de Souza, ano de ingresso 1998, para obtenção do título de Mestre em Ciências. A Banca Examinadora foi composta pelos seguintes professores:

MEMBROS DA BANCA	ASSINATURA
Presidente:. **Prof. Dr. Carlos Alberto Maziero (PUCPR) – orientador**	
Prof. Dr. Alcides Calsavara (PUCPR)	
Prof. Dr. Manoel Camillo de Oliveira Penna Neto (PUC **- PR)**	
Prof. Dr. Luiz Eduardo Buzato (UNICAMP - SP)	

De acordo com as normas regimentais a Banca Examinadora deliberou sobre os conceitos a serem atribuídos e que foram os seguintes:

MEMBROS DA BANCA	CONCEITOS
Presidente: **Prof. Dr. Carlos Alberto Maziero (PUCPR) – orientador**	APROVADO
Prof. Dr. Alcides Calsavara (PUCPR)	APROVADA
Prof. Dr. Manoel Camillo de Oliveira Penna Neto (PUC **– PR)**	APROVADO
Prof. Dr. Luiz Eduardo Buzato (UNICAMP - SP)	APROVADO
Conceito Final	APROVADO

Observações da Banca Examinadora

_____

_____

Profº Dr. Carlos Alberto Maziero
Diretor do Programa de Pós-Graduação em Informática Aplicada–PUCPR

Rua Imaculada Conceição, 1155 – Prado Velho – CEP 80215-901 – Telefone: +55 41 330-1669 / + 55 41 330-1654
Fax: + 55 41 330- 1669 – Curitiba – Paraná – Brasil - E-mail:ppgia@rla01.pucpr.br

Souza, Cristina Verçosa Pérez Barrios de

Uso de Reflexão Computacional em Aplicações da Plataforma J2EE™.

Curitiba, 2001.

163 p.

Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná. Departamento de Informática.

I. J2EE 2.Componentes 3.Reflexão Computacional

I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Departamento de Informática II.t

AGRADECIMENTOS

Ao professor Carlos Maziero, pela excelente orientação, apesar da sobrecarga de trabalho.

Ao professor Alcides Calsavara, pela a leitura crítica e as valiosas sugestões.

Ao professor Robert Burnett, pelo apoio e confiança.

Ao SERPRO – Serviço Federal de Processamento de Dados – pela oportunidade de participar do seu programa de incentivo à educação. E aos amigos e colegas de trabalho, pela torcida e paciência.

Ao meu marido, Nei Ricardo, pelo carinho, amizade e compreensão nos inúmeros momentos dedicados a este trabalho.

Aos meus pais, Roberto e Zila, e a toda minha família, por sempre terem me incentivado a crescer e a querer saber mais.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

.EAR	*Enterprise Archive*
.JAR	*Java™ Archive*
.WAR	*Web Archive*
CCM	CORBA *Component Model*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
COTS	*Component Off-The-Shelf*
EJB	*Enterprise JavaBean*
EIS	*Enterprise Information System*
DCE	*Distributed Computing Environment*
DCOM	*Distributed COM*
DLL	*Dynamic Link Library*
GUI	*Graphical User Interface*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Secure Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
IDL	*Interface Definition Language*
IIOP	*Internet Inter-ORB Protocol*
IT	*Information Technology*
J2EE	*Java™ 2 Platform, Enterprise Edition*
J2SE	*Java™ 2 Platform, Standard Edition*
JCP	*Java™ Community Process*
JDBC	*Java™ Database Connectivity*
JDK	*Java™ Development Kit*
JMS	*Java™Messaging Service*

JNDI	*Java™ Naming and Directory Interface*
JSP	*Java™ Server Pages*
JTA	*Java™ Transaction API*
JTS	*Java™Transaction Service*
MOP	*Meta-Object Protocol*
OMG	*Object Management Group*
ORB	*Object Request Broker*
OLTP	*Online Transaction Processing*
OTS	*Object Transaction Service*
OSF	*Open Software Foundation*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
SDK	*Software Development Kit*
SSL	*Secure Socket Layer*
TI	Tecnologia da Informação
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
VSA	*Visual Studio for Applications*
WORA	*Write Once, Run Anywhere™*
XML	*eXtensible Markup Language*

RESUMO

Uma das principais preocupações das aplicações servidoras se concentra em como prover interoperabilidade e portabilidade – a fim de fornecer uma solução duradoura e abrangente. Essas habilidades em particular recebem especial atenção na Plataforma Java para Corporações. Também denominada de Java 2 *Platform Enterprise Edition*, ou J2EE, ela define uma arquitetura Java unificada, centrada em serviços e baseada em componentes, que promove interoperabilidade, portabilidade e ambiente de execução consistente para aplicações corporativas.

O paradigma reflexivo, por sua vez, torna possível que as computações observem e modifiquem as propriedades de seu comportamento. Permite assim que a aplicação controle seu comportamento atuando sobre si mesma. No que se refere aos padrões Java, essas habilidades reflexivas não estão completamente especificadas. Todavia, a plataforma J2EE define uma série de facilidades para composição de aplicação, baseadas em seus padrões para componentes, que geram aberturas para customizar e modificar o comportamento da aplicação – ou seja, utilizar princípios de reflexão.

Esse trabalho visa, portanto, utilizar características de reflexão computacional na plataforma J2EE, mantendo todas as suas propriedades de interoperabilidade, portabilidade e consistência de ambiente. Para tanto, pretende-se valer apenas de seus padrões para construção de aplicação, através da abordagem denominada de MOP (*Meta-Object Protocol*) em Tempo de Implantação – ou mais especificamente, Intercessão em Tempo de Implantação –, cujo objetivo é o de fornecer um grau a mais de flexibilização na implementação de aplicações corporativas.

RESUMO

Uma das principais preocupações das aplicações servidoras se concentra em como prover interoperabilidade e portabilidade – a fim de fornecer uma solução duradoura e abrangente. Essas habilidades em particular recebem especial atenção na Plataforma Java para Corporações. Também denominada de Java 2 *Platform Enterprise Edition*, ou J2EE, ela define uma arquitetura Java unificada, centrada em serviços e baseada em componentes, que promove interoperabilidade, portabilidade e ambiente de execução consistente para aplicações corporativas.

O paradigma reflexivo, por sua vez, torna possível que as computações observem e modifiquem as propriedades de seu comportamento. Permite assim que a aplicação controle seu comportamento atuando sobre si mesma. No que se refere aos padrões Java, essas habilidades reflexivas não estão completamente especificadas. Todavia, a plataforma J2EE define uma série de facilidades para composição de aplicação, baseadas em seus padrões para componentes, que geram aberturas para customizar e modificar o comportamento da aplicação – ou seja, utilizar princípios de reflexão.

Esse trabalho visa, portanto, utilizar características de reflexão computacional na plataforma J2EE, mantendo todas as suas propriedades de interoperabilidade, portabilidade e consistência de ambiente. Para tanto, pretende-se valer apenas de seus padrões para construção de aplicação, através da abordagem denominada de MOP (*Meta-Object Protocol*) em Tempo de Implantação – ou mais especificamente, Intercessão em Tempo de Implantação –, cujo objetivo é o de fornecer um grau a mais de flexibilização na implementação de aplicações corporativas.

ABSTRACT

One of the main concerns on developing server applications focuses on how to provide interoperability and portability – aiming to supply a durable and widespread solution. These abilities in particular take special attention from the Java Platform for the Enterprise. Also named Java 2 Platform Enterprise Edition, or J2EE, it defines an unified, service-centered, and component-based architecture, which promotes interoperability, portability and a consistent runtime environment for enterprise applications.

The reflective paradigm, in its turn, makes possible computations to observe and modify properties of their behavior. So, it allows applications to control its own behavior, acting upon itself. Considering the Java standards, these reflective abilities are not quite specified. However, the J2EE platform defines several application composition facilities, based on its component standards, that give the possibility of customizing and modifying the application's behavior – i.e., of using reflection principals.

This work aims, therefore, to use features of computational reflection on J2EE platform, maintaining all of its interoperability, portability, and environment consistency properties. In order to do that, we intend to use just the J2EE's application construction standards, through the approach called Deploy-Time MOP (Meta-Object Protocol) – or more specifically, Deploy-Time Intercession –, providing then one more flexibility level on implementing enterprise applications.

ABSTRACT

One of the main concerns on developing server applications focuses on how to provide interoperability and portability – aiming to supply a durable and widespread solution. These abilities in particular take special attention from the Java Platform for the Enterprise. Also named Java 2 Platform Enterprise Edition, or J2EE, it defines an unified, service-centered, and component-based architecture, which promotes interoperability, portability and a consistent runtime environment for enterprise applications.

The reflective paradigm, in its turn, makes possible computations to observe and modify properties of their behavior. So, it allows applications to control its own behavior, acting upon itself. Considering the Java standards, these reflective abilities are not quite specified. However, the J2EE platform defines several application composition facilities, based on its component standards, that give the possibility of customizing and modifying the application's behavior – i.e., of using reflection principals.

This work aims, therefore, to use features of computational reflection on J2EE platform, maintaining all of its interoperability, portability, and environment consistency properties. In order to do that, we intend to use just the J2EE's application construction standards, through the approach called Deploy-Time MOP (Meta-Object Protocol) – or more specifically, Deploy-Time Intercession –, providing then one more flexibility level on implementing enterprise applications.

CAPÍTULO 1 - INTRODUÇÃO

A influência da Internet aumentou a necessidade por uma maior e melhor interação entre aplicações distribuídas. Os desenvolvedores têm respondido a essa demanda com uma variedade de produtos Web que permitem ao usuário expandir seu alcance aos recursos da Rede. Dentro desse contexto, para que aplicações servidoras ofereçam uma solução durável e de grande alcance, é preciso que possuam duas habilidades básicas: interoperabilidade e portabilidade. Com suporte adequado, tais aplicações devem então ser capazes de interagir com outras, independente do equipamento ou sistema operacional em que se encontrem, bem como mudar de plataforma base de forma transparente sempre que necessário. A sofisticação dessas habilidades pode, portanto, determinar a utilidade e o tempo de vida das aplicações distribuídas – que atualmente ganharam um grande impulso com a especificação da Plataforma Java para Corporações.

1.1. Plataforma Java

A tecnologia de componentes revolucionou a forma de desenvolver complexos sistemas de informação através da combinação e extensão de blocos reutilizáveis de software. Nessa linha, os JavaBeans (lançados pela Sun Microsystems em 1996) emergiram rapidamente como um importante padrão de componentes para aplicações cliente, com as vantagens de portabilidade e desenvolvimento através de ferramentas visuais [ORFALI & HARKEY, 1998]. Contudo, os JavaBeans não foram projetados para criar aplicações servidoras. A JVM (Java *Virtual Machine*) possibilita que uma aplicação execute em qualquer sistema operacional – portabilidade WORA (*"Write Once, Run Anywhere™"*) –, porém componentes servidores precisam de serviços adicionais, não providos diretamente pela JVM, mas sim por uma infra-estrutura de sistemas distribuídos [ORFALI & HARKEY, 1998].

A especificação da arquitetura Enterprise JavaBeans (ou EJB, lançada pela Sun Microsystems no fim de 1998), veio suprir essa demanda, estendendo o modelo original dos componentes JavaBeans para suporte a aplicações servidoras. Isso é possível porque os EJBs estão preparados para suportar um conjunto de serviços essenciais (nome, transação, segurança, etc.) providos por diferentes infra-estruturas de sistemas distribuídos (p. ex. CORBA). Por conseguinte, os EJBs definem uma arquitetura Java baseada em componentes servidores.

A contínua evolução desses padrões conduziu, enfim, à especificação da Plataforma Java para Corporações (lançada pela Sun Microsystems em 1999). Também denominada Java 2 *Platform Enterprise Edition*, ou J2EE, define uma arquitetura Java unificada, baseada em componentes, centrada em serviços e habilitada para aplicações multi-camada, fornecendo o suporte de ambiente necessário aos componentes servidores EJB. Seu lançamento agrega uma série de especificações (seção 3.2.2), cujo objetivo é o de auxiliar os desenvolvedores a alcançar o ideal WORA no lado servidor. Proporciona, dessa forma, um ambiente de execução integrado, consistente e atestado, que garante uma determinada qualidade de serviço e assegura portabilidade e interoperabilidade para aplicações corporativas.

1.2. Reflexão Computacional

Em termos gerais, o paradigma reflexivo permite que as computações observem e modifiquem as propriedades de seu comportamento. Torna possível, então, que a aplicação controle seu comportamento atuando sobre si mesma. Para tanto, a parte funcional (nível base) de uma aplicação é separada de suas partes não funcionais (nível meta), ficando o nível base com os métodos e procedimentos da aplicação em si, e o nível meta com as funções de controle e gerenciamento da aplicação. Essa abordagem é conhecida como protocolo de meta-objeto, ou MOP (*Meta-Object Protocol*), detalhado na seção 4.2.

Dessa forma, a reflexão computacional permite independência entre nível base e nível meta, fornecendo flexibilidade de implementação, uma vez que alterar o gerenciamento de nível meta não implica em alterar, ou mesmo afetar, a implementação dos algoritmos da aplicação, localizados no nível base.

1.3. Motivação

Essa breve descrição da reflexão computacional já denota os benefícios que as computações podem obter das capacidades reflexivas. Estas, por sua vez, permitem que as aplicações tenham autoconsciência de seu comportamento e estado, e também capacidade de alterá-los, usando meta-informações nas decisões sobre o que fazer em seguida [SOBEL & FRIEDMAN, 1998].

No que se refere à Plataforma Java, tais capacidades reflexivas integrais não estão completamente especificadas, sendo esse o tema de estudo de várias abordagens: [OLIVA, 1998], [TATSUBORI, 1999] e [GOLM & KLEINÖDER, 1998]. No entanto, essas propostas implicam em um certo grau de alteração na especificação

Java padrão, o que tem como conseqüência o comprometimento das vantagens corporativas e de ambiente oferecidas pela plataforma J2EE.

Contudo, a J2EE define facilidades para composição de aplicação que favorecem o desacoplamento de funcionalidade em componentes lógicos, e encorajam a reutilização de código orientado a componente. Tais características, aliadas à facilidade de alteração de suas entradas de ambiente (usadas para configuração, controle e gerenciamento), geram aberturas que permitem customizar o comportamento de componentes – ou seja, utilizar os princípios de reflexão – no momento da montagem de aplicação.

Uma vez estabelecida uma forma sistematizada e sustentada de utilizar reflexão na plataforma J2EE – definida pelo modelo de integração detalhado no Capítulo 5 –, a proposta é validada e criticada através de alguns experimentos que visam a construção flexível de aplicações servidoras. Esses experimentos (Capítulo 6) concentram-se em: (1) validar e viabilizar o modelo de integração através de uma implementação direta da proposta de Intercessão em Tempo de Implantação; e (2) criticar e analisar mais profundamente o modelo, levantando seus prós e contras, através da elaboração do projeto de uma aplicação mais complexa que proporciona a replicação de um serviço via reflexão computacional.

1.4. Proposta

Esse trabalho objetiva, portanto, propor uma abordagem consistente para utilizar reflexão computacional na plataforma J2EE, mantendo todas as suas características de interoperabilidade, portabilidade e consistência de ambiente. Com esse propósito, é apresentada a abordagem MOP (*Meta-Object Protocol*) em Tempo de Implantação, ou mais especificamente Intercessão em Tempo de Implantação – como detalhado no Capítulo 5.

Esta abordagem utiliza-se apenas da facilidade de composição das aplicações J2EE, visando possibilitar a alteração do comportamento de componentes de negócio de forma sistematizada e modular, baseada no paradigma reflexivo. Fornece assim um grau a mais de flexibilização de implementação nas situações onde é preciso introduzir controle e / ou alterar o comportamento da aplicação corporativa como um todo.

1.5. Estrutura da Dissertação

Objetivando apresentar as tecnologias e mecanismos utilizados e propostos de forma progressiva, esse trabalho está dividido em sete Capítulos, como descrito a seguir.

- Capítulo 1 – Introdução. Faz uma apresentação geral do contexto da dissertação.
- Capítulo 2 – Componentes. Fornece uma descrição de componentes, ressaltando suas principais características e vantagens, a fim de justificar porque esse mecanismo de construção de aplicação é o fundamento dos modernos *frameworks* de desenvolvimento.
- Capítulo 3 – Plataforma Java para Corporações. Apresenta mais detalhadamente a evolução dos padrões Java até a definição da Plataforma Java para Corporações, ressaltando suas principais características e vantagens, a fim de justificar sua escolha como plataforma base de sustentação para a proposta da dissertação.
- Capítulo 4 – Reflexão Computacional. Faz um apanhado geral dos principais conceitos da reflexão computacional, relaciona as características reflexivas existentes no padrão Java, e faz uma breve análise de algumas abordagens reflexivas baseadas em Java – visando traçar um paralelo comparativo com a proposta da dissertação. Por fim, analisa e justifica a escolha de algumas habilidades reflexivas possíveis e interessantes de serem aplicadas à plataforma J2EE – o fundamento desse trabalho.
- Capítulo 5 – Modelo de Integração Proposto. Detalha e justifica a proposta de utilização de reflexão computacional na plataforma J2EE, através da abordagem de Intercessão em Tempo de Implantação, tendo como base as tecnologias e mecanismos apresentados nos Capítulos anteriores.
- Capítulo 6 – Exemplos de Aplicação. Relaciona dois exemplos de aplicação onde é possível validar, através de uma implementação e de análises criticas, a proposta da dissertação.
- Capítulo 7 – Conclusões e Trabalhos Futuros. Apresenta as conclusões obtidas nesse trabalho, e como é possível prosseguir com o mesmo.

CAPÍTULO 2 - COMPONENTES

O presente Capítulo descreve os principais conceitos e vantagens dos componentes de software, objetivando demonstrar porque esse mecanismo de construção de aplicação é o adotado pelas modernas arquiteturas de software – exatamente o caso da plataforma J2EE.

Com esse propósito, o Capítulo inicia com uma definição geral de componentes, na seção 2.1, prosseguindo com uma relação de termos e conceitos afins na seção 2.2. A seção 2.3 descreve os fundamentos de composição de aplicação baseada em componentes, enfatizando a estratificação em camadas lógicas, e a seção 2.4 apresenta resumidamente os principais padrões de mercado para componentes.

Por fim, a seção 2.5 encerra o Capítulo com uma análise onde são relacionadas as vantagens e conceitos fundamentais sobre componentes, utilizados ao longo da dissertação.

2.1. Definição Geral

Componentes são unidades binárias – ou pedaços pré-construídos de código de aplicação – destinadas à construção de sistemas de software. Para isso, têm fronteiras bem definidas que facilitam sua combinação com outros componentes, permitindo a produção de aplicações customizadas e completas [ORFALI & HARKEY, 1998].

As fronteiras dos componentes, denominadas de interfaces (seção 2.2.4), são especificadas na forma de contratos com dependências explícitas de contexto (seção 2.2.1). Dessa forma, a produção, aquisição e distribuição de componentes pode ocorrer de forma independente, o que facilita o processo de construção de sistemas [SZYPERSKI, 1998], bem como habilita a reutilização de software (seção 2.2.2).

2.2. Termos e Conceitos

Esta seção faz um apanhado dos principais termos e conceitos pertinentes a componentes de aplicação.

2.2.1. Modelo de Componentes

Genericamente, componentes executam dentro de uma construção denominada *container*, que provê um contexto de aplicação. É esse contexto que habilita a interação entre vários componentes, podendo fornecer também gerenciamento e serviços de controle para os mesmos. Um modelo de componentes, portanto, define a arquitetura básica de um padrão de componentes, especificando a estrutura de suas interfaces e os mecanismos através dos quais o componente interage com seu *container* e com outros componentes. Ou seja, o modelo de componentes especifica como o componente expõe suas interfaces, métodos e eventos, provendo as diretrizes para a criação e implementação dos mesmos [ORFALI & HARKEY, 1998]. Existem diferentes modelos de componentes (seção 2.4), e cada um define seus próprios contratos e requisitos de arquiteturas.

Observa-se que, embora os componentes obedeçam aos rigorosos padrões de um modelo, o processo de montagem – ou composição – da aplicação permite uma customização significativa das aplicações baseadas em componentes (seção 2.3).

2.2.2. Reutilização

A reutilização também é uma forma de criar softwares melhores. Construir novas aplicações a partir de componentes prontos e testados favorece o planejamento e a redução de prazos e custos de desenvolvimento, provendo previsibilidade e permitindo a existência de linhas de produção de software [CHAPPEL, 1996].

Os componentes, portanto, permitem altos níveis de reutilização – que a orientação a objetos por si só não permite (seção 2.2.3) –, e a existência de um mercado de compra e venda de componentes, uma vez que são unidades binárias de software.

2.2.3. Componentes e a Orientação a Objetos

O paradigma da orientação a objetos (OO) não é uma *revolução*, mas uma *evolução* do paradigma estruturado. Ele definiu novas unidades de modularização – classes e objetos –, que visam permitir a construção de sistemas mais robustos e com maiores facilidades de reutilização, manutenção e evolução.

Contudo, uma visão mais atual e realista da abordagem orientada a objetos permite observar que não ocorreu uma disseminação generalizada de bibliotecas de classes para o desenvolvimento de sistemas. O encapsulamento, herança e polimorfismo dos objetos oferecem uma forma clara de reutilizar funcionalidades. No

entanto, apenas essas características não foram suficientes para trazer os níveis de reutilização esperados. A razão para isso é que a tecnologia tradicional de objetos apresenta alguns obstáculos para a criação de um mercado de objetos reutilizáveis [CHAPPEL, 1996]:

- Distribuição de objetos implica na distribuição de seu código fonte[1], sendo que a distribuição de seu código binário restringe sua utilização aos mesmos ambiente de desenvolvimento e compilador;
- Por estar a nível de código fonte, a reutilização entre objetos criados em diferentes linguagens não é possível; e
- Quando um objeto é alterado, é preciso recompilar toda a aplicação.

A tecnologia de componentes, contudo, não apresenta esses obstáculos, oferecendo um mecanismo efetivo de reutilização de software, através de blocos de software binários e discretos, com determinadas interfaces.

Ainda deve ser ressaltado que um componente não é necessariamente um objeto, ou um conjunto de objetos, pois também é possível desenvolver componentes através do paradigma estruturado (seção 2.4.2). Porém, tanto a tecnologia de objetos quanto a tecnologia de componentes visam basicamente reutilização e, como conseqüência, o mais provável é que os componentes "ganhem vida" através de objetos.

Quando os componentes são desenvolvidos segundo o paradigma da orientação a objetos, agregam as vantagens do alto potencial de reutilização dos componentes com a robustez e facilidade de evolução e manutenção da OO. Pode-se afirmar, por conseguinte, que objetos são inerentemente centralizados e intra-aplicações, enquanto componentes são distribuídos e inter-aplicações.

Em comum, componentes e classes têm o fato de realizarem (implementarem) interfaces (seção 2.2.4). Suas diferenças básicas são [BOOCH et al., 2000]: classes são abstrações lógicas, com atributos e operações, e componentes são representações físicas, que vivem em nós de rede e só possuem operações através de interfaces.

---

[1] Embora a reutilização de código fonte seja razoável dentro de uma corporação, o mesmo não é verdade entre corporações [CHAPPEL, 1996].

2.2.4. Interfaces

As interfaces[2] definem os pontos de acesso ao componente, descrevendo a face que o mesmo apresenta a (ou requer de) outros componentes da aplicação [BOOCH et al., 2000].

Genericamente, uma interface é uma coleção de métodos abstratos (apenas define a assinatura do método, não seu algoritmo), que especifica os serviços providos por um componente (ou classe) – são os componentes (ou as classes) que realizam[3] a interface. O conceito de interface visa facilitar a combinação de diferentes elementos, funcionando como um contrato de construção que permite que clientes e fornecedores montem aplicações baseando-se nesse padrão. Um componente, como na Fig. 2.1 (ou classe, como na Fig. 2.2), pode realizar ou depender de (importar) interfaces [BOOCH et al., 2000].

A Fig. 2.1 apresenta um exemplo em UML de composição de aplicação, provida por interfaces: o componente "Quiosque.exe" depende da (importa) interface "IConta", realizada (implementada) pelo componente "ContaEspecial". Nesses exemplo, são apresentadas duas representações de uma interface: a forma canônica na parte superior da figura, e a forma estendida na parte inferior da figura. A forma estendida permite visualizar os métodos abstratos da interface que deverão ser implementados pelo componente que a realiza. É dessa forma que as interfaces possibilitam a codificação genérica, que aumenta a portabilidade do código.

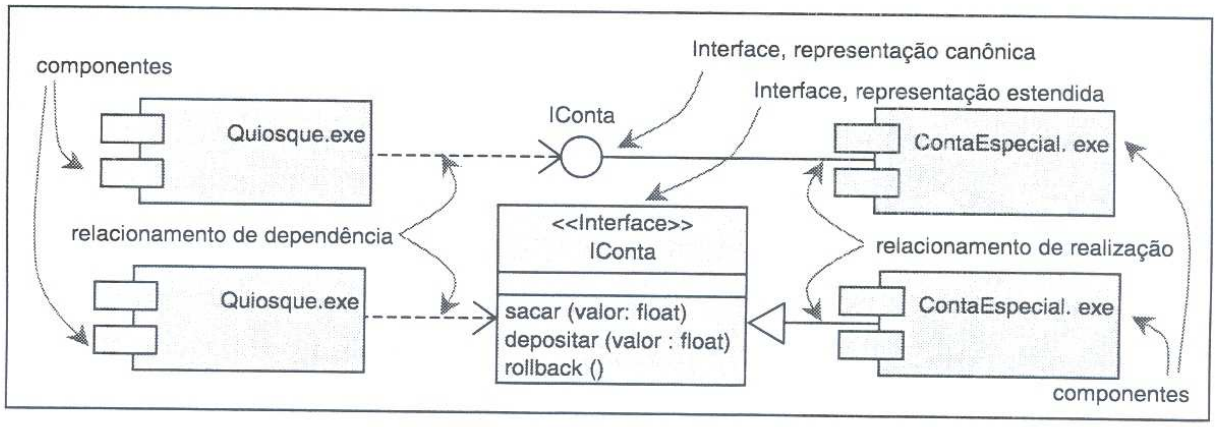

Fig. 2.1. Composição / montagem de aplicação através de interface.

---

[2] Um componente pode ter uma ou várias interfaces, dependendo do seu padrão de implementação.
[3] O conceito de **realização** especifica o relacionamento entre uma interface e o componente, ou classe, que implementa suas operações; uma interface pode ser realizada por diferentes componentes ou classes; e um componente, ou classe, pode realizar diferentes interfaces.

Da mesma forma que na Fig. 2.1, a Fig. 2.2 apresenta um exemplo em UML de relacionamento entre classes, provido por interfaces: a classe "TelaSaque" depende da interface "IConta", realizada pela classe "ContaCorrente". Também nesse exemplo, estão ilustradas as duas formas de representação de interface.

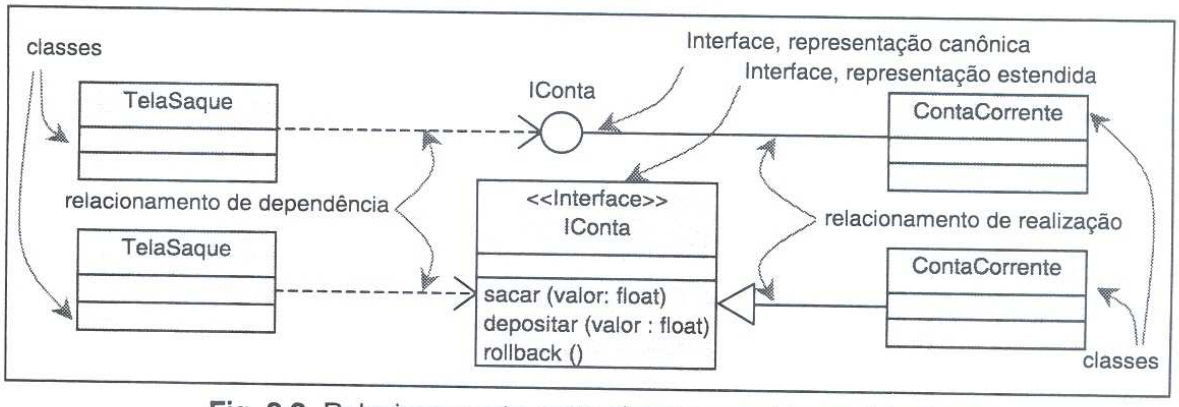

Fig. 2.2. Relacionamento entre classes, provido por interfaces.

Quando desenvolvidos através da OO, os componentes representam um pacote físico de elementos lógicos, e suas interfaces ultrapassam as fronteiras lógica e física da seguinte maneira: a mesma interface realizada por um componente (elemento físico) será encontrada realizada por uma classe (elemento lógico), que por sua vez encontra-se implementada dentro de um componente [BOOCH et al., 2000].

2.2.5. Módulos

Os componentes podem ser distribuídos – para posterior utilização em aplicações customizadas – através de módulos. São os módulos que agrupam um determinado conjunto de componentes com funções afins, provendo um espaço de nome (*namespace*) específico para um domínio de aplicação (assuntos de RH, regras de seguros, controle de estoque, etc.).

Os módulos são disponibilizados através de pacotes, que podem ser compartilhados entre diferentes produtos de software e servir a diferentes projetos. O termo "Componente de Prateleira", ou COTS, *Component Off-The-Shelf*, refere-se à disponibilização comercial de pacotes de componentes – ou seja, representa de forma plena o conceito de reutilização de software.

2.3. Modelos de Arquiteturas

A arquitetura de um sistema deve prover um modelo que prepare o sistema para suas prováveis evoluções, tais como: aumento do número de usuários, novas funcionalidades, novas formas de acesso e distribuição, etc. Existem vários modelos

de arquitetura que auxiliam o desenvolvimento de aplicações. Os modelos clássicos são referenciados como *patterns* de arquitetura, descritos na subseção a seguir.

2.3.1. *Patterns* de Arquitetura

Um *pattern* é uma solução comum para um problema freqüente em um dado contexto. Ele especifica um conhecimento coletado das experiências em um certo domínio, que auxilia a entender, especificar, construir e documentar um sistema [GAMA et al., 1995].

Nessa seção, são abordados os *patterns* de arquitetura que se concentram em elementos de alta granularidade e em interações entre subsistemas e sistemas, como os *patterns* de Três-Camadas (*Three-Tier*) e de Cliente-Servidor (*Client-Server*). Eles definem a estrutura de um modelo de implantação e sugerem como os elementos que compõem o sistema (sistema operacional, banco de dados, subsistemas legados e subsistemas a serem desenvolvidos) devem ser alocados em nós de rede [JACOBSON et al., 1999].

Juntos, os *patterns* Cliente-Servidor e Três-Camadas auxiliam na estratificação do modelo de um sistema, cuja funcionalidade pode ser particionada (Fig. 2.3) nas camadas lógicas:

- Camada de Apresentação: faz tratamento de vídeo, teclado e mouse, e validação de dados;
- Camada de Negócio: implementa a lógica do negócio – as funções de negócio; e
- Camada de Acesso a Dados: recupera e atualiza a informação armazenada.

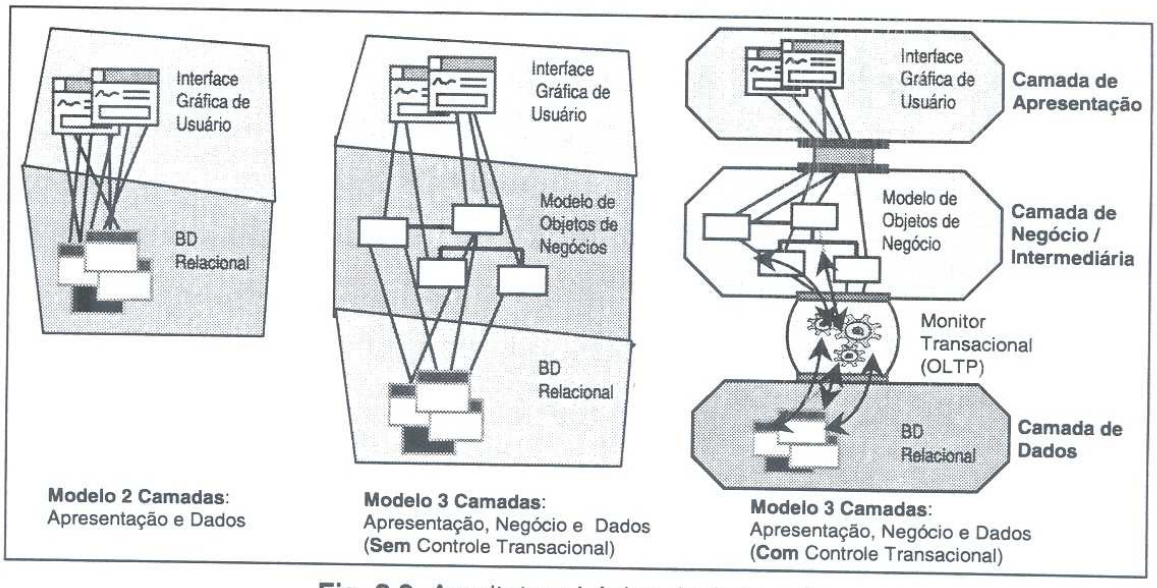

Fig. 2.3. Arquitetura Lógica de Aplicação.

Existem diferentes alternativas de distribuição desses elementos lógicos através dos computadores de uma rede (Fig. 2.4). É o modelo da arquitetura que vai definir o mecanismo de colaboração usado entre tais elementos e a definição padrão de seus componentes.

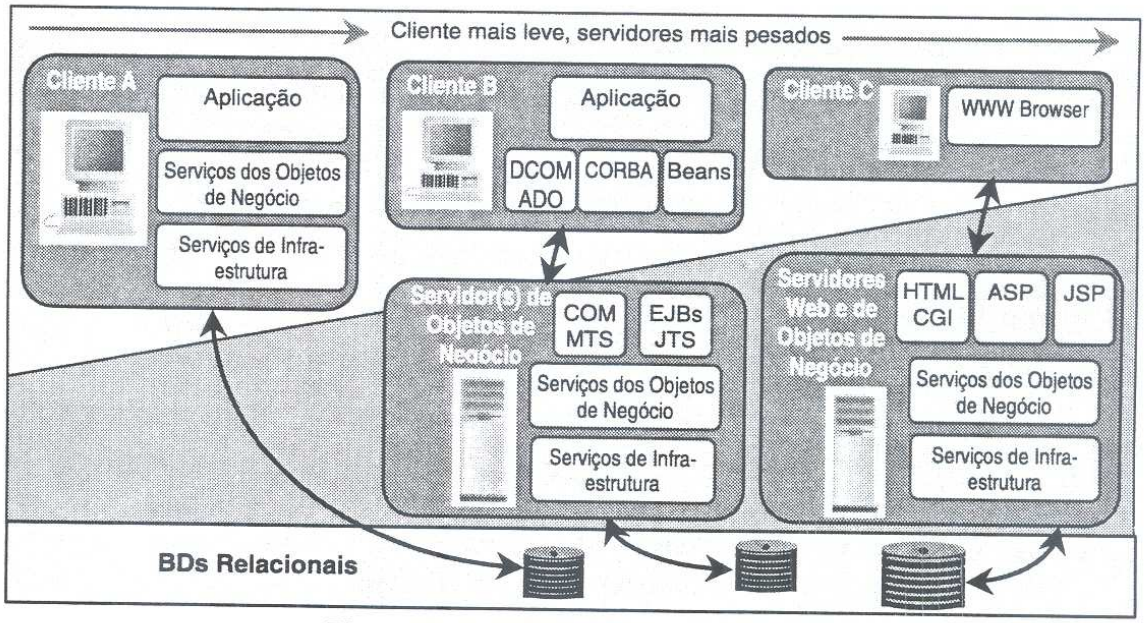

Fig. 2.4. Arquitetura Física de Aplicação.

A Fig. 2.3 também apresenta modelo de arquitetura de Duas-Camadas (*Two-Tier*), mais antigo, onde a lógica de apresentação está no cliente, e a lógica de negócio e de dados é assumida pelo banco de dados (via *Stored Procedures*) [DeLOTTINVILLE, 1994]. Nesse caso, o BD fica sobrecarregado com funções de sistema operacional – para o tratamento dos objetos de negócio (seção 2.3.2) –, e seu *throughput* pode tornar-se um gargalo na escalabilidade e no desempenho da aplicação (um cliente por conexão).

Para contornar essas dificuldades, surgiu o modelo de arquitetura Três-Camadas, também chamado mais recentemente de Multi-camada (*Multitier*). Nesse caso, os servidores de aplicação – gerenciados pelas tecnologias tradicionais de OLTP (*Online Transaction Processing*), como a do Monitor Transacional – desempenham o papel do que é chamado de Camada Intermediária (*Middle Tier*) [KRAMER, 1996], onde está a lógica de negócio da aplicação e os serviços de infra-estrutura próprios dos sistema distribuídos (nome, transação, etc.).

Todas as regras de combinação desses elementos devem estar bem definidas nas diferentes infra-estruturas de sistemas distribuídos, de forma a proporcionar interoperabilidade.

2.3.2. Objetos de Negócio

O termo objetos de negócio provê uma forma natural de descrever conceitos independentes de aplicação, tais como cliente, pedido, pagamento, paciente, etc.. Esses conceitos encorajam uma visão de software que transcende ferramenta, aplicações, banco de dados e outros conceitos de sistemas. A última promessa da tecnologia de objetos e de componentes é prover esses elementos de granulação média, que comportam-se mais como uma entidade reconhecida do mundo real. Em suma, objetos de negócio são aplicações a nível de componentes (mini-aplicações), que podem ser usadas de forma não previsível, uma vez que são independentes de uma aplicação em particular. Os objetos de sistema, por sua vez, representam entidades que apenas fazem sentido para sistemas de informação e programadores – não são algo que o usuário final reconheça [ORFALI & HARKEY, 1998].

Os objetos de negócio são ideais para criar soluções Cliente-Servidor de Três-Camadas (seção 2.3.1), pois são diretamente mapeados na lógica de negócios da camada intermediária.

2.4. Padrões de Componentes

Os padrões de componentes são influenciados pelo mercado. Eles devem ser implementados e utilizados por um número significativo de empresas, de forma a permitir portabilidade e compatibilidade. Atualmente, os principais padrões de mercado para componentes são: (1) o CORBA da OMG; (2) o COM / DCOM da Microsoft; e (3) o JavaBean / Enterprise JavaBean da Sun Microsystems, descritos resumidamente nas subseções a seguir.

2.4.1. CORBA da OMG

O principal objetivo da arquitetura CORBA (*Common Object Request Broker Architecture*) da OMG (*Object Management Group*) é o de habilitar a interoperação aberta para uma grande variedade de linguagens, plataformas e implementações. Para tal, seu princípio básico é: produtos CORBA-compatíveis não podem interoperar em nível binário; ao invés, devem comprometer-se com protocolos de alto nível – o IIOP (*Internet Inter-ORB Protocol*), provido pelos ORBs (*Object Request Brokers*) [SZYPERSKI, 1998].

Existem três mecanismos básicos CORBA (Fig. 2.5) que permitem a invocação de métodos em objetos [SZYPERSKI, 1998]:

- Um conjunto de interfaces de invocação (*invocation interfaces*): habilita a ligação entre objetos distribuídos, realizando o *marshal* (empacotamento) dos argumentos da invocação;
- O ORB: localiza o objeto destinatário e o método invocado, transportando os argumentos, atuando como um "corretor" / intermediário entre a requisição do cliente e o serviço de um objeto.
- Um conjunto de adaptadores de objetos (*object adapters*): realiza o *unmarshal* (desempacotamento) dos argumentos na ponta destino e invoca o método solicitado no objeto destinatário.

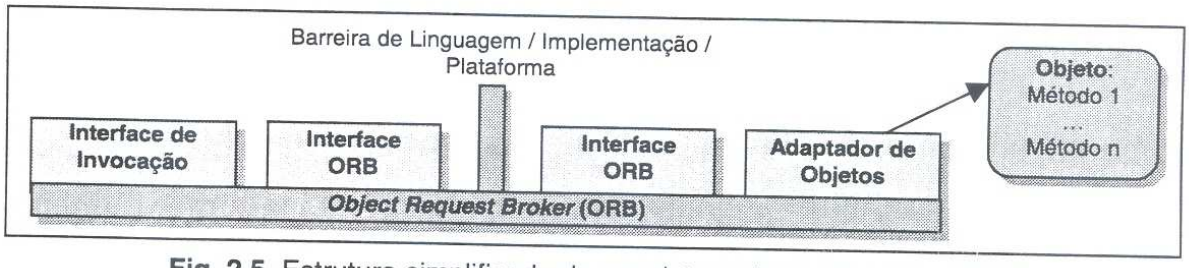

Fig. 2.5. Estrutura simplificada de um sistema baseado em um ORB.

Para que as Interfaces de Invocação e os Adaptadores de Objetos interoperem é preciso que todas as interfaces de objetos estejam descritas em uma linguagem comum – o que habilita a construção de *marshallings* e *unmarshallings* genéricos. Assim, todas as linguagens CORBA-compatíveis devem ter ligações com uma linguagem comum, a IDL (*Interface Definition Language*) da OMG, que permite chamadas de e para uma linguagem em particular.

Interfaces expressas em OMG IDL são compiladas por um compilador OMG IDL e depositadas no "Repositório de Interfaces", que todo ORB deve ter. Através da interface do ORB, as interfaces compiladas podem ser recuperadas deste repositório. Um compilador OMG IDL deve ser usado basicamente para gerar (Fig. 2.6) [SZYPERSKI, 1998]:

- *Stub* (*proxy* do lado cliente): é instanciado e parece com um objeto local, mas envia todas as invocações através do ORB, fazendo efetivamente o *marshalling* para o objeto real.
- *Skeleton* (*stub* do lado servidor): recebe invocações, faz o *unmarshalling* dos argumentos e direciona as invocações para o objeto real.

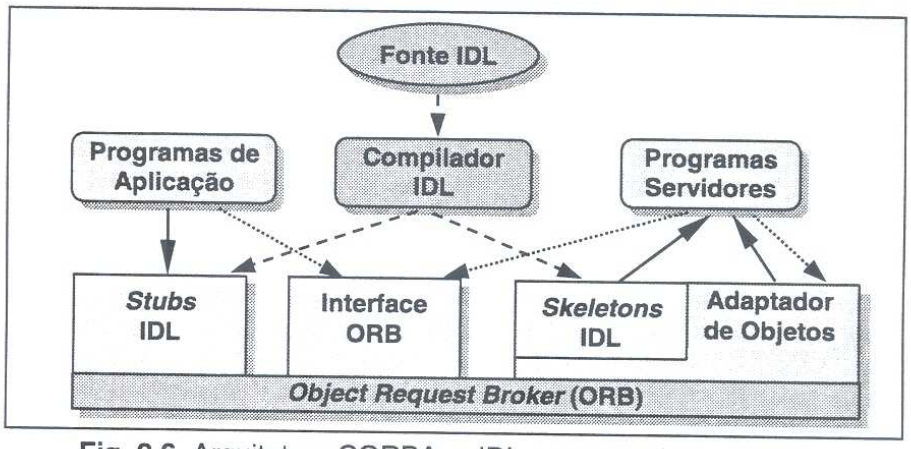

Fig. 2.6. Arquitetura CORBA: o IDL gera o *stub* e o *skeleton*.

➢ **CCM – CORBA Component Model**

O Modelo de Componentes CORBA, o CORBA *Component* Model (CCM), especifica componente como um novo tipo básico, uma extensão / especialização do tipo objeto, que pode ser especificado em IDL e representado no Repositório de Interfaces. Um componente é denotado por uma referência de componente, que é uma especialização de uma referência de objeto, e uma definição de componente é uma especialização e extensão da definição de uma interface [OMG, 1999].

Um tipo componente é instanciado para criar entidades concretas com estado e identidade, e deve encapsular sua representação interna e implementação. A superfície do componente, definida por sua descrição IDL, é seu ponto de acesso para seus clientes. Os componentes suportam uma variedade de características de superfícies, chamadas portas, que podem ser dos seguintes tipos [OMG, 1999]:

- Facetas (*Facets*): interfaces providas para interação com clientes;
- Receptáculos (*Receptacles*): pontos de conexão que permitem que componente use uma referência provida por um agente externo;
- Fontes de Evento (*Event Sources*): pontos de conexão que emitem eventos de um certo tipo para consumidores de evento, ou para um canal de eventos;
- Escoadouros de Evento (*Event Sinks*): pontos de conexão para onde eventos de um determinado tipo podem ser enviados;
- Atributos (*Attributes*): valores expostos através de operações de acesso e de alteração – destinados basicamente para configuração do componente.

Um componente pode prover múltiplas referências de objetos, chamadas de facetas, capazes de suportar diferentes interfaces CORBA. Contudo, um componente

tem uma única referência distinta, cuja interface corresponde à definição do componente, chamada de "Interface Equivalente" – sua face para clientes. É ela que permite que os clientes naveguem pelas demais facetas (demais interfaces) do componente, e se conectem com suas portas, como ilustrado na Fig. 2.7 [OMG, 1999].

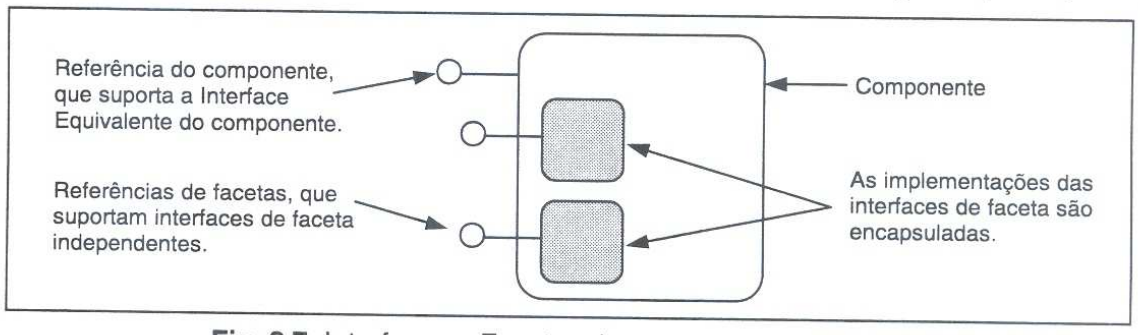

Fig. 2.7. Interfaces e Facetas de componentes CORBA.

- **Facilidades de Composição e Implantação**

O Modelo de Componentes CORBA, o CCM, especifica que uma ou mais implementações de componentes podem ser agrupadas em pacotes, cujas propriedades são definidas em um arquivo descritor (com vocabulário XML), visando futura implantação [OMG, 1999]. Além dos esquemas para descrever pacotes, também é possível identificar no CCM referências a ferramentas para implantação de componentes individuais e montagem de aplicação, e uma linguagem para definição de implementação de componentes, a CIDL (*Component Implementation Definition Language*). Tal conjunto de características visa facilitar a composição e a distribuição de aplicações baseada em componentes CORBA.

2.4.2. COM / DCOM da Microsoft

Um componente COM (*Componente Object Model*), ou Modelo de Objetos Componentes, é o fundamento Microsoft sobre o qual todos os componentes de software da sua plataforma são embasados. Um componente COM suporta mais de uma interface, cada uma contendo métodos – para chamar os métodos em uma interface, o cliente deve adquirir um ponteiro separado para cada interface do componente COM (Fig. 2.8).

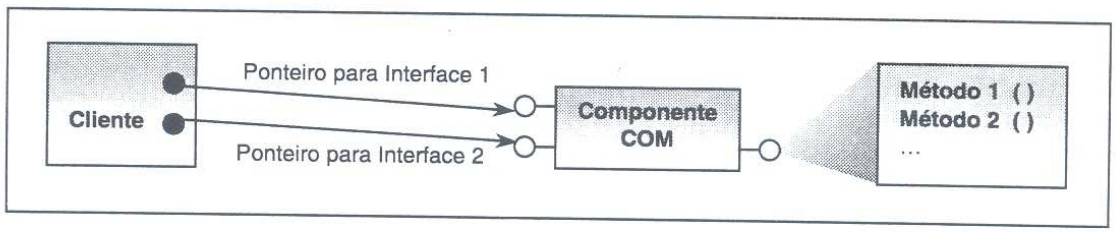

Fig. 2.8. Serviços de um componente COM acessados via interface e métodos.

O padrão para interface COM é a nível binário, o que permite que qualquer linguagem, orientada ou não a objetos, possa criar componentes COM. A regra para esse padrão binário é [CHAPPEL, 1996]: o ponteiro do cliente para uma interface aponta para um ponteiro dentro do componente, que por sua vez aponta para uma tabela *vtable*, que guarda os ponteiros para os métodos do objeto (Fig. 2.9). A *vtable* sempre aponta primeiro para os métodos da interface *IUnknown* – da qual toda interface COM é herdeira –, que possui os métodos [CHAPPEL, 1996]:

- *QueryInterface:* utilizada pelo cliente para encontrar o ponteiro de outra interface do objeto. Usa como parâmetro o identificador da interface que deseja.
- *AddRef* e *Release:* utilizadas para manter um contador de referência para componente, que controla sua existência em memória.

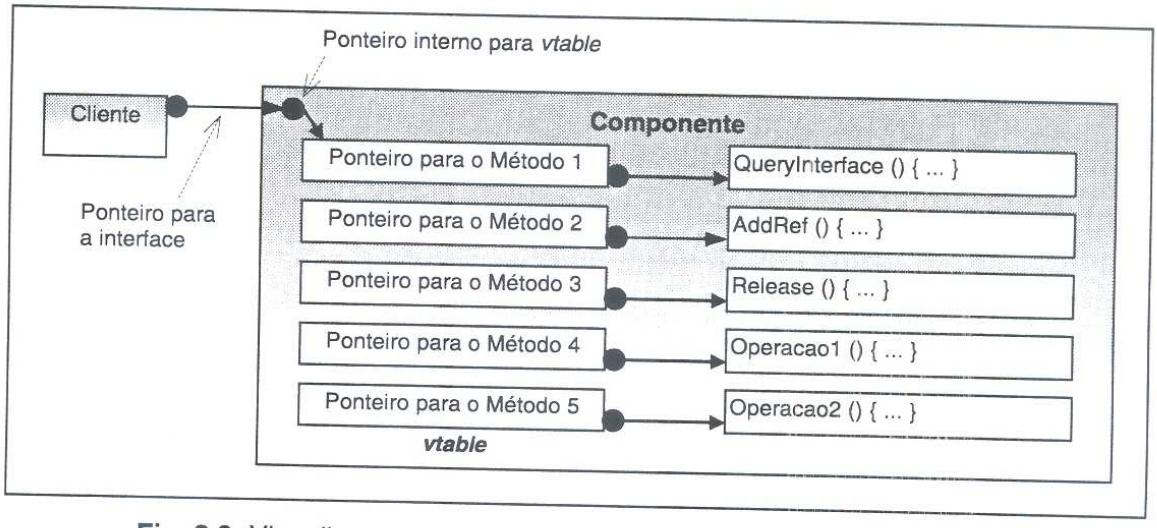

Fig. 2.9. Visualização da interface binária de um componente COM.

O COM não impõe uma forma obrigatória para descrever uma interface, seus métodos e parâmetros, porém existe uma ferramenta padrão: a COM's *Interface Definition Language* – COM's IDL. Ela é uma extensão da IDL utilizada nas *Remote Prodecures Calls* da Microsoft (MS RPC), que emprestam sua sintaxe do OSF DCE (*Open Software Foundation's Distributed Computing Environment).* A Fig. 2.10 exibe um exemplo de especificação MS IDL para interface.

```
[ object, uuid (E7CD0D00-1827-11CF-9946-444553540000) ]
interface ITeste : IUnknown {
   import "unknown.idl";

  HRESULT Metodo1    ([in] OLECHAR word[31], [out] boolean * found);
  HRESULT Metodo2    ([in] OLECHAR word[31]);
}
```

Fig. 2.10. Exemplo de MS IDL para interface.

➢ **DCOM**

O Modelo Distribuído de Objetos Componentes, o *Distributed COM* (DCOM) é uma extensão da tecnologia COM para suporte à comunicação entre componentes em diferentes computadores, interligados via LAN ou WAN, promovendo assim a transparência de acesso. Atualmente, os modelos COM e DCOM são referenciados como uma única tecnologia, denominada de COM+.

Invocar um método implementado em um equipamento remoto também baseia-se nos conceitos de *proxy* e *stub*, e nesse caso uma *Remote Procedure Call* (RPC) é feita do cliente para o servidor (Fig. 2.11). A Microsoft adota sua própria RPC, a MS RPC, que também empresta sua sintaxe da RPC do OSF DCE. A MS RPC e sua utilização DCOM é conhecida como *Object RPC* (ORPC) [CHAPPEL, 1996].

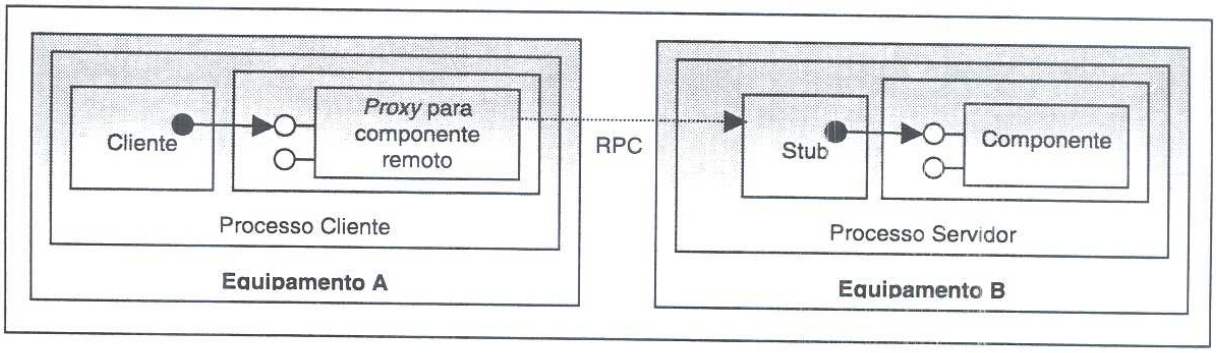

Fig. 2.11. Acesso a um componente DCOM em um servidor remoto.

➢ **Facilidades de Composição e Implantação**

O COM+ faz parte da nova estratégia da Microsoft para ambiente de desenvolvimento, o *Visual Studio for Applications*. Também denominado de VSA, este IDE, ou ambiente de desenvolvimento integrado, pretende prover uma forma completa de customização para aplicações Web, habilitando desenvolvedores a modificar e a estender aplicações Web [MICROSOFT, 2000b]. Seu lançamento está previsto para o segundo semestre de 2001, sendo que sua primeira versão apenas contemplará a linguagem Visual Basic.NET [MICROSOFT, 2001].

A característica máxima dos ambientes Microsoft é sua habilidade visual de desenvolvimento através de sofisticadas ferramentas. O VSA repete esse preceito, fornecendo para o lado servidor a console de um *Server Explorer* (Explorador de Servidor) que auxilia desenvolvedores a acessar e manipular os recursos dos computadores para o qual eles tenham permissão. É possível, portanto, adicionar visualmente um componente (DLL), ou outro recurso (banco de dado), de um

computador remoto no projeto de uma aplicação servidora [MICROSOFT, 2000a] – um avanço em relação ao antigo sistema de registro manual de componentes, via console.

Apesar das facilidades visuais para montagem de aplicação servidora customizada, é importante ressaltar que a plataforma de software deve ser exclusivamente Microsoft. Além disso, o conceito de distribuição de pacotes de aplicação é direcionado para o lado cliente: existem serviços que habilitam usuários a fazer *download* e instalação de componentes em ambiente Internet / Intranet. Para tal, há especificações de unidades de distribuição, ou pacotes de aplicação cliente (com extensão .CAB, de *cabinet file*), cujo conteúdo é descrito em arquivos do tipo *Open Software Desciption* (.OSD), também com vocabulário XML. Ainda há outros formatos de distribuição de pacote de aplicação cliente (executáveis para auto-extração de aplicação e arquivos .ZIP para applets), porém ressalta-se que não há a mesma correspondência de distribuição para o lado servidor.

2.4.3. JavaBeans da Sun Microsystems

A especificação dos JavaBeans define um modelo de componentes (seção 2.2.1) baseado nas classes Java, que possibilita que diferentes fornecedores de software possam criar e distribuir componentes Java. Por definição, um JavaBean, ou apenas bean, é um componente de software reutilizável que pode ser manipulado visualmente dentro de uma ferramenta de desenvolvimento. Um bean pode ser visível – elemento GUI como um botão – ou invisível, porém sempre pode ser composto visualmente com outros beans através de um IDE (*Integrated Development Environment*), ou ambiente de desenvolvimento integrado.

- **Bean x Applets**

Um applet é uma mini aplicação. Embora os applets possam ser arranjados em em uma página HTML para interagir, o mecanismo que provê essa comunicação não é padronizado: é implementado de forma diferente (e geralmente não confiável) em diferentes browsers e em diferentes ambientes Java [LEMAY et al., 1996]. Logo, tal solução é considerada ad hoc, e um applet é, portanto, em geral considerado isolado dos elementos que executam no mesmo ambiente [SZYPERSKI, 1998].

Os beans, por outro lado, podem facilmente ser combinados para formar uma aplicação que executa tanto em modo *standalone* quanto em browsers Web – embutida em uma página HTML, de forma semelhante aos applets. Os beans sabem, portanto, não só como interagir entre si, mas também com seu *container* [SZYPERSKI, 1998].

➢ **Fundamentos**

Os beans foram projetados com o propósito de habilitar um modelo de utilização duplo, da seguinte forma [SZYPERSKI, 1998]: (1) *Design time* (tempo de projeto), onde os beans podem ser montados e configurados dentro de um ambiente integrado de desenvolvimento; (2) *Runtime* (tempo de execução), onde é possível retirar do bean o código necessário apenas em tempo de projeto, provendo uma versão mais enxuta para distribuição.

Dessa forma, um bean pode dinamicamente customizar sua aparência e funcionalidade, verificando se ele se encontra em *design time* ou em *runtime.*

Não há uma linguagem de definição de elemento para descrever um bean, como uma IDL (*Interface Definition Language*), ou linguagem de definição de interface. Logo, para que um bean revele suas interfaces, é definida uma série de convenções de nomes que são usadas para identificar métodos, eventos e propriedades do bean, que por sua vez são recuperados automaticamente via ferramenta de desenvolvimento.

Um bean não é obrigado a herdar características de qualquer outra classe base. Beans visíveis devem ser herdeiros da `java.awt` de forma que possam ser adicionados a *containers* visuais, mas beans invisíveis não precisam ser herdeiros da `java.awt`. Ainda é interessante observar que o modelo JavaBean objetiva primariamente o desenvolvimento via ferramentas visuais, porém ele é totalmente utilizável por pessoas, pois todas as suas principais APIs foram projetadas para trabalhar bem, tanto com ferramentas quanto com pessoas. A seguir, as características fundamentais que distinguem um bean são descritas mais detalhadamente [ORFALI & HARKEY, 1998].

- Suporte a Propriedades

  Os componentes possuem um estado. São as propriedades do componentes que identificam e expõem suas informações de estado, definido as características do bean. Logo, uma propriedade é um atributo discreto e nomeado que pode ser usado para ler e modificar o estado de um bean – tipicamente via editor de propriedades de uma ferramenta de desenvolvimento. O modelo JavaBean corrobora os princípios de encapsulamento, logo um bean não permite que elementos externos manipulem diretamente suas propriedades. Ao invés, devem ser usados os métodos de acesso *get / set* para cada variável. As propriedades, portanto, podem ser usadas tanto para customização quanto para programação. A

API JavaBeans suporta tanto propriedades tipo *single-value* (de um único valor) quanto propriedades *indexed* (indexadas). Adicionalmente, as propriedades podem ser *bound* (ligadas) e *constrained* (restritas). Uma propriedade *bound* notificará as partes interessadas – via disparo de evento – sempre que seu valor for modificado. Já uma propriedade *constrained* permite que as partes interessadas vetem sua modificação. O bean é responsável por especificar o comportamento de suas propriedades e por emitir os eventos que elas geram.

- Suporte à Introspecção

  Uma infra-estrutura de componentes deve definir os mecanismos pelos quais o componente expõe seus métodos, propriedades e eventos ao mundo exterior. O modelo JavaBeans provê uma facilidade de introspecção de alto nível que permite a uma ferramenta de desenvolvimento descobrir as interfaces do bean. Esta facilidade é construída sobre as classes das APIs de Reflexão Java, fornecidas pela JDK. Os desenvolvedores podem definir o comportamento de seus beans tanto usando as convenções de nome do JavaBeans, *design patterns*[4], quanto provendo explicitamente seus meta-dados através de uma classe **BeanInfo**.

- Suporte à Customização

  O fato de um componente ser adequado para trabalhar em uma ferramenta visual de desenvolvimento encoraja sua larga reutilização por programadores não especializados, e esse é um dos objetivos do modelo JavaBeans. Por essa razão, um bean pode ser facilmente customizado em uma ferramenta de desenvolvimento pela alteração de suas propriedades.

- Suporte a Eventos

  Uma metáfora simples de comunicação que pode ser utilizada para conectar beans. Esta é uma conexão fracamente acoplada, ideal para interligar componentes. Um bean pode ser um *source* (fonte) ou um *listener* (ouvinte) de um evento, e é através de uma ferramenta de desenvolvimento que *sources* e *listeners são* conectados.

---

[4] O termo *design pattern* usado na especificação JavaBeans [SUN, 1997a], refere-se aos nomes e tipos de assinaturas convencionais para conjuntos de métodos e/ou interfaces que são usadas para propósitos padrão.

- Suporte à Persistência

Deve ser possível armazenar a instância de um componente para sua posterior recuperação. Adicionalmente, uma ferramenta de desenvolvimento requer componentes que suportem alguma forma de persistência. Por exemplo, uma ferramenta permite a customização de um componente pela modificação de suas propriedades e deve, por conseqüência, poder comunicar ao mesmo que salve seu novo estado modificado. Os beans se beneficiam dos serviços de serialização da JDK para automaticamente salvar e recuperar seus estados, bem como para usar uma forma simples de fornecer versão para seus componentes, habilitando dessa forma que novos beans guardem estados de suas versões anteriores. Os beans são serializados em um arquivo (**.ser**), que pode ser empacotado para distribuição em um arquivo **.jar** (Java *Archive*, padrão Java desde a JDK 1.1, que permite a compressão e o armazenamento de um conjunto de arquivos relacionados [ORFALI & HARKEY, 1998]).

➢ **Limitações**

O modelo de componentes JavaBeans oferece a grande vantagem da transparência de implementação e da facilidade de desenvolvimento através de ferramentas visuais, também conhecida como *toolability*. Contudo, os beans não foram projetados para criar servidores de aplicação. A JVM possibilita que uma aplicação execute em qualquer sistema operacional – portabilidade WORA: "*Write Once, Run Anywhere™*" –, porém componentes servidores precisam de serviços adicionais (de nome, de transação, de segurança, etc.), e também de interoperabilidade aberta, todos não providos diretamente pela JVM. Tais facilidades devem então ser providas por uma infra-estrutura de sistemas distribuídos, exatamente como considerado pela Arquitetura Enterprise JavaBeans – um padrão específico para componentes servidores [ORFALI & HARKEY, 1998].

Por conseguinte, os padrões para componentes Java, propostos pela Sun Microsystems, envolvem basicamente dois tipos: (1) os componentes JavaBeans para aplicações clientes, e (2) os componentes Enterprise JavaBeans – descritos em detalhes no Capítulo 3 deste documento – para aplicações servidoras. A principal característica comum a ambos é a portabilidade WORA provida pela linguagem Java.

2.5. Conclusões do Capítulo

De forma geral, as características fundamentais dos componentes de software podem ser resumidas como:

- São unidades binárias e padronizadas para a composição de aplicações customizadas, com contratos bem definidos que permitem sua colaboração com outros componentes – ou produtos de terceiros – dentro de um determinado contexto;
- São unidades de distribuição independentes, portanto passíveis de substituição e reutilização;
- Reduzem prazos e custos de desenvolvimento, pois as aplicações podem utilizar-se de componentes comprados prontos, os chamados componentes de prateleira; e
- Permitem a formação de uma base de soluções para um conjunto de aplicações.

Essas características oferecem grandes vantagens para o desenvolvimento de sistemas, permitindo concluir que a construção de aplicações multi-camadas baseadas em componentes orientados a objetos são a grande tendência do futuro da computação. Além de tais componentes aumentarem a produtividade, pois são voltados para reutilização e evolução, seus padrões e interfaces garantem interoperabilidade e portabilidade. Adicionalmente, aplicações multi-camada têm o planejamento de seu projeto favorecido – orientado pela estratificação em camadas lógicas (seção 2.3) –, o que habilita uma pronta evolução do sistema como um todo, mantendo escalabilidade e desempenho.

Ainda, a adoção desse tipo de aplicação também possibilita a definição de times, ou equipes de desenvolvimento, pois separa os desenvolvedores em diferentes especialidades: os que desenvolvem componentes, os que integram componentes, os que implantam aplicações customizadas em determinados ambientes operacionais, e os que administram todo o sistema.

Enfim, é esse conjunto de benefícios que está conduzindo as principais plataformas de sistemas distribuídos a encaminharem-se na direção desse tipo de aplicação, como é o caso do CORBA da OMG, do DCOM da Microsoft, e também da plataforma Java para Corporações, a J2EE, que fundamenta-se diretamente sobre os componentes JavaBeans e Enterprise JavaBeans (este último detalhado no Capítulo 3).

CAPÍTULO 3 - PLATAFORMA JAVA PARA CORPORAÇÕES

O presente Capítulo objetiva destacar os aspectos da Plataforma Java que são utilizados nesta dissertação. Como trata-se de uma tecnologia bastante recente, cabe uma explanação um pouco mais detalhada em relação aos seus conceitos e especificidades, principalmente no que diz respeito a componentes de aplicação.

Com esse intuito, a seção 3.1 do Capítulo apresenta a Arquitetura Enterprise JavaBeans, que descreve o padrão Java para componentes servidores. A seção 3.2, por sua vez, apresenta as principais definições da Plataforma Java para Corporações (*The Enterprise Java Platform*) da Sun Microsystems, que engloba uma série de padrões Java, incluindo JavaBeans e Enterprise JavaBeans, a fim de propiciar um ambiente consistente para aplicações corporativas.

O Capítulo encerra com a seção 3.3, onde é feita uma análise da plataforma, visando justificar sua escolha como tecnologia de suporte à proposta desse trabalho.

3.1. Arquitetura Enterprise JavaBeans

A especificação da arquitetura Enterprise JavaBeans (EJB) estende logicamente o modelo de componentes JavaBeans para suportar componentes servidores. Os Enterprise JavaBeans, ou apenas enterprise beans, podem ser combinados e customizados – via ferramentas de desenvolvimento compatíveis com a tecnologia EJB – de forma a habilitar a criação de novas aplicações servidoras.

Em síntese, os Enterprise JavaBeans definem uma arquitetura para computação distribuída baseada em componentes servidores, que habilita o desenvolvimento e a implantação (*deployment*) de aplicações de negócios distribuídas. Tais aplicações têm escalabilidade, são transacionais, suportam multi-usuários, e têm portabilidade – são escritas apenas uma vez e depois podem ser implantadas em qualquer plataforma servidora que suporte a especificação Enterprise JavaBeans [SUN, 1999b].

3.1.1. Histórico

No início dos anos 90, os provedores de sistemas de informação começaram a responder às demandas de aplicação através do modelo de aplicação multi-camadas, que basicamente separa a lógica do negócio dos serviços do sistema e da interface do

usuário, colocando-a em uma camada intermediária, de *middle-tier* (seção 2.3.1). A evolução de novos serviços de *middle-tier* – monitores transacionais, *middleware* orientado a mensagem, ORBs e outros – deu ímpeto adicional a essa nova arquitetura. Ainda, o crescente uso da Internet e Intranet em aplicações corporativas contribuiu para uma maior ênfase em clientes mais leves e mais fáceis de instalar.

O projeto multi-camadas simplifica o desenvolvimento, a implantação e a manutenção de aplicações, possibilitando que os desenvolvedores concentrem-se nas especificidades da programação da lógica de negócio, pois contam com o apoio dos serviços de infra-estrutura e de aplicações cliente (*standalone* ou em browsers Web). Uma vez desenvolvida, a lógica de negócio pode ser implantada nos servidores mais apropriados às necessidades de uma organização. Mas o fato do modelo multi-camada ter sido implementado até agora em uma variedade de padrões divergentes tem limitado a habilidade dos desenvolvedores de eficientemente construir aplicações de componentes padronizadas (capazes de serem implantadas em uma larga variedade de plataformas), ou de prontamente escalar aplicações para adequá-las às condições de negócios em constante modificação.

Na divisão JavaSoft da Sun Microsystems, diversos esforços de desenvolvimento apontavam para o que se tornaria a tecnologia EJB. Primeiro, a tecnologia dos servlets mostrou que os desenvolvedores estavam ávidos para criar ambientes tipo CGI (*Common Gateway Interface*) que pudessem executar em qualquer servidor Web com suporte à plataforma Java. Segundo, a tecnologia JDBC (*Java Database Connectivity*) forneceu um modelo que casava as características WORA da linguagem de programação Java com os sistemas de gerenciamento de banco de dados. Finalmente, a arquitetura de componentes JavaBeans demonstrou a grande utilidade de encapsular conjuntos completos de funções em componentes, prontamente reutilizáveis e fáceis de configurar, no lado cliente. A convergência desses três conceitos – comportamento servidor escrito na linguagem de programação Java, conectores para habilitar o acesso a sistemas empresarias existentes (SGBDs) e componentes modulares e fáceis de implantar – levou ao padrão EJB [SUN, 1999a].

Aplicações multi-camadas baseadas em componentes estão definindo a evolução do futuro da computação. E a tecnologia dos Enterprise JavaBeans oferece componentes com benefícios de portabilidade e escalabilidade para uma vasta gama de servidores, juntamente com desenvolvimento, implantação e manutenção simplificados.

3.1.2. Características dos Componentes EJB

Os enterprise beans são componentes servidores destinados a aplicações corporativas orientadas a transação, cujas características essenciais são [SUN, 1999b]:

- Armazenar a lógica de negócio, em Java, que opera os dados de uma empresa – segue o padrão WORA: um enterprise Bean pode ser implantado em múltiplas plataformas sem recompilação ou modificação de código fonte.
- Facilitar a programação: desenvolvedores não precisam conhecer detalhes de transação, de gerenciamento de estado, de *multi-threading* e de APIs de baixo nível em geral – os serviços de informação (transação, atributos de segurança, ...) são separados das classes dos enterprise beans, e o acesso aos mesmos é intermediado por seu(s) *Container*(s).
- Possibilitar a customização em tempo de implantação pela edição de suas entradas de ambiente (ver seção 3.1.4).
- Definir contratos (ver seção 3.1.3) que permitem que componentes EJBs de diferentes fornecedores possam interoperar entre si em tempo de execução, e também com aplicações não-Java, via protocolos CORBA.
- Permitir que fornecedores de software possam estender seus produtos existentes para suportar Enterprise JavaBeans.

Em termos práticos, tais características permitem que um enterprise bean seja implantado em um *container* EJB, dentro de qualquer servidor EJB, mesmo se diferentes servidores implementem seus serviços de diferentes maneiras. O modelo EJB garante essa portabilidade através de um conjunto padrão de contratos (ver seção 3.1.3) entre o *container* e o enterprise bean, e entre o cliente e o *container*. No entanto, o enterprise bean não obriga o uso de um sistema de *container* específico. Ou seja, um fornecedor pode adaptar qualquer servidor de aplicação para suportar a tecnologia Enterprise JavaBeans adicionando suporte para esses contratos definidos na especificação EJB [THOMAS, 1998].

3.1.3. Contratos

Basicamente, existem o dois tipos de contratos EJB: o contrato da Visão de Cliente e o contrato do Componente [SUN, 1999b], como descrito nas subseções a seguir.

➢ **Contrato da Visão de Cliente**

É o contrato entre o cliente e um *container*, que provê um modelo de desenvolvimento uniforme para aplicações que usam enterprise beans, e também habilita o uso de ferramentas de desenvolvimento e a reutilização de componentes.

O cliente de um enterprise bean pode ser um outro enterprise bean implantado no mesmo ou em outro *container*, ou ainda pode ser um programa Java arbitrário (um applet ou um servlet). Essa visão de cliente também pode ser mapeada para ambientes não-Java, tais como clientes CORBA escritos em outras linguagens de programação.

O contrato de visão de cliente é implementado através de um conjunto específico de interfaces, definidas pelo desenvolvedor do EJB. O *container* EJB invoca essas interfaces que envolvem o EJB (*wrappers*) em certos momentos da execução. Isso significa que o cliente não interage diretamente com o enterprise bean, mas sim com essas interfaces, cujas classes são geradas automaticamente pelo próprio *container*. A visão de cliente, ilustrada na Fig. 3.3, inclui [THOMAS, 1998] as interfaces *Home* e *Remote*, como descrito a seguir.

- Interface *Home*: provê acesso aos serviços de ciclo de vida do bean. Os clientes podem usá-la para criar e destruir instâncias do bean. O *container* automaticamente registra esta interface para cada classe de bean instalada no mesmo, através da API *Java Naming and Directory Interface* (JNDI, ver seção 3.2.3). Isso permite que o cliente localize a interface *Home* de uma classe de bean para criar uma nova instância do mesmo. Quando um cliente cria ou localiza um bean, o *container* retorna a sua interface Remote. A Fig. 3.1 apresenta um exemplo de codificação da interface *Home* que o Provedor de Enterprise Bean (seção 3.1.6) deve prover, juntamente com o enterprise bean propriamente dito.

```
1:  package Beans;

2:  import java.rmi.RemoteException;
3:  import javax.ejb.CreateException;
4:  import javax.ejb.EJBHome;

5:  public interface CalcHome extends EJBHome {
6:    Calc create() throws CreateException, RemoteException;
7:  }
```

Fig. 3.1. Exemplo de codificação da interface *Home*.

- Interface *Remote*: provê acesso aos métodos de negócio do enterprise bean. Permite que o *container* intercepte todas as operações feitas no enterprise bean.

Cada vez que o cliente invoca um método, sua requisição passa pelo *container* antes de ser delegada ao enterprise bean, e é dessa forma que o *container* implementa o gerenciamento de estado, o controle de transação, os serviços de segurança e de persistência transparentemente, tanto para o cliente quanto para o enterprise bean, atuando como elemento de ligação entre o enterprise bean e o servidor EJB. A interface *Remote* estende a interface `javax.ejb.EJBObject`. A Fig. 3.2 apresenta um exemplo de codificação da interface *Remote* que o Provedor de Enterprise Bean (ver seção 3.1.6) também deve fornecer.

```
1:  package Beans;

2:  import javax.ejb.EJBObject;
3:  import java.rmi.RemoteException;

4:  public interface Calc extends EJBObject {
5:    public double calcBonus(int multiplier, double bonus)
6:      throws RemoteException;
7:  }
```

Fig. 3.2. Exemplo de codificação da interface *Remote*.

➢ **Contrato do Componente**

É o contrato entre um enterprise bean e seu *Container*. Basicamente, determina quais os requisitos que devem ser seguidos pelos [SUN, 1999b]:

- Desenvolvedores do EJB: devem a implementar os métodos de negócio na classe do enterprise bean e especificar as interfaces *Home* e *Remote*. Ver seção 3.1.6.
- Desenvolvedores do *container*: devem garantir que *container* delegue a invocação feita pelo cliente aos métodos do EJB e que o *container* implemente as classes das interfaces *Home* e *Remote*. Ver seção 3.1.6.

A Fig. 3.3 ilustra as principais interdependências entre o enterprise bean, o *container*, o servidor EJB e o cliente.

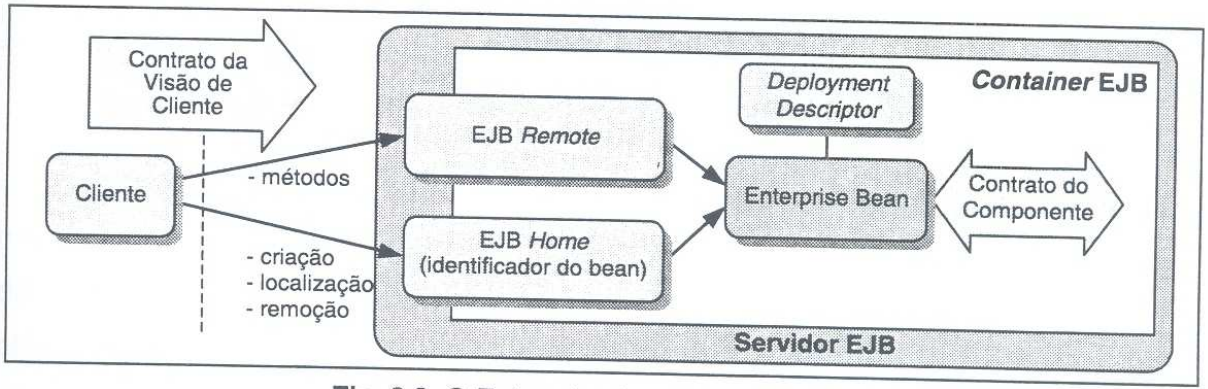

Fig. 3.3. O Enterprise JavaBean *Container*.

3.1.4. O *Deployment Descriptor*

As regras associadas com o gerenciamento de ciclo de vida, transações, segurança e persistência de um enterprise bean estão definidas em um arquivo associado denominado *deployment descriptor* – descritor de implantação, ou entrada de ambiente, escrito em XML (*eXtensible Markup Language*).

Essas regras são especificadas declarativamente em tempo de desenvolvimento – manualmente ou via ferramenta de desenvolvimento / implantação. Em tempo de execução, o *container* executa serviços de acordo com os valores descritos por esse arquivo [THOMAS, 1998].

A Fig. 3.4 apresenta um exemplo de um *deployment descriptor* para componente EJB (seção 3.2.7), com algumas entradas de ambiente definidas para: nome do componente (linha 10) na aplicação, tipo do componente (linha 7), nome da interface *Home* (linha11), nome da interface *Remote* (linha 12), entre outros. Observa-se que os nomes das interfaces *Home* e *Remote* seguem o padrão de nome (*namespace*) dos exemplos das Fig. 3.1 e Fig. 3.2 (`package Beans`).

```
1:  <?xml version="1.0" encoding="Cp1252"?>

2:  <!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
      JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

3:  <ejb-jar>
4:    <description>Session Bean para cálculo de multiplicação</description>
5:    <display-name>CalcJar</display-name>
6:    <enterprise-beans>
7:      <session>
8:        <description>no description</description>
9:        <display-name>CalcEJB</display-name>
10:       <ejb-name>CalcEJB</ejb-name>
11:       <home>Beans.CalcHome</home>
12:       <remote>Beans.Calc</remote>
13:       <ejb-class>Beans.CalcEJB</ejb-class>
14:       <session-type>Stateless</session-type>
15:       <transaction-type>Bean</transaction-type>
16:     </session>
17:   </enterprise-beans>
18: </ejb-jar>
```

Fig. 3.4. Exemplo de *deployment descriptor*.

Para cada instância de enterprise bean, o *Container* EJB gera uma instância de um objeto de contexto que mantém informações sobre o gerenciamento das regras e do estado corrente da instância, obtido originalmente do *deployment descriptor*. Esse objeto de contexto é utilizado tanto pelo *container* EJB quanto pelo enterprise bean para coordenar transações, segurança, persistência, entre outros serviços.

3.1.5. Componentes *Session* e *Entity*

Os componentes enterprise beans podem ser de dois tipos: *session* ou *entity*. O objetivo dessa divisão feita pela arquitetura EJB é o de oferecer componentes flexíveis para as aplicações servidoras [SUN, 1999b], como descrito a seguir.

➢ **Componente tipo *Session***

Um componente tipo *session*, ou de sessão, é um componente servidor que visa suprir os seguintes requisitos de aplicação:

- Representar uma sessão com um único cliente, mantendo o estado conversacional (*stateful*) das múltiplas invocações de métodos realizadas pelo cliente, ou representar um serviço sem estado (*stateless*) para múltiplos clientes;
- Poder estar ciente de uma transação e atualizar dados de um banco de dados;
- Ter uma identificação única gerada e gerenciada por seu *container*;
- Ser eliminado quando seu *container* sofrer uma parada: o cliente deve então restabelecer a conexão (criar um novo componente *session*) para continuar a computação.

A Fig. 3.5 apresenta um exemplo de codificação de *session* bean. Para esse exemplo, as interfaces *Home* e *Remote* correspondentes são as apresentadas nas Fig. 3.1 e Fig. 3.2 – observa-se que todos os espaços de nomes das classes e interfaces nesses exemplos seguem o mesmo padrão (`package Beans`). O *deployment descriptor* correspondente é o da Fig. 3.4.

```
1:  package Beans;

2:  import java.rmi.RemoteException;
3:  import javax.ejb.SessionBean;
4:  import javax.ejb.SessionContext;

5:  public class CalcEJB implements SessionBean {
6:    public double calcBonus(int multiplier, double bonus) {
7:      double calculo = (multiplier * bonus);
8:      return calculo;
9:    }
10:   public void ejbCreate() { }
11:   public void setSessionContext(SessionContext ctx) { }
12:   public void ejbRemove() { }
13:   public void ejbActivate() { }
14:   public void ejbPassivate() { }
15:   public void ejbLoad() { }
16:   public void ejbStore() { }
17: }
```

Fig. 3.5. Exemplo de codificação de um componente *session* bean.

➢ **Componente tipo *Entity***

Um componente tipo *entity*, ou entidade, é um componente servidor que visa suprir os seguintes requisitos de aplicação:

- Representar um objeto de negócio, que pode ser compartilhado entre múltiplos clientes;
- Ter um ciclo de vida diretamente atrelado a um banco de dados – o identificador único de um *entity* é a chave primária de uma entidade em um banco de dados;
- Sobreviver a uma parada do *Container* EJB – o componente *entity*, sua chave primária e sua referência remota são mantidas. Se o estado de um componente *entity* tiver sido atualizado por uma transação no momento da parada do *container*, o estado do *entity* é automaticamente reiniciado com o estado existente após o último *commit*. Essa falha, porém, não é totalmente transparente ao cliente, que recebe uma exceção se chamar esse *entity* após a parada.

3.1.6. Papéis da Arquitetura

Com o objetivo de orientar a tarefa de desenvolvimento de enterprise beans, a arquitetura Enterprise JavaBeans definiu seis papéis distintos, realizados pelos diferentes participantes do ciclo de vida de um EJB, como descrito nas subseções a seguir [SUN, 1999b].

➢ **Provedor de Enterprise Bean**

O Provedor de Enterprise Bean, ou apenas Provedor de Bean, é quem produz os enterprise beans. Seu produto de saída é um arquivo **ejb-jar**, que contém um *deployment descriptor* e um ou mais enterprise beans – com suas respectivas classes (onde estão os métodos de negócio) e interfaces *Home* e *Remote* –, como nos exemplos das Figs. 12, 13, 15 e 16.

O Provedor de Bean deve ser um especialista no domínio de aplicação, mas não necessariamente um especialista em transação, concorrência, segurança, distribuição, etc., uma vez que essas responsabilidades são competência do *container* EJB.

➢ **Montador de Aplicação**

O Montador de Aplicação (*Application Assembler*) é quem combina vários enterprise beans em grandes unidades de implantação de aplicação. Sua entrada é um ou mais arquivos **ejb-jar** produzidos pelo(s) Provedor(es) de Bean, e sua saída é

um ou mais arquivos **ejb-jar**, que contêm os enterprise beans juntamente com suas instruções de montagem da aplicação, inseridas em um *deployment descriptor.*

O Montador de Aplicação é um especialista no domínio que compõe as aplicações que usam enterprise beans. Embora familiarizado com a funcionalidade provida pelas interfaces *Home* e *Remote*, não precisa conhecer a implementação do enterprise bean.

- **Implantador**

O Implantador (*Deployer*) obtém um ou mais arquivos **ejb-jar** produzidos pelo Provedor de Bean ou pelo Montador de Aplicação e implanta seu(s) enterprise bean(s) em um ambiente operacional específico – que inclui um Servidor EJB e respectivo *container.*

O Implantador é especialista em um determinado ambiente operacional. Deve resolver as dependências externas declaradas pelo Provedor de Bean e seguir as instruções de montagem da aplicação definidas pelo Montador de Aplicação (p. ex. mapeamento das funções de segurança definidas no bean para as contas de um ambiente operacional). Para executar essas tarefas, o *Deployer* usa as ferramentas providas pelo Provedor de *Container* EJB.

- **Provedor de Servidor EJB**

O Provedor de Servidor EJB é um especialista na área de gerenciamento de transação distribuída, objetos distribuídos e outros serviços de baixo nível de sistema. Tipicamente, ele é um fornecedor de Sistema Operacional, de *middleware* ou de banco de dados. A arquitetura EJB assume que as funções do Provedor de Servidor EJB e do Provedor de *Container* EJB são realizadas pelo mesmo fornecedor de software, portanto, não define requisitos de interfaces para o Provedor de Servidor EJB.

- **Provedor de *Container* EJB**

O Provedor de *Container* EJB (ou apenas Provedor de *Container*) provê as ferramentas de implantação necessárias para a implantação dos enterprise beans, bem como o suporte em tempo de execução para as instâncias do enterprise bean implantado.

Pela perspectiva do enterprise bean, o *container* é parte do ambiente operacional em questão. O *runtime* do *container* provê seus enterprise beans com gerenciamento transacional e de segurança, distribuição em rede e gerenciamento de

recursos, que geralmente compõem uma plataforma servidora. O Provedor de *Container* isola o enterprise bean das especificidades de um determinado Servidor EJB, provendo uma série de APIs padrão (JDK, EJB, JDBC, JNDI, JTA e JavaMail, descritas na seção 3.2.3.) entre o enterprise bean e o *container* – que definem o contrato de componente do EJB.

- **Administrador de Sistema**

O Administrador de Sistema é responsável por configurar e administrar a infra-estrutura de rede e computação da empresa, que inclui o Servidor EJB e o *container.* Também é responsável por inspecionar o bem-estar dos enterprise beans de aplicação implantados.

A arquitetura EJB não define contratos para gerenciamento e administração de sistema. O Administrador de Sistema usa ferramentas de gerenciamento e monitoração fornecidas pelos Provedores do Servidor EJB e do *Container* para realizar essas tarefas.

A Tab. 3.1 exemplifica os papéis EJB do cenário da Fig. 3.6: uma aplicação Web de acesso a informações sobre empregados de um certa empresa.

Exemplo de Organização	Papel EJB Desempenhado	Produto
Empresa X Especializada em integração de aplicação	▪ **Provedor de Bean**	**EJB A** genérico que permite que aplicações Java acessem os módulos de folha de pagamento de sistemas ERP.
Empresa Y Especializada em desenvolvimento de aplicações Web	▪ **Provedor de Bean** ▪ **Montador de Aplicação**	**Aplicação Web** que permite que os empregados de uma certa empresa acessem (folha de pagamento através do **EJB A** de **X**) e atualizem seus registros de informação (cadastro através dos **EJB B e EJB C** próprios).
Empresa Z Fornecedora de software servidor.	▪ **Provedor de *Container* EJB** ▪ **Provedor de Servidor EJB**	Um **Servidor EJB** e um ***Container EJB***.
Empresa ACME Seu pessoal de TI comprou a **Aplicação Web** de **Y** para o ambiente de **Z**	▪ **Deployer** ▪ **Provedor de Bean** ▪ **Administrador de Sistema**	Aplicação para que seus empregados acessem seus dados e folha de pagamento, em dados armazenados em seu sistema ERP. O pessoal de TI implanta, configura, integra (com os produtos de **Z**) e monitora a **Aplicação Web** na infra-estrutura de **ACME**.

Tab. 3.1. Exemplos para os Papéis EJB.

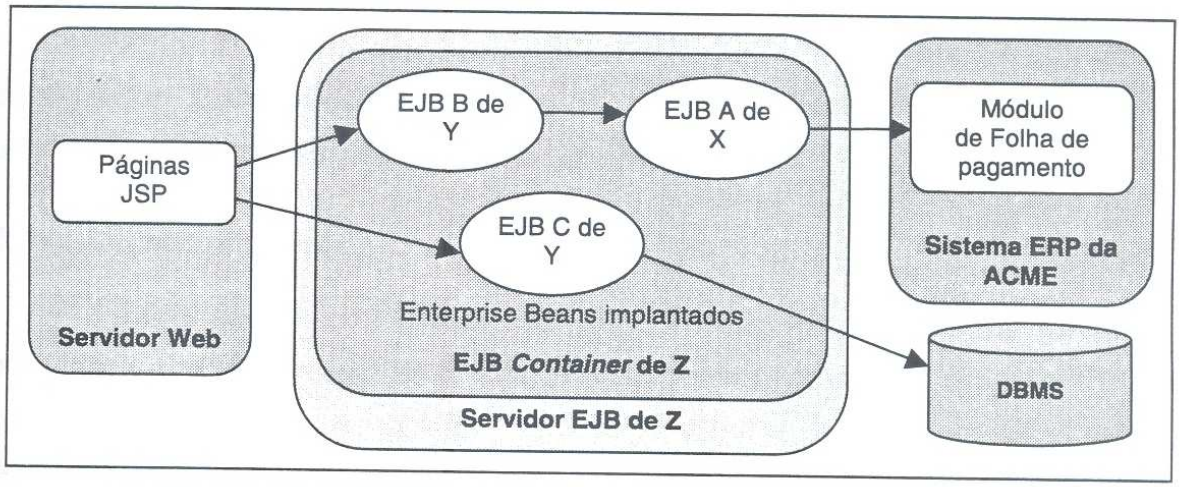

Fig. 3.6. Cenário: Aplicação Web para empregados de uma empresa.

3.1.7. Distribuição de Serviços

A especificação EJB determina o uso da API de Invocação de Método Remota Java - *Remote Method Invocation* (RMI) - para prover acesso aos enterprise beans. A RMI também suporta comunicação através do protocolo de comunicação padrão do CORBA, o *Internet InterORB Protocol* – IIOP. Os enterprise beans que se baseiam no subconjunto RMI-IIOP da RMI têm interoperabilidade com uma infra-estrutura de componentes distribuída multi-linguagem, pois qualquer cliente CORBA pode acessar enterprise beans e estes também podem acessar qualquer servidor CORBA [THOMAS, 1998].

Quando o protocolo RMI-IIOP é usado, o cliente comunica-se com o enterprise bean usando *stubs* para os objetos do lado servidor. Os *stubs* implementam, ou realizam, as interfaces *Home* e *Remote* (Fig. 3.7).

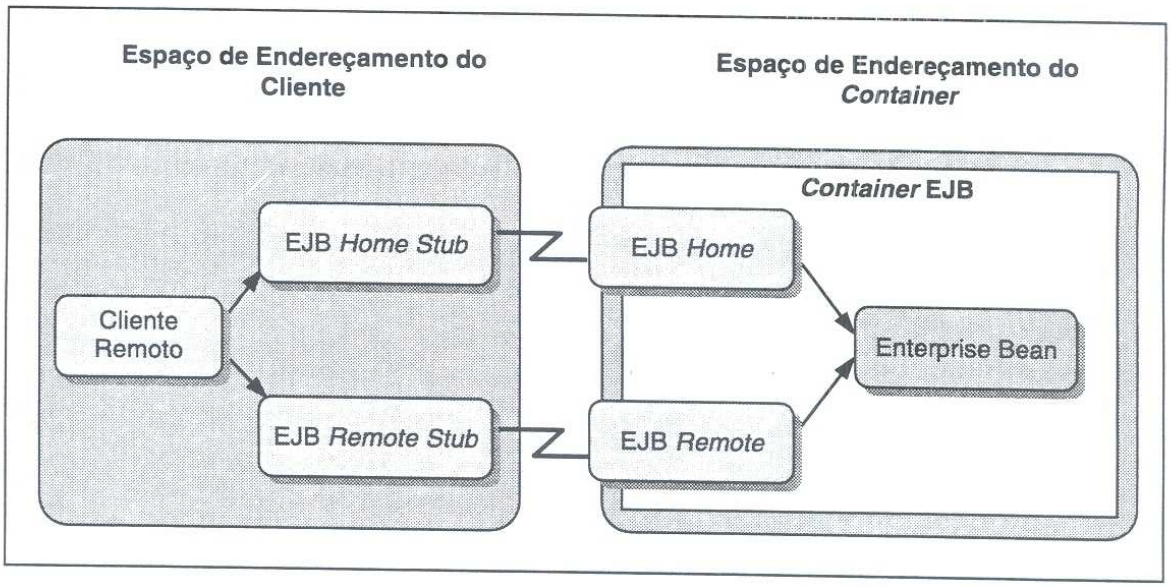

Fig. 3.7. Localização dos *stubs* EJB cliente.

Os *stubs* de comunicação usados no lado cliente são gerados, em tempo de implantação do enterprise bean, pelas ferramentas de desenvolvimento do Provedor de *Container* EJB. Esses *stubs* são padrão, se o *container* usa o RMI-IIOP como protocolo de distribuição; caso contrário, são específicos do *container* em questão.

3.1.8. Gerenciamento de Transação

A especificação EJB recomenda [SUN, 1999b], mas não obriga, transações baseadas na API de Serviço de Transação Java – *Java Transaction Service* (JTS). Ou seja, o *container* não é obrigado a suportá-la. A API JTS é a tecnologia Java que faz ligação com a especificação do Serviço de Transação de Objeto CORBA – *Object Transaction Service* (OTS). Ela provê interoperabilidade transacional via protocolo IIOP, e objetiva propagar transações entre servidores. Portanto, é direcionada a fornecedores que implementem uma infra-estrutura de processamento de transação para *middleware* corporativo. Por exemplo, um fornecedor de Servidor EJB pode usar a implementação JTS como gerenciador de transação base.

Já as aplicações EJB comunicam-se com o serviço de transação usando a API de Transação Java – *Java Transaction API* (JTA), obrigatoriamente suportada pelo *container*. Ela provê uma interface de programação para iniciar e permitir a entrada em transações, e para realizar *commits* e *rollbacks* nas mesmas, de acordo com o padrão OTS da OMG [THOMAS, 1998].

3.1.9. Restrições de Programação

O Provedor de Bean deve respeitar algumas restrições de programação para garantir a portabilidade do EJB. Em resumo, um EJB **NÃO** deve [SUN, 1999b]:

- Tentar gerenciar ou usar *threads*, pois são funções exclusivas do *container* EJB;
- Usar as funcionalidades da AWT (input / output), pois muitos servidores EJB não permitem interação direta entre o EJB e a console / teclado do sistema;
- Usar o pacote `java.io` para tentar acessar arquivos ou diretórios no sistema de arquivos – as APIs de sistema não são as mais adequadas para manipulação de dados, devendo-se preferir APIs como a JDBC;
- Tentar ouvir ou aceitar conexões em um *socket*, ou usar um *socket* para *multicast* – não é permitido que instâncias de EJB sejam servidoras de rede, pois as funções de rede são da competência do *container* EJB;

- Tentar criar um *class loader*, obter o *class loader* corrente, estabelecer o contexto do *class loader*, assinalar ou criar um gerente de segurança, parar a JVM, ou alterar os *streams* de entrada, de saída e de erro, pois essas são funcionalidades do *container* EJB;
- Tentar carregar uma biblioteca nativa (código executável específico para uma plataforma), pois essa funcionalidade é da competência do *container* EJB;
- Tentar acessar ou modificar os objetos de configuração de segurança – *Policy, Security, Provider, Signer* e *Identity*, também funções do *container* EJB.
- Tentar passar o apontador `this` como um argumento ou resultado de um método – ao invés, deve usar o `getEJBObject()` do `SessionContext` ou do `EntityContext`.
- Tentar usar a API de Reflexão Java para acessar informação que as regras de segurança da linguagem de programação Java tornam indisponíveis.

Essas regras de programação garantem a portabilidade das implementações dos enterprise JavaBeans entre diferentes *containers*. Sua adoção não deve comprometer componentes que possuam apenas métodos referentes às regras de negócio de uma aplicação. Se o comprometimento ocorrer, a lógica do componente deve ser revista.

3.1.10. Considerações

Os Enterprise JavaBeans provêem um modelo *container* de componentes servidores elegante e simples, baseado na especificação das interfaces EJBs. Suas principais características podem ser resumidas como:

- Produtividade, baseada na automação provida pelos ambientes de desenvolvimento EJB;
- Alta capacidade de customização, sem necessidade de acesso ao código fonte;
- Portabilidade WORA no lado servidor; e
- Consonância com os padrões de interoperabilidade CORBA, visando prover propagação de contextos de transação e segurança, e tratamento de clientes multi-linguagens.

Contudo, deve ser ressaltado que todos esses benefícios só são atingidos se houver completa concordância com o padrão, que por sua vez restringe aspectos mais avançados de programação (seção 3.1.9). Por essa razão, o modelo EJB entende o

Provedor de EnterpriseBean (seção 3.1.6) como um programador não altamente especializado, que não precisa e também não quer enveredar por aspectos mais complexos de programação. Ao invés, prefere fiar-se nos serviços de uma infra-estrutura de sistemas distribuídos, concentrando-se nos aspectos de negócio da aplicação – o que também incrementa sua produtividade.

3.2. Java 2 Platform, Enterprise Edition

A arquitetura Enterprise JavaBeans (EJB), vista na seção 3.1, define o modelo de componentes servidores Java que é parte integrante de um *framework* maior conhecido como *Java 2 Platform, Enterprise Edition*, ou apenas plataforma J2EE. Ambas, plataforma J2EE e arquitetura EJB, têm objetivos similares. Do ponto de vista da J2EE, o modelo de componentes EJB é o *backbone* do modelo de programação J2EE – que considera outros componentes além dos EJBs (ver seção 3.2.5). Já da perspectiva EJB, a plataforma J2EE complementa as especificações EJB com [SUN, 1999c]: (1) uma especificação completa das APIs que um desenvolvedor de enterprise bean pode usar; (2) uma definição de um ambiente de programação onde os enterprise beans são usados como componentes da lógica do negócio.

Em suma, a J2EE define uma plataforma unificada que provê suporte de ambiente à arquitetura EJB. Sua especificação agrega uma série de padrões que visam auxiliar os desenvolvedores a alcançarem o ideal WORA no lado servidor.

3.2.1. Histórico

Em abril de 1997, a Sun Microsystems anunciou sua iniciativa de desenvolver uma Plataforma Java para Corporações (*Java Platform for the Enterprise*). Para tanto, encorajou o desenvolvimento de uma coleção de Extensões Java Padrão (*Java Standard Extensions*), conhecida como APIs *Enterprise* Java, ou APIs Java Corporativas. Essas APIs tinham o objetivo de prover interfaces de programação independentes do fornecedor de *middle-tier, s*endo que sua pedra fundamental a API EJB [THOMAS, 1999]. Observa-se que a arquitetura EJB definiu um padrão de componentes servidores e uma interface independente de fornecedor para aplicações servidoras Java. No entanto, apenas o modelo EJB não é suficiente para garantir portabilidade, interoperabilidade e consistência de plataforma, pois não especifica detalhes de implementação – tais como comunicação e protocolos de transação –, apenas os indica. Ao mesmo tempo, as APIs *Enterprise* Java ainda eram classificadas como extensão padrão da plataforma Java, ou seja, não havia garantias que essas APIs estariam instaladas em um sistema específico.

Esses problemas foram contornados pela definição da plataforma J2EE. Através dela, a Sun especificou todo um mecanismo capaz de validar um sistema como apto a suportar um ambiente Java completo, com todas as APIs *Enterprise* Java. Isso significa que uma plataforma J2EE validada provê, com certeza, um ambiente de execução (*runtime*) Java integrado e consistente, que garante uma certa qualidade de serviço e assegura portabilidade e interoperabilidade para aplicações.

3.2.2. Visão Geral do Ambiente

Os serviços providos por aplicações corporativas devem, de forma geral, prover alta disponibilidade, segurança, confiabilidade e escalabilidade. Na maioria dos casos, eles são arquitetados como aplicações multi-camada, onde a camada intermediária, também chamada de camada de serviços, executa um importante papel. É ela quem manipula as requisições dos clientes, protegendo-os da complexidade existente em se lidar com recursos de EIS (*Enterprise Information Systems*), ou Sistemas de Informação Corporativos. Assim, as aplicações clientes não têm que lidar com acesso a bando de dados, ou executar complexas regras de negócio, ou ainda conectar-se a aplicações legadas. Esse trabalho é feito pela camada de serviços, de forma transparente, para os clientes.

A plataforma J2EE proporciona essas facilidades, provendo uma arquitetura baseada em componentes, centrada em serviços e habilitada para aplicações multi-camada, cujo *backbone* é constituído pelos componentes EJB (ver Fig. 3.8).

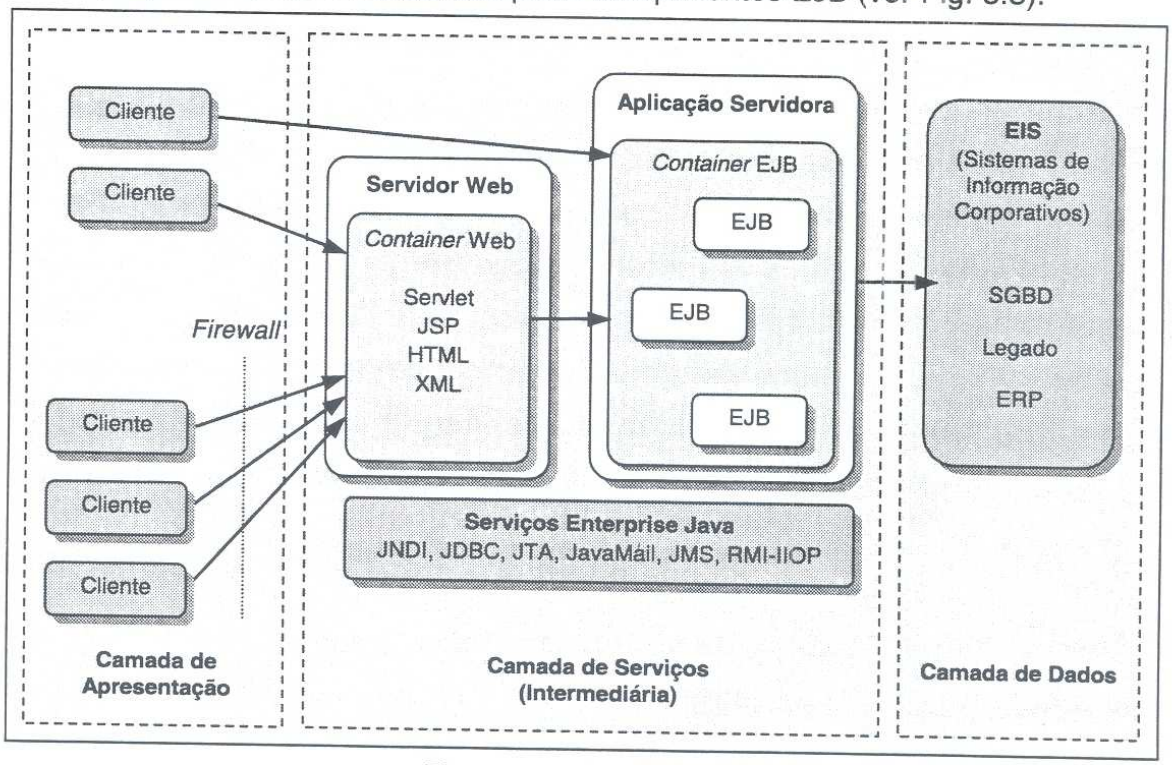

Fig. 3.8. Ambiente J2EE.

Como o ilustrado da Fig. 3.8, a Aplicação Servidora suporta componentes EJB, atendendo diferentes tipos de cliente. Ela combina as tecnologias tradicionais de OLTP (*Online Transaction Processing*) com as novas tecnologias de objetos distribuídos para prover desempenho, escalabilidade e *runtime* robusto, reduzindo a complexidade e o custo de desenvolvimento de serviços corporativos. Mais detalhes sobre Aplicações Servidoras J2EE podem ser vistos nas seções 3.2.5, 3.2.6 e 3.2.7.

A J2EE alcança esses benefícios pela definição de uma plataforma padrão composta dos seguintes lançamentos [SUN, 1999e] [SUN, 1999g] [THOMAS, 1999]:

- J2EE *Platform Specification*: (Especificação da Plataforma J2EE) define as APIs *Enterprise* Java (ver seção 3.2.3) que devem ser suportadas para garantir mínima qualidade de serviço, indicando as versões dessas APIs e as práticas que irão assegurar compatibilidade, portabilidade e integração.
- J2EE *Application Programming Model*: (Modelo de Programação de Aplicação J2EE) é o modelo de programação (ver seção 3.2.6) que visa auxiliar o desenvolvimento de aplicações corporativas multi-camada para a plataforma J2EE. Inclui exemplos e *design patterns* bem sucedidos para corporações.
- J2EE *Compatibility Test Suite* (CTS): (Conjunto de Testes de Compatibilidade para a J2EE) é o conjunto de testes (ver seção 3.2.3) utilizado por um fornecedor de software para verificar se sua implementação da plataforma J2EE é compatível com a especificação padrão J2EE. Ou seja, valida se um produto está aderente à plataforma J2EE.
- J2EE *Reference Implementation*: (Implementação de Referência – J2EE SDK) é uma implementação da especificação da plataforma J2EE (ver seção 3.2.3), que visa demonstrar suas capacidades, bem como prover uma definição operacional da mesma. É usada por fornecedores de software como um padrão para determinar o que sua implementação deve fazer sob determinadas circunstâncias geradas por aplicações, e por desenvolvedores para verificar a portabilidade de sua aplicação. Também é usada como plataforma para execução do J2EE *Compatibility Test Suit*, e está disponível, assim como seu código fonte, para livre utilização.

3.2.3. APIs *Enterprise* Java

As APIs *Enterprise* Java, extensões Java padrão, provêem acesso aos serviços da camada intermediária – nome, segurança, transação, mensagem, banco de dados, etc. Elas estratificam os serviços de infra-estrutura heterogêneos e de multi-

fornecedores, pois cada API fornece uma interface de programação comum para um tipo genérico de serviço – como descrito a seguir [THOMAS, 1999]:

- EJB (*Enterprise JavaBeans*): define um modelo de componentes servidores que provê portabilidade entre servidores de aplicação.
- JNDI (*Java Naming and Directory Interface*): provê acesso aos serviços corporativos de nome e diretório, como DNS, NDS, NIS+, LDAP e COS Naming.
- RMI-IIOP (*Remote Method Invocation*): cria interfaces remotas para computação distribuída na plataforma Java. Essa extensão usa o protocolo de comunicação IIOP (*Internet Inter-ORB Protocol*) padrão.
- Java IDL (*Interface Definition Language*): cria interfaces remotas para suportar comunicação CORBA na plataforma Java. Provê conectividade que habilita que aplicações J2EE invoquem operações em serviços remotos usando a IDL da OMG e o IIOP. Inclui o compilador *IDL-to-Java* e um ORB *lightweight* que suporta IIOP.
- Servlets e JSP (*Java Server Pages*): são componentes servidores que executam em um Servidor Web e suportam a geração de HTML dinâmico e gerenciamento de sessão para clientes em browsers.
- JMS (*Java Messaging Service API*): suporta comunicação assíncrona através de vários sistemas de mensagens, tais como enfileiramento confiável (*reliable queueing*) e serviço de *publish-and-subscribe*.
- JTA (*Java Transaction API*): permite que aplicações e servidores J2EE acessem transações de forma independente de implementação. Ou seja, especifica a interface Java padrão entre um gerenciador de transação e as partes envolvidas em um sistema de transação distribuído. Mais especificamente, é a interface, a nível de aplicação, usada pelo *container* e pelos componentes de aplicação para demarcar as fronteiras de uma transação.
- JTS (*Java Transaction Service*): especifica um gerenciador de transação que suporta a API JTA e implementa o mapeamento Java da especificação OTS (*Object Transaction Service*) da OMG. Propaga transação via IIOP
- JDBC (*Java Database Access*): provê acesso uniforme a banco de dados relacionais, tais como DB2, Informix, Oracle, SQL Server e Sybase.
- JavaMail: provê um protocolo de *framework* independente para construir aplicações de envio e recebimento de correio.

➢ **J2SE, J2EE e J2EE SDK**

Um vez que nem todas as aplicações Java necessitam do conjunto completo das APIs Java, a Sun Microsystems dividiu seu conjunto de APIs nos subconjuntos J2SE e J2EE. A partir desses subconjuntos, a Sun ainda definiu uma implementação de referência para a J2EE, denominada de J2EE SDK. As diferenças básicas entre J2SE, J2EE e J2EE SDK estão descritas a seguir [THOMAS, 1999] [SUN, 1999f]:

- *Java™ 2, Standard Edition* (J2SE): é o SDK padrão da tecnologia Java, necessário para suportar aplicações Java de propósito geral - *the Java Core Classes*.
- *Java ™2, Enterprise Edition* (J2EE): é um super-conjunto da J2SE, que define uma configuração Java estendida para suporte a sistemas de aplicação corporativas (*enterprise class*). Inclui as classes das APIs do núcleo Java (J2SE) e as APIs *Enterprise* Java.
- *Java ™ 2, SDK Enterprise Edition* (J2EE SDK): é a implementação da Sun Microsystems da plataforma J2EE, que necessita da J2SE e da J2EE para executar. Seu objetivo é prover uma definição operacional da J2EE através de uma arquitetura para desenvolvimento, implantação e execução de aplicações em ambiente distribuído. Para auxiliar o desenvolvimento e implantação, a Sun ainda provê uma *Application Deployment Tool* (Ferramenta de Implantação de Aplicação), que habilita o desenvolvedor a construir, validar e implantar aplicações corporativas em um servidor J2EE.

3.2.4. Distribuição dos Serviços

Como já mencionado, a plataforma J2EE provê um conjunto predefinido de APIs Java e serviços para suportar aplicações corporativas. A plataforma completa pode ser implementada em um único sistema, ou seus serviços podem ser distribuídos através de vários sistemas. Contudo, as APIs especificadas devem estar incluídas em algum local dentro do sistema total. A Fig. 3.9 exibe uma visão geral abstrata da plataforma J2EE, sendo que as partes que a compõe estão descritas nas subseções a seguir [SUN, 1999e] [SUN, 1999c].

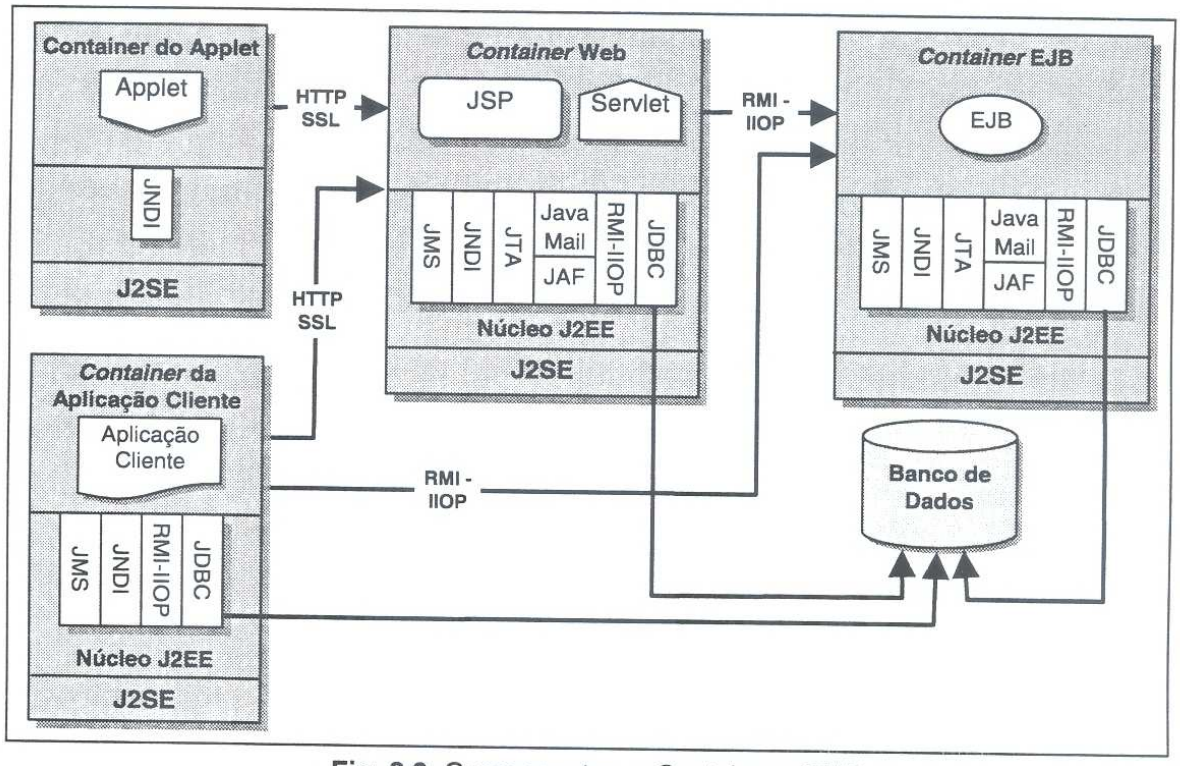

Fig. 3.9. Componentes e *Containers* J2EE.

➢ **Núcleo J2EE**

Um Provedor de Produto J2EE (ver seção 3.2.8) implementa o Núcleo J2EE (J2EE *Server Core*) usando tipicamente uma infra-estrutura existente de processamento de transação – justamente para poder fornecer serviços de gerenciamento de transação –, combinada com a tecnologia Java 2: Máquina Virtual J2SE + APIs de Serviço J2EE + *containers* para componentes servlets, JSP e EJB.

➢ **Serviços**

Como já mencionado, o conjunto de serviços padrão da J2EE simplifica a implementação de aplicações, e também permite que componentes e aplicações sejam customizadas em tempo de implantação – o que possibilita que os mesmos utilizem os recursos disponíveis no ambiente de implantação. Esses serviços padrão são acessados pelas APIs *Enterprise* Java, ou APIs J2EE, e se encarregam basicamente das atribuições descritas a seguir.

- Serviços de Nome: provêem acesso a um ambiente de nome JNDI a aplicações clientes, enterprise beans e componentes Web. É o *container* que provê ao componente J2EE um contexto de nome JNDI. Um componente J2EE localiza seu contexto de nome usando as interfaces JNDI. Um componente cria um objeto `javax.naming.InitialContext` que possibilita a procura pelo contexto de nome do componente.

- Serviços de Implantação: as aplicações J2EE são implantadas como um conjunto de unidades aninhadas. Cada unidade contém um *deployment descriptor* – um arquivo texto baseado em XML cujos elementos descrevem declarativamente como montar e implantar a unidade em um ambiente específico. São os *deployment descriptors* que armazenam os elementos de customização de serviços da plataforma J2EE, tais como serviços de nome, transação e segurança.
- Serviços de Transação: as transações dividem uma aplicação em uma série de unidades atômicas de trabalho independentes. Se uma unidade pôde ser totalmente completa, é dita *committed*. Senão, o sistema desfaz (*rollback*) todas as suas operações. As transações simplificam o desenvolvimento, pois libera o Provedor de Componente de Aplicação (seção 3.2.8) de questões como, por exemplo, recuperação e programação multi-usuário. Um componente J2EE localiza seu contexto de transação usando as interfaces JTA.
- Serviços de Segurança: garantem que recursos sejam acessados apenas por usuários autorizados. Um Provedor de Componente de Aplicação declara o mecanismo de segurança no *deployment descriptor* do componente.

3.2.5. Componente de Aplicação

Um componente é uma unidade de software a nível de aplicação. A plataforma J2EE suporta os seguintes tipos de componentes: aplicações clientes, applets, componentes Enterprise JavaBeans e componentes Web.

Os componentes J2EE não são aplicações *standalone*; eles dependem de uma entidade de sistema chamada de *container*. É o *container* que provê o contexto de aplicação, ou o ambiente de execução, para um ou mais componentes de aplicação, e também o gerenciamento e os serviços de controle. Em termos práticos, o *container* provê um processo de sistema operacional, ou uma *thread* , no qual é possível executar o componente.

Os componentes de aplicação da plataforma J2EE estão descritos nas subseções a seguir [SUN, 1999e] [SUN, 1999c]:

- **Aplicações Clientes**

Geralmente são programas GUI *standalone* escritos em Java. Executam em um computador *desktop*, dentro de sua própria JVM – nesse caso o *container* da aplicação, que fornece acesso aos serviços e APIs da J2EE(J2SE e APIs JDBC, RMI-IIOP, JNDI e JMS).

➢ **Applets**

São componentes GUI móveis que executam em um browser Web ou em um outro *container* para applet. Os applets requerem que seu *container* suporte a J2SE, o modelo de programação applet e a JNDI.

➢ **Componentes Web**

São entidades de software que geram a interface de usuário para uma aplicação baseada em Web. A J2EE especifica dois tipos de componentes Web: os Servlets e as páginas JavaServer Pages (JPS), sendo que ambos executam no servidor Web.

- Servlet: é um programa que estende a funcionalidade de um servidor Web, gerenciando requisições HTTP para serviços, e dinamicamente gerando uma resposta – um documento HTML ou XML. Dessa forma, proporciona uma ponte entre múltiplos clientes e aplicações de *back-end*. Os servlets também gerenciam informações de sessões de clientes. O *container* dos servlets deve suportar a J2SE, os servlets, e as APIs JDBC, JTA, JavaMail, JAF, RMI-IIOP, JNDI e JMS.
- Java Server Pages (JSP): também é uma tecnologia que estende um servidor Web, habilitando a geração dinâmica de conteúdo para um cliente Web. A página JSP é um documento tipo texto que descreve como processar uma requisição e criar uma resposta. Contém: um modelo de dados para formatar o documento Web (com elementos HTML ou XML); e elementos JSP e *scriptlets* para gerar o conteúdo dinâmico do documento Web – também pode usar componentes JavaBeans ou Enteprise JavaBeans para realizar procedimentos mais complexos. O *container* dos servlets deve suportar a J2SE, o JSP e as APIs JDBC, JTA, JavaMail, JAF, RMI-IIOP, JNDI e JMS.

➢ **Componentes Enterprise JavaBeans (EJB)**

São os componentes que contêm a lógica de negócio de uma aplicação corporativa, e que executam em um servidor de aplicação. O *container* EJB deve suportar a J2SE, e as APIs EJB, JDBC, JTA, JavaMail, JAF, RMI-IIOP, JNDI e JMS.

3.2.6. Modelo de Programação

Existem vários cenários de aplicações considerados pelo modelo de programação J2EE, que visam oferecer diversidade. As decisões e escolhas a nível de aplicação são, portanto, um acordo entre a riqueza e a complexidade funcional que se deseja implementar.

A Fig. 3.10 a seguir apresenta alguns desses cenários essenciais, incluindo aqueles onde tanto o *container* Web quanto o *container* EJB – e potencialmente ambos – podem ser contornados, ou tratados como entidades opcionais [SUN, 2000].

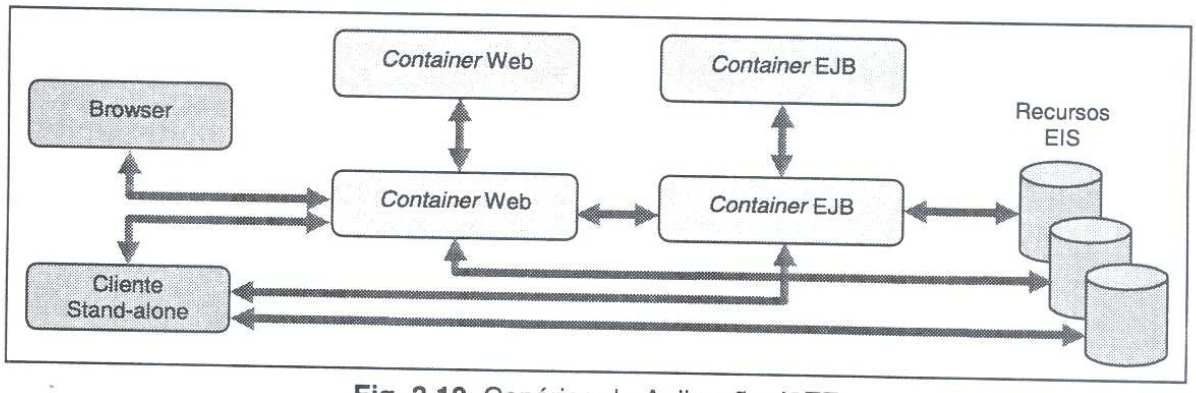

Fig. 3.10. Cenários de Aplicação J2EE.

A Fig. 3.10 reflete um modelo de aplicação multi-camada que assume a presença de *containers* Web e EJB – proposta definida com base nos seguintes requisitos corporativos:

- A necessidade de fazer rápidas e freqüentes mudanças na apresentação (aparência) da aplicação.
- A necessidade de particionar a aplicação em blocos de apresentação e lógica de negócio, de forma a incrementar a modularidade.
- A necessidade de distribuir tarefas entre recursos humanos com diferentes especialidades, de forma a obter um trabalho cooperativo, porém com independência.
- O fato de haver desenvolvedores familiarizados com aplicações corporativas de *back-office,* porém sem ter necessariamente capacitação em projetos gráfico e de novas tecnologias.
- A necessidade de implantar componentes transacionais através de múltiplas plataformas de hardware e software, de forma independente da tecnologia de banco de dados.

Embora seja razoável considerar a lógica de apresentação como descartável (a aparência da aplicação "envelhece" rapidamente), ainda há uma certa inércia associada à lógica de negócio, mais acentuada no caso dos bancos de dados, e de dados de forma geral. Assim, pode-se afirmar que a medida que se distancia dos recursos de EIS (*Enterprise Information Systems*, ou Serviços de Informação

Corporativos), a volatilidade do código de aplicação aumenta, ou seja, a "vida de prateleira" da aplicação diminui significativamente [SUN, 2000].

Por essas razões, o modelo de programação J2EE se concentra em promover um modelo que antecipa o crescimento, encoraja a reutilização de código orientado a componente e promove a comunicação entre camadas. É justamente a integração de camadas lógicas (montagem da aplicação) que fundamenta o modelo de programação J2EE.

- **Modelo de Aplicação Multi-camada**

Como visto na Fig. 3.10, o modelo de programação J2EE considera vários cenários de aplicações – apesar de não haver tendências implícitas favorecendo um cenário em detrimento de outro. Adicionalmente, um produto J2EE não deve impedir a realização de qualquer um desses cenários [SUN, 2000].

Dentre esses cenários suportados, há destaque para o cenário denominado de Modelo de Aplicação Multi-camada (Fig. 3.11). Nele, o *container* Web hospeda componentes Web dedicados quase que exclusivamente a manipular a lógica de apresentação da aplicação. A entrega de conteúdo Web dinâmico para clientes é responsabilidade de páginas JSP (suportadas por servlets). O *container* EJB, por sua vez, hospeda componentes de aplicação que, por um lado, respondem às requisições da camada Web e, por outro lado, acessam os recursos EIS. A habilidade de desacoplar o acesso a dados de questões sobre como realizar interações com o usuário final é um ponto forte desse cenário em particular. Dessa forma, a aplicação é implicitamente escalável. Porém, mais importante, a funcionalidade de *back-office* da aplicação é relativamente isolada da aparência da aplicação para o usuário final.

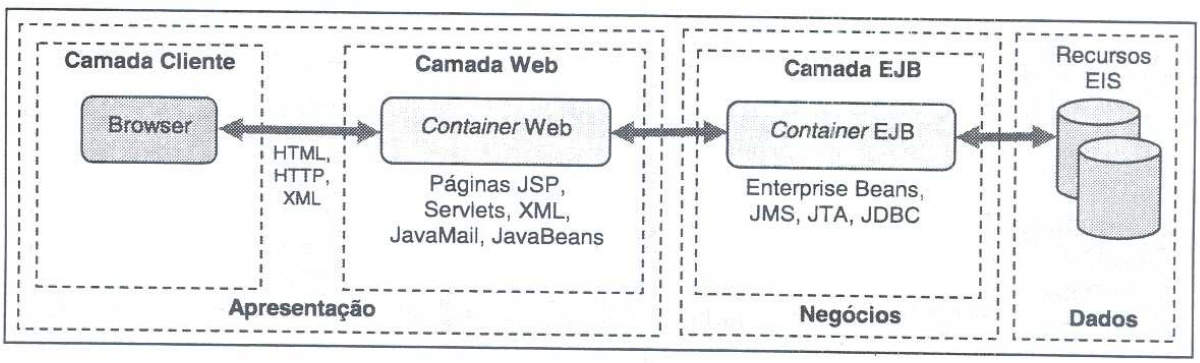

Fig. 3.11. Modelo de Aplicação Multi-camada.

Na camada Web, a questão sobre usar páginas JSP ou servlets aparece repetidamente. O modelo de programação J2EE promove a tecnologia JSP como facilidade de programação preferida em um *container* Web. As páginas JSP se

fundamentam na funcionalidade dos servlets, porém o modelo de programação J2EE toma a posição de que as páginas JSP são mais naturalmente adequadas a engenheiros Web. O *container* Web é portanto mais otimizado para a criação de conteúdo dinâmico destinado a clientes Web, e o uso de tecnologia JSP deve ser visto como norma, enquanto o uso de servlets deve ser considerado uma exceção [SUN, 2000].

- **O *Pattern* MVC**

O modelo de programação J2EE adota o *pattern* de projeto MVC (*Model-View-Controller*), ou Modelo-Visualização-Controlador, cujo objetivo é auxiliar o processo de "quebrar", ou decompor uma aplicação em componentes lógicos que podem ser arquitetados mais facilmente, incrementando a flexibilidade e reutilização de código [GAMA et al., 1995]. Isso é feito dividindo a funcionalidade da aplicação em objetos envolvidos com a preservação e com a apresentação de dados, minimizando o grau de acoplamento entre os mesmos. Originalmente, o *pattern* MVC foi desenvolvido para mapear as tarefas tradicionais de entrada, processamento e saída de um modelo de interação gráfica com o usuário. Não obstante, ele também é preciso em mapear esses conceitos em um domínio de aplicações corporativas mutli-camadas baseadas em Web.

No *pattern* MVC, o *Model* representa as regras de negócio e de dados da aplicação. O *View* trata da apresentação da aplicação, ou seja, da renderização do conteúdo do *Model*. Quando o *Model* é alterado, é responsabilidade do *View* manter a consistência com sua representação. Também é o *View* que transmite as atividades do usuário para o *Controller*.

O *Controller*, por sua vez, define o comportamento da aplicação, interpretando as ações do usuário enviadas pelo *View*, e mapeando-as em ações a serem realizadas pelo *Model*. Ou seja, o *Controller* gerencia a interação do usuário com o *View*, e as invocações ao *Model*. Em um cliente GUI *stant-alone*, as atividades do usuário podem ser a ativação de botões ou seleções de menu. Em uma aplicação Web, elas aparecem como requisições HTTP GET e POST. As ações realizadas no *Model* incluem a ativação de processos de negócios ou a alteração do estado do *Model*. Baseado nas atividades do usuário e na saída dos comandos do *Model*, o *Controller* seleciona uma *View* para ser renderizada como parte da resposta a essa requisição do usuário. Usualmente, há um *Controller* para cada conjunto de funcionalidades da aplicação. Por exemplo: em geral, aplicações de recursos humanos

têm um *Controller* para gerenciamento das interações com os empregados, e um outro *Controller* para gerneciamento das interações com o pessoal de recursos humanos.

A Fig. 3.12 descreve os relacionamentos entre o *Model*, o *View* e o *Controller* em uma aplicação MVC. Ela também ilustra como o modelo de Aplicação Multi-camada da J2EE adota o MVC, implementando-o através de páginas JPS, EJBs e componentes JavaBeans.

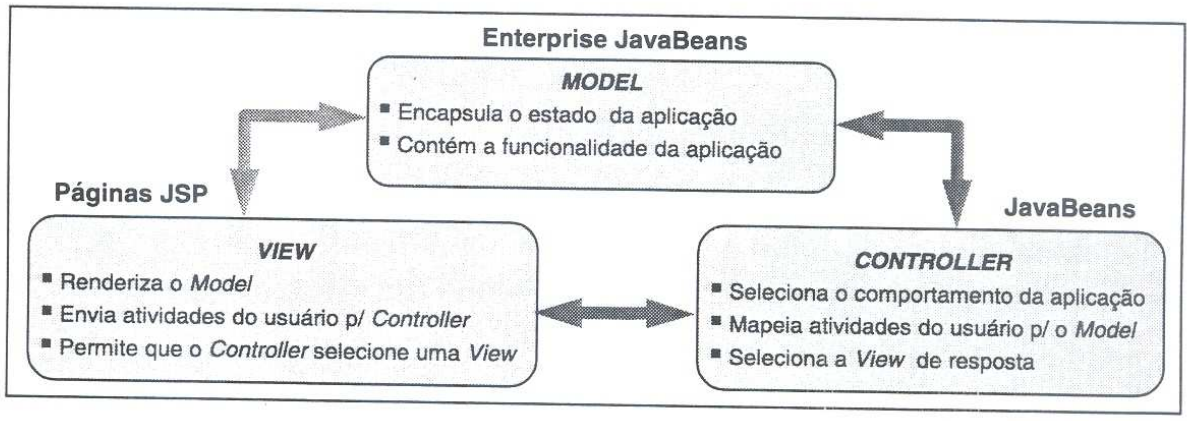

Fig. 3.12. *Pattern* MVC em aplicações J2EE Multi-camada.

3.2.7. Aplicação Corporativa

Uma aplicação corporativa (*enterprise application*) é feita de [SUN, 1999f]: (1) um ou mais enterprise beans; (2) de componentes Web ou componentes de aplicação de cliente; e (3) de um *deployment descriptor* referente ao conjunto da aplicação corporativa.

Observa-se que cada componente (Web, EJB, cliente) e cada aplicação também têm seu próprio *deployment descriptor*. É no *deployment descriptor* que estão declarados os vários atributos associados como a aplicação / componente (como autorização de segurança e gerenciamento de transação). Esses atributos serão usados pelos *containers* J2EE para customizar o comportamento da aplicação no momento de sua implantação (*deployment time*). Uma *Application Deployment Tool* (seção 3.2.3) automaticamente cria todos os *deployment descriptors* necessários, à medida que novos componentes e aplicações são desenvolvidos com auxílio da mesma [SUN, 1999f].

A Fig. 3.13 representa o conteúdo de uma aplicação J2EE corporativa típica, com as seguintes particularidades em relação ao empacotamento de componentes e aplicações:

- A Aplicações Cliente e seus *deployment descriptors* são empacotados em arquivos **.jar**.
- Os Componentes Web (JSP e Servlets), todos os seus arquivos relacionados (HML, .GIF, etc.) e seu *deployment descriptor* são empacotados em arquivos do tipo **.jar**, que nesse caso são chamados de *Web Archive* (Arquivo Web), cuja extensão é **.war**.
- Os Componentes Enterprise JavaBeans, juntamente com todos os seus arquivos de classes e seu *deployment descriptor*, são empacotados em arquivos **.jar**.
- A Aplicação Corporativa, que contém os arquivos de todos os seus componentes, mais o *deployment descriptor* da própria aplicação, é empacotada em um arquivo do tipo **.jar**, nesse caso chamado de *Enterprise Archive* (Arquivo Corporativo), cuja extensão é **.ear**. Assim que um arquivo **.ear** é criado (pela *Application Deployment Tool*), ele está pronto para ser implantado em um servidor J2EE.

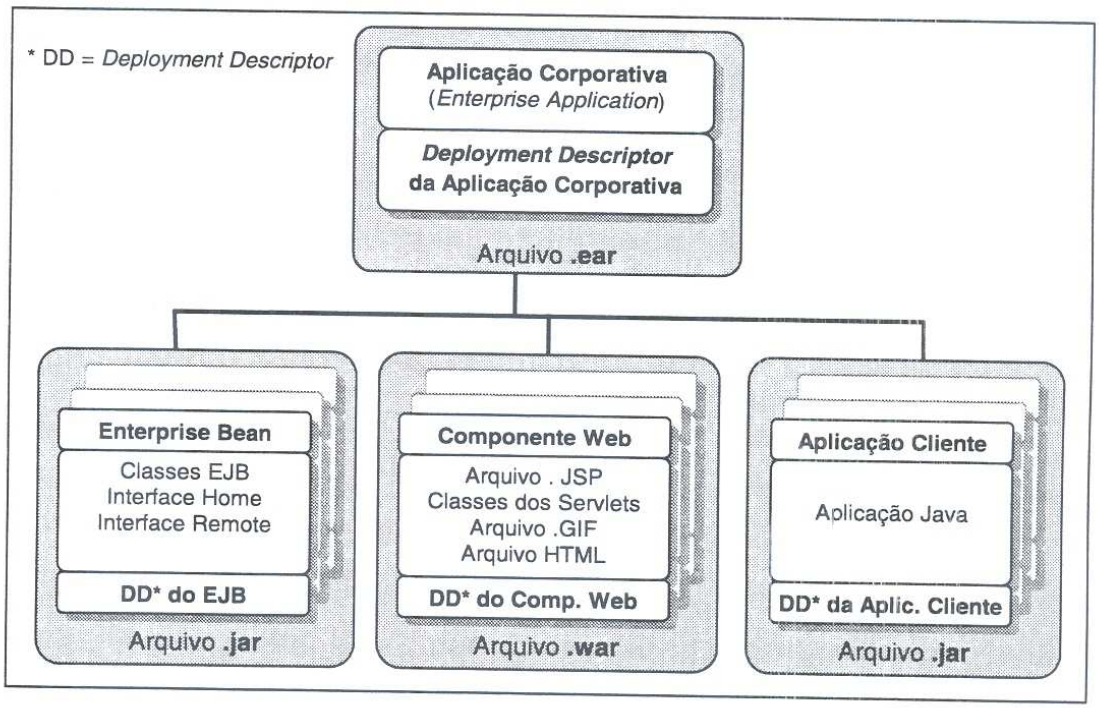

Fig. 3.13. Aplicação Corporativa.

3.2.8. Papéis da Plataforma

O processo de montagem de uma aplicação que possui diversos componentes passa pelas fases de projeto, desenvolvimento e implantação. Em geral, os papéis são definidos para auxiliar na identificação das tarefas realizadas pelos vários envolvidos no desenvolvimento, implantação e execução de uma aplicação J2EE.

Os papéis da plataforma J2EE [SUN, 2000], descritos nas seções a seguir, provêem portanto uma visão orientada a tarefa do processo de montagem de aplicação. Subconjuntos de alguns desses papéis estão definidos em outras especificações, como na do Enterprise JavaBeans (Provedor de Enterprise Bean, Provedor de *Container* EJB, Provedor de Servidor EJB, vistos na seção 3.1.6). Observa-se que esses papéis podem ser preenchidos / realizados através de uma *Application Deployment Tool*.

- **Provedor de Produto J2EE**

É um fornecedor de sistema operacional, de banco de dados, ou de servidor Web. Implementa um produto J2EE provendo os *containers* para componente, as APIs da plataforma J2EE e outras características definidas na especificação J2EE. Também pode prover ferramentas de implantação e gerenciamento, que não são definidas pela especificação J2EE.

- **Provedor de Componente de Aplicação**

Produz os blocos de construção de uma aplicação J2EE. Geralmente é especialista no desenvolvimento de componentes reutilizáveis, e tem conhecimento no domínio do negócio da aplicação. Não precisa conhecer o ambiente operacional onde seus componentes serão utilizados. Pode criar documentos HTML, JSP, enterprise beans, etc. Define os atributos declarativos dos componentes nos *deployment descriptor*, e empacota componentes em arquivos **.jar** e **.war**.

- **Montador de Aplicação**

Reúne um conjunto de componentes desenvolvidos por Provedores de Componente de Aplicação e monta uma aplicação completa J2EE, empacotando um arquivo **.ear**. Sua especialidade é prover soluções para um domínio específico de problema. Pode não estar familiarizado com o código fonte dos componentes que usa, porém usa descritores declarativos (XML) dos componentes para saber como montar aplicações a partir dos mesmos. Também não precisa conhecer o ambiente operacional onde a aplicação será utilizada. É o responsável por prover instruções de montagem, descrevendo dependências externas da aplicação que o Implantador (*Deployer*) resolverá no processo de implantação.

- **Implantador**

Ou *deployer*, é um especialista em um ambiente operacional, responsável pela implantação de componentes e aplicações J2EE em tal ambiente. Resolve as

dependências externas das aplicações, declaradas pelo Provedor de Componente de Aplicação e pelo Montador de Aplicação.

O processo de implantação geralmente envolve dois estágios: primeiro, o implantador (via ferramenta de implantação) gera as classes e interfaces adicionais que habilitam o *container* a gerenciar os enterprise beans em tempo de execução. Essas classes são específicas do *container* em questão. Em seguida, o implantador executa a instalação dos enterprise beans e de suas classes e interfaces adicionais no *container* EJB.

Em certas situações, um implantador qualificado pode customizar a lógica do negócio dos enterprise beans no momento de sua implantação. Tipicamente, ele usa as ferramentas do *container* para escrever um código de aplicação relativamente simples que envolve (*wrap*) os métodos de negócio do bean.

- **Administrador de Sistema**

É o responsável pela configuração e administração de uma infra-estrutura de rede e de computação de uma empresa. Também é o responsável pela supervisão da boa execução de aplicações J2EE.

- **Provedor de Ferramenta**

Provê ferramentas usadas para o desenvolvimento e empacotamento de componentes de aplicação. Futuras versões da plataforma J2EE devem definir mais interfaces que permitirão que as ferramentas de desenvolvimento sejam independentes de plataforma.

3.2.9. Considerações

O grande desafio para os profissionais de TI hoje em dia é o de eficientemente desenvolver e implantar aplicações distribuídas para uso tanto por Intranets corporativas quanto por Internets. As empresas que alcançarem essa habilidade terão uma considerável vantagem estratégica na economia de informação.

Com essa perspectiva, a plataforma J2EE visa um conjunto padrão de tecnologias Java que facilitam o desenvolvimento, implantação e gerenciamento de aplicações corporativas. Suas principais características e vantagens podem ser resumidas como:

- Define um ambiente Java corporativo consistente e certificado (um "selo" J2EE de integridade), que garante suporte a serviços corporativos críticos;

- Oferece um ambiente de programação de alta produtividade, que estratifica a solução em camadas lógicas, permitindo a definição de times de desenvolvimento;
- É independente de fornecedor, o que permite que as corporações escolham a solução que melhor preencha suas demandas.

3.3. Conclusões do Capítulo

As especificações e definições da plataforma J2EE preenchem diversas lacunas existentes nos atuais ambientes corporativos, principalmente no que diz respeito ao conjunto das características: portabilidade, interoperabilidade, produtividade e consistência de ambiente, dentro de um contexto que permita evolução com escalabilidade e desempenho – propriedades essenciais em aplicações servidoras.

Assim sendo, o domínio das características e capacidade de expansão da plataforma J2EE, mesmo na sua atual concepção (Dezembro de 2000), pode ser encarado como um diferencial positivo, tanto para os profissionais quanto para as empresas que lidam com tecnologia da informação. Esse leque de vantagens altamente promissor contribuiu para determinar a adoção da J2EE como plataforma de sustentação para a proposta desta dissertação.

A partir dessa decisão, e fundamentando-se apenas nos recursos e especificações da J2EE, pretende-se utilizar na mesma algumas características de reflexão computacional (detalhadas no Capítulo 4), com o objetivo de fornecer um grau a mais de flexibilidade no desenvolvimento de aplicações corporativas.

Por fim, é importante destacar que, ao se adotar essa plataforma, deve-se estar ciente de que não se obterá apenas vantagens corporativas. Também existirão restrições de programação, especificamente aquelas inerentes aos componentes enterprise beans, que não recomendam a utilização de várias primitivas Java, sob pena de quebrar a portabilidade do componente EJB e gerar conflito com as atribuições do *container* EJB (seção 3.1.9). Contudo, isso não constitui um problema se tais primitivas não forem o foco principal da aplicação em questão, exatamente o caso das aplicações corporativas, onde a meta principal, como já enfatizado, é produtividade com habilidades de interoperabilidade e portabilidade.

CAPÍTULO 4 - REFLEXÃO COMPUTACIONAL

O presente Capítulo objetiva apresentar os principais conceitos da reflexão computacional, destacando aqueles que são utilizados nesta dissertação.

Com esse intuito, o Capítulo inicia com a seção 4.1, onde é apresentada uma definição mais abrangente sobre reflexão. A seção 4.2 explora a reflexão aplicada à programação orientada a objetos – abordagem de meta-objetos –, e o conceito de reificação. A seção 4.3 apresentada os dois tipos fundamentais de reflexão – a introspecção e a intercessão. A seção 4.4 apresenta as características reflexivas já existentes no padrão Java (em mais detalhes do que os vistos na seção 2.4.3). A seção 4.5 descreve algumas abordagens reflexivas baseadas em Java, que exploram e propõem a extensão das características reflexivas Java padrão.

Por fim, o Capítulo encerra com a seção 4.6, onde é feita uma analise dos conteúdos expostos, verificando seus prós e contras em relação à plataforma J2EE, a fim de justificar a escolha de algumas habilidades reflexivas possíveis e interessantes de serem aplicadas à plataforma – base da proposta da dissertação.

4.1. Definição Geral

Intuitivamente, os sistemas de reflexão computacional permitem que as computações observem e modifiquem as propriedades de seu próprio comportamento. Por analogia, o conceito de reflexão talvez seja melhor explicado pelo estudo de autoconsciência (*self-awareness*) da Inteligência Artificial: "Aqui estou andando rua abaixo na chuva. Uma vez que eu estou ficando ensopado, devo abrir meu guarda-chuva". Esse fragmento de pensamento é um exemplo que revela uma autoconsciência de comportamento e estado, que por sua vez leva à alteração nos mesmos comportamento e estado [SOBEL & FRIEDMAN, 1998].

Mais especificamente, a reflexão computacional é o processo em que um sistema pode analisar seu próprio comportamento e atuar sobre o mesmo. Em um sistema convencional, a computação é realizada em dados que representam entidades externas ao sistema. No entanto, um sistema reflexivo deve conter dados que representam aspectos estruturais e computacionais do próprio sistema. Também deve ser possível acessar e manipular tais dados a partir do sistema e, principalmente, tais dados devem estar conectados causalmente ao comportamento do sistema. Ou

seja, alterações nesses dados devem causar alterações no comportamento do sistema e vice-versa [FABRE et al., 1995].

4.2. Protocolos de Meta-Objetos

O objetivo no desenvolvimento do protocolo de meta-objeto (*Meta-Objetc Protocl*, MOP) foi o de permitir que uma linguagem fosse extensível o suficiente para admitir a inclusão de novas características, mantendo ao mesmo tempo sua simplicidade, poder de programação, compatibilidade e desempenho com versões anteriores. Assim, o MOP permite a extensão de uma linguagem, que então passa a abrir sua abstração e implementação à intervenção do programador. Para isso, são usadas as técnicas de orientação a objetos e de reflexão computacional, que organizam uma arquitetura de nível meta, onde é possível adicionar as características que estendem a linguagem de programação [KICZALES et al., 1993].

Por conseguinte, quando a reflexão computacional é aplicada à programação orientada a objetos, tem-se a abordagem de meta-objetos. Ela estrutura os objetos em dois níveis: (1) nível base (objeto-base) e (2) nível meta (meta-objeto) [LAU, 1996]. Cada objeto-base ***x*** está associado com um meta-objeto ***^x***, que representa tanto aspectos estruturais quanto computacionais de ***x***, aspectos estes que podem ser gerenciados dinamicamente através de computações realizadas em ***^x*** (Fig. 4.1). As chamadas aos métodos do objeto-base são desviadas com o propósito de ativar meta-métodos que permitem a modificação do comportamento do objeto-base ou a adição de funcionalidades a seus métodos. Essa abordagem possibilita a separação dos aspectos funcionais e não funcionais de uma aplicação, permitindo que os métodos e procedimentos da aplicação em si sejam tratados no nível base, enquanto o controle e o gerenciamento da aplicação são tratados no nível meta. Dessa forma, a reflexão torna possível abrir a implementação de um sistema sem revelar detalhes desnecessários de sua implementação, fornecendo flexibilidade de implementação [FABRE et al., 1995].

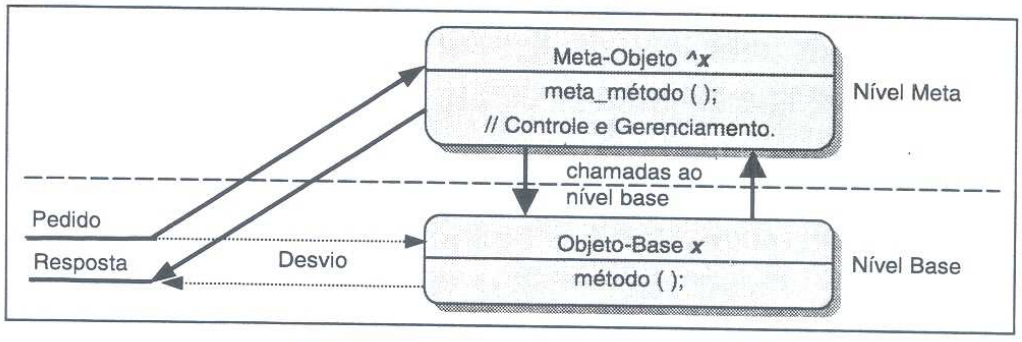

Fig. 4.1. Reflexão Computacional.

4.2.1. Reificação

A reificação (*reification*), ou materialização, é o ato de converter algo que estava previamente implícito, ou não expresso, em algo explicitamente formulado, que é então disponibilizado para manipulação conceitual, lógica ou computacional. Portanto, é através do processo de reificação que o nível meta obtém as informações estruturais internas dos objetos do nível base, tais como métodos e atributos. Contudo, o comportamento do nível base, dado pelas interações entre objetos, não pode ser completamente modelado apenas pela reificação estrutural dos objetos-base. Para tanto, é preciso intermediar as operações de nível base e transformá-las em informações passíveis de serem manipuladas (analisadas e possivelmente alteradas) pelo nível meta [OLIVA, 1998].

4.3. Tipos de Reflexão

Seguindo o raciocínio exposto em 4.2.1, a reflexão computacional pode ser particionada genericamente em dois tipos de funções [TATSUBORI, 1999] [GOLM & KLEINÖDER, 1998]: a introspecção e a intercessão, descritas nas subseções a seguir.

O programa que realiza a reflexão computacional, introspecção e / ou intercessão, é chamado de programa de nível meta, enquanto o programa executado na computação reflexiva é chamado de programa de nível base [TATSUBORI, 1999].

4.3.1. Introspecção

A introspecção (*introspection*), também referenciada como reflexão estrutural (*structural reflection*), refere-se ao processo de obter informação estrutural do programa e usá-la no próprio programa. Isso é possível através da representação, em nível meta, dessa informação do programa, feita por um processo de reificação.

4.3.2. Intercessão

A intercessão (*intercession*), também referenciada como reflexão comportamental (*behavioral reflection*), ou ainda como interceptação[5], refere-se ao processo de alterar o comportamento do programa no próprio programa. Isso pode ser realizado, por exemplo, através do ato de interceder, intermediar, ou ainda interceptar [FABRE et al., 1995] operações de nível base – como operações de invocação de

---

[5] De fato, o termo mais usado na literatura para designar reflexão comportamental é interceptação, porém sua semântica enfatiza bloqueio, obstáculo. Por essa razão, adotou-se o termo intercessão [GOLM & KLEINÖDER, 1998] [TATSUBORI, 1999], cujo significado – interceder, intermediar – é mais apropriado.

método, ou operações de estabelecimento (*setting*) / obtenção (*getting*) de valores de atributos –, que podem assim ser reificadas e, consequentemente, manipuladas pelo nível meta [OLIVA, 1998].

4.4. Reflexão Java Padrão

Desde a introdução da linguagem Java pela Sun, mais e mais características reflexivas são incorporadas à mesma. Um dos primeiros componentes reflexivos, por exemplo, é o gerenciador de segurança, que controla o acesso a diversas classes vitais, tais como as classes **File** e **Sockets**. Cada vez que um método dessas classes é executado, o gerenciador de segurança é chamado e questionado se o invocador tem o direito de executar tal método [GOLM & KLEINÖDER, 1998]. Ou seja, é uma intercessão específica de método.

4.4.1. ClassLoader

Um outro importante componente reflexivo Java é o carregador de classe (*class loader*), que transforma um *array* de bytes em uma classe. Esse processo acontece quando a JVM usa carregadores de classes para carregar arquivos **.class** e criar objetos. Como os carregadores de classe são instâncias de subclasses da classe **ClassLoader**, provida como API Java, os programadores podem definir novas subclasses da mesma em programas Java.

Assim, em uma nova subclasse da **ClassLoader**, os programadores podem alterar o comportamento do programa pela modificação do *bytecode* carregado – gerando intercessão genérica de método. Embora o custo de carga e de modificação de *bytecodes* não seja pequeno, ainda assim é útil. Nesse processo, deve ser levado em conta a dificuldade desse tipo de programação, pois a manipulação direta de *bytecodes* não é trivial [TATSUBORI, 1999].

4.4.2. API de Reflexão Java

A API de Reflexão Java oferece uma forma limitada de transformar uma classe em uma entidade que pode ser analisada e manipulada por um programa [GOLM & KLEINÖDER, 1998], ou seja, de fazer a reflexão estrutural. Ela foi introduzida com o JDK 1.1, concentrando-se basicamente nos aspectos de introspecção, e pode ser usada para [SUN, 1997b]:

- construir novas instâncias de classes e novos *arrays;*
- acessar e modificar campos de objetos e classes;

- invocar métodos em objetos e classes;
- acessar e modificar elementos de *arrays*.

Com a API de Reflexão, os programadores podem facilmente manipular classes desconhecidas (campos, métodos e construtores), possibilitando a implementação de browsers de objeto, JavaBeans ou serialização de objeto. Exemplo: a habilidade reflexiva de atuar sobre si mesmo para dinamicamente descobrir suas características estruturais (propriedades, métodos e eventos) é estendida aos beans através das APIs Introspecção Java (ver seção 2.4.3), providas por uma camada construída sobre o núcleo da API de Reflexão Java. Essa capacidade nata de introspecção visa tornar o bean customizável e manipulável de dentro de uma ferramenta visual de desenvolvimento [ORFALI & HARKEY, 1998].

Contudo, deve ser observado esse tipo de informação estrutural sobre as classes e seus membros geralmente não está disponível em tempo de compilação. A Fig. 4.2 exibe um pedaço de código que utiliza a API de Reflexão Java para obter o nome de uma classe, invocar um método em um objeto e assim por diante.

```
Object X = ...
Class  classX = X.getClass ();
Field  field = classX.getField ("name");
String name = (String) field.get (p);
```

Fig. 4.2. Exemplo de utilização da API de Reflexão Java.

É importante frisar que a API de Reflexão apenas habilita reflexão estrutural (introspecção) em tempo de execução, não oferecendo mecanismos de intercessão.

4.4.3. Considerações

A reflexão comportamental, que permite a intercessão de propósito geral de métodos, não é ofertada diretamente por nenhuma especificação Java da Sun. Essa é uma das principais motivações de vários estudos que propõem o acréscimo dessa característica reflexiva ao padrão Java.

Esses estudos são apresentados como arquiteturas reflexivas baseadas em Java, e visam permitir que sistemas desenvolvidos na linguagem de programação Java se beneficiem de um mecanismo de reflexão unificado. Os estudos que orientaram e motivaram esta dissertação foram: o sistema metaXa [GOLM & KLEINÖDER, 1998], o OpenJava [TATSUBORI, 1999] e o Guaraná [OLIVA, 1998].

4.5. Arquiteturas Reflexivas Baseadas em Java

Esta seção descreve resumidamente as abordagens adotadas pelas arquiteturas reflexivas do sistema metaXa, do OpenJava e do Guaraná.

4.5.1. metaXa

O metaXa [GOLM & KLEINÖDER, 1998], conhecido formalmente como MetaJava, é um sistema reflexivo projetado para permitir reflexão estrutural e alguma forma de reflexão comportamental – ao menos para invocação de método – em sistemas baseados em Java. Ele consiste do programa de aplicação (o sistema base), do meta-sistema e de funções IPC do sistema operacional subjacente para conectar o nível base ao nível meta. Ou seja, a abordagem metaXa para reflexão não é baseada em linguagem, mas em sistema. Logo, não é a linguagem que é estendida, e sim a JVM, o que caracteriza o metaXa como um interpretador Java estendido com habilidades de intercessão em tempo de execução.

Sua abordagem para transferência de controle do nível base para o nível meta é feita através de eventos (p. ex. invocação de métodos ou acesso a variáveis), que são lançados pelo nível base e entregues ao nível meta. O meta-sistema então pode avaliar os eventos e reagir de uma maneira específica. A passagem de evento é síncrona, logo a computação base é suspensa enquanto o meta-objeto processa o evento [GOLM & KLEINÖDER, 1998].

O metaXa também permite o empilhamento de meta-objetos. Sua principal desvantagem é a falta de suporte de linguagem para programação reflexiva, ou seja falta extensão de sintaxe de linguagem. Durante a configuração do nível meta, o programador deve referir-se a certas entidades do modelo e nesse momento podem ocorrer erros de programação (p. ex. referências inválidas a métodos) que poderiam ser detectados por um compilador [GOLM & KLEINÖDER, 1998].

4.5.2. Guaraná

O Guaraná [OLIVA, 1998] é uma arquitetura reflexiva independente de linguagem que visa alto grau de reutilização de código de nível meta, simplicidade, flexibilidade e segurança. Sua implementação foi efetivada através da modificação de uma implementação aberta de JVM, o que caracteriza sua abordagem para reflexão como a de MOP em Tempo de Execução – similar à solução adotada pelo sistema metaXa. Ou seja, a implementação do Guaraná para Java também é baseada em um interpretador Java estendido. Nesse caso, nem a linguagem Java, nem o formato dos

bytecodes são alterados. Isso permite que qualquer aplicação Java existente possa tornar-se reflexiva através de sua execução nessa JVM modificada [OLIVA, 1998]. É o núcleo do Guaraná que fica responsável pelas funções de: intercessão e reificação de operação; ligação e invocação dinâmica para objetos do nível meta; e manutenção de meta-informação estrutural.

Os meta-objetos dessa arquitetura podem ser combinados através de *composers*, que permitem que vários meta-objetos sejam associados com um objeto de aplicação, habilitando a criação de blocos de construção de meta-objetos, usados para prover complexas configurações de nível meta. Como os *composers* são meta-objetos, eles também podem ser combinados entre si, provendo um mecanismo de composição hierárquico [OLIVA, 1998].

4.5.3. OpenJava

O OpenJava [TATSUBORI, 1999] é uma extensão da sintaxe da linguagem Java padrão, com o propósito de prover reflexão estrutural e comportamental. As características estendidas do OpenJava são especificadas por um programa de nível meta em tempo de compilação. Assim, se não houver nenhuma declaração de nível meta, o OpenJava é idêntico ao Java.

Internamente, o compilador OpenJava opera em três estágios: recebe o código fonte base estendido (OpenJava, que contém as declarações dos meta-objetos), realiza o pré-processamento que traduz o OpenJava para o Java, e por fim realiza a compilação Java regular, gerando *bytecodes* para uma JVM padrão. Sua diferença em relação aos compiladores Java regulares está nas bibliotecas de nível meta, em adição às bibliotecas regulares [TATSUBORI, 1999]. Uma visão geral do processamento do compilador OpenJava é exibida na Fig. 4.3.

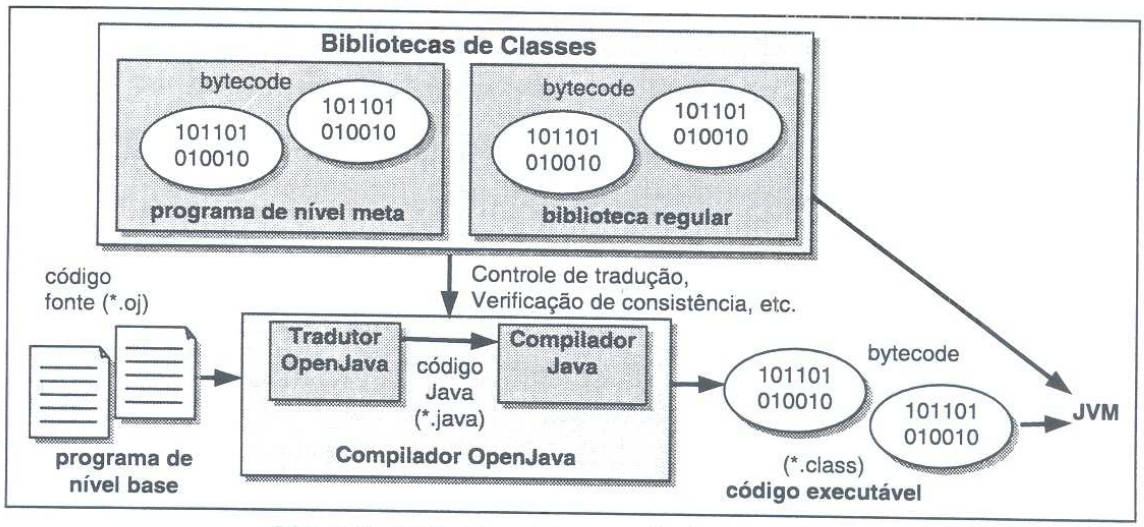

Fig. 4.3. Visão Geral do compilador OpenJava.

O OpenJava adota uma abordagem de MOP em Tempo de Compilação, pois seu requisito básico é a eficiência na execução de aplicações, evitando o custo do MOP em Tempo de Execução [TATSUBORI, 1999] – como nos caso dos interpretadores Java estendidos. Contudo, neste caso, só é possível tornar reflexivas as classes cujo código fonte está disponível para modificação – o que não ocorre nos MOP em Tempo de Execução.

4.6. Conclusões do Capítulo

A reflexão computacional (comportamental e / ou estrutural) permite um alto grau de flexibilidade aos sistemas computacionais, pois possibilita a extensão de sua capacidade de gerenciamento e de sua adaptabilidade a novos requisitos – uma demanda sempre presente em sistemas de informação. Consequentemente, diversos estudos, como os citados anteriormente, vêm sendo realizados com o intuito de prover tais características reflexivas às plataformas em geral, e mais especificamente à plataforma Java.

O ideal seria prover tais vantagens com um mínimo de alteração na plataforma em questão, garantindo desempenho e compatibilidade com os sistemas de aplicação existentes. Avaliando as soluções apresentadas, a abordagem de MOP em Tempo de Execução não só não altera a linguagem de programação Java base, como também não modifica o formato dos *bytecodes*. Entretanto, podem ocorrer erros de referência (que poderiam ser corrigidos por um pré-processador) no momento de adicionar características reflexivas às aplicações, e adicionalmente, a JVM deve ser estendida para poder suportar as características reflexivas.

Já a abordagem de MOP em Tempo de Compilação, apesar de conseguir desempenho, altera a linguagem Java base, e apenas permite tornar reflexivas as aplicações cujo código fonte está disponível – sendo esta sua maior restrição.

Ao se decidir como proporcionar reflexão na plataforma J2EE, essas informações devem ser bem avaliadas. Também deve-se ter em mente que, ao se trabalhar com componentes encapsulados, torna-se implícito que seu código fonte não está disponível (como proposto em [TATSUBORI, 1999]). Qualquer acesso ao mesmo apenas se dá através de sua interface. Da mesma forma, extensões na JVM (como proposto em [GOLM & KLEINÖDER, 1998] e [OLIVA, 1998]) podem implicar em problemas de compatibilidade com a plataforma (p. ex., gerando conflitos com as atribuições do *container*), o que levaria ao comprometimento de um dos maiores trunfos da J2EE, que é justamente o ambiente Java corporativo, consistente e

integrado. E ainda, soluções corporativas baseadas em componentes são intrinsecamente de menor granularidade que as soluções objetivadas pelas abordagens OpenJava, Guaraná e metaXa. Isto altera o enfoque objetivado pela aplicação: maior granularidade, menor modularidade, solução mais ad hoc; menor granularidade, maior modularidade, solução de maior abrangência, característica das soluções servidoras corporativas.

Como já mencionado (seção 4.2.2), a plataforma J2EE já contém implicitamente características de introspecção (reflexão estrutural) – fundamentadas sobre as APIs de Reflexão Java –, que inclusive são utilizadas por ferramentas de desenvolvimento para levantar informações estruturais (métodos, atributos) dos componentes [ORFALI & HARKEY, 1998]. Ao mesmo tempo, o modelo de programação da plataforma J2EE favorece a montagem de aplicação pela combinação de blocos lógicos (seção 3.2.6), gerando a possibilidade de inserir reflexão comportamental (intercessão) apenas nos chamados componentes de negócio. Tal reflexão estaria restrita a um determinado módulo funcional da aplicação, não estendendo-se a toda plataforma, como proposto pelas arquiteturas reflexivas Java metaXa, Guaraná e OpenJava, que propõem uma nova perspectiva para o padrão Java. Porém, ainda assim é possível alterar o comportamento da aplicação, uma vez que é na lógica de negócio que estão os métodos que definem seu domínio funcional – e ao mesmo tempo permite-se que o padrão Java vigente seja mantido de forma integral. Consequentemente, mantém-se a idéia de um nível meta capaz de controlar a aplicação, com independência em relação ao nível base.

Enfim, o conjunto dessas características presumem uma alternativa para implementação de reflexão computacional na plataforma J2EE, ao menos para invocação de método de negócio – intercessão ou reflexão comportamental (ver seção 4.3.2) –, abordagem esta denominada de **Intercessão em Tempo de Implantação**, cujo principal preceito é o de preservar a compatibilidade com o padrão J2EE. O Capitulo 5 a seguir desenvolve essa proposta, através de um modelo de integração.

CAPÍTULO 5 - MODELO DE INTEGRAÇÃO PROPOSTO

O presente Capítulo detalha o modelo de integração que propõe a utilização de reflexão computacional na plataforma J2EE – Intercessão em Tempo de Implantação. Para tal, utiliza os conceitos e conclusões sobre tecnologias e mecanismos expostos nos Capítulos anteriores. Com esse propósito, o Capítulo inicia com a seção 5.1, onde são apresentados formalmente os preceitos e diretrizes da proposta. A seção 5.2 explora em detalhes o modelo de integração, com exemplos de código e arquitetura de implementação. Por fim, o Capítulo encerra com a seção 5.3, onde é feita uma analise do modelo que permite prosseguir para o Capítulo 6, onde são apresentados estudos de caso visando validar a proposta e ressaltar suas vantagens.

5.1. Diretrizes da Proposta

O modelo de integração proposto foi elaborado a partir do paralelo comparativo (Capítulo 4) dado pela análise das arquiteturas reflexivas metaXa, Guaraná e OpenJava – cujas soluções orientaram a elaboração da abordagem MOP em Tempo de Implantação (*Deploy-time* MOP), no sentido de evidenciar quais restrições as mesmas teriam na plataforma J2EE, e quais alternativas poderiam ser consideradas. Em síntese, essas arquiteturas fundamentam-se em uma das seguintes abordagens:

- Abordagem MOP em Tempo de Execução (*Runtime* MOP), caracterizada por um interpretador Java estendido, com habilidades de intercessão em tempo de execução (JVM estendida); este é o princípio básico do metaXa e do Guaraná; e
- Abordagem MOP em Tempo de Compilação (*Compile-Time* MOP), caracterizada por um pré-processador de código fonte base, modificado a fim de prover habilidades de intercessão e introspecção em tempo de compilação (linguagem Java estendida) – apenas classes cujo código fonte está disponível podem tornar-se reflexivas; este é o princípio básico do OpenJava.

5.1.1. Contexto J2EE

Elegendo a preservação das características e vantagens do contexto J2EE como principal preceito da proposta, foi possível verificar que os princípios do OpenJava, MetaXa e Guaraná não poderiam ser adotados, uma vez que ou propõem novos padrões Java, ou se baseiam em práticas nem sempre acessíveis na plataforma J2EE. Tal situação é melhor explicada a partir dos seguintes fatores da J2EE:

(1) Aplicações J2EE trabalham com componentes de negócio que, a princípio, são encapsulados – os COTS, que não disponibilizam seu código fonte; e

(2) Modificações nos padrões Java (p. ex. extensões na JVM) podem implicar em problemas de compatibilidade com a plataforma, gerando comprometimento de seu ambiente Java corporativo, consistente e integrado.

Por conseguinte, ao se considerar a plataforma J2EE, é possível inferir que o fator (1) impossibilita a abordagem MOP em Tempo de Compilação, e o fator (2) não recomenda a abordagem MOP em Tempo de Execução.

Em resumo, os fatos e decisões que orientaram a confecção da proposta podem ser listados como na Tab. 5.1 a seguir.

CONDUÇÃO DA PROPOSTA	
Motivação	A reflexão comportamental, que permite a intercessão de propósito geral de métodos, não é ofertada diretamente pela especificação Java, sendo este o motivo de vários estudos que propõem este acréscimo ao Java ([GOLM & KLEINÖDER, 1998], [TATSUBORI, 1999] e [OLIVA, 1998]).
Restrição	O Implantador / Montador de Aplicação J2EE, ao trabalhar com componentes EJB prontos (componentes de prateleira, encapsulados), não tem acesso ao seu código fonte. Logo, não pode manipular ou estender código dos componentes para alterar suas funcionalidades.
Preceitos	Não alterar as características da plataforma J2EE. Apenas usar os princípios definidos por sua arquitetura, mantendo compatibilidade a linguagem Java, *bytecode*, JVM, APIs Enterprise Java e ambiente corporativo consistente
Formalização da Proposta	Proporcionar intercessão (reflexão comportamental) de propósito geral aos métodos dos componentes de negócio da Plataforma J2EE, visando flexibilidade de implementação nas situações onde é necessário introduzir controle e / ou alterar o comportamento da aplicação corporativa. Essa proposta, portanto, preconiza a abordagem de **Intercessão em Tempo de Implantação** – ou mais genericamente, MOP em Tempo de Implantação.

Tab. 5.1. Fatos e decisões que orientaram a Proposta.

5.1.2. Convergência de Vantagens

Para valer-se tanto das vantagens da reflexão computacional quanto da plataforma J2EE, é preciso primeiramente atender aos fatores (1) e (2), levantados na seção 5.1.1. O modelo de programação da plataforma J2EE responde a esses quesitos, pois, ao favorecer a montagem de aplicação pela combinação de blocos lógicos, gera a possibilidade de utilizar reflexão comportamental (intercessão) apenas

na camada de negócio, representada pelos componentes de negócio. É um mecanismo reflexivo restrito apenas a um módulo da aplicação, mas que ainda assim permite a alteração do comportamento de toda a aplicação, uma vez que é a camada de negócios que define sua funcionalidade básica.

A Tab. 3, a seguir, resume quais recursos da plataforma J2EE foram adotados para efetivar o modelo de integração, de forma a atender aos preceitos expostos.

RECURSOS J2EE QUE APOIAM A PROPOSTA	
Modelo de Programação	O Modelo de Aplicação Multi-camada, auxiliado pelo *pattern* MVC, favorece a habilidade de estratificar a aplicação em camadas lógicas: a lógica de negócio (ou os aspectos funcionais da aplicação corporativa) fica isolada na Camada de Negócio, representada pelos componentes de negócio. É exclusivamente nessa camada lógica que a proposta se concentra.
Componente de Negócio	Os enterprise JavaBeans, ou EJBs, são o tipo de componente J2EE (seção 3.2.5) destinado a guardar a lógica de negócio da aplicação corporativa; ou seja, são os EJBs que definem a Camada de Negócio da aplicação J2EE. Por conseqüência, a intercessão de seus métodos proporciona a habilidade de alterar o comportamento da aplicação como um todo.
Deployment Descriptors	A facilidade de composição da aplicação corporativa pela edição das suas declarações XML (existentes nos *deployment descriptors*) permite a alteração, ou redirecionamento, da referência a componentes EJB (nome JNDI) no momento da montagem, ou composição da aplicação. Isso gera a possibilidade de inserção de um nível meta de controle através de um outro componente – denominado meta-componente.

Tab. 5.2. Principais recursos da plataforma J2EE que sustentam a Proposta.

As decisões de projeto, exibidas nas tabelas 5.1 e 5.2, permitem a definição do mecanismo de suporte para a proposta: o meta-componente – detalhado na Tab. 5.3.

META-COMPONENTE	
Princípio	Princípio MOP: provê um nível meta (meta-componente) capaz de controlar a aplicação, com independência em relação ao nível base (componente de negócio base), e que é habilitado em tempo de implantação.
Implementação	Deve ser inserido no momento de montagem da aplicação, em tempo de implantação, via redirecionamento (alteração de nome JNDI no *deployment descriptor*), e entre a chamada cliente e o componente de negócio base.
Atribuição	Prover novas funcionalidades (controle, gerência, alteração de comportamento, etc.) à aplicação corporativa J2EE, sem necessidade de acesso ou alterações no código fonte do componente de negócio base.

Tab. 5.3. Atribuições e princípio de funcionamento do Meta-componente.

5.2. Detalhamento do Modelo de Integração

O presente projeto, como mencionado na seção anterior, adota o Modelo de Aplicação Multi-camada da J2EE com o *pattern* MVC, sendo que a intercessão de funcionalidade é realizada pela execução dos passos descritos na Tab. 5.4, a seguir.

Passos	DESCRIÇÃO
1º	As novas funcionalidades que se quer adicionar à aplicação J2EE devem ser implementadas como um novo componente EJB, o meta-componente (há a necessidade de um Provedor de Bean, descrito na seção 3.1.6), que deve ter as mesmas interfaces (*Home* e *Remote*, seção 3.1.3) do componente EJB base que se quer interceptar.
2º	O meta-componente deve ser inserido na aplicação J2EE no momento da sua composição, sendo referenciado com o nome JNDI original do componente EJB base – este último deve adotar um novo nome JNDI dentro do contexto da aplicação. Ocorre então o redirecionamento de componentes pela alteração de seus nomes JNDI.
3º	O meta-componente deve se encarregar de chamar, quando e se necessário, o componente EJB base – logo deve saber o novo nome JNDI do EJB base.
4º	O *namespace* (*package* Java) do componente EJB base não é alterado, uma vez que é definido a nível de código fonte; assim, para o correto redirecionamento dos componentes, o *namespace* do meta-componente deve ser referenciado no componente que representa o *Controller* no MVC do Modelo de Aplicação Multi-camada (seção 3.2.6), ou seja, no JavaBean – que é quem de fato invoca os componentes de negócio (*Model* no MVC) em nome dos clientes.

Tab. 5.4. Seqüência para intercessão de funcionalidade em aplicações J2EE.

Os procedimentos exibidos na Tab. 5.4 visam minimizar significativamente as alterações na aplicação J2EE, necessárias à implementação da proposta.

Também deve ser ressaltado que a inserção do meta-componente na aplicação J2EE não é percebida na ponta do cliente, pois todo o redirecionamento de nomes acontece na camada de serviços da aplicação corporativa, orientada por sua vez pelo MVC: componentes Web (o *View*), componentes JavaBeans (o *Controller*), meta-componente EJB e o componente EJB base (o *Model*).

5.2.1. Representação Gráfica

A seqüência de passos exposta na Tab. 5.4 encontra-se ilustrada nas Fig. 5.1 e Fig. 5.2. A primeira, Fig. 5.1, exibe o modelo da aplicação J2EE **sem** a inserção do

meta-componente. A evolução desse sistema, pela inclusão de novas funcionalidades através de um meta-componente, é exibida na Fig. 5.2.

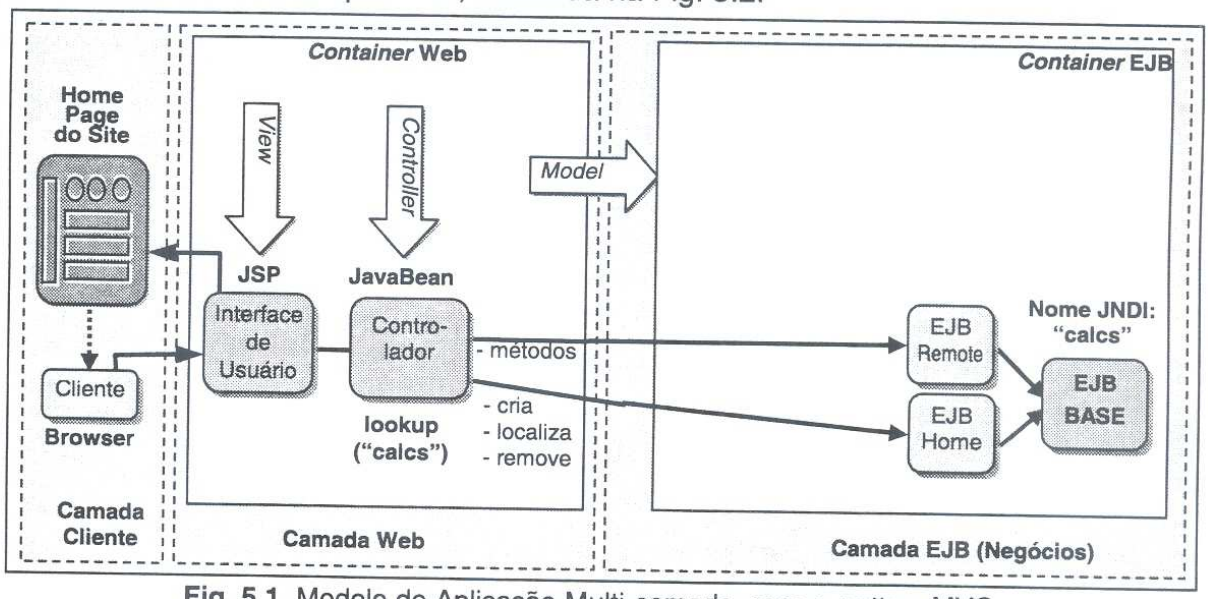

Fig. 5.1. Modelo de Aplicação Multi-camada, com o *pattern* MVC.

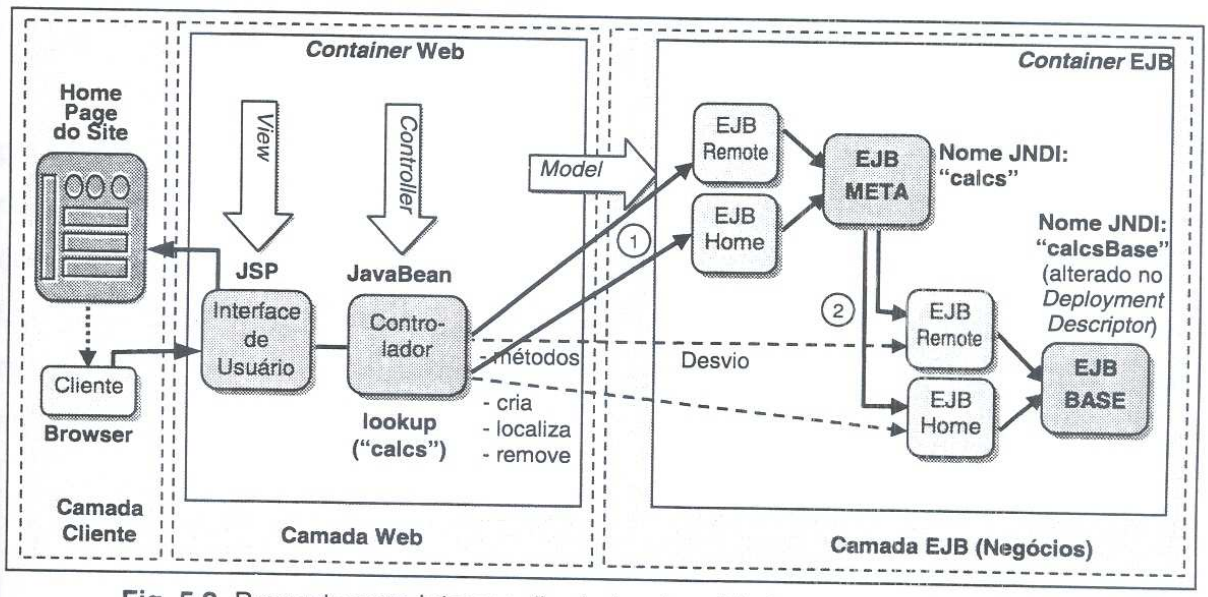

Fig. 5.2. Proposta para intercessão de funcionalidade em aplicações J2EE.

A Fig. 5.2 exibe o meta-componente EJB desviando as chamadas ao componente EJB base (ver seção 4.2), pois está identificado com o nome JNDI original e tem suas interfaces idênticas às do componente base – provê assim um nível meta capaz de controlar o nível base.

5.2.2. Codificação

A Fig. 5.3 a seguir exibe um código JSP (*View*) que captura as ações do usuário – formulário HTML, das linhas 4 a 8. As linhas 9 a 13 contêm os *scriplets* e as marcações (*tags*) JavaBeans próprios do JSP. A linha 10 faz a referência ao JavaBean (*Controller*) que irá tratar das ações do usuário. A linha 11 define uma variável JSP

que irá receber (na linha 12) o conteúdo do formulário HTML. A linha 13 invoca indiretamente um método (na linha 17 da Fig. 5.4) do JavaBean. Observa-se que essa chamada segue a convenção de nome JavaBeans (seção 2.4.3) para método e propriedade: *set*NomeDaPropriedade / *get*NomeDaPropriedade.

```
2:  <HTML>
3:  <BODY>
4:  <P> Entre com o numero do seu identificador:
5:  <FORM METHOD="GET" ACTION="Bonus.jsp">
6:    <INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT>
7:    <INPUT TYPE="Enviar" VALUE="Submit">
8:    <INPUT TYPE="Limpar">
9:  </FORM>
10: <!-- Scriptlet e Tags JavaBeans. -->
11: <jsp:useBean id = "jbonus" class = "JBonusBean"/>
12: <%! String sMult; %>
13: <%  sMult = request.getParameter("MULTIPLIER");%>
14: <jsp:setProperty name = "jbonus" property="strMult" value="<%=sMult%>"/>
15: </BODY>
16: <HTML>
```

Fig. 5.3. Exemplo JPS: arquivo "Bonus.jsp" – o *View* no MVC.

A Fig. 5.4 a seguir exibe o código do JavaBean referenciado e chamado na página JSP da Fig. 5.3. É esse JavaBean que se encarrega de invocar o componente de negócio da aplicação (*Model*), que nesse exemplo é um meta-componente – observa-se que o *namespace* "MetaBeans", na linha 3, denota essa diferenciação. De fato, a troca do nome do pacote do EJB que se quer chamar é a **única** alteração que deve ser feita no JavaBean *controller* , quando se troca o componente EJB base pelo meta-componente EJB.

```
1:  import javax.naming.*;
2:  import javax.rmi.PortableRemoteObject;
3:  import MetaBeans.*;
4:  public class JBonusBean {
5:    private String       strMult;
6:    MetaBeans.CalcHome   homecalc;
7:    public JBonusBean() {
8:      try{
9:        InitialContext ctx = new InitialContext();
10:       Object objref = ctx.lookup("calcs");
11:       homecalc = (CalcHome)
12:                  PortableRemoteObject.narrow (objref,CalcHome.class);
13:     } catch (javax.naming.NamingException e) {
14:       e.printStackTrace();
15:     }
16:   }
17:   public void setStrMult (String strMult) {
18:     this.strMult = strMult;
19:   }
20: }
```

Fig. 5.4. Exemplo JavaBean: arquivo "JBonusBean.java" – o *Controller* no MVC.

Ainda na Fig. 5.4, a linha 1 explicita a importação da API JNDI: como o JavaBean invoca componentes EJBs, deve ter acesso ao serviço de nome, invocado na linha 9 da figura. A linha 10 pede, ao serviço de nome, a referência do EJB registrado como "Calcs" na aplicação – no exemplo, esse é o nome JNDI originalmente usado pelo do EJB base, que passa a ser o do meta-componente, alterado na entrada do *deployment descriptor.*

```
1:  package MetaBeans;

2:  import java.rmi.RemoteException;
3:  import javax.ejb.SessionBean;
4:  import javax.ejb.SessionContext;
5:  import Beans.*;

6:  public class CalcEJB implements SessionBean {
7:  Beans.CalcHome homecalcBase;
8:  double         calculo;

9:    public double calcBonus(int multiplier, double bonus) {
10:     try {
11:       InitialContext ctx = new InitialContext();
12:       Object objref = ctx.lookup("calcsBase");
13:       homecalcBase = (Beans.CalcHome)
14:          PortableRemoteObject.narrow(objref, Beans.CalcHome.class);
15:        Beans.Calc theCalculation = homecalcBase.create();
16:        calculo = theCalculation.calcBonus(multiplier, bonus);
17:        calculo = calculo * 3;
18:     } cath (javax.ejb.CreateException e) {
19:         e.printStackTrace();
20:     } catch (java.rmi.RemoteException e) {
21:         e.printStackTrace();
22:     }
23:       return calculo;
24:
25:   }
26:  ...
```

Fig. 5.5. Exemplo meta-componente: "CalcEJB.java", parte do *Model* no MVC.

A Fig. 5.5 apresenta o código do meta-componente EJB. Observar a definição do *namespace* "MetaBeans" na linha 1, e a importação do pacote "Beans.*" do componente EJB base, na linha 5. Esse recurso permite que o meta-componente chame o componente base, quando for preciso.

Assim, para poder invocar o componente EJB base, o meta-componente da Fig. 5.5 procura por sua referência com o novo nome JNDI "calcsBase", na linha 12. Deve-se ter o cuidado de fazer todas as referências às interface *Home* e *Remote* do componente EJB base com seu nome completo explicitado "Beans.CalcHome" e "Beans.Calc" (nas linhas 7, 13, 14 e 15), para evitar confusão com as interfaces (idênticas) do próprio meta-componente.

Notar que o método "`calcBonus`" do meta-componente, na linha 9, deve ter a mesma assinatura que o método-base correspondente, que por sua vez é invocado na linha 16. A alteração da funcionalidade desse método é sugerida na linha 17.

5.2.3. Composição / Implantação da Aplicação Corporativa

A Fig. 5.6 a seguir exibe o redirecionamento de nomes JNDI feito através de uma *Application Deployment Tool*, provida pela J2EE SDK (seção 3.2.3), onde é efetivada a Intercessão em Tempo de Implantação. Essa ferramenta permite a edição visual das entradas XML do *deployement descriptor* no momento da montagem da aplicação J2EE, o que facilita o trabalho de inserção do meta-componente.

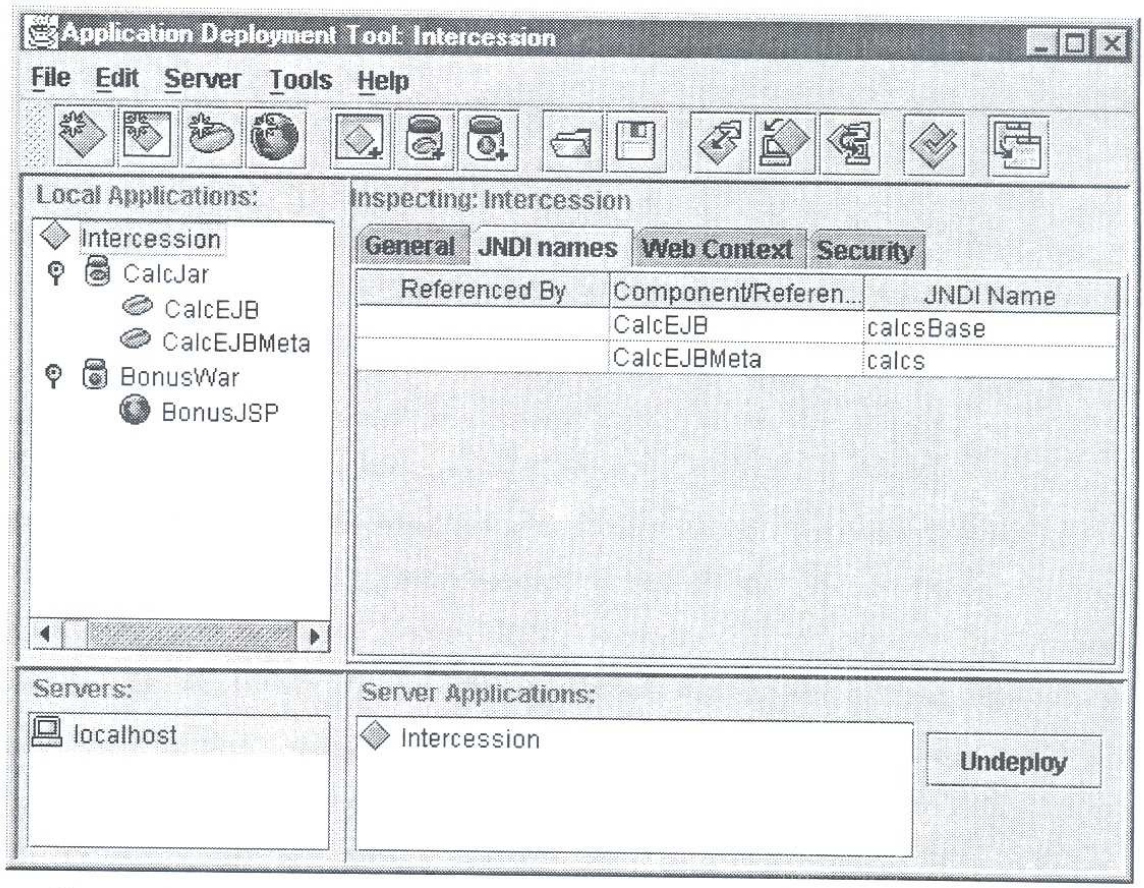

Fig. 5.6. Exemplo de inserção de meta-componente, através de ferramenta gráfica.

O painel superior à esquerda da Fig. 5.6 expõe uma aplicação J2EE, cujo nome é "Intercession", empacotada em arquivo "intercession.ear" (seção 3.2.7). Ela é composta de um módulo web (*View* e *Controller*), cujos elementos estão empacotados em um arquivo "BonusWar.war", e de um módulo EJB (*Model*), cujos elementos estão empacotados em um arquivo "CalcJar.jar". Este último contém dois componentes EJBs: (1) o componente de negócio base "CalcEJB", com o nome JNDI "calcsBase"; e o meta-componte de negócio "CalcEJBMeta", cujo nome JNDI é "calcs" (comparar com a Fig. 5.2).

Ainda na Fig. 5.6, o painel superior à direita faz a introspecção da aplicação J2EE "Intercession", exibindo algumas guias onde é possível alterar as entradas XML do seu *deployment descriptor*.

Uma vez que todas as entradas XML da aplicação estejam configuradas para o ambiente operacional em questão, ela pode ser validada também na *Application Deplyoment Tool*, valendo-se dos testes de compatibilidade, também fornecidos pela especificação da plataforma J2EE (seção 3.2.2). Se a consistência das configurações contidas no *deployment descriptor* for confirmada, a aplicação J2EE pode então ser implantada no ambiente operacional escolhido. Essa situação pode ser observada na Fig. 5.6, nos painéis inferiores da janela: o servidor J2EE é referenciado como "*localhost*" e a aplicação implantada no mesmo é a "Intercession".

5.3. Conclusões do Capítulo

Uma análise do modelo de integração apresentado permite observar que sua implementação é ao mesmo tempo simples e versátil, o que pode ser verificado pelos procedimentos sistematizados nas seções 5.2.1, 5.2.2 e 5.2.3. Esses procedimentos, por sua vez, corroboram todos os princípios da proposta, expostos na seção 5.1.

Em resumo, a proposta para prover reflexão comportamental de propósito geral aos métodos de um componente de negócio EJB é implementada através da abordagem de Intercessão em Tempo de Implantação. Nela, um meta-componente EJB, com interfaces idênticas às do componente EJB base, é inserido na aplicação ao ser referenciado com o nome JDNI original do componente EJB base – redirecionamento realizado em tempo de implantação. Tal decisão visa minimizar significativamente todas as alterações necessárias na aplicação J2EE no momento de utilizar habilidades de intercessão na mesma.

Ainda, devido à escolha do modelo de programação multi-camada, a inserção do meta-componente não é percebida na ponta do cliente, pois todo o redirecionamento acontece na camada de serviços da aplicação corporativa.

Enfim, o conjunto de facilidades do modelo de integração oferece um meio efetivo para utilizar reflexão comportamental em aplicações pré-construídas da plataforma J2EE, visando flexibilidade de implementação nas situações onde é necessário modificar sua funcionalidade – como quando é preciso alterar de seu comportamento ou adaptá-la a novos requisitos. Para melhor visualizar os benefícios advindos do modelo, o Capítulo 6 propõe alguns projetos que se apoiam no mesmo.

CAPÍTULO 6 - EXEMPLOS DE APLICAÇÃO

O presente Capítulo apresenta dois exemplos de aplicação que baseiam sua implementação na proposta para utilização de reflexão computacional em aplicações da plataforma J2EE.

Com esse propósito, o Capítulo inicia com a seção 6.1, onde é apresentado um exemplo de utilização de intercessão com o objetivo de realizar a contabilização de chamadas a métodos de negócio. A seção 6.2 propõe um exemplo mais elaborado de utilização de intercessão, em uma aplicação que utiliza um contexto reflexivo para prover replicação.

O Capítulo é encerrado na seção 6.3, onde é feita uma analise das vantagens e desvantagens obtidas por esses exemplos de aplicação, em relação ao mecanismo de intercessão proposto.

6.1. Aplicação de Contabilização

O modelo de integração apresentado no Capítulo 5 permite de imediato a implementação de uma aplicação que realiza a contabilização das ações executadas sobre um componente de negócio, inserindo assim funcionalidade de controle na aplicação, sem alterar – ou mesmo afetar – os algoritmos que definem sua funcionalidade básica.

Com base no mecanismo de intercessão proposto na seção 5.1, esta Aplicação de Contabilização é implementada pelos seguintes componentes:

- Componentes Web – *View* do MVC
  - **Calculos.jsp**: página JSP que promove interação com o usuário, permitindo as chamadas aos métodos de negócio da aplicação (funcionalidade básica).
  - **Viewer.jsp**: página JSP que promove interação como o usuário, permitindo consulta à contabilização feita sobre os métodos de negócio (funcionalidade de controle).

- Componentes Web – o *Controller* do MVC
  - **JBonusCredBean**: JavaBean que interpreta as ações do usuário enviadas pelo *View* "Calculos.jsp", mapeando-as em ações a serem realizadas pelo *Model* – nesse caso, faz referência ao meta-componente, auxiliando assim a prover intercessão.
  - **JViewerBean**: JavaBean que interpreta as ações do usuário enviadas pelo *View* "Viewer.jsp", mapeando-as em ações a serem realizadas pelo *Model* – nesse caso, faz referência ao meta-componente auxiliar para consulta à contabilização.
- Componentes EJB: o *Model* do MVC
  - **Beans::CalcEJB**: componente de negócio base
  - **BeansMeta::CalcEJB**: meta-componente que realiza intercessão
  - **LogPack::LogEJB**: meta-componente auxiliar para registro da contabilização em uma base de dados

A modelagem da Aplicação de Contabilização, exibida nos diagramas UML (*Unified Modeling Language*) da subseção a seguir, é fornecida com o objetivo apresentar sua visualização formal e gráfica, fornecer um guia para sua construção, e documentar as principais decisões tomadas para sua implementação. A codificação completa da aplicação encontra-se nas seções 1.1. a 1.6, do "Anexo A – Código do Exemplo: Aplicação de Contabilização" deste documento.

6.1.1. Modelagem UML

Genericamente, os modelos são construídos para comunicar a estrutura e o comportamento desejado de sistemas de informação, auxiliando a ilustrar e controlar sua arquitetura (melhor compreensão / reaproveitamento). A UML, ou Linguagem de Modelagem Unificada, é uma linguagem padrão para elaboração de estruturas de software. Pode ser empregada para a visualização, a especificação e a documentação de artefatos[6] de um sistema de software [BOOCH et al., 2000].

Um diagrama é uma apresentação gráfica de um modelo, ou conjunto de elementos. A UML admite nove diagramas, sendo que cada um exibe uma diferente perspectiva do sistema: diagrama de classes, de objetos, de casos de uso, de

[6] Artefato é um conjunto de informações utilizado ou produzido por um processo de desenvolvimento de software [BOOCH et al., 2000].

seqüências, de colaborações, de estados, de atividades, de componentes e de implantação.

Para a Aplicação de Contabilização, são necessários apenas os diagramas de classes, de seqüência e de componentes para retratar de forma satisfatória suas especificidades relevantes.

- **Diagramas de Classe**

A Fig. 6.1 apresenta o diagrama de classes, em alto nível de abstração, da Aplicação de Contabilização. Nele, é enfatizado o relacionamento de dependência entre os pacotes da aplicação – que definem os *namespaces* dos componentes –, bem como seu conteúdo público. Os JavaBeans "JBonusCredBean" e "JViewerBean" não estão agrupados em pacotes, e são representados por suas classes nesse diagrama.

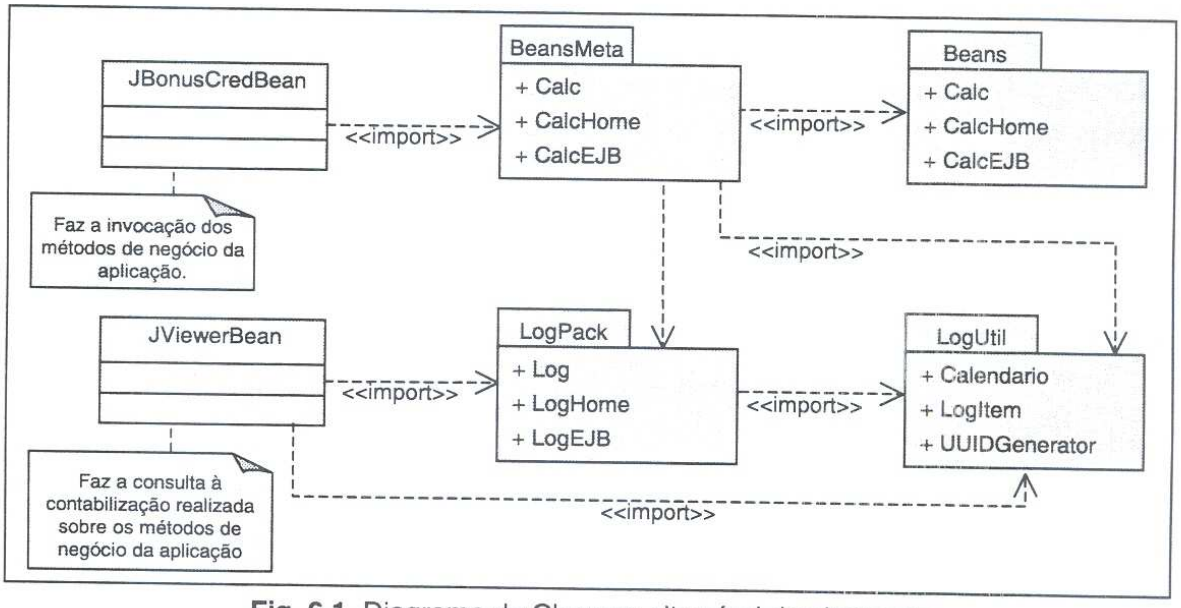

Fig. 6.1. Diagrama de Classes: alto nível de abstração.

A Fig. 6.2 apresenta um diagrama de classes mais detalhado da Aplicação de Contabilização, enfatizando não só o relacionamento entre seus pacotes, mas também o relacionamento entre as classes e interfaces que os compõem, com seus respectivos métodos e propriedades.

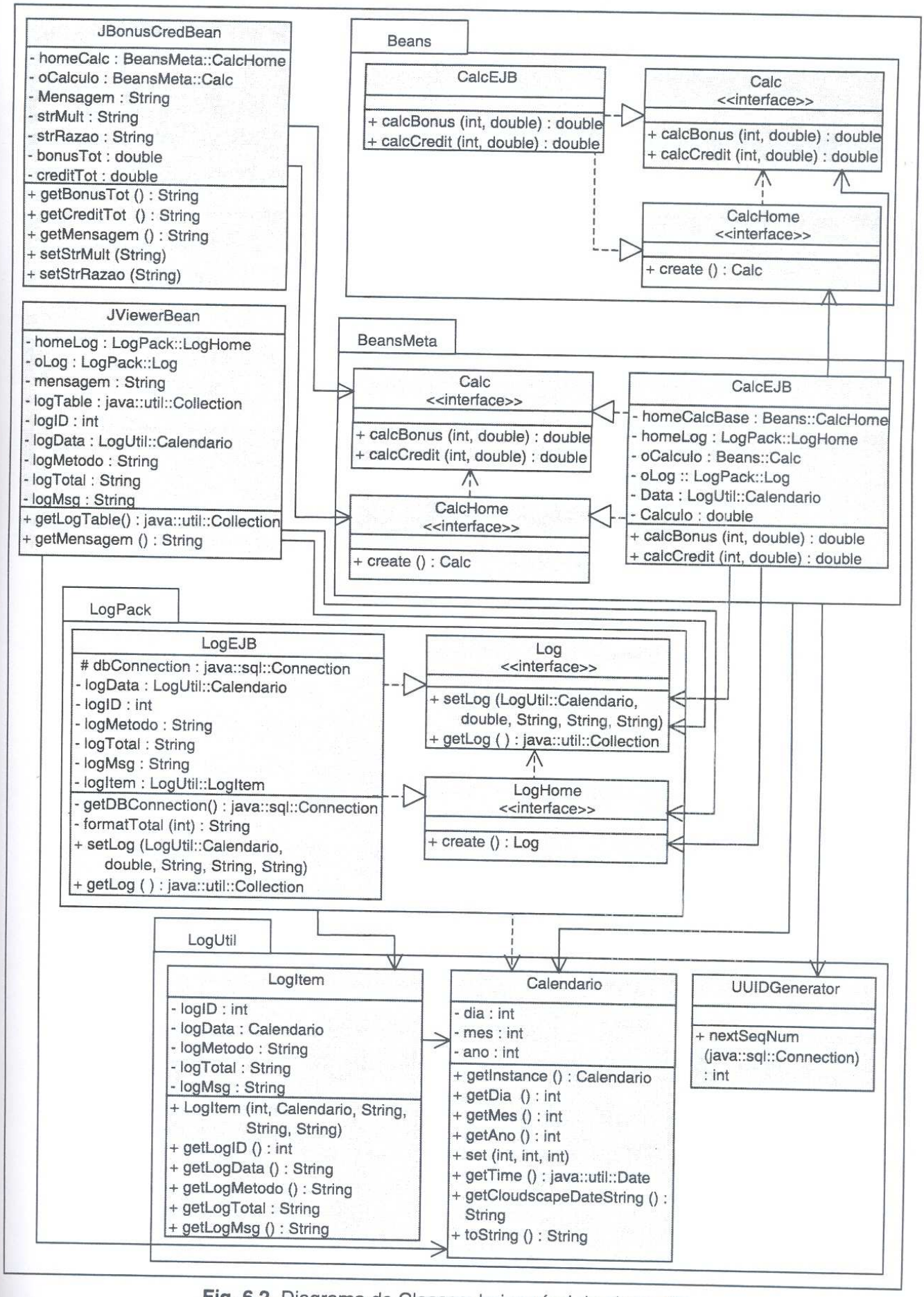

Fig. 6.2. *Diagrama de Classes: baixo nível de abstração.*

➢ **Diagramas de Interação**

A Fig. 6.3 apresenta o diagrama de interações (diagrama de seqüência) realizadas quando da chamada aos métodos de negócio – contidos no componente base "Beans::CalcEJB" da Aplicação de Contabilização. A intercessão é efetivada pelo meta-componente "BeansMeta::CalcEJB", invocado por JavaBean "JBonusCredBean". Dessa forma, é inserido o mecanismo de contabilização, cujas informações são manipuladas pelo meta-componente auxiliar "LogPack::LogEJB" (faz a interação com um banco de dados, referenciado na aplicação via nome JNDI "jdbc/LogDB").

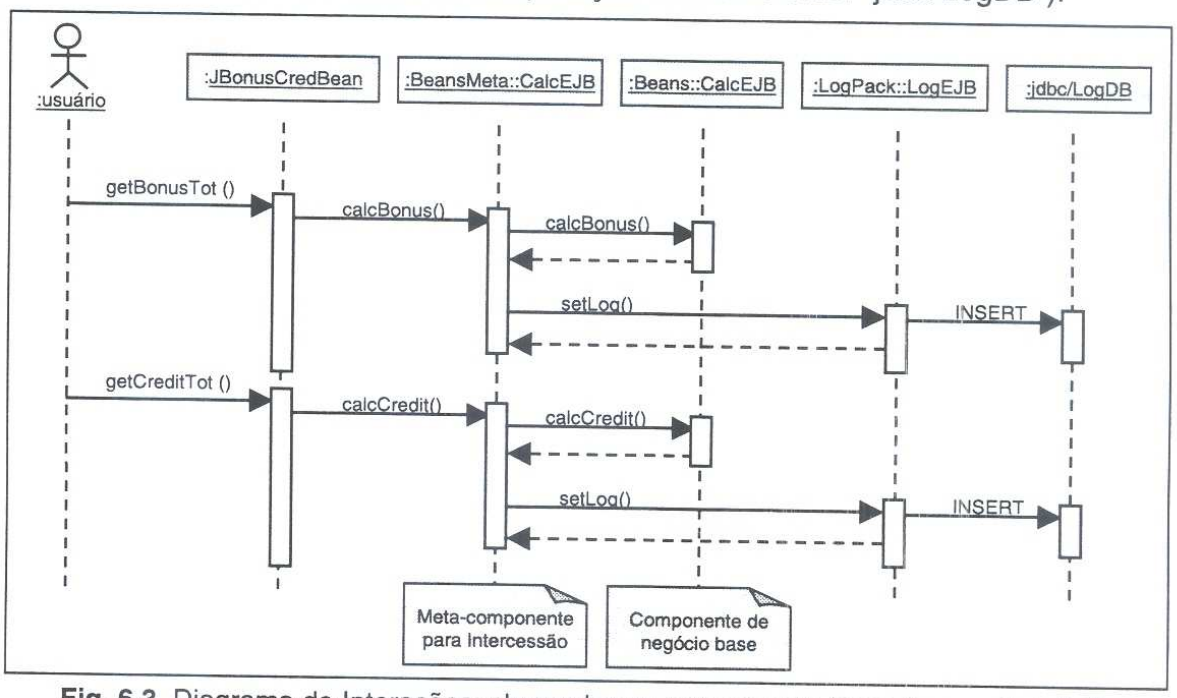

Fig. 6.3. Diagrama de Interações: chamadas ao componente base Beans::CalcEJB.

A Fig. 6.4 apresenta o diagrama de interações realizadas para a visualização da contabilização feita sobre os métodos de negócio do componente base "Beans::CalcEJB". Nesse caso, apenas é preciso invocar a funcionalidade de controle provida pelo meta-componente auxiliar "LogPack::LogEJB".

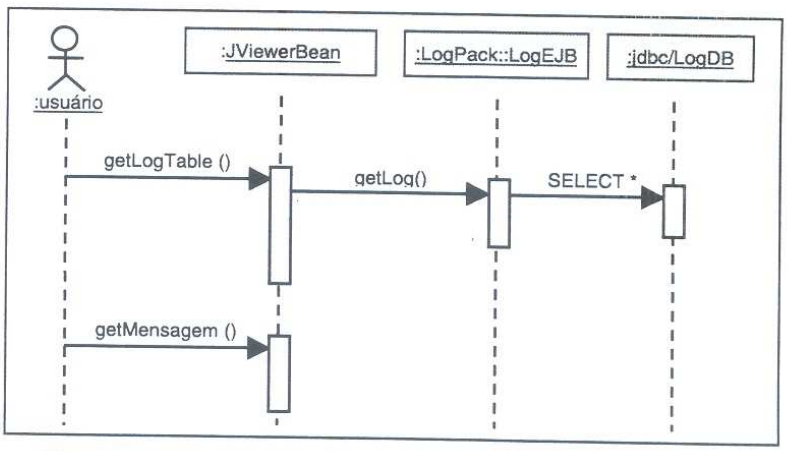

Fig. 6.4. Diagrama de Interações: consulta à contabilização.

➢ **Diagrama de Componentes**

A Fig. 6.5 apresenta o diagrama de componentes da Aplicação de Contabilização. Nele, é enfatizada a construção física da aplicação, indicando como a construção de seu executável está definida.

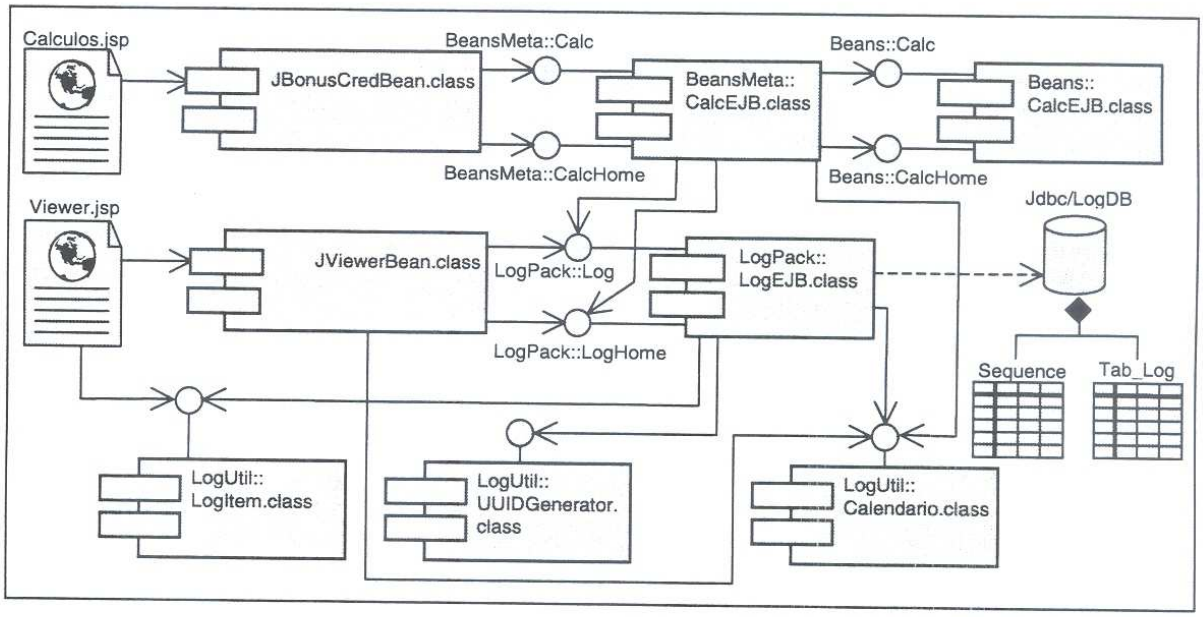

Fig. 6.5. Diagrama de Componentes da Aplicação de Contabilização.

6.1.2. Montagem via Ferramenta

A construção da Aplicação de Contabilização foi realizada com auxílio de ferramenta de implantação, a *Application Deployment Tool*, embutida na J2EE SDK (seção 3.2.3). Ela cria automaticamente todos os descritores de implantação (*deployment descriptors*, detalhados na seção 1.7 do Anexo A deste documento), bem como instala a aplicação no servidor J2EE – o servidor fornecido com a J2EE SDK.

A Fig. 6.6 ilustra como a ferramenta de implantação auxilia a incluir na aplicação os nomes JNDI previamente definidos. Ainda na Fig. 6.6, os campos "JNDI Name" são atualizados nos arquivos XML de *deployment descriptors* da aplicação, no momento em que a mesma é salva em um arquivo **.ear** (seção 3.2.7). Nesse caso, o arquivo **.ear** da Aplicação de Contabilização chama-se "BonusCredEV.ear", e o nome pelo qual ele é referenciado na ferramenta é "BonusCredEV" – como ilustrado na barra de título da Fig. 6.6.

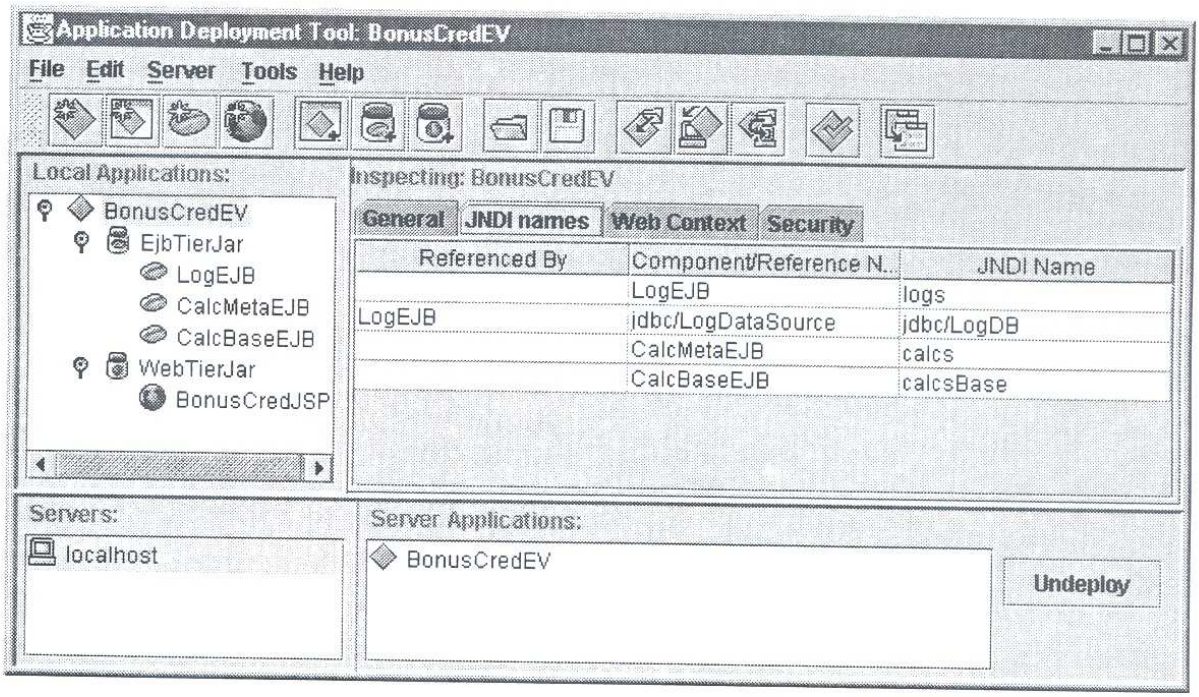

Fig. 6.6. Configuração dos nomes JNDI da Aplicação de Contabilização.

É portanto através da *Application Deployment Tool* que a intercessão em tempo de implantação é efetivada, como o exibido na Fig. 6.6: a introdução do meta-componente é realizada via redirecionamento entre componentes EJB pela alteração de seus nomes JNDI – informação mantida no *deployment descriptor*.

Uma aplicação corporativa J2EE armazena seus componentes Web em um arquivo **.war**, e os mesmos podem ser acessados via browser. Para tanto, deve ser configurado o contexto Web da aplicação, associado diretamente com esse arquivo **.war**. Assim, pela Fig. 6.7, o contexto Web da Aplicação de Contabilização está associado ao arquivo "WebTierJar", este definido no campo "WAR File". O contexto Web raiz da aplicação, por sua vez, está definido no campo "Context Root", cujo valor é "BonusCred".

Isso significa que para acessar a Aplicação de Contabilização via Web o browser aponta para os endereços http://localhost:8000/BonusCred/Calculos.jsp (para acessar seus métodos de negócio) e http://localhost:8000/BonusCred/Viewer.jsp (para visualizar o controle realizado sobre a aplicação, feita pela contabilização implementada em nível meta).

Cabe uma observação em relação ao serviço Web oferecido pela J2EE SDK: ele utiliza a porta 8000 em sua configuração default, e é uma implementação de referência, mais conhecida como TomCat. Maiores detalhes sobre o mesmo podem ser encontrados no site http://java.sun.com/products/jsp/tomcat, ou diretamente em http://jakarta.apache.org/tomcat.

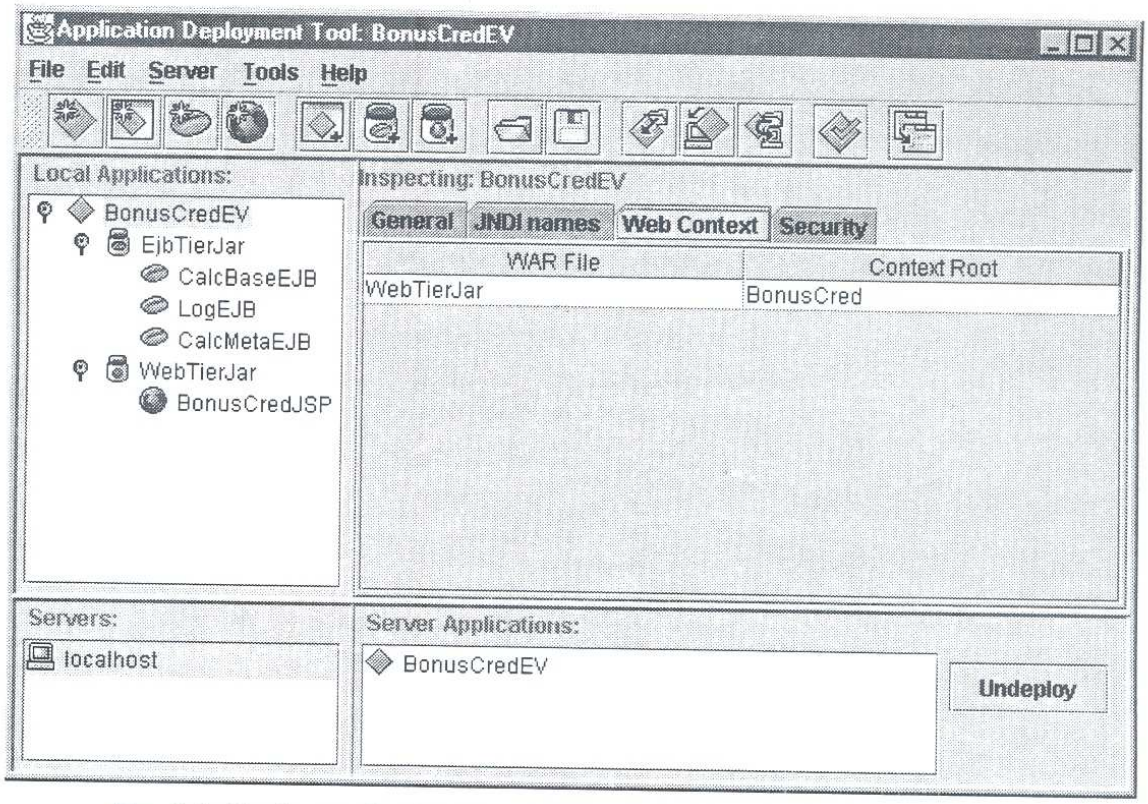

Fig. 6.7. Configuração do contexto Web da Aplicação de Contabilização.

A Fig. 6.8 apresenta os detalhes de implantação do componente "Beans::CalcEJB": é referenciado na ferramenta como "CalcBaseEJB" (comparar com a Fig. 6.6), está contido no arquivo **.jar**, cujo nome é "EjbTierJar", e é um EJB do tipo *session stateless* (seção 3.1.5).

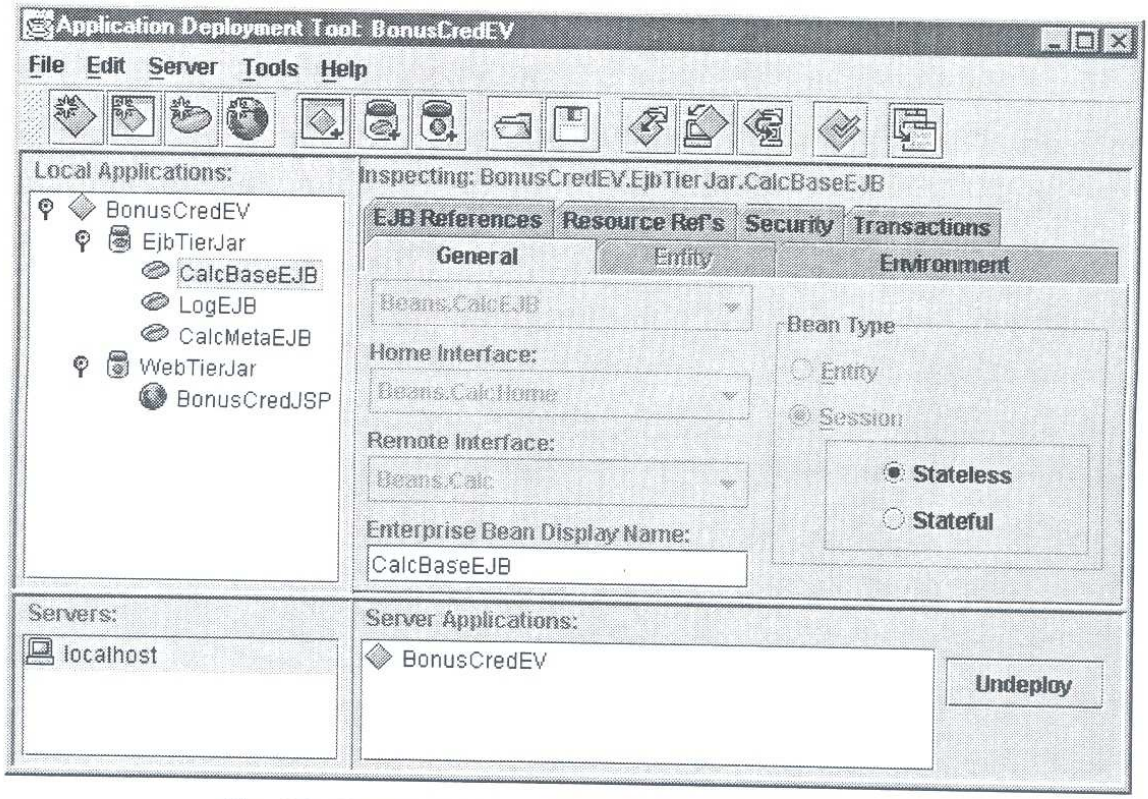

Fig. 6.8. Componente de Negócio Base: CalcBaseEJB.

A Fig. 6.9 e a Fig. 6.10 apresentam os detalhes de implantação dos EJBs "BeansMeta::CalcEJB" e "LogPack::LogEJB": são referenciados como "CalcMetaEJB" e "LogEJB" (comparar com a Fig. 6.6), estão contidos no "EjbTierJar", e são do tipo *session stateless*.

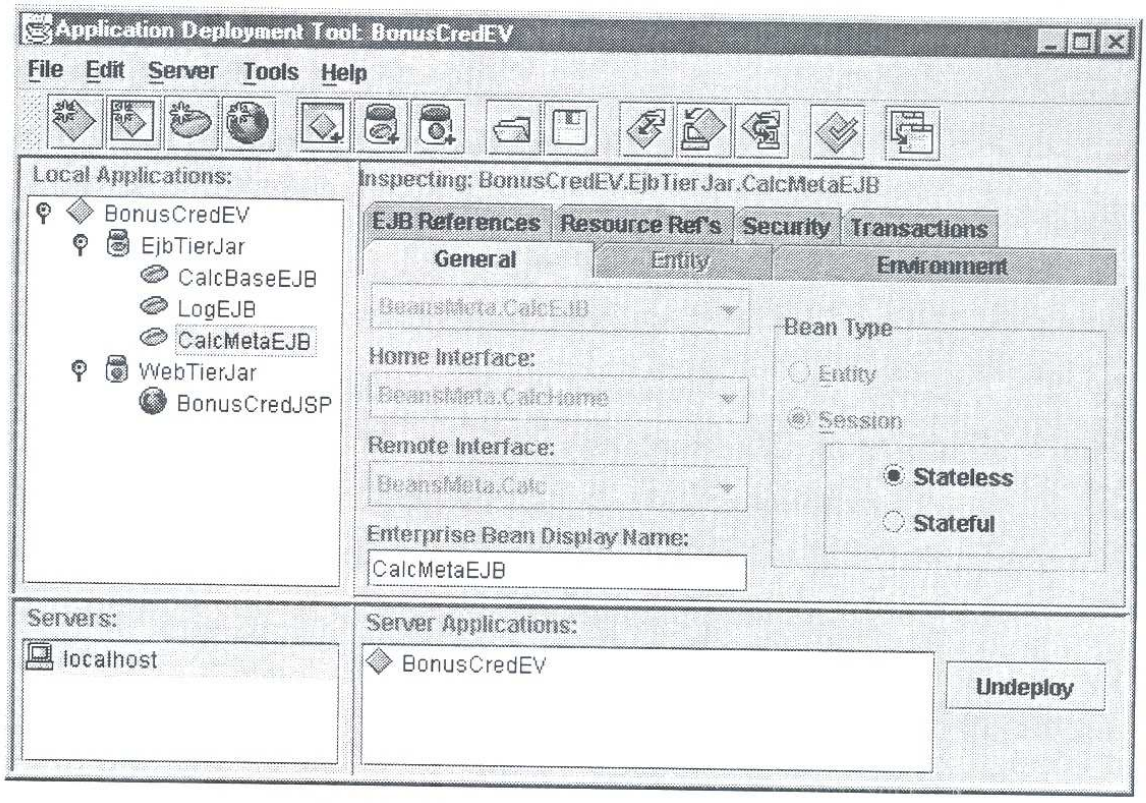

Fig. 6.9. Meta-componente: CalcMetaEJB.

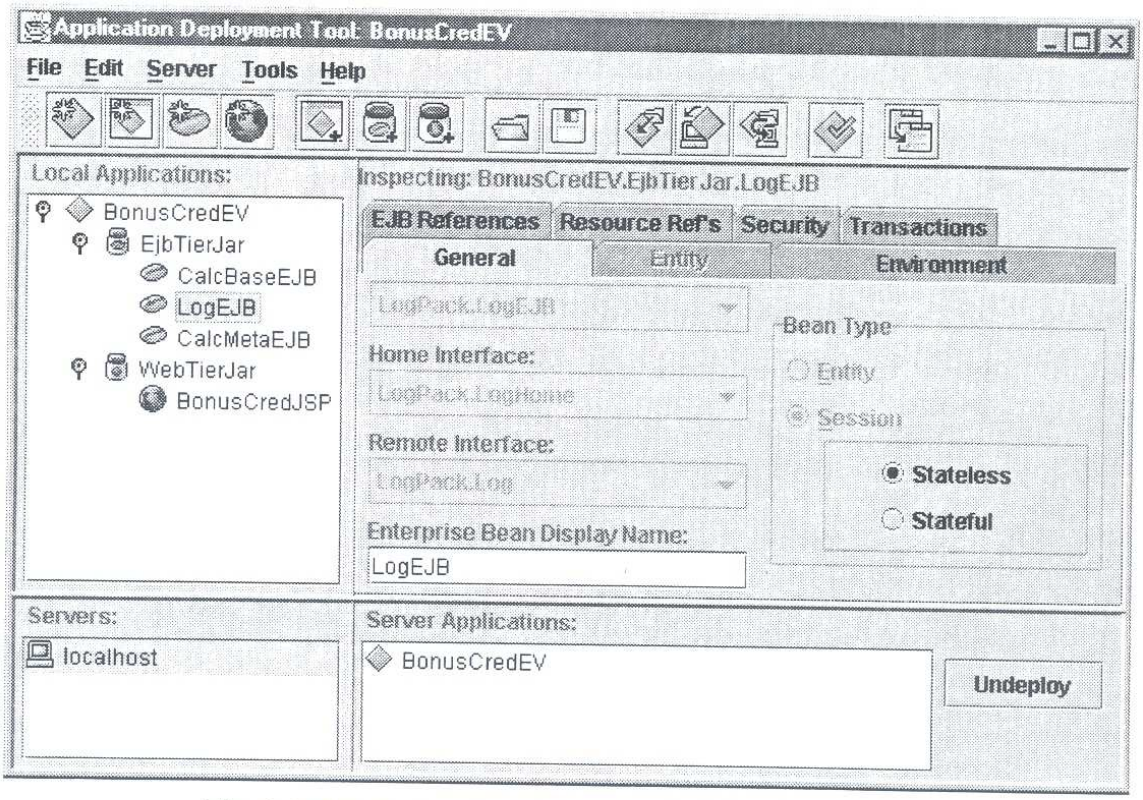

Fig. 6.10. Meta-componente para contabilização: LogEJB.

6.1.3. Interfaces para Usuário

De acordo com o modelo de programação da J2EE (seção 3.2.6) adotado pela proposta de intercessão, a interação com o cliente é feita através de páginas JSP. Assim, para acessar a Aplicação de Contabilização, são oferecidas dois tipos de interface para usuário: (1) uma para acesso aos métodos de negócio, sua funcionalidade básica, e (2) outra para acesso às informações de contabilização da aplicação, que provêem as funcionalidades de controle.

A Fig. 6.11 apresenta a interface para as funcionalidades básicas da aplicação, implementada pela página "Calculos.jsp". Seus detalhes de codificação são exibidos na seção 1.5.1 do Anexo A deste documento.

É interessante relembrar que o *Controller* do modelo – no caso o componente Web "JBonusCredBean.class", cujo código encontra-se na seção 1.5.2 do Anexo A – deve fazer referência ao *namespace* do meta-componente ("`import package BeansMeta.*;`") para efetivar a intercessão.

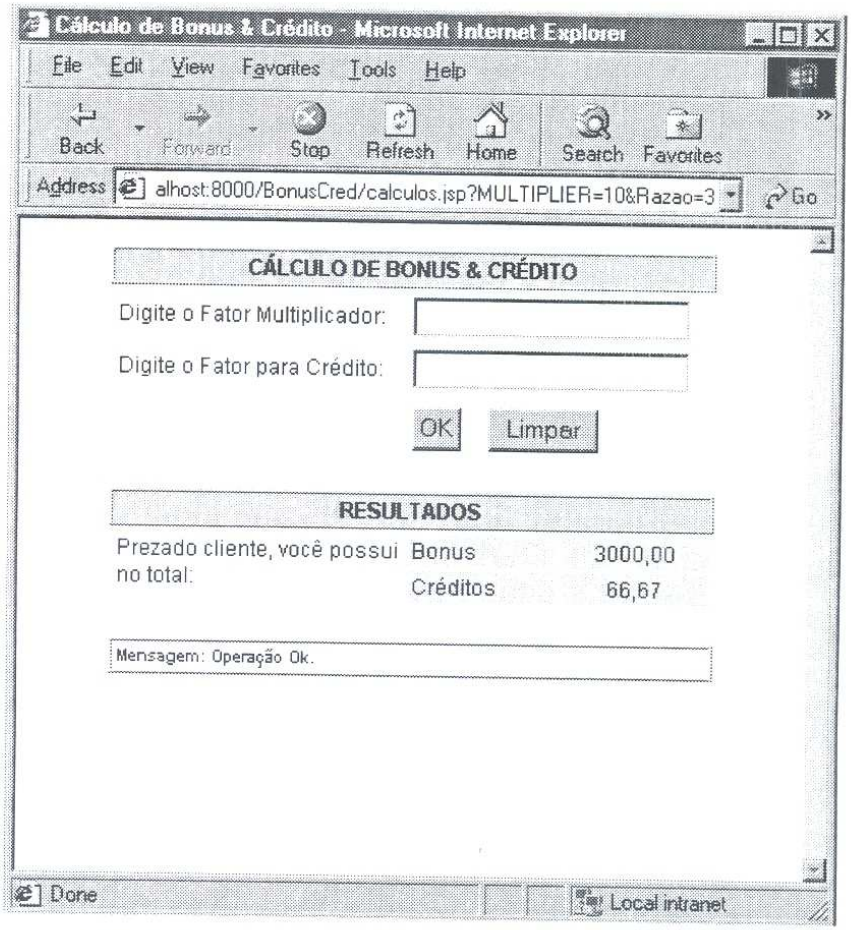

Fig. 6.11. Interface do Cliente: página "Calculos.jsp".

A fig. 6.12 apresenta a interface para as funcionalidades de controle da aplicação, implementada pela página "Viewer.jsp". Seus detalhes de codificação também podem ser encontrados na seção 1.6.1 do Anexo A deste documento.

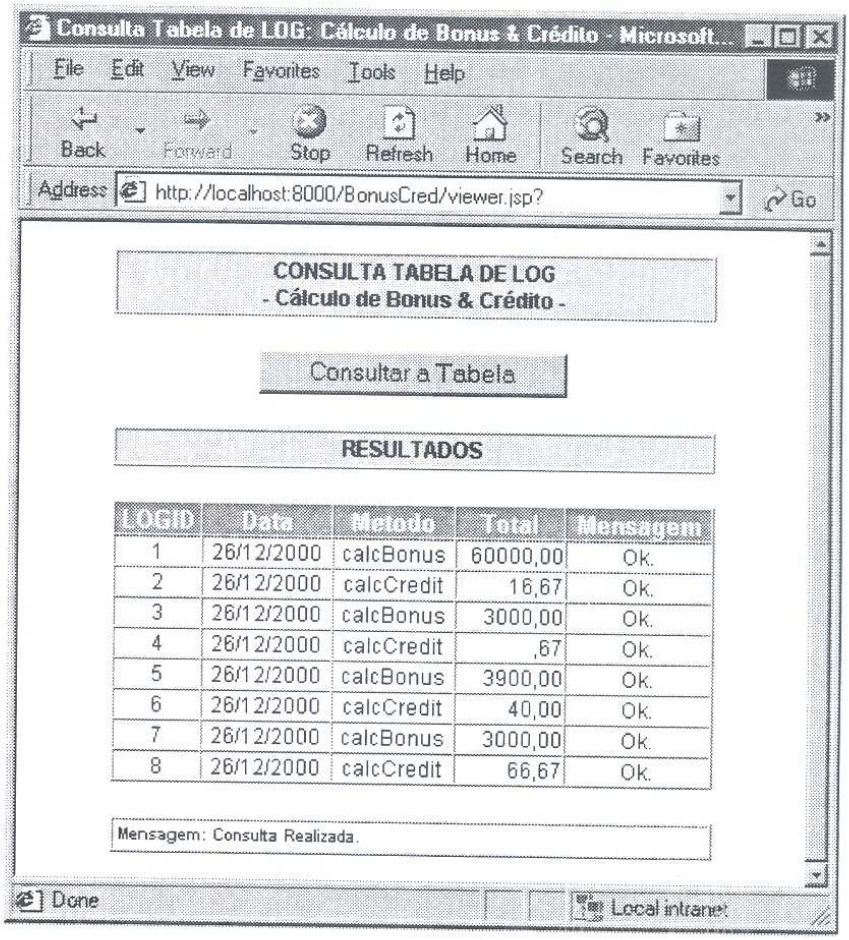

Fig. 6.12. Interface do Cliente: página "Viewer.jsp".

6.1.4. Desempenho

Todo mecanismo de reflexão inflige uma certa latência às aplicações, devido ao desvio do fluxo da computação realizado pelo nível meta.

Assim, a partir de exemplos de aplicação, é possível estimar o quanto o mecanismo de reflexão proposto – Intercessão em Tempo de Implantação – afeta o desempenho da aplicação corporativa J2EE. Ou seja, é possível avaliar o quão custoso é desacoplar a lógica de controle da lógica funcional, com o objetivo de obter maior modularidade e facilidade de manutenção. Para tanto, foram consideradas as seguintes situações:

(1) Chamada Simples a um Componente EJB **com** Intercessão em Tempo de Implantação (Chamada Simples COM ITI) – um meta-componente se interpõe entre a chamada do cliente e o componente de negócio base;

(2) Chamada Simples a um Componente EJB **sem** Intercessão em Tempo de Implantação (Chamada Simples SEM ITI) – a chamada ao componente de negócio base é direta;

(3) Aplicação de Contabilização **com** Intercessão em Tempo de Implantação (Aplic. Contabiliz. COM ITI) – um meta-componente separa a lógica de controle (contabilização) da lógica da aplicação; e

(4) Aplicação de Contabilização **sem** Intercessão em Tempo de Implantação (Aplic. Contabiliz. SEM ITI) – o próprio componente base se encarrega também de fazer a contabilização, estendo sua funcionalidade sem prover modularização.

Vale a pena destacar que estas comparações forçam uma situação a princípio incompatível com a proposta da dissertação, que concentra-se em prover intercessão em componentes encapsulados – logo não permitiria a extensão do código desses componentes para inclusão de lógica de controle. Contudo, essa avaliação é possível – pois encontra-se em ambiente experimental e controlado – e válida, no sentido de fornecer uma noção do custo do desacoplamento de funcionalidade.

Prosseguindo, as situações consideradas em (1) e em (2) objetivam prover um parâmetro do custo de desacoplamento o mais independente possível de aplicação. Ou seja, o custo apenas em relação ao uso ou não do desvio de fluxo de computação por um meta-componente. Pode-se associar a situação (1) com a Fig. 5.2, e a situação (2) com a Fig. 5.1, onde estão representadas graficamente, e de forma genérica, a utilização ou não da proposta de Intercessão em Tempo de Implantação.

A situação (3) é a própria implementação da Aplicação de Contabilização, documentada na seção 6.1.1. E por fim, na situação (4), a Aplicação de Contabilização tem sua lógica funcional e de controle contidas em um único componente EJB – este executa os métodos de negócio e gerencia informação de contabilização. Observa-se que ambas as situações (3) e (4) se valem do componente "LogPack::LogEJB" para armazenamento e recuperação da informação de controle. Portanto, a diferença de desempenho entre (3) e (4) deve-se apenas ao desvio provocado pela adição de um nível meta, ou seja, pela adição de mais um EJB na aplicação.

Em todos os casos (1), (2), (3) e (4), a tomada de tempo (milisegundos) das amostragens foi realizada na página JSP da aplicação corporativa, que de fato invoca o componente EJB base. Ainda, nesse experimento o tempo de resposta não incluiu tráfego em rede, uma vez que o serviço Web (componentes Web), o serviço J2EE (componentes EJB) e o serviço de nomes (acessado via JNDI) sempre estão no

mesmo equipamento. Inclusive, para melhor avaliar a influência de equipamento no custo do desacoplamento, foram utilizadas duas diferentes máquinas, onde todas as quatro situações consideradas foram experimentadas:

- Equipamento **A**: Pentium III 736MHz, 128Mbytes de RAM, Windows 2000;
- Equipamento **B**: Pentium I 233MHz, 128Mbytes de RAM, Windows NT 4.0.

A Fig. 6.13 ilustra uma estimativa do desempenho das situações (1), (2), (3) e (4) no Equipamento **A**. Em cada uma das quatro situações, foram realizadas entre 400 e 500 amostras de chamadas ao componente EJB base, para obtenção de seu tempo de resposta médio.

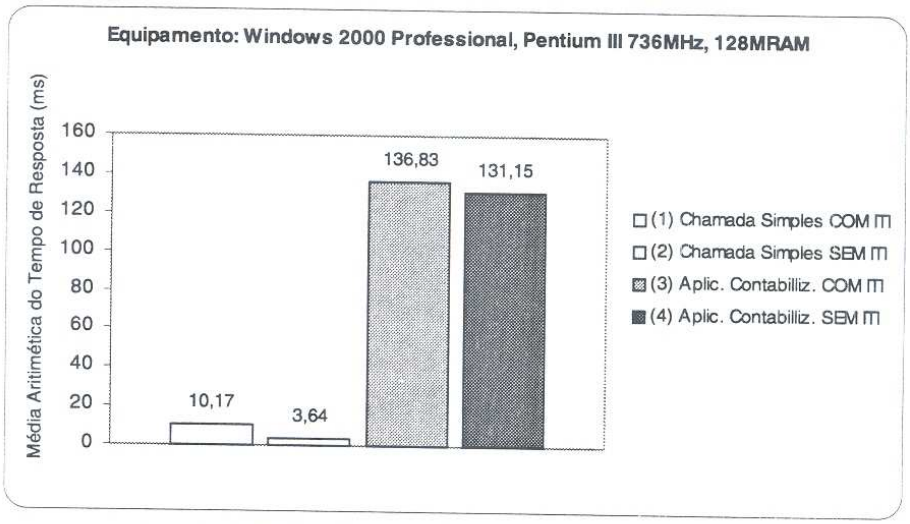

Fig. 6.13. Estimativa de Desempenho: Equipamento **A**.

A Fig. 6.14, por sua vez, faz o mesmo experimento da Fig. 6.13, porém no Equipamento **B**.

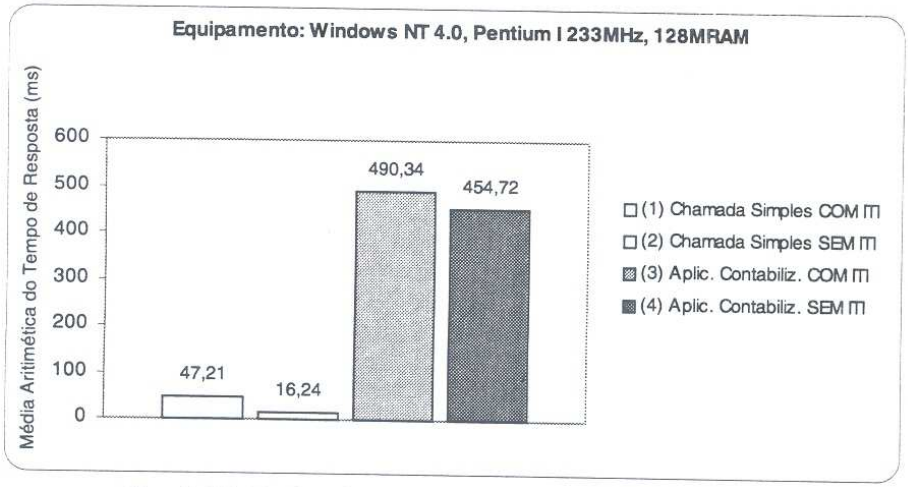

Fig. 6.14. Estimativa de Desempenho: Equipamento **B**.

Constata-se, pelas Fig. 6.13 e Fig. 6.14, que existe um valor mais ou menos constante que é acrescido ao tempo de resposta sempre que a Intercessão em Tempo de Implantação (ITI) é utilizada na aplicação. A diferença para a Aplicação de Contabilização e para a Chamada Simples, em cada equipamento experimentado, é exibida na Tab. 6.1 a seguir.

Win NT 4.0 - Pentium I 233MHz – 128Mbyes RAM	
Diferença do Tempo de Resposta: Aplic. Contabiliz. (Com ITI – Sem ITI) =	**35,62** ms
Diferença do Tempo de Resposta: Chamada Simples (Com ITI – Sem ITI) =	**30,97** ms
Win 2000 - Pentium III 736MHz – 128Mbytes RAM	
Diferença do Tempo de Resposta: Aplic. Contabiliz. (Com ITI – Sem ITI) =	**5,68** ms
Diferença do Tempo de Resposta: Chamada Simples (Com ITI – Sem ITI) =	**6,53** ms

Tab. 6.1. Estimativa: Custo do Desvio do Fluxo de Computação.

Logo, o desvio devido à inserção de um meta-componente na aplicação (uso a Intercessão em Tempo de Implantação), aumenta o tempo de resposta em um valor constante, dependente de equipamento e de sistema operacional. Esse acréscimo corresponde à localização de um novo componente (utilização do serviço de nome), à criação do mesmo dentro do contexto da aplicação, e a mais uma manipulação de parâmetros. Nos experimento realizados, o custo no Equipamento **A** ficou em torno de 6 ms, e no Equipamento **B**, em torno de pouco mais de 33 ms.

É importante ressaltar que esta diferença estima *apenas o tempo de resposta entre componentes Web e componente EJB*. Não inclui o tempo de rede da ponta cliente (o browser), nem o processamento de página HTML dinâmica para o cliente, e nem o processamento de compilação JIT da página JSP – que é transformada em classe no primeiro acesso à aplicação multi-camada J2EE.

Concluindo, pode-se afirmar que o custo de desacoplamento da lógica de controle (uso de Intercessão em Tempo de Implantação) acresce em um valor constante o tempo de resposta interno de chamada a componentes EJB, em relação à mesma solução sem a separação de funcionalidades. Esse é um preço pago pela flexibilização de implementação objetivada pela proposta. Para aplicações um pouco mais complexas, como o caso da Aplicação de Contabilização, trata-se de uma diferença proporcionalmente muito pequena em relação ao tempo de resposta total, e não é perceptível pela ponta cliente. A capacidade de processamento, os tempos de rede, de gravação em disco e de processamento de página dinâmica são muito mais relevantes. Logo, os ganhos com a separação de interesses proporcionada pela

proposta de intercessão apresentam-se muito mais interessantes, pois favorecem futuras manipulações na aplicação, bem como melhor modularização e legibilidade.

6.1.5. Considerações

O exemplo de Aplicação de Contabilização apresentado é de fato bem simples, sendo sua principal pretensão a de ilustrar, de forma detalhada, as perspectivas de utilização oferecidas por essa proposta para intercessão. Daí a preocupação em fornecer não apenas o código fonte para sua implementação, mas também as diretivas de implantação da aplicação via ferramenta. Assim também pretende-se evidenciar mais concretamente, através do exemplo implementado, as vantagens de montagem de aplicação oferecidas pela plataforma J2EE – um dos seus grandes atrativos.

Com esse intuito, foi desenvolvido o "Anexo A – Código do Exemplo: Aplicação de Contabilização", já citado algumas vezes ao longo desta seção 6.1, e que também inclui, em sua seção 1.8, os passos a serem realizados para execução da aplicação: instalação de produtos, configuração de ambiente e diretivas de implantação.

Ainda, para o armazenamento e manipulação das informações de contabilização, foi adotada JDBC, visando adequação às restrições de programação para EJBs (seção 3.1.9). Por esse motivo, foi utilizado o banco de dados Cloudscape, fornecido junto com a J2EE SDK, sendo que sua configuração para correta atuação na Aplicação de Contabilização também encontra-se no Anexo A, em sua seção 1.9.

6.2. Aplicação de Replicação

Esta seção propõe um exemplo mais elaborado para utilização do modelo de integração visto no Capítulo 5. Para tal, apresenta um modelo de aplicação que utiliza o mecanismo de intercessão proposto para a J2EE com o objetivo de prover replicação.

6.2.1. Replicação

O esforço para construir sistemas distribuídos tolerantes a falhas[7] é bem antigo. De fato, um problema inerente a esse tipo de sistema é sua potencial vulnerabilidade a falhas, pois, se um único nó sofrer um *crash*[8], a disponibilidade de todo o sistema pode ser comprometida. Contudo, a própria natureza distribuída desse

---

[7] É o desvio de um serviço entregue, mediante à especificação do sistema [LAPRIE, 1990].
[8] Tipo de falha onde um elemento do sistema não responde a mais nenhuma requisição do cliente [LAU, 1996].

tipo de sistema também provê os meios para aumentar sua confiabilidade e disponibilidade: a distribuição permite introduzir redundância, que torna o sistema como um todo mais confiável que suas partes individuais [DÉFANO et al., 1998].

A redundância, por sua vez, geralmente é implementada através de técnicas de replicação. Embora a replicação seja uma idéia intuitiva e prontamente compreendida, sua implementação não é trivial.

Assim, são as técnicas de replicação que fornecem uma alternativa para que os serviços em um sistema de informação continuem a ser fornecidos mesmo na ocorrência de falhas, provendo portanto confiabilidade e disponibilidade. Para isso, as unidades de replicação, denominadas réplicas, são distribuídas em diversos pontos da rede, e sua coordenação define protocolos que asseguram controle de concorrência, consistência e transparência do conjunto, e recuperação em caso de falha. Isso habilita a noção de serviço abstrato para o cliente, pois o que é visto como um único serviço, na realidade é um grupo de serviços replicados [LAU, 1996].

Existem várias técnicas de replicação, como descrito nas subseções a seguir, que mascaram falhas individuais das réplicas membros de um conjunto de serviços replicados; cada uma provê um diferente protocolo de coordenação de réplicas.

- **Replicação Ativa**

Na replicação ativa, ou abordagem de máquina de estados, cada réplica manipula a requisição recebida do cliente e envia uma resposta. Em outras palavras, as réplicas comportam-se independentemente e a técnica consiste em garantir que todas as réplicas recebam as requisições na mesma ordem. Essa técnica (Fig. 6.15) é apreciada porque seu tempo de resposta é baixo, mesmo na ocorrência de um *crash*.

A técnica tem, contudo, duas importantes desvantagens: (1) a redundância do processamento implica em um alto uso de recursos, e mais importante (2) as requisições têm que ser manipuladas de maneira determinística – caso contrário, pode haver inconsistência de estado entre as réplicas. Determinismo significa que o resultado da operação depende apenas do estado inicial da réplica e da seqüência de operações que já foram executadas.

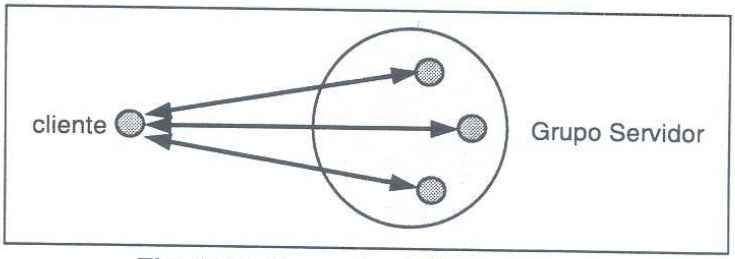

Fig. 6.15. Grupo de réplicas ativas.

➢ Replicação Passiva

Na replicação passiva, ou replicação primário-*backup*, uma das réplicas – chamada primário – se encarrega de receber todas as requisições dos clientes e retornar as respostas. As réplicas *backups* interagem apenas com o primário, recebendo mensagens de atualização de estado (*checkpoints*), o que garante a consistência do grupo. Essa técnica de replicação é bastante útil, uma vez que requer menos poder de processamento que a replicação ativa, e não faz nenhuma suposição em relação ao determinismo do processamento de uma requisição. Contudo, a implementação da replicação passiva requer um mecanismo para concordância com a réplica primária – como um serviço *membership*[9] [DÉFANO et al., 1998]. Se a réplica primária falha, um dos *backups* toma seu lugar. Se a mesma cai antes de enviar a resposta para o cliente, o cliente sofrerá *timeout*. O cliente deve então aprender a identidade da nova réplica primária e reenviar a requisição. Isso leva a um aumento do tempo de resposta no caso de falha, que o torna inadequado no contexto de aplicações de tempo crítico. Em suma, a replicação passiva, ilustrada na Fig. 6.16, não mascara totalmente as falhas para o cliente, porém o fato do cliente comunicar-se apenas com a réplica primária simplifica a interação cliente/servidor.

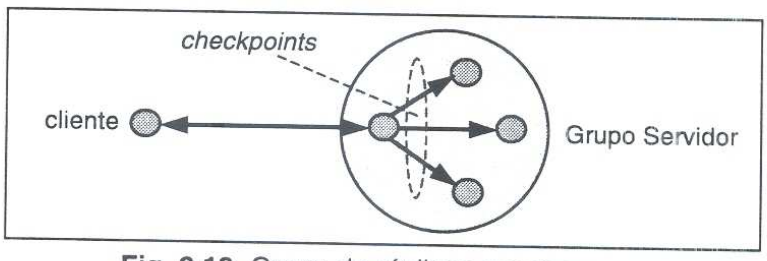

Fig. 6.16. Grupo de réplicas passivas.

➢ Replicação Semi-Ativa

A replicação semi-ativa foi introduzida para contornar o problema do não-determinismo da replicação ativa, no contexto de aplicações de tempo crítico. É baseada na replicação ativa e estendida com a noção de *líderes* e *seguidores*. Enquanto o processamento normal de uma requisição é executado por todas as réplicas, é responsabilidade do líder executar as partes não determinísticas do processamento e informá-las aos seguidores.

---

[9] É um serviço usado para manter a lista de membros dos grupos no sistema; manipula requisições explícitas dos processos membros para se juntar e para deixar um grupo; também remove membros suspeitos [DÉFANO et al., 1998].

➢ **Replicação Semi-Passiva**

A replicação semi-passiva tem similaridades com a replicação passiva. Em ambas, existe uma única réplica (o primário na replicação passiva) que (1) processa uma requisição recebida de um cliente, e (2) atualiza os *backups* depois de processar cada requisição. Pelo fato de uma requisição ser manipulada apenas por uma única réplica, o processamento de requisições pode ser não determinístico. A principal diferença entre as duas técnicas de replicação é que na replicação semi-passiva, a seleção da réplica que processará as requisições de clientes é baseada no paradigma do "coordenador rotativo"[10], no qual a decisão do consenso leva à atualização do valor de estado das demais réplicas – valor de estado obtido pelo processamento da requisição de cliente feito pela réplica coordenadora do *round* (ciclo) em questão [DÉFANO et al., 1998].

Na replicação semi-passiva, as seguintes características da replicação passiva são mantidas: (1) na ausência de *crash*, ou suspeita de falha, a requisição é manipulada por uma única réplica, e (2) o processamento de uma requisição não necessita ser determinístico – uma vez que a réplica que manipula a requisição do cliente impõe seu estado sobre as demais.

Uma diferença menor entre a replicação passiva e a replicação semi-passiva é que, na replicação semi-passiva, o cliente envia sua requisição para todas as réplicas e cada réplica envia uma resposta de volta ao cliente. Portanto, não há necessidade de o cliente conhecer o primário, nem de o cliente ter *timeouts* para detectar o *crash* no primário (e reenviar a requisição): similarmente à replicação ativa, na replicação semi-passiva os efeitos de falhas são completamente mascarados para o cliente [DÉFANO et al., 1998].

6.2.2. Replicação no Contexto Reflexivo

Uma das maneiras de implementar técnicas de replicação é através da utilização da reflexão computacional. Para tanto, o nível base da aplicação fica com a parte funcional da mesma – o serviço a ser replicado – e o nível meta com um determinado protocolo de coordenação entre as réplicas – replicação ativa ou passiva, por exemplo. Isso permite flexibilidade de implementação, pois alterar os protocolos do nível meta não afeta a implementação dos algoritmos da aplicação do nível base.

---

[10] No paradigma do coordenador rotativo, o algoritmo passa por uma sucessão de *rounds*, e em cada *round*, um processo diferente é o coordenador. Mais detalhes no Anexo B desse documento.

Assim, os detalhes de como a replicação é implementada são escondidos no nível meta e não aparecem no código da aplicação em si [FABRE et al., 1995].

6.2.3. Definições

Para ser possível delinear o modelo da Aplicação de Replicação proposto, é preciso antes avaliar as restrições do escopo de atuação da aplicação, para então definir os preceitos que orientarão as decisões de projeto adotas na aplicação. A seqüência desse raciocínio encontra-se descrita nas subseções a seguir.

- **Restrições**

Como visto na seção 6.2.1, um dos grandes problemas da replicação é manter a consistência de estado entre as réplicas. Se for possível prover o determinismo, a técnica de replicação adotada pode ser a da replicação ativa. Caso contrário, deverá haver uma forma de obter o estado de uma réplica líder, que será imposto sobre o estado das demais réplicas.

Os componentes servidores da plataforma J2EE visados pelo modelo de integração proposto são os componentes de prateleira (COTS), inerentemente encapsulados – não permitem o acesso ao seu código fonte. Adicionalmente, os EJBs não possuem suporte específico para introspecção de alto nível – o que os difere dos JavaBeans (seção 2.4.3). Mesmo sendo possível utilizar a API de Reflexão[11] Java na classe do EJB [SUN, 1999b] para descobrir seus membros (construtores, métodos e campos), tal trabalho de implementação, além de improdutivo, corre o risco de ir de encontro às restrições de segurança da linguagem Java (seção 3.1.9).

Por essas razões, o ato de obter o estado dos componentes EJBs, a fim de utilizá-los nos algoritmos de replicação, não serão discutidos no atual trabalho. Entende-se que tais definições fogem ao escopo da proposta de utilização intercessão apresentada, e são da competência da especificação da plataforma J2EE, uma vez que devem ser tratadas como padrão para atender ao quesito WORA de portabilidade de aplicação servidora.

Para então ser possível trabalhar com uma aplicação que ofereça a replicação no contexto reflexivo, nos moldes propostos por esse trabalho, a fim de prover tolerância a falha e disponibilidade, parte-se da premissa que o componente servidor EJB a ser replicado deve impreterivelmente prover todos métodos que modifiquem o estado do componente, definidos de antemão na sua interface *Remote* (seção 3.1.3).

➢ **Preceitos**

Em síntese, os seguintes preceitos, considerando as restrições levantadas em 6.2.3.1, orientam a definição desta proposta de Aplicação de Replicação:

1. Ser compatível com o modelo de integração proposto para utilização de intercessão na plataforma J2EE (seção 5.1);
2. Definir um modelo de aplicação que ofereça replicação de serviço visando tolerância a falha do tipo *crash*, e disponibilidade;
3. O serviço do componente EJB a ser replicado deve, de antemão, fornecer publicamente todos os métodos que alterem seu estado, definidos em sua interface *Remote*;
4. Não basear-se em um serviço de *membership* – não oferecido pela J2EE;
5. Não ser necessário prover o determinismo entre as réplicas, evitando operações específicas para garantir uma ordem única nas alterações aplicadas às réplicas [OLIVEIRA et al., 1998]; e
6. Em relação ao comportamento dos links de comunicação – que deve ser observado quando da comunicação entre as réplicas –, é considerado o modelo de sistema **assíncrono** [GUERRAOUI & SHIPER, 1996] que não define nenhum limite de tempo (*delay*) para transmissão de mensagem[12].

➢ **Decisões de Projeto**

Para estar em consonância com os preceitos apresentados em 6.2.3.2, a proposta para a Aplicação de Replicação segue as seguintes diretivas de projeto:

1. Adotar a técnica de replicação semi-passiva apresentada em [DÉFANO et al., 1998], que não necessita de um serviço de *membership*, provê tolerância a falha do tipo *crash* e disponibilidade;
2. A técnica de replicação semi-passiva baseia-se na comunicação *multicast* entre as réplicas, e para prover esse tipo de comunicação entre componentes EJB deve ser utilizado um serviço de mensagens acessado via API JMS (Java *Message Service* API, que descreve uma comunicação assíncrona) da J2EE, obedecendo as restrições de programação dos EJBs (seção 3.1.9).

---

[11] Considerada uma forma de introspecção de baixo nível [ORFALI & HARKEY, 1998].
[12] É requerido apenas que os canais de comunicação tenham confiabilidade: uma mensagem m enviada do processo p_i para p_j, é recebida em algum momento por p_j [GUERRAOUI & SHIPER, 1996].

Definidas as principais diretivas de projeto, pode-se então partir para seu detalhamento, como o exibido na seção 6.2.4 a seguir.

6.2.4. Modelo – Adoção da Replicação Semi-Passiva

Na replicação semi-passiva, a seleção do primário não é baseada em um serviço de *membership*: é baseada no paradigma do "coordenador rotativo" (*rotating coordinator*) que tem sido usado para resolver o problema de concenso. Isso permite a abordagem de dois-tempos: um valor de *timeout* agressivo pode ser usado para garantir rápida reação ao *crash* do primário, porém a suspeita incorreta do primário, devido ao valor agressivo de *timeout,* não leva o primário a ser excluído do grupo de servidores[13]. A replicação semi-passiva garante rápida reação à ocorrência de *crash*, sem o alto custo de uma suspeita incorreta de falha [DÉFANO et al., 1998].

Adcionalmente, o fato de não requerer um serviço de *membership* possibilita que a replicação semi-passiva não faça referência à recuperação de processo [DÉFANO et al., 1998].

- **Visão Geral da Solução**

De forma semelhante à replicação passiva, na técnica de replicação semi-passiva, a réplica primária manipula as requisições e, após o processamento de cada requisição, envia uma mensagem de atualização aos *backups*. Na solução de [DÉFANO et al., 1998], baseada em uma seqüência de problemas de consenso[14], cada consenso decide a mensagem de atualização.

Em outras palavras, expressar a replicação semi-passiva como uma seqüência de consensos esconde o papel do primário dentro do algoritmo de consenso. Um processo *p* atua como primário (i.e., manipula as requisições do cliente) precisamente quando *p* é o coordenador no algoritmo de consenso. Portanto, em cada instância do problema de consenso, o mecanismo para seleção de um coordenador é o mecanismo que seleciona o primário.

---

[13] Um *timeout* agressivo é custoso: envolve a remoção de um processo e em seguida sua reintegração ao grupo. Um *timeout* conservador pode levar a um período de *blackout* do sistema, muitas vezes inaceitável [DÉFANO et al., 1998].

[14] O detalhamento do problema de consenso encontra-se no Anexo B desse documento.

➢ **Solução em Boa Execução**

Uma "boa execução" é uma execução na qual nenhum processo servidor sofre *crash* e nenhuma suspeita de falha é gerada. Nesse caso, a solução geral leva a um esquema similar ao de comunicação da replicação passiva (Fig. 6.17). Seja a Fig. 6.17 a representação da execução do consenso número *k*. O processo servidor *réplica*$_1$ é o coordenador inicial do consenso *k* e também o primário. Após ter recebido a requisição do cliente, o primário *réplica*$_1$ manipula a requisição. Uma vez que o processamento esteja realizado, *réplica*$_1$ tem o valor inicial para o consenso *k*. De acordo com o protocolo de consenso, *replica*$_1$ faz a difusão (*multicast*) da mensagem de atualização *upd para os backups*, e espera por mensagens de *ack*. Uma vez que as mensagens de *ack* tenham sido recebidas (da maioria, na verdade), o processo *réplica*$_1$ pode decidir sobre *upd* e fazer *multicast* da mensagem *decide* para os *backups*. Tão logo a mensagem *decide* é recebida, os servidores atualizam seus estados, e enviam uma resposta (*reply*) para o cliente.

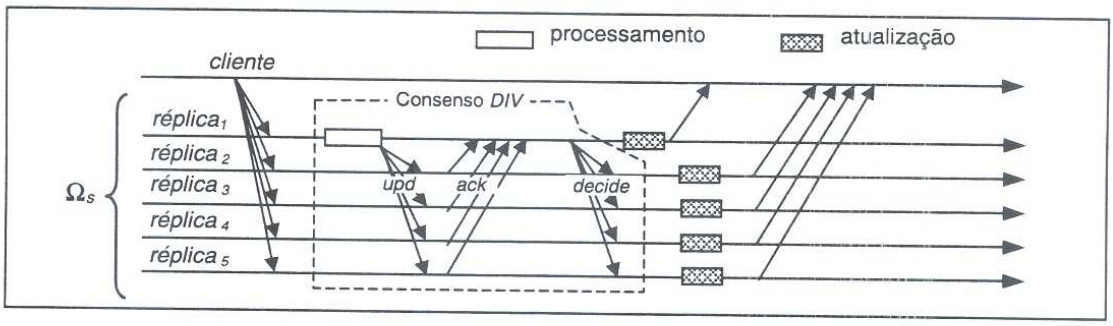

Fig. 6.17. Replicação Semi-passiva (boa execução).

➢ **Solução no Caso de *Crash***

Nessa avaliação, é considerado não haver suspeitas incorretas. Como ilustrado na Fig. 6.18, o cenário de pior caso acontece quando o primário *réplica*$_1$ (o coordenador inicial do algoritmo de consenso) sofre *crash* imediatamente após o processamento da requisição do cliente, porém antes de estar apto a enviar a mensagem de atualização *upd* aos *backups* (comparar com a Fig. 6.17).

Nessa situação, se o primário *réplica*$_1$ sofre *crash*, os *backups* suspeitam então de *réplica*$_1$, enviam uma mensagem negativa de *ack*, ou seja, um *nack*, à *réplica*$_1$ e começam um novo ciclo (*round*). O processo *réplica*$_2$ torna-se o coordenador para o novo ciclo, ou seja, torna-se o novo primário, e espera por mensagens de *estimate* da maioria dos servidores. De forma a obter um valor inicial, o novo primário *réplica*$_2$ processa a requisição recebida pelo cliente (Fig. 6.18), e desse ponto em diante, o

cenário é similar ao caso de "boa execução" da seção anterior (comparar com a Fig. 6.17).

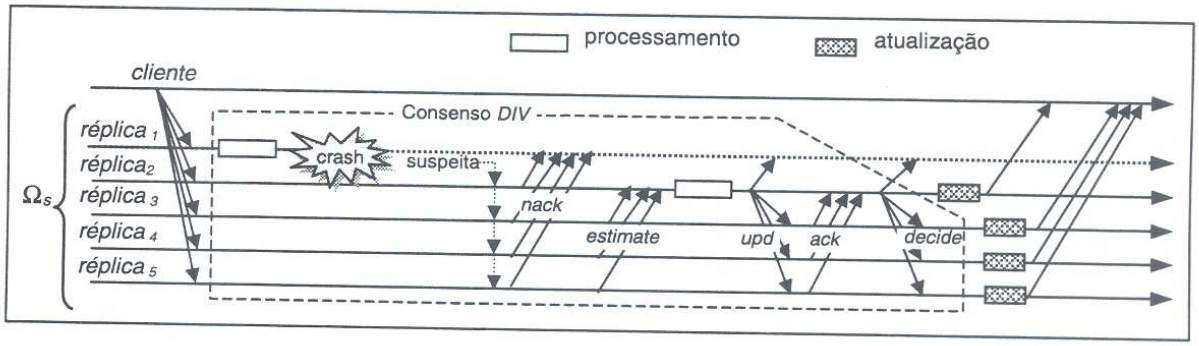

Fig. 6.18. Replicação Semi-passiva com uma Falha (pior caso).

➢ Representação Gráfica

Visando a implementação da replicação semi-passiva na J2EE, e mantendo a concordância como as definições de 6.2.3, o seguinte esquema de comunicação é apresentado na Fig. 6.19, a fim de suprir a necessidade de *multicast*.

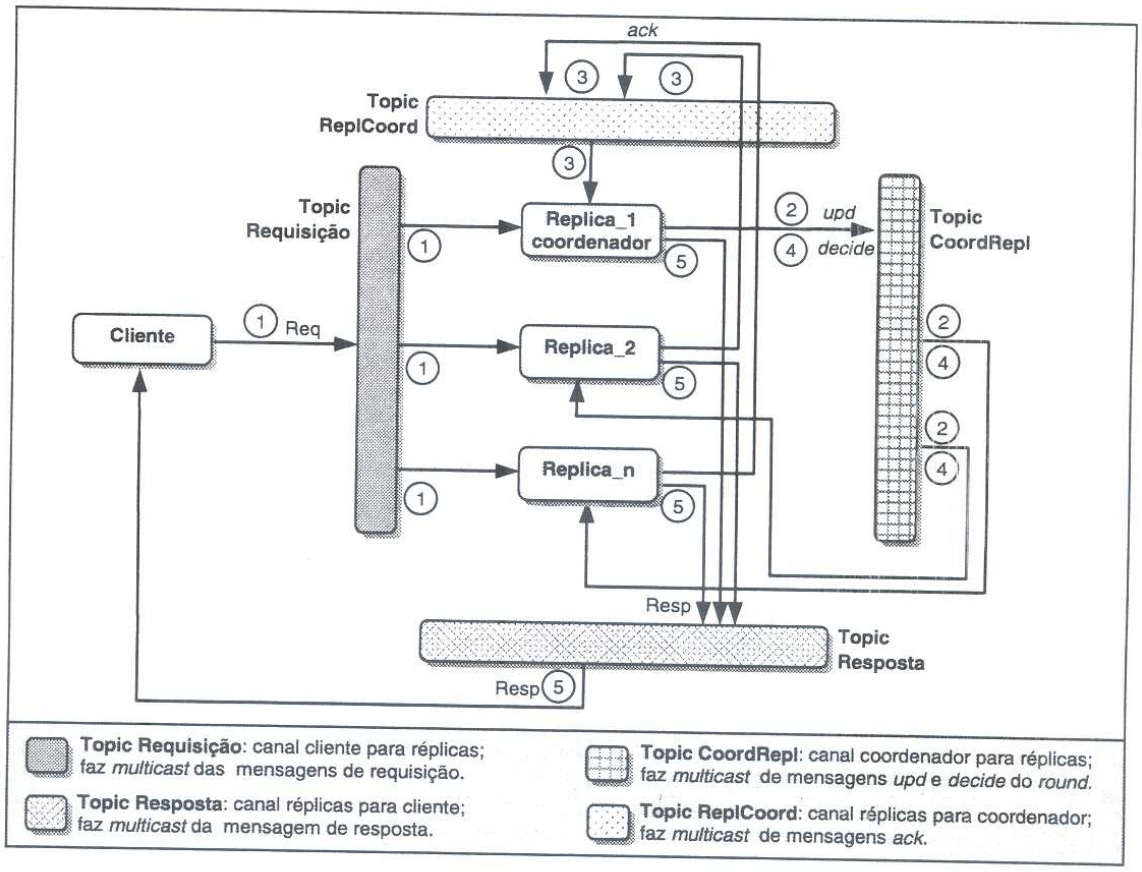

Fig. 6.19. Esquema de comunicação J2EE para replicação semi-passiva.

Os canais de mensagem definidos para habilitar a comunicação *multicast* – na Fig.6.17, "TopicRequisição" e "TopicResposta", "TopicRepliCoord" e "TopicCoordRepl" – seguem as diretivas do serviço JMS da J2EE[15], de acordo com o domínio *Publish / Subscribe*, ou Pub / Sub, descrito na seção 1.1.2 do Anexo C deste documento.

Nessa situação, pretende-se montar um grupo de réplicas – no nível meta da aplicação, via meta-componentes – onde cada participante torna-se membro do grupo ao se inscrever, ou assinar, determinados canais de mensagens [OLIVEIRA et al., 1998].

Os canais de mensagens foram definidos a fim de atender ao algoritmo de replicação semi-passiva (Fig. 6.17 e Fig. 6.18), e visam atender aos seguintes requisitos:

- "TopicRequisição": canal para recebimento das requisições do cliente: todas as réplicas devem assiná-lo de forma a receberem requisições simultaneamente; apenas o cliente irá publicar mensagens no canal.
- "TopicResposta": canal para envio de resposta ao cliente: todas as réplicas publicam suas respostas (idênticas, garantidas pelo algoritmo de replicação) no canal; apenas o cliente assina o canal – ele também tem a possibilidade de considerar somente a primeira resposta recebida.
- "TopicReplCoord": canal de envio de mensagens das réplicas para o coordenador: todas as réplicas assinam o canal, e todas estão habilitadas para publicar no mesmo; porém apenas a réplica primária, ou a coordenadora, irá processar as mensagens recebidas durante seu ciclo (*round*) de coordenação – as demais devem descartar a mensagem.
- "TopicCoordRepl": canal de envio de mensagens do coordenador para as réplicas: todas as réplicas assinam o canal e estão habilitadas para publicar no mesmo; porém apenas a réplica primária, ou a coordenadora, irá publicar mensagens durante seu ciclo (*round*) de coordenação.

Uma codificação genérica para canais de mensagem do domínio Pub/Sub é apresentada em detalhes na seção 1.3.1 do Anexo C deste documento. Esse exemplo de codificação objetiva apenas ilustrar a facilidade de comunicação proporcionada pelo serviço JMS da J2EE. Nele, cada canal é um componente que também recebe

---

[15] Os detalhes do Serviço JMS utilizados na proposta da Aplicação de Replicação estão no Anexo C desse documento.

um nome JNDI, o que facilita sua utilização e inserção em aplicações corporativa J2EE.

Os canais de mensagem do tipo "Topics" devem ser criados e configurados previamente pelo Provedor JMS e disponibilizado para acesso na Aplicação de Replicação, através de seu JNDI, como ilustrado na Fig. 6.20.

Ainda na Fig. 6.20, encontra-se esboçado o modelo básico para montagem da Aplicação de Replicação, valendo-se da proposta para utilização de intercessão, definida na seção 5.1. Observa-se que cada *Container* EJB – que contém o meta-componente e o componente de negócio base – **deve** estar localizado em um equipamento distinto, a fim de prover a replicação do serviço base.

Cabe ressaltar que a replicação provida dessa maneira provê tolerância a falha do tipo *crash* e disponibilidade apenas para o serviço base. Nesse caso, são **pontos únicos de falha** o próprio serviço JMS, o serviço Web, e o serviço provido pelo meta-componente "EJB META Requisição" (Fig. 6.20), que se encarrega de disparar as requisições do cliente via "TopicRequisição" e receber as respostas das solicitações via "TopicResposta".

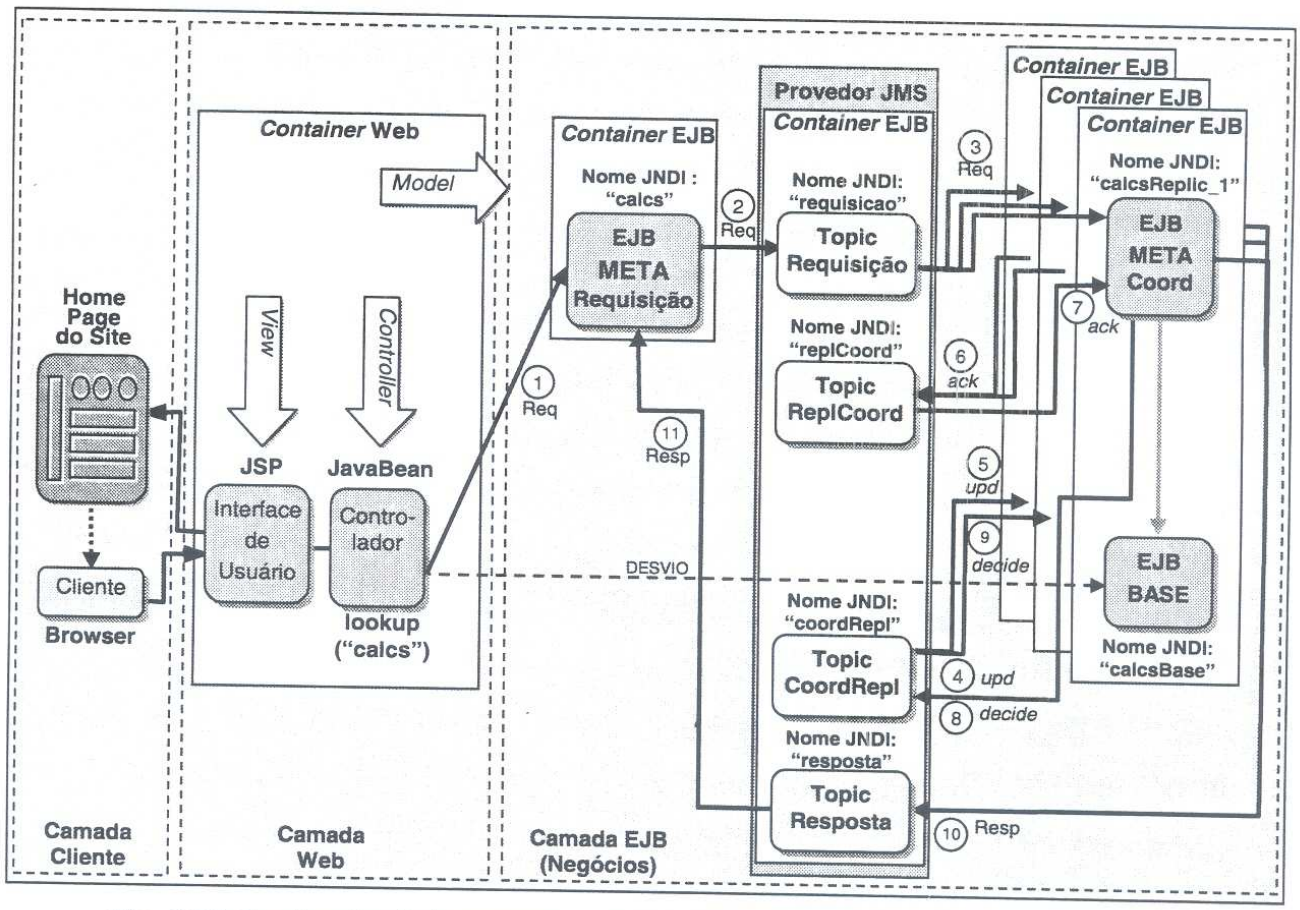

Fig. 6.20. Replicação via Intercessão – Perspectiva do modelo multi-camada J2EE.

Para o correto controle das mensagens trocadas entre as réplicas, as mensagens devem ser mapeadas em diversos campos de controle, possibilitando assim sua identificação precisa ao longo da execução do algoritmo de replicação semi-passiva – por exemplo, qual requisição acabou de ser recebida, qual já foi manipulada, qual ainda não foi enviada para atualização das réplicas, etc..

Para se adequar à essa necessidade da aplicação, o serviço JMS suporta um tipo de mensagem denominada *MapMessage*, ou mensagem mapeada, onde cada campo da mesma pode ser definido pelo usuário, da maneira mais conveniente para a aplicação. Um exemplo de codificação para utilizar esse tipo de mensagem encontra-se na seção 1.3.2 do Anexo C deste documento.

6.2.5. Considerações

O modelo para Aplicação de Replicação apresentado nessa seção 6.2 tem como principal vantagem o fato de esconder os detalhes de como a replicação é implementada no nível meta, possibilitando incluir essa funcionalidade em componentes EJB encapsulados – provendo portanto transparência e separação de interesses aplicados à tolerância a falhas. Contudo, o modelo tem várias restrições, sendo as principais: (1) os pontos únicos de falha descritos em 6.2.4.4, e (2) o fato dessa solução apenas ser possível para componentes EJB encapsulados que disponibilizem, através de sua interface *Remote*, todos os métodos que modificam seu estado.

6.3. Conclusões do Capítulo

Os exemplos de aplicação apresentados nesse Capítulo pretendem explorar as perspectivas da abordagem de Intercessão em Tempo de Implantação. Nesse sentido, eles cumprem seu papel, uma vez que revelam as várias restrições do modelo de integração, bem como ressaltam as características positivas do mesmo, no que se refere à adição de novas funcionalidades à serviços existentes, valendo-se principalmente das facilidades de montagem da plataforma.

A viabilidade da proposta para utilização de intercessão foi comprovada basicamente pela implementação do exemplo da Aplicação de Contabilização, que segue rigorosamente aos preceitos definidos em 5.1, e que também explora em detalhes a J2EE – evidenciando de forma concreta um dos grandes atrativos da plataforma: suas facilidades para montagem de aplicação.

O exemplo da Aplicação de Replicação, por sua vez, teve como principal preocupação levantar os prós e contras da proposta para utilização de intercessão no planejamento de uma aplicação que oferece a replicação de um serviço. Nesse sentido, foi confeccionado um projeto macro da solução, com indicações de como proceder com sua implementação. Esse exercício já possibilitou uma série de questionamentos, citados ao longo de 6.2, que podem inclusive vir a contestar a viabilidade dessa Aplicação de Replicação, uma vez que as restrições levantadas acabam por ter um grande peso na solução proposta. Contudo, foi exatamente para obter esse tipo de avaliação crítica do modelo de integração visto no Capítulo 5 que esse estudo foi realizado. O mesmo também possibilitou a realização várias pesquisas sobre replicação que subsidiaram as conclusões sobre a abordagem de utilização dessas técnicas na plataforma J2EE.

Enfim, é deixado como perspectiva de trabalho futuro a implementação da proposta para a Aplicação de Replicação.

CAPÍTULO 7 - CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho de pesquisa apresenta um empenho no sentido de projetar e implementar uma proposta que permita a utilização de características de reflexão computacional em aplicações servidoras. Mais especificamente, foi dada ênfase a aplicações corporativas que concentram-se em interoperabilidade e portabilidade, dentro de um contexto evolutivo que admita escalabilidade e desempenho. Todas essas características são pertinentes à plataforma J2EE, e determinaram sua escolha como plataforma base para a proposta desta dissertação.

Como conseqüência, o estudo dos requisitos para utilização de reflexão na plataforma J2EE permitiu realizar uma minuciosa avaliação da mesma, no sentido de levantar suas restrições e validar suas propriedades, a fim de utilizá-las para atingir os objetivos desta dissertação.

Adicionalmente, também foi obtida, como resultado desta pesquisa, uma coletânea de conceitos e definições que subsidiaram as conclusões do trabalho, principalmente no que diz respeito a:

- Componentes:
  - Unidades binárias, padronizadas para permitir colaboração com outros componentes dentro de um determinado contexto, propiciando a montagem de aplicações customizadas com portabilidade e interoperabilidade;
  - Entidades independentes, portanto passíveis de substituição e reutilização;
  - Mecanismo de construção de aplicação adotado pelas modernas arquiteturas de software.
- J2EE:
  - Padrão Java para ambiente corporativo altamente promissor, que permite a definição de soluções em sistemas distribuídos criticamente fundamentadas;
  - Sua padronização, contudo, impõe restrições de programação que visam assegurar portabilidade e consistência de ambiente.

- Reflexão Computacional:
  - Proporciona um alto grau de flexibilidade aos sistemas, habilitando a extensão de sua capacidade de controle e adaptação, sem acessar ou modificar as funcionalidades básicas da aplicação;
  - Pode ser particionada genericamente em dois tipos de funções: a introspecção (reflexão estrutural) e a intercessão (reflexão comportamental);
  - O programa que realiza a reflexão computacional, introspecção e / ou intercessão, é chamado de programa de nível meta, enquanto o programa que é executado na computação reflexiva é chamado de programa de nível base.

As próximas subseções organizam as conclusões e resultados do trabalho da seguinte forma: a seção 7.1 sintetiza a condução da proposta, recapitulando os fatos e decisões que definiram sua confecção; a seção 7.2 faz uma rápida análise da viabilidade da proposta em plataformas concorrentes à J2EE; a seção 7.3 apresenta as perspectivas de continuação do trabalho; por fim, a seção 7.4 encerra o Capítulo listando resumidamente os principais resultados desta pesquisa.

7.1. Intercessão em Tempo de Implantação

O modelo de integração apresentado no Capítulo 5 foi confeccionado com base na análise comparativa das arquiteturas reflexivas metaXa [GOLM & KLEINÖDER, 1998], Guaraná [OLIVA, 1998] e OpenJava [TATSUBORI, 1999].

Suas abordagens MOP em Tempo de Execução (metaXa e Guaraná) e MOP em Tempo de Compilação (OpenJava) não são recomendadas para a plataforma J2EE, justamente por propor um novo padrão ou por se basear em práticas nem sempre possíveis na plataforma. Como a preservação das características da J2EE é o principal preceito da proposta, obtém-se os seguintes impedimentos em relação às abordagens das arquiteturas reflexivas estudadas:

(1) Aplicações J2EE muito provavelmente não disponibilizam seu código fonte, pois utilizam componentes de negócio encapsulados de baixa granularidade (COTS) – o que impede o MOP em Tempo de Compilação; e

(2) Modificações nos padrões J2EE podem acarretar em problemas de compatibilidade com a plataforma – o que não recomenda o MOP em Tempo de Execução.

Então, para aliar as vantagens da reflexão com as da plataforma J2EE, foram utilizadas as facilidades para composição de aplicação recomendadas pelo Modelo de Programação Multi-camada da J2EE. Ao favorecer o desacoplamento de funcionalidade em componentes lógicos, ele possibilita a utilização de reflexão comportamental (intercessão) apenas na camada de negócio, que encerra a funcionalidade básica da aplicação, e é representada pelos componentes EJBs. Por conseguinte, interceptar os métodos dos EJBs – através da inserção (pelo redirecionamento de nomes) de um meta-componente EJB – permite a alteração do comportamento da aplicação como um todo. Ou seja, gera uma alternativa para utilizar princípios de reflexão computacional na J2EE, através da abordagem MOP em Tempo de Implantação (*Deploy-Yime* MOP), ou mais especificamente Intercessão em Tempo de Implantação.

7.2. Plataformas Concorrentes à J2EE - Considerações

As plataformas de mercado diretamente concorrentes à Plataforma Java para Corporações são (ver Capítulo 2) a Plataforma CORBA da OMG e a Plataforma DCOM, ou COM+, da Microsoft. Ambas visam oferecer um conjunto de soluções para aplicações corporativas, similar ao da J2EE. As subseções a seguir analisam a viabilidade da proposta de Intercessão em Tempo de Implantação nessas plataformas.

7.2.1. Viabilidade da Proposta no CORBA

O Modelo de Componentes CORBA, CCM (CORBA *Component Model*), baseou-se no modelo de componentes servidores da J2EE [MELEWS, 2000], os EJBs, para definir um modelo de componentes análogo (ver seção 2.4.1) para diferentes linguagens de programação – inclusive para o Java, sendo que a especificação CCM para Java é a mesma do EJB [MELEWS, 2000] [OMG, 1999].

A similaridade entre o padrão CORBA e o padrão EJB visa mais complementação e compatibilidade do que concorrência, pois tanto a Sun quanto a OMG recomendam uma combinação: CORBA para sistemas legados e EJB para novas aplicações multi-camada [MELEWS, 2000]. Dessa forma, componentes EJB encarregam-se de proporcionar facilidade de utilização e rápidos desenvolvimento e implantação para novas aplicações servidoras, enquanto o CORBA oferece soluções de conectividade aberta. De fato, o ambiente corporativo Java considera a existência de ao menos um ORB CORBA para prover interoperabilidade com sistemas implementados em outras linguagens. A compatibilidade, por sua vez, concentra-se no

fato do CCM estender o lado servidor CORBA de modo similar ao proporcionado pelos componentes servidores EJB, provendo essa facilidade para linguagens não Java.

Diante do exposto, a abordagem de Intercessão em Tempo de Implantação encontra no padrão CORBA suporte de implementação e implantação correspondente ao do padrão J2EE, viabilizando-a na plataforma da OMG. Ou seja, o princípio da abordagem pode ser repetido na plataforma CORBA, através de seus padrões específicos. Ressalta-se, contudo, que a ênfase em ferramentas da J2EE, a "*toolability*" [ORFALI & HARKEY, 1998], não tem a mesma correspondência no CORBA. Apesar de considerar o uso de ferramentas na montagem e implantação de aplicações baseadas em componentes (admite que fornecedores de software independentes ofereçam suas próprias ferramentas), o CORBA não as especifica, e muito menos fornece implementações de referência – como a J2EE com sua J2EE SDK –, o que impacta diretamente no quesito produtividade. Daí o ambiente J2EE ser considerado muito mais produtivo e fácil de utilizar, enquanto o CORBA demanda programadores mais especializados na criação de sistemas distribuídos [ORFALI & HARKEY, 1998].

7.2.2. Viabilidade da Proposta no COM+

Pelo exposto na seção 2.4.2, sobre as facilidades de composição e implantação na plataforma Microsoft, pode-se constatar que a abordagem de Intercessão em Tempo de Implantação não encontra no padrão COM+ o mesmo suporte de implementação e implantação do padrão J2EE. Contudo, isso não inviabiliza a utilização dos princípios de reflexão nessa plataforma, nos moldes propostos, visto que seu modelo de componentes comporta os conceitos básicos de reutilização e independência de módulos (camada de negócio). Ou seja, é possível inserir meta-componentes na aplicação servidora, de forma visual e em tempo de implantação, visando alteração do comportamento. Contudo, para efetivar essa nova proposta, o modelo de integração apresentado no Capítulo 5 deve ser revisto e adaptado às especificações do COM+.

7.3. Perspectivas

A partir dos resultados obtidos, tanto no projeto implementado quanto no proposto, acredita-se que no futuro novos trabalhos poderão ser desenvolvidos, contribuindo em termos de complementação e de evolução do esforço atual. A continuação dessa pesquisa pode então concentrar-se em:

- Implementar a proposta da Aplicação de Replicação, apresentada na seção 6.2;
- Implementar a proposta Aplicação de Replicação com uma técnica diferente da replicação semi-passiva (p.ex., replicação passiva), a fim de verificar na prática a flexibilidade de alteração do código de controle, sem interferência na funcionalidade básica de aplicação;
- Construir novas aplicações que utilizem a proposta de Intercessão em Tempo de Implantação para introduzir controle e alteração de comportamento (novas funcionalidades) em aplicações corporativas J2EE prontas; e
- Revisar e adaptar a proposta de Intercessão em Tempo de Implantação para os padrões OMG CORBA e Microsoft COM+, a fim de analisar e apontar as principais diferenças desses modelos em relação à proposta em si, ao desempenho e às facilidades de implementação e implantação.

7.4. Resultados

Valendo-se das premissas expostas na seção 7.1, o modelo de integração apresentado atingiu seu objetivo, uma vez que conseguiu implementar essa característica da reflexão computacional (intercessão implementada em tempo de implantação), verificando-a na prática: uma implementação simples foi efetivada (ver Capítulo 6) no sentido de validar proposta.

Em síntese, o modelo de integração proposto pela abordagem de Intercessão em Tempo de Implantação é endereçado a aplicações que contenham componentes de negócio EJB encapsulados (COTS) e visa permitir, a partir de procedimentos sistematizados e auxiliados por ferramenta de implantação, que a aplicação como um todo possa ser alterada de forma mais elegante e modular, mantendo intactas suas funcionalidades básicas.

Tais facilidades, portanto, provêem como resultado maior flexibilidade de desenvolvimento e implementação nas situações onde é necessário introduzir controle e / ou modificar a funcionalidade da aplicação (alteração de comportamento ou adaptação a novos requisitos), que é precisamente a meta almejada pela proposta da dissertação.

Por fim, cabe ressaltar que a plataforma J2EE de fato comprovou ser, através deste trabalho de pesquisa, uma alternativa altamente promissora para ambientes corporativos. A organização de seu conjunto de serviços e facilidades oferece um leque de vantagens cada vez mais robusto, reconhecido inclusive por ambientes

concorrentes. Consequentemente, o domínio das características e capacidades desta plataforma pode ser encarado como um diferencial positivo, visto que seus preceitos estão contribuindo ativamente para definir a evolução do futuro da computação, principalmente no que diz respeito a aplicações corporativas.

ANEXO A – CÓDIGO DO EXEMPLO: APLICAÇÃO DE CONTABILIZAÇÃO

Dado o exemplo da Aplicação de Contabilização, apresentado na seção 6.1, seguem os códigos fonte da solução e procedimentos para sua execução, descritos nas subseções a seguir.

1.1. Pacote Beans::*

O pacote "Beans::*" contém os arquivos que definem o componente de negócio base "Beans::CalcEJB". São seus métodos de negócio que sofrerão intercessão, a fim de inserir um mecanismo – via meta-componente – de contabilização das ações executadas no mesmo.

1.1.1. Arquivo Beans\Calc.java

```
//-------------------------------------------------------------
// Arquivo: Calc.java
//-------------------------------------------------------------
package Beans;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
//-------------------------------------------------------------
// INTERFACE: Calc
//
// Interface Remote para o enterprise bean CalcEJB.
// Define a assinatura dos métodos que serao disponibilizados
// por CalcEJB.
//-------------------------------------------------------------
public interface Calc extends EJBObject {
  public double calcBonus (int multiplier, double bonus)
                              throws RemoteException;
  public double calcCredit(int rate, double bonus) throws RemoteException;
}
```

1.1.2. Arquivo Beans\CalcHome.java

```
//-------------------------------------------------------------
// Arquivo: CalcHome.java
//-------------------------------------------------------------
package Beans;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
//-------------------------------------------------------------
// INTERFACE: CalcHome
//
// Interface Home para o enterprise bean CalcEJB.
// Chamada via JNDI
//-------------------------------------------------------------
public interface CalcHome extends EJBHome {
  Calc create() throws CreateException, RemoteException;
}
```

1.1.3. Arquivo Beans\CalcEJB.java

```
//-------------------------------------------------------------
// Arquivo: CalcEJB.java
//-------------------------------------------------------------
package Beans;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
//-------------------------------------------------------------
// CLASSE: CalcEJB
//
// Implementacao de SessionBean para confeccao do
// componente de negocio base Beans.CalcEJB.
//-------------------------------------------------------------
public class CalcEJB implements SessionBean {

//-------------------------------------------------------------
//                      METODOS DE NEGOCIO
//-------------------------------------------------------------
//-------------------------------------------------------------
// Metodo: calcBonus
// Faz o calculo do Bonus.
// Retorno: calculo do Bonus
//-------------------------------------------------------------
public double calcBonus (int multiplier, double bonus) {
double calc = (multiplier*bonus);
return calc;
}
//-------------------------------------------------------------
// Metodo: calcCredit
// Faz o calculo do Credito.
// Retorno: calculo do Credito
//-------------------------------------------------------------
public double calcCredit (int rate, double bonus) {
double calc = (bonus / rate);
return calc;
}
//-------------------------------------------------------------
//             METODOS HERDADOS DO SessionBean
//-------------------------------------------------------------
public void ejbCreate()    { }
public void setSessionContext(SessionContext ctx) { }
public void ejbRemove()    { }
public void ejbActivate()  { }
public void ejbPassivate() { }
public void ejbLoad()      { }
public void ejbStore()     { }
}
```

1.2. Pacote BeansMeta::*

O pacote "BeansMeta::*" contém os arquivos que definem o meta-componente de negócio "BeansMeta::CalcEJB". Este por sua vez se encarrega de executar o mecanismo de contabilização das ações executadas no componente base, o "Beans::CalcEJB", provendo assim a intercessão.

1.2.1. Arquivo BeansMeta\Calc.java

```
//-------------------------------------------------------------
// Arquivo: Calc.java
//-------------------------------------------------------------
package BeansMeta;
import javax.ejb.EJBObject;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.sql.SQLException;
//-------------------------------------------------------------
// INTERFACE: Calc
//
// Interface Remote para o enterprise bean CalcEJB.
// Define a assinatura dos métodos que serao disponibilizados
// por CalcEJB.
//-------------------------------------------------------------
public interface Calc extends EJBObject {
  public double calcBonus (int multiplier, double bonus) throws
               CreateException, RemoteException, SQLException;
  public double calcCredit(int rate, double bonus) throws
               CreateException, RemoteException, SQLException;
}
```

1.2.2. Arquivo BeansMeta\CalcHome.java

```
//-------------------------------------------------------------
// Arquivo: CalcHome.java
//-------------------------------------------------------------
package BeansMeta;
import javax.naming.NamingException;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
//-------------------------------------------------------------
// INTERFACE: CalcHome
//
// Interface Home para o enterprise bean CalcEJB.
// Chamada via JNDI
//-------------------------------------------------------------
public interface CalcHome extends EJBHome {
  Calc create() throws NamingException, CreateException, RemoteException;
}
```

1.2.3. Arquivo BeansMeta\CalcEJB.java

```
//-------------------------------------------------------------
// Arquivo: CalcEJB.java
//-------------------------------------------------------------
package BeansMeta;
import javax.rmi.PortableRemoteObject;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.sql.SQLException;
import Beans.*;
import LogPack.*;
import LogUtil.*;
```

```
//-------------------------------------------------------------
// CLASSE: CalcEJB
//
// Implementacao de SessionBean para confeccao do
// Meta-componente BeansMeta.CalcEJB para o componente
// Beans.CalcEJB.
//-------------------------------------------------------------
public class CalcEJB implements SessionBean {
  Beans.CalcHome     homecalcBase;
  Beans.Calc         oCalculo;
  LogPack.LogHome    homeLog;
  LogPack.Log        oLog;
  LogUtil.Calendario Data;
  double             calculo;
//-------------------------------------------------------------
//                      METODOS DE NEGOCIO
//-------------------------------------------------------------
//-------------------------------------------------------------
// Metodo: calcBonus
// Faz o calculo do Bonus. Chama EJB para log, o LogEJB.
// Retorno: calculo do Bonus
//-------------------------------------------------------------
  public double calcBonus(int multiplier, double bonus)
      throws CreateException, RemoteException, SQLException {
    try{
        oCalculo = homecalcBase.create();
        calculo  = oCalculo.calcBonus(multiplier, bonus);
        calculo  = calculo * 3;
        oLog     = homeLog.create();
        this.Data = LogUtil.Calendario.getInstance();
        oLog.setLog (this.Data,calculo,"calcBonus", "Ok.");
     } catch (java.sql.SQLException e) {
       throw new SQLException    ("Falha Log de Bonus " + e);
     } catch (javax.ejb.CreateException e) {
       throw new CreateException ("Falha Log - Bonus " + e);
     } catch (java.rmi.RemoteException e) {
       throw new RemoteException ("Falha Log - Bonus " + e);
     } return calculo;
  }
//-------------------------------------------------------------
// Metodo: calcCredit
// Faz o calculo do Credito. Chama EJB para log, o LogEJB
// Retorno: calculo do Credito
//-------------------------------------------------------------
  public double calcCredit(int rate, double bonus)
           throws CreateException, RemoteException, SQLException {
    try{
        oCalculo = homecalcBase.create();
        calculo  = oCalculo.calcCredit(rate, bonus);
        calculo  = calculo * 2;
        oLog     = homeLog.create();
        this.Data = LogUtil.Calendario.getInstance();
        oLog.setLog (this.Data,calculo,"calcCredit", "Ok.");
     } catch (java.sql.SQLException e) {
       throw new SQLException    ("Falha Log de Credito " + e);
     } catch (javax.ejb.CreateException e) {
       throw new CreateException ("Falha Log - Credito " + e);
     } catch (java.rmi.RemoteException e) {
       throw new RemoteException ("Falha Log - Credito " + e);
     }
   return calculo;
  }
```

```
//-------------------------------------------------------------
//              METODOS HERDADOS DO SessionBean
//-------------------------------------------------------------
//-------------------------------------------------------------
// Metodo: ejbCreate
// Metodo chamado no momento de criacao do enterprise bean
// CalcEJB. Localiza via JNDI os entrprise beans "calcsBase"
// e "logs".
//-------------------------------------------------------------
  public void ejbCreate() throws NamingException {
   try{
       InitialContext ctx = new InitialContext();
       Object objref1 = ctx.lookup("calcsBase");
       Object objref2 = ctx.lookup("logs");

       homecalcBase = (Beans.CalcHome)
           PortableRemoteObject.narrow(objref1, Beans.CalcHome.class);
       homeLog      = (LogPack.LogHome)
           PortableRemoteObject.narrow(objref2, LogPack.LogHome.class);
   } catch (javax.naming.NamingException e) {
     throw new NamingException("Falha ao procurar " + e);
   }
 }
  public void setSessionContext(SessionContext ctx) { }
  public void ejbRemove() { }
  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void ejbLoad() { }
  public void ejbStore() { }
}
```

1.3. Pacote LogPack::*

O pacote "LogPack::*" contém os arquivos que definem o meta-componente de negócio "LogPack::LogEJB". Este por sua vez registra em um banco de dados as ações realizadas no componente de negócio base – "Beans::CalcEJB". O meta-componente "BeansMeta::CalcEJB", ao interceptar os métodos do "Beans::CalcEJB", faz as chamadas ao "LogPack.LogEJB".

1.3.1. Arquivo LogPack\Log.java

```
//-------------------------------------------------------------
// Arquivo: Log.java
//-------------------------------------------------------------
package LogPack;
import java.util.Collection;
import java.sql.SQLException;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import LogUtil.*;
//-------------------------------------------------------------
// INTERFACE: Log
//
// Interface Remote para o enterprise bean LogEJB.
// Define a assinatura dos métodos que serao disponibilizados
// por LogEJB.
//-------------------------------------------------------------
```

```
public interface Log extends EJBObject {
    public void setLog (Calendario lData, double lTotal,
        String lMetodo, String lMsg) throws SQLException, RemoteException;
    public Collection getLog() throws SQLException, RemoteException;
}
```

1.3.2. Arquivo LogPack\LogHome.java

```
//-------------------------------------------------------------
// Arquivo: LogHome.java
//-------------------------------------------------------------
package LogPack;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
//-------------------------------------------------------------
// INTERFACE: LogHome
//
// Interface Home para o enterprise bean LogEJB.
// Chamada via JNDI
//-------------------------------------------------------------
public interface LogHome extends EJBHome {
    public Log create() throws CreateException, RemoteException;
}
```

1.3.3. Arquivo LogPack\LogEJB.java

```
//-------------------------------------------------------------
// Arquivo: LogEJB.java
//-------------------------------------------------------------
package LogPack;
import java.util.ArrayList;
import java.util.Collection;
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.text.NumberFormat;
import java.text.DecimalFormat;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.EJBException;
import javax.ejb.CreateException;
import javax.sql.DataSource;
import LogUtil.*;
//-------------------------------------------------------------
// CLASSE: LogEJB
//
// Implementacao de SessionBean para
// controle de Log.
//-------------------------------------------------------------
public class LogEJB implements SessionBean {
    protected Connection dbConnection = null;
    private Calendario logData = Calendario.getInstance ();
    private int        logID;
    private String     logMetodo;
    private String     logTotal;
    private String     logMsg;
```

```
    private LogItem    logItem;
//-------------------------------------------------------------
//                      METODOS DE NEGOCIO
//-------------------------------------------------------------
//-------------------------------------------------------------
// Metodo: getLog
// Faz uma consulta a tabela Tab_Log e retorna seu
// conteudo como uma java.util.Collection.
// Retorno: colecao com o conteudo de Tab_Log.
//-------------------------------------------------------------
  public Collection getLog() throws SQLException {
      ArrayList al = new ArrayList();
      Integer   integerAno, integerMes, integerDia;
      int       Ano, Mes, Dia;
      try {
        dbConnection    = getDBConnection();
        Statement stmt = dbConnection.createStatement();
        ResultSet rs    = stmt.executeQuery("select * from tab_log");
        while (rs.next()) {
            this.logID       = rs.getInt    ("LOGID");
            String nData = rs.getString("DATA");
            Integer Ano    = new Integer(nData.substring(0,4));
            Ano            = integerAno.intValue();
            Integer Mes    = new Integer(nData.substring(5,7));
            Mes            = integerMes.intValue();
            Integer Dia    = new Integer(nData.substring(8,10));
            Dia            = integerDia.intValue();
            this.logData.set (Ano, Mes, Dia);
            this.logTotal     = rs.getString("TOTAL");
            this.logMetodo    = rs.getString("METODO");
            this.logMsg       = rs.getString("MSG");
            logItem = new LogItem(logID, logData, logTotal, logMetodo,
            logMsg);
            al.add(logItem);
        }
        rs.close();
        stmt.close();
       } catch(java.sql.SQLException e) {
          throw new SQLException ("ERRO na Log_TABLE: SELECT !! " + e);
       } finally {
           if (dbConnection != null) dbConnection.close();
       }
        return al;
    }
//-------------------------------------------------------------
// Método: setLog
// Inclui uma linha na tabela Tab_Log.
//-------------------------------------------------------------
  public void setLog  (Calendario lData, double lTotal,
                       String lMetodo, String lMsg)
  throws SQLException {
      try {
        dbConnection = getDBConnection();
        this.logID = UUIDGenerator.nextSeqNum(dbConnection);
        Statement stmt = dbConnection.createStatement();
        String queryStr = "INSERT INTO TAB_LOG " +
                          "(LOGID,DATA,TOTAL,METODO,MSG)" +
                          "VALUES ("    +
                              + this.logID + "," +
                          "'" + lData.getCloudscapeDateString() + "'," +
                          "'" + formatTotal(lTotal) + "'," +
                          "'" + lMetodo + "'," +
```

```
                              "'" + lMsg     + "' )";
          int resultCont = stmt.executeUpdate(queryStr);
          if (resultCont != 1 ){
               throw new java.sql.SQLException("ERRO na Log_TABLE: INSERT !!
                                                    resultCont = " +
resultCont);
          }
          stmt.close();
        } catch(java.sql.SQLException e) {
           throw new SQLException ("SQLException em create: " + e);
        } finally {
            if (dbConnection != null) dbConnection.close();
        }
}
//-------------------------------------------------------------
//              METODOS HERDADOS DO SessionBean
//-------------------------------------------------------------
//-------------------------------------------------------------
// Metodo: ejbCreate
// Metodo chamado no momento de criacao do enterprise bean
// LogEJB.
//-------------------------------------------------------------
    public void ejbCreate() throws CreateException {
        try{
             dbConnection = getDBConnection();
        }catch (java.sql.SQLException e) {
             throw new CreateException ("SQL Exception in create:" + e);
        }
        finally {
          try {
            dbConnection.close();
          }catch (java.sql.SQLException e) {
           throw new javax.ejb.CreateException("SQLException create:" + e);
          }
        }
    }

    public void setSessionContext(SessionContext sc) {
      this.sc = sc;
    }
    public  void ejbRemove()    {}
    public  void ejbPassivate() {}
    public  void ejbActivate()  {}
    private SessionContext sc = null;
    private Context ctx       = null;
    private DataSource ds     = null;
//-------------------------------------------------------------
//                      METODOS UTILITARIOS
//-------------------------------------------------------------
//-------------------------------------------------------------
// Método: getDBConnection
// Procura via JNDI o nome do bacno de dados que armazena
// os dados da tabela Tab_Log.
// Retorno: a conexao com o DB
//-------------------------------------------------------------
    private Connection getDBConnection() throws SQLException {
        Connection connection;
        try {
            InitialContext ic = new InitialContext();
            DataSource ds = (DataSource) ic.lookup("jdbc/LogDB");
            connection = ds.getConnection();
        } catch (javax.naming.NamingException e) {
```

```
            throw new EJBException("Log: falha de nome " + e);
        } catch (java.sql.SQLException e) {
            throw new EJBException("Log: SQLException " + e);
        }
        return connection;
    }
//-------------------------------------------------------------
// Metodo: formatTotal
// Formata uma entrada do tipo double em uma string com
// duas casas decimais.
// Retorno: String da entrada double
//-------------------------------------------------------------
    private String formatTotal(double total){
        NumberFormat nf = NumberFormat.getInstance();
        DecimalFormat df = (DecimalFormat)nf;
        df.setMinimumFractionDigits(2);
        df.setMaximumFractionDigits(2);
        String pattern = "##.00";
        df.applyPattern(pattern);
        df.setDecimalSeparatorAlwaysShown(true);
        return df.format(total);
    }
}
```

1.4. Pacote LogUtil::*

O pacote "LogUtil::*" contém os arquivos cujas funcionalidades auxiliam as atividades do meta-componente de negócio "LogPack::LogEJB", no momento de manipular as informações de contabilização armazenadas em banco de dados.

1.4.1. Arquivo LogUtil\Calendario.java

```
//-------------------------------------------------------------
// Arquivo: JViewerBean.java
//-------------------------------------------------------------
package LogUtil;
import java.util.Date;
import com.sun.xml.tree.XmlDocument;
import com.sun.xml.tree.ElementNode;
//-------------------------------------------------------------
// CLASSE: Calendario
//
// Essa classe representa um calendário.
//-------------------------------------------------------------
public class Calendario extends Object implements java.io.Serializable{
    private int mes;
    private int dia;
    private int ano;
//-------------------------------------------------------------
// CLASSE: Calendario
//
// Essa classe representa um calendário.
//-------------------------------------------------------------
private Calendario(int ano, int mes, int dia){
   this.mes = mes;
   this.dia= dia;
   this.ano = ano;
}
```

```
//-------------------------------------------------------------
// Metodo: getInstance
// Obtem a data atual do sistema, retorna um objeto calendario.
// Retorno: Data atual do sistema
//-------------------------------------------------------------
public static Calendario getInstance(){
   java.util.Calendar c = java.util.Calendar.getInstance();
   int m = c.get(java.util.Calendar.MONTH);
   int d = c.get(java.util.Calendar.DATE);
   int a = c.get(java.util.Calendar.YEAR);
   return new Calendario (a,m,d);
}
//-------------------------------------------------------------
// Metodo: getMes
// Obtem o mes do objeto calendario.
// Retorno: mes
//-------------------------------------------------------------
public int getMes(){
  return mes;
}
//-------------------------------------------------------------
// Metodo: getDia
// Obtem o dia do objeto calendario.
// Retorno: dia
//-------------------------------------------------------------
public int getDia(){
   return dia;
}
//-------------------------------------------------------------
// Metodo: getAno
// Obtem o ano do objeto calendario.
// Retorno: ano
//-------------------------------------------------------------
public int getAno(){
   return ano;
}
//-------------------------------------------------------------
// Metodo: set
// Atualiza a data do objeto calendario.
//-------------------------------------------------------------
public void set(int ano, int mes, int dia){
   this.mes = mes;
   this.dia = dia;
   this.ano = ano;
}
//-------------------------------------------------------------
// Metodo: getDate
// Transforma a data do obj. calendario para o formato java.util.Date.
// Retorno: java.util.Date
//-------------------------------------------------------------
public Date getTime(){
   return new Date(ano, mes, dia);
}
//-------------------------------------------------------------
// Metodo: setTime
// Atualiza a data do objeto calendario a partir de uma
// entrada do tipo java.util.Date
//-------------------------------------------------------------
public void setTime(java.util.Date date){
   java.util.Calendar c = java.util.Calendar.getInstance();
   c.setTime(date);
   this.dia = c.get(java.util.Calendar.DATE);
```

```
        this.mes = c.get(java.util.Calendar.MONTH);
        this.ano = c.get(java.util.Calendar.YEAR);
    }
    //-------------------------------------------------------------
    // Metodo: clear
    // Inicializa o conteudo do objeto calendario.
    // Retorno: mes
    //-------------------------------------------------------------
    public void clear(){
        this.dia = -1;
        this.mes = -1;
        this.ano = -1;
    }
    //-------------------------------------------------------------
    // Metodo: getFullDateString
    // Obtem o formato mm/dd/yyyy.
    // Retorno: String da data
    //-------------------------------------------------------------
    public String getFullDateString(){
        return ((mes + 1 < 10)? "0" : "") + (mes + 1) + "/" + ((dia < 10)? "0" :
    "") +  dia + "/" + ((ano < 10)? "0" : "") + ano;
    }
    //-------------------------------------------------------------
    // Metodo: toString
    // Transforma o conteudo de calendario em String.
    // Retorno: String da data
    //-------------------------------------------------------------
    public String toString(){
        return "[ano=" + ano + ", mes=" + mes + ", dia=" + dia + "]";
    }
    //-------------------------------------------------------------
    // Metodo: getCloudscapeDateString
    // Transforma o conteudo de calendario para o formato de data
    // suportado pelo SGBD Cloudscape.
    // Retorno: String da data
    //-------------------------------------------------------------
    public String getCloudscapeDateString(){
        return ano + "-" + ((mes + 1 < 10)? "0" : "") + (mes + 1) + "-" + ((dia
    < 10)? "0" : "") + dia;
    }
    }
```

1.4.2. Arquivo LogUtil\UUIDGenerator.java

```
    //-------------------------------------------------------------
    // Arquivo: UUIDGenerator.java
    //-------------------------------------------------------------
    package LogUtil;
    import java.sql.Connection;
    import java.sql.Statement;
    import java.sql.ResultSet;
    import java.sql.SQLException;
    //-------------------------------------------------------------
    // CLASSE: UUIDGenerator
    //
    // Essa classe eh usada para gerar uma chave primaria
    // unica para o LogEJB, na tabela Tab_Log.
    //-------------------------------------------------------------
    public class UUIDGenerator implements java.io.Serializable {
```

```
//-------------------------------------------------------------
// Metodo: nextSeqNum
// Esse metodo obtem o proximo numero na sequencia
// para insercao na Tab_Log, e atualiza o numero de
// sequencia. Uma base de dados é usada para armazenar
// esse numero de sequencia na tabela Sequence.
//
// Retorno: o próximo número de seqüência.
//-------------------------------------------------------------
public static int nextSeqNum(Connection dbConnection)
   throws SQLException {
        int seqNum = 0;
        Statement s = dbConnection.createStatement();
        ResultSet rs = s.executeQuery
            ("SELECT seqnum FROM sequence for update");
        if(rs.next())
            seqNum = rs.getInt(1);
        int resultCount = s.executeUpdate
                        ("update sequence set seqnum = seqnum + 1");
        if ( resultCount != 1 )
            throw new SQLException
                        ("Nao foi possivel atualizar a tabela: Sequence");
        s.close();
        return (seqNum);
}
}
```

1.4.3. Arquivo LogUtil\LogItem.java

```
//-------------------------------------------------------------
// Arquivo: LogItem.java
//-------------------------------------------------------------
package LogUtil;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Collection;
import java.io.Serializable;
//-------------------------------------------------------------
// CLASSE: LogItem
//
// Essa classe eh usada para guardar o conteúdo de uma linha
// da tabela Tab_Log.
//-------------------------------------------------------------
public class LogItem implements java.io.Serializable {
    private int        logID;
    private Calendario logData;
    private String     logMetodo;
    private String     logTotal;
    private String     logMsg;
//-------------------------------------------------------------
// Metodo: LogItem
// Construtor.
//-------------------------------------------------------------
    public LogItem (int lID, Calendario lData, String lTotal,
                    String lMetodo, String lMsg) {
        this.logID     = lID;
        this.logData   = lData;
        this.logTotal  = lTotal;
        this.logMetodo = lMetodo;
        this.logMsg    = lMsg;
    }
```

```
//-------------------------------------------------------------
// Metodo: getLogID
// Retorno: identificador unico da linha.
//-------------------------------------------------------------
    public int getLogID() {
        return logID;
    }
//-------------------------------------------------------------
// Metodo: getLogData
// Retorno: data do log.
//-------------------------------------------------------------
    public String getLogData() {
        return logData.getDia() + "/" + logData.getMes() + "/" +
logData.getAno();
    }
//-------------------------------------------------------------
// Metodo: getLogTotal
// Retorno: o total do calculo que foi realizado.
//-------------------------------------------------------------
    public String getLogTotal() {
        return logTotal;
    }
//-------------------------------------------------------------
// Metodo: getLogMetodo
// Retorno: o nome do método realizado.
//-------------------------------------------------------------
    public String getLogMetodo() {
        return logMetodo;
    }
//-------------------------------------------------------------
// Metodo: getLogMsg
// Retorno: a mensagem de erro ou de boa execucao.
//-------------------------------------------------------------
    public String getLogMsg() {
        return logMsg;
    }
}
```

1.5. Camada Web: *View* e *Controller* para o Componente Beans::CalcEJB

Os arquivos da camada Web para acesso ao componente de negócio base "Beans::CalcEJB" são a página JSP "Calculos.jsp" e o java bean "JBonusCredBean.class", o *view* e o *contoller* respectivamente. Esse acesso é intercedido pelo meta-componente "BeansMeta::CalcEJB", a fim de inserir um mecanismo de contabilização de ações.

1.5.1. Arquivo Calculos.jsp

```
<HTML>
<BODY BGCOLOR = "WHITE">
<HEAD>
<TITLE>Cálculo de Bonus & Crédito </TITLE>
</HEAD>
<CENTER>
<TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPPADING="0">
  <TR>
    <TD ALIGN="CENTER" BGCOLOR="#CCCCCC">
```

```
        <FONT FACE="ARIAL" SIZE="2"><B>CÁLCULO DE BONUS &
 CRÉDITO</B></FACE></TD>
    </TR>
 <TABLE>
 <P>

 <TABLE WIDTH="80%" BORDER="0" CELLSPACING="3" CELLPPADING="0">
    <TR>
      <TD><FONT FACE="arial" SIZE="2">Digite o Fator
 Multiplicador:</FACE></TD>
      <TD><INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT></TD>
    </TR>
    <TR>
      <TD><FONT FACE="arial" SIZE="2">Digite o Fator para
 Crédito:</FACE></TD>
      <TD><INPUT TYPE="TEXT" NAME="Razao"></INPUT></FACE></TD>
    <TR>
    <TR>
      <TD ALIGN="RIGHT"> </TD>
      <TD ALIGN="LEFT">
         <FORM METHOD="GET" ACTION="calculos.jsp">
           <INPUT TYPE="SUBMIT" VALUE="OK">     
           <INPUT TYPE="RESET"  VALUE="Limpar"></TD>
         </FORM>
 </TABLE>

 <jsp:useBean id = "jbonus" class = "JBonusCredBean"/>

 <%! String sMult, sRazao; %>
 <%
   sMult = request.getParameter("MULTIPLIER");
   sRazao = request.getParameter("Razao");
 %>

 <jsp:setProperty name = "jbonus" property="strMult"  value="<%=sMult%>"/>
 <jsp:setProperty name = "jbonus" property="strRazao" value="<%=sRazao%>"/>

 <TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPADING="0">
   <TR>
     <TD ALIGN="CENTER" BGCOLOR="#CCCCCC">
         <FONT FACE="ARIAL" SIZE="2"><B>RESULTADOS</B></FACE></TD>
   </TR>
 <TABLE WIDTH="80%" BORDER="0" CELLSPACING="3" CELLPADING="0">
   <TR>
     <TD WIDTH="50%" ROWSPAN="2" VALIGN="TOP"><FONT FACE="arial" SIZE="2">
         Prezado cliente, você possui no total: </FACE></TD>
     <TD WIDTH="25%" BGCOLOR="#E0E0E0"><FONT FACE="arial" SIZE="2">
         Bonus</FACE></TD>
     <TD WIDTH="25%" BGCOLOR="#E0E0E0" ALIGN="CENTER">
        <FONT FACE="arial" SIZE="2">
        <jsp:getProperty name="jbonus" property="bonusTot"/></FACE></TD>
   </TR>
   <TR>
     <TD WIDTH="25%" BGCOLOR="#E0E0E0"><FONT FACE="arial" SIZE="2">
         Créditos</FACE></TD>
     <TD WIDTH="25%" BGCOLOR="#E0E0E0" ALIGN="CENTER">
        <FONT FACE="arial" SIZE="2">
        <jsp:getProperty name="jbonus" property="creditTot"/></FACE></TD>
   </TR>
 </TABLE>
 <P>
 <TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPADDING="2">
```

```
  <TR>
    <TD><FONT FACE="arial" SIZE="1"><B>Mensagem:</B>
      <jsp:getProperty name = "jbonus" property="mensagem"/></FACE></TD>
  </TR>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

1.5.2. Arquivo JBonusCredBean.java

```
//-------------------------------------------------------------
// Arquivo: JBonusCredBean.java
//-------------------------------------------------------------
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.sql.SQLException;
import java.text.NumberFormat;
import java.text.DecimalFormat;
import BeansMeta.*;
//-------------------------------------------------------------
// CLASSE: JBonusCredBean
//
// Implementacao de SessionBean para confeccao do
// componente de negocio base Beans.CalcEJB.
//-------------------------------------------------------------
public class JBonusCredBean {
  private String strMult, strRazao, mensagem;
  private double bonusTot, creditTot;
  CalcHome       homecalc;
//-------------------------------------------------------------
// Metodo: JBonusCredBean
// Construtor.
//-------------------------------------------------------------
  public JBonusCredBean(){
    try{
      InitialContext ctx = new InitialContext();
      Object objref = ctx.lookup("calcs");
      homecalc = (CalcHome)PortableRemoteObject.narrow(objref,
CalcHome.class);
    } catch (javax.naming.NamingException e) {
      e.printStackTrace();
      this.mensagem = "NamingException: " + e;
    }
  }
//-------------------------------------------------------------
// Metodo: getBonusTot
// Retorno: a propriedade privada bonusTot, formatada
//          como uma String com duas casas decimais.
//-------------------------------------------------------------
  public String getBonusTot(){
    if(strMult != null)
    {
      Integer integerMult   = new Integer(strMult);
      int     multiplicador = integerMult.intValue();

      try {
           double bonus     = 100.00;
           Calc   oCalculo  = homecalc.create();
           double oBonus    = oCalculo.calcBonus(multiplicador, bonus);
           bonusTot         = oBonus;
      }
```

```
            catch (javax.naming.NamingException e) {
              e.printStackTrace();
              this.mensagem = "NamingException: " + e;

          }  catch (javax.ejb.CreateException e) {
              e.printStackTrace();
              this.mensagem = "CreateException: " + e;
          }
              catch (java.rmi.RemoteException e) {
              e.printStackTrace();
              this.mensagem = "RemoteException: " + e;
          }
              catch (java.sql.SQLException e) {
              e.printStackTrace();
              this.mensagem = "SQLException: " + e;

          }
          mensagem = "Operação Ok.";
          return formatTotal (this.bonusTot);

        }
        else
        {
          this.bonusTot = 0;
          this.mensagem = "Valores não informados.";
          return formatTotal (this.bonusTot);
        }
      }

//-------------------------------------------------------------
// Metodo: getCreditTot
// Retorno: a propriedade privada creditTot, formatada
//          como uma String com duas casas decimais.
//-------------------------------------------------------------
  public String getCreditTot(){
    if(strRazao != null)
    {
      Integer integerRazao   = new Integer(strRazao);
      int     razao          = integerRazao.intValue();

      try {
           double bonus     = 100.00;
           Calc   oCalculo  = homecalc.create();
           double oCredito  = oCalculo.calcCredit(razao, bonus);
           creditTot        = oCredito;
      }
         catch (javax.naming.NamingException e) {
           e.printStackTrace();
           this.mensagem = "NamingException: " + e;

      }  catch (javax.ejb.CreateException e) {
           e.printStackTrace();
           this.mensagem = "CreateException: " + e;
      }
           catch (java.rmi.RemoteException e) {
           e.printStackTrace();
           this.mensagem = "RemoteException: " + e;
      }
           catch (java.sql.SQLException e) {
           e.printStackTrace();
           this.mensagem = "SQLException: " + e;
```

```
        }
         mensagem = "Operação Ok.";
         return formatTotal (this.creditTot);
      }
      else
      {
        this.creditTot = 0;
        this.mensagem  = "Valores não informados.";
        return formatTotal (this.creditTot);
      }
    }


 //-----------------------------------------------------------
 // Metodo: getMensagem
 // Retorno: a propriedade privada Mensagem
 //-----------------------------------------------------------
   public String getMensagem(){
     return this.mensagem;
   }
 //-----------------------------------------------------------
 // Metodo: getStrMult
 // Retorno: a propriedade privada strMult
 //-----------------------------------------------------------
   public String getStrMult(){
     return this.strMult;
   }
 //-----------------------------------------------------------
 // Metodo: setStrMult
 // Atualiza a propriedade privada strMult
 //-----------------------------------------------------------
   public void setStrMult(String strMult){
    this.strMult = strMult;
   }
 //-----------------------------------------------------------
 // Metodo: setStrRazao
 // Atualiza a propriedade privada strRazao
 //-----------------------------------------------------------
   public void setStrRazao(String strRazao){
    this.strRazao = strRazao;
   }
 //-----------------------------------------------------------
 // Metodo: formatTotal
 // Formata uma entrada do tipo double em uma string com
 // duas casas decimais.
 // Retorno: String da entrada double
 //-----------------------------------------------------------
  private String formatTotal(double total){

         NumberFormat nf = NumberFormat.getInstance();
         DecimalFormat df = (DecimalFormat)nf;
         df.setMinimumFractionDigits(2);
         df.setMaximumFractionDigits(2);
         String pattern = "##.00";
         df.applyPattern(pattern);
         df.setDecimalSeparatorAlwaysShown(true);
         return df.format(total);
     }
}
```

1.6. Camada Web: *View* e *Controller* para o componente LogPack::LogEJB

Os arquivos da camada Web para acesso ao meta-componente de negócio "LogPack::LogEJB" são a página JSP "Viewer.jsp" e o java bean "JViewerBean.class", o *view* e o *contoller* respectivamente. Esse acesso é feito com o objetivo de visualizar a contabilização das ações realizadas no componente base "Beans::CalcEJB". A contabilização é armazenada em um banco de dados, que por sua vez é acessado pelo componente "LogPack::LogEJB".

1.6.1. Arquivo Viewer.jsp

```
<%@ page import="LogUtil.LogItem" %>
<%@ page import="java.util.Collection" %>
<%@ page import="java.util.Iterator" %>

<HTML>
<BODY BGCOLOR = "WHITE">
<HEAD>
<TITLE>Consulta Tabela de LOG: Cálculo de Bonus & Crédito </TITLE>
</HEAD>
<CENTER>

<TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPPADING="0">
  <TR>
    <TD ALIGN="CENTER" BGCOLOR="#CCCCCC">
      <FONT FACE="ARIAL" SIZE="2">
        <B>CONSULTA TABELA DE LOG<BR>
       - Cálculo de Bonus & Crédito -</B></FACE></TD>
  </TR>
</TABLE>

<FORM METHOD="GET" ACTION="viewer.jsp">
<INPUT TYPE="SUBMIT" VALUE="Consultar a Tabela">
</FORM>

<jsp:useBean id = "jviewer" class = "JViewerBean"/>

<P>
<TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPADING="0">
  <TR>
    <TD ALIGN="CENTER" BGCOLOR="#CCCCCC">
        <FONT FACE="arial" SIZE="2"><B>RESULTADOS</B></FACE></TD>
  <TR>
</TABLE>
<P>
<%
   Collection todosItens = null;
   todosItens = jviewer.getLogTable();
   if (todosItens != null && !todosItens.isEmpty()) {
%>
    <P>
    <TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPADDING="0">
      <TR>
       <TD ALIGN="CENTER" BGCOLOR="#888888">
          <FONT FACE="ARIAL" COLOR="white" SIZE="3"><B>LOGID
</B></FONT></TD>
       <TD ALIGN="CENTER" BGCOLOR="#888888">
```

```
            <FONT FACE="ARIAL" COLOR="white" SIZE="3"><B>Data
</B></FONT></TD>
          <TD ALIGN="CENTER" BGCOLOR="#888888">
            <FONT FACE="ARIAL" COLOR="white" SIZE="3"><B>Total
</B></FONT></TD>
          <TD ALIGN="CENTER" BGCOLOR="#888888">
            <FONT FACE="ARIAL" COLOR="white"
SIZE="3"><B>Metodo</B></FONT></TD>
          <TD ALIGN="CENTER" BGCOLOR="#888888">
            <FONT FACE="ARIAL" COLOR="white" SIZE="3"><B>Erro
</B></FONT></TD>
          <TD ALIGN="CENTER" BGCOLOR="#888888">
            <FONT FACE="ARIAL" COLOR="white"
SIZE="3"><B>Mensagem</B></FONT></TD>
        </TR>
<%
    for (Iterator it = todosItens.iterator(); it.hasNext();){
      LogItem item = (LogItem) it.next();
      // Loop para obter cada item encontrado na tabela de log.
%>

 <TR>
   <TD ALIGN="CENTER"><FONT FACE="ARIAL" SIZE="2"><%= item.getLogID()     %>
</TD>
   <TD ALIGN="CENTER"><FONT FACE="ARIAL" SIZE="2"><%= item.getLogData()   %>
</TD>
   <TD ALIGN="RIGHT" ><FONT FACE="ARIAL" SIZE="2"><%= item.getLogTotal()  %>
</TD>
   <TD ALIGN="CENTER"><FONT FACE="ARIAL" SIZE="2"><%= item.getLogMetodo()%>
</TD>
   <TD ALIGN="CENTER"><FONT FACE="ARIAL" SIZE="2"><%= item.getLogErro()   %>
</TD>
   <TD ALIGN="CENTER"><FONT FACE="ARIAL" SIZE="2"><%= item.getLogMsg()    %>
</TD>
 </TR>
<%
    } //Fim do loop.
%>
    </TABLE>
<% } else { %>
<%-- A busca falhou. --%>
    <FONT SIZE="5" FACE="ARIAL" COLOR="BLUE">
      Tabela de Log vazia.
    </FONT>
<% } %>
<P>
<TABLE WIDTH="80%" BORDER="1" CELLSPACING="0" CELLPADDING="2">
  <TR>
    <TD><FONT FACE="arial" SIZE="1"><B>Mensagem:</B>
        <jsp:getProperty name = "jviewer"
property="mensagem"/></FACE></TD>
  </TR>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

1.6.2. Arquivo JViewerBean.java

```
//-------------------------------------------------------------
// Arquivo: JViewerBean.java
//-------------------------------------------------------------
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;

import LogPack.*;
import LogUtil.*;
//-------------------------------------------------------------
// CLASSE: JViewerBean
//
// Essa classe eh usada como "Controller", intermediando
// a chamada do cliente para a consulta a Tab_Log.
//-------------------------------------------------------------
public class JViewerBean {
    LogHome            homeLog;
    private String     mensagem = new String("Consulta não realizada.");

    private Collection logTable;

    private int        logID;
    private Calendario logData;
    private String     logMetodo;
    private String     logTotal;
    private String     logMsg;
//-------------------------------------------------
// Metodo: JBiewerBean
// Construtor
//-------------------------------------------------
  public JViewerBean(){
    try{
      InitialContext ctx = new InitialContext();
      Object objref = ctx.lookup("logs");

      homeLog = (LogHome)PortableRemoteObject.narrow(objref,
LogHome.class);
    } catch (javax.naming.NamingException e) {
      e.printStackTrace();
    }
  }

//-------------------------------------------------
// Metodo: getLogTot
// Faz uma chamada ao LogEJB, que acessa a tabela
// Tab_Log. Retorna uma java.util.Collection.
// Retorno: colecao com o conteudo de Tab_Log.
//-------------------------------------------------
  public Collection getLogTable() {
    try {
          Log    theLog   = homeLog.create();

          logTable   = theLog.getLog();

        } catch (javax.ejb.CreateException e) {
          e.printStackTrace();
          mensagem = "CreateException";
```

```
        }
            catch (java.rmi.RemoteException e) {
            e.printStackTrace();
            mensagem = "rmi.RemoteException";
          }
            catch (java.sql.SQLException e) {
            e.printStackTrace();
            mensagem = "SQLException";
          }
            mensagem = "Consulta Realizada.";
            return this.logTable;
  }
  //-------------------------------------------------
  // Metodo: getMensagem
  // Retorno: mesagem de erro ou de boa execucao.
  //-------------------------------------------------
    public String getMensagem(){
      return this.mensagem;
    }
  }
```

1.7. *Deployment Descriptors* da Aplicação de Contabilização

Os descritores de implantação, *deployment descriptors*, da Aplicação de Contabilização devem possuir o conteúdo exibido nas subseções a seguir. Os mesmos não precisam ser criados manualmente, ou seja, suas entradas podem ser definidas na ferramenta de implantação embutida na J2EE SDK, a *Application Deployment Tool* (seção 3.2.7), invocada pelo *batch* "C:\%J2EE_HOME%\bin\deploytool.bat".

1.7.1. Application.xml

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN' 'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
<application>
  <display-name>BonusCredEV</display-name>
  <description>Application description</description>
  <module>
    <web>
      <web-uri>war-ic.war</web-uri>
      <context-root>BonusCred</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb-jar-ic.jar</ejb>
  </module>
</application>
```

1.7.2. Ejb-jar.xml

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
  <description>Componentes enterprise beans</description>
  <display-name>EjbTierJar</display-name>
```

```
    <enterprise-beans>
      <session>
        <display-name>CalcBaseEJB</display-name>
        <ejb-name>CalcBaseEJB</ejb-name>
        <home>Beans.CalcHome</home>
        <remote>Beans.Calc</remote>
        <ejb-class>Beans.CalcEJB</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
      </session>
      <session>
        <display-name>LogEJB</display-name>
        <ejb-name>LogEJB</ejb-name>
        <home>LogPack.LogHome</home>
        <remote>LogPack.Log</remote>
        <ejb-class>LogPack.LogEJB</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
        <resource-ref>
    <res-ref-name>jdbc/LogDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
        </resource-ref>
      </session>
      <session>
        <display-name>CalcMetaEJB</display-name>
        <ejb-name>CalcMetaEJB</ejb-name>
        <home>BeansMeta.CalcHome</home>
        <remote>BeansMeta.Calc</remote>
        <ejb-class>BeansMeta.CalcEJB</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
      </session>
    </enterprise-beans>
  </ejb-jar>
```

1.7.3. Sun-j2ee-ri.xml

```
  <?xml version="1.0" encoding="Cp1252"?>
  <j2ee-ri-specific-information>
    <server-name></server-name>
    <rolemapping>
      <role name="gold_customer">
        <groups>
    <group name="gold" />
        </groups>
      </role>
      <role name="customer">
        <principals>
    <principal>
      <name>j2ee</name>
    </principal>
        </principals>
        <groups>
    <group name="cust" />
        </groups>
      </role>
    </rolemapping>
    <web>
      <display-name>WebTierJar</display-name>
      <context-root>BonusCred</context-root>
    </web>
```

```
<enterprise-beans>
  <ejb>
    <ejb-name>CalcBaseEJB</ejb-name>
    <jndi-name>calcsBase</jndi-name>
  </ejb>
  <ejb>
    <ejb-name>LogEJB</ejb-name>
    <jndi-name>logs</jndi-name>
    <resource-ref>
<res-ref-name>jdbc/LogDataSource</res-ref-name>
<jndi-name>jdbc/LogDB</jndi-name>
    </resource-ref>
  </ejb>
  <ejb>
    <ejb-name>CalcMetaEJB</ejb-name>
    <jndi-name>calcs</jndi-name>
  </ejb>
</enterprise-beans>
</j2ee-ri-specific-information>
```

1.7.4. Web.xml

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN' 'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>
<web-app>
  <display-name>WebTierJar</display-name>
  <description>no description</description>
  <servlet>
    <servlet-name>BonusCredJSP</servlet-name>
    <display-name>BonusCredJSP</display-name>
    <description>no description</description>
    <jsp-file>Calculos.jsp</jsp-file>
  </servlet>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
</web-app>
```

1.8. Procedimentos Para Execução da Aplicação de Contabilização

1. A Aplicação de Contabilização executa sobre a plataforma Java™ 2 Enterprise Edition, a J2EE SDK (seção 3.2.3), que deve ser instalada no equipamento a ser utilizado – nesse exemplo, foi adotada a plataforma Windows NT. É assumido que a mesma encontra-se instalada em "C:\J2sdkee1.2.1", que também é o valor da variável de ambiente "%J2EE_HOME%".

2. É assumido que a J2SE (seção 3.2.3), JDK Java padrão, está instalada em "C:\jdk1.3", também o valor da variável de ambiente "%JAVA_HOME%".

3. Deve-se então ir para o diretório "%J2EE_HOME%\bin" e executar o *batch* "cloudscape.bat" que inicia o serviço de banco de dados Cloudscape da seguinte forma: "C:\j2eesdk1.2.1\bin\cloudscape –start".

4. Executar o script "cloudscape.sql", como descrito na seção 1.9 desse Anexo A, a fim de configurar o banco de dados Cloudscape, criando as tabelas a serem utilizadas pela Aplicação de Contabilização.

5. Para configurar a base de dados criada no passo anterior de forma que ela seja acessível pelos serviços da J2EE, acrescentar a seguinte linha na propriedade "jdbc.datasources", definida no arquivo "%J2EE_HOME%\config\default.properties":

```
|jdbc/LogDB|jdbc:cloudscape:rmi:CloudscapeDB;create=true
```

6. Deve-se então Ir para o diretório "%J2EE_HOME%\bin" e executar o *batch* "j2ee.bat", que inicia os serviços padrão da J2EE, da seguinte maneira:

   "C:\ j2eesdk1.2.1\bin\j2ee –verbose".

7. Também do diretório "%J2EE_HOME%\bin", executar o *batch* "deploytool.bat", que inicia a ferramenta de implantação embutida na J2EE, da seguinte maneira: "c:\j2eesdk1.2.1\bin\deploytool".

8. No menu "File" menu, clicar no item "Open Application...", e selecionar o arquivo que contém a Aplicação de Contabilização, por exemplo "C:\ BonusCredEV.ear".

9. No menu "Tools", clicar no item "Deploy Application", e ir pelas janelas de diálogo até completar a implantação. Pode-se escolher usar valores default.

10. Abrir um browse Web e direcioná-lo para o endereço http://localhost:8000/BonusCred/Calculos.jsp, a fim de chamar os métodos do componente de negócio base.

11. Abrir um browse Web e direcioná-lo para o endereço http://localhost:8000/BonusCred/Viewer.jsp, a fim de visualizar a contabilização feita sobre o componente de negócio base da aplicação.

1.9. Configuração do Banco de Dados Cloudscape

O banco de dados utilizado nessa aplicação é o Cloudscape, embutido na J2EE SDK (seção 3.2.3). Mais informações podem ser encontradas no site http://www.cloudscape.com.

Para configurar a base de dados do Cloudscape, de forma que ele aceite as tabelas referenciadas pela aplicação, o arquivo que contém o script "Cloudscape.sql" –

exibido na seção 1.9.1 do Anexo A, a seguir – deve ser chamado no *prompt* do sistema da seguinte maneira:

```
C:\%JAVA_HOME%\bin\java -Dcloudscape.system.home=%J2EE_HOME%\cloudscape
 -classpath %J2EE_HOME%\lib\cloudscape\client.jar;
            %J2EE_HOME%\lib\cloudscape\tools.jar;
            %J2EE_HOME%\lib\cloudscape\cloudscape.jar;
            %J2EE_HOME%\lib\cloudscape\RmiJdbc.jar;
            %J2EE_HOME%\lib\cloudscape\license.jar -ms16m -mx32m
            COM.cloudscape.tools.ij  cloudscape.sql
```

1.9.1. Arquivo Cloudscape.sql

```
connect
'jdbc:rmi://localhost:1099/jdbc:cloudscape:CloudscapeDB;create=true'
;
drop table tab_log
;
drop table sequence
;
create table tab_log (
    logid    int not null,
    data     date not null,
    metodo   varchar(80) not null,
    total    varchar(80) not null,
    msg      varchar(80)  not null
)
;
create table sequence (
    seqnum int not null
)
;
INSERT INTO sequence values (1)
;
```

ANEXO B – PROBLEMA DE CONSENSO DIV

Esse anexo é um resumo do problema de Consenso DIV apresentado em [DÉFANO et al., 1998].

1.1. Clientes e Servidores

Um único serviço replicado é considerado. Os processos que implementam esse serviço são chamados de "processos servidores", e os que enviam invocações são chamados de "processos cliente". O conjunto de processos servidores é denotado por Ω_s , e o conjunto de processos clientes é denotado por Ω_c. É assumido uma maioria de processos corretos em Ω_s. O número de processos no conjunto de servidores é denotado por $n = | \Omega_s |$.

1.2. Consenso com Valores Iniciais Protelados

Essa seção introduz a definição de Consenso com Valores Iniciais Protelados, ou Consenso DIV (*Deferred Initial Values*), usado no algoritmo de replicação semi-passiva.

1.2.1. O Problema

Na definição clássica do problema de consenso, cada processo p_i começa com um valor inicial v_i, e o processo tem que concordar com um valor de decisão comum. No consenso DIV, um processo p_i não precisa ter seu valor inicial definido no início do algoritmo de consenso – o algoritmo pode requisitar um valor inicial de p_i em um tempo posterior.

1.2.2. Obtendo Valores Iniciais

A interface de um algoritmo de consenso é expressa como uma função *propose* (v_i), onde o argumento v_i é o valor inicial do processo chamador da função *propose*. Isso não é adequado para o consenso com valores iniciais protelados. Ao invés, é proposto expressar a interface do algoritmo de consenso DIV como uma função *DIVpropose* (*getInitialVal*), com a função *getInitialVal* como argumento. A função *getInitialVal* retorna um valor inicial e é chamada durante a execução do algoritmo de consenso, quando um valor inicial é necessário.

Quando se resolve o consenso DIV usando o paradigma do coordenador rotativo[16], a função *getInitialVal* pode ser chamada por algum processo p_i apenas quando p_i é o coordenador. Portanto, na ausência de falhas, apenas um processo chama a função *getInitialValue*.

1.2.3. Resolvendo o Consenso DIV

O algoritmo requer uma maioria de processos corretos. Em boas execuções, apenas o coordenador do primeiro *round* (processo p_1) chama a função *getInitialVal*, e o problema de consenso DIV é resolvido no primeiro *round* (Fig. A.1). Com uma ocorrência de *crash* (e nenhuma suspeita incorreta), dois *rounds* são necessários para resolver o problema: o processo p_1, coordenador do primeiro *round*, chama a *getInitialVal* e sofre *crash*. Os processos sobreviventes movem-se para o segundo *round*. O processo p_2, coordenador do segundo *round*, por sua vez chama a *getInitialVal*.

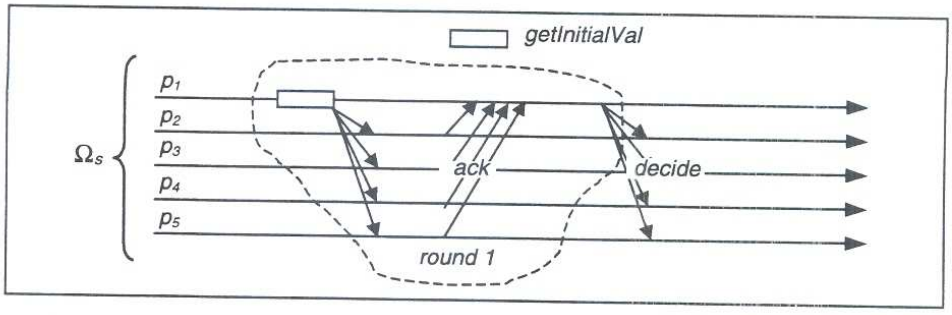

Fig. A.1. Consenso DIV (boa execução).

1.3. Um Seqüência de Consensos DIV

Na solução para replicação semi-passiva baseada em um seqüência de consensos DIV, cada problema de consenso decide sobre a mensagem de atualização enviada pela réplica primário para os *backups*. Isso requer que o valor inicial de cada problema de consenso seja uma um valor de atualização. Considerando o problema de consenso número *k*. como o valor inicial de consenso *k* é uma valor de atualização, um processo servidor pode ter um valor inicial para consenso *k* apenas depois de ter manipulado uma nova requisição recebida de um cliente. Contudo, apenas o primário manipula as requisições de cliente na técnica de replicação semi-passiva, ou apenas o primário tem o valor inicial para consenso *k*. Dessa forma não há problemas, contanto que o primário não sofra *crash*, nem seja suspeito de ter sofrido *crash*.

---

[16] No paradigma do coordenador rotativo, o algoritmo passa por uma sucessão de *rounds*, e em cada *round*, um processo diferente é o coordenador.

Por conseguinte, o primário impõe seu valor inicial como valor de decisão, e portanto os *backups* não necessitam ter uma valor inicial de consenso *k*. Contudo, se o primário sofre *crash*, um dos backups tem que manipular a requisição para obter o valor inicial para consenso número *k*. Esse processamento acontece após o início do consenso número *k*, e explica a necessidade por valores iniciais protelados no contexto do problema de consenso.

1.3.1. Notação

A fim de expressar o algoritmo da replicação semi-passiva, é apresentada a seguinte notação:

- req^{j}_{c} : requisição j^{th} (j-ésima) enviada por um cliente *c*.
- upd^{i}_{s} : mensagem de atualização número *i*, gerada pelo servidor *s*.
- res^{i}_{s} : resposta número *i* para um cliente, gerada pelo servidor *s*.
- $state^{i}_{s}$: estado do servidor *s*, após o processamento de upd^{i}_{s}.
- $handle_s$: $(req^{j}_{c}, state^{i-1}_{s}) \rightarrow (upd^{i}_{s}, res^{i}_{s})$: transformação de uma requisição req^{j}_{c} em uma mensagem de atualização (manipulação da requisição).
- *apply* : $(upd^{i}_{s}, state^{i-1}_{s}) \rightarrow state^{i}_{s}$: modificação do estado devido à aplicação de uma mensagem de atualização (possivelmente $s = s'$).

1.3.2. Algortimo Completo

É fornecido o algoritmo completo, que expressa a técnica de replicação semi-passiva como uma seqüência de consenso DIV.

O algoritmo, executado por todos os processos servidores, é dado na Fig. A.2. Cada servidor *s* gerencia um inteiro k_s (linha 4), que identifica a instância corrente do problema de consenso. Cada processo servidor também manipula as variáveis $recv_s$ e $hand_s$ (linha 2, 3):

- $recv_s$ é um conjunto contendo as requisições recebidas por *s* dos clientes. Sempre que *s* recebe uma nova requisição, ele a inclui em $recv_s$ (linhas 5, 6);
- $hand_s$ é um conjunto que consiste das requisições que foram manipuladas (não necessariamente por *s*). Um novo consenso é iniciado por *s* sempre que o consenso anterior for terminado, e $recv_s$ - $hand_s$ não é vazio (linha 11). No fim do consenso, a requisição que foi manipulada é inserida por *s* em $hand_s$ (linha 16).

A função *getInitVal* é discutida no próximo parágrafo. A parte principal do algoritmo consiste das linhas 11 – 16: na linha 13, a função de consenso DIV é chamada e retorna a decisão (req_c, $upd^{k\,s}_{s'}$, $res^{k\,s}_{s'}$):

- req_c é a requisição que foi manipulada.
- $upd^{k\,s}_{s'}$ é a atualização resultante da manipulação de s' sobre req_c.
- $res^{k\,s}_{s'}$ é a resposta que deveria ser enviada ao cliente c.

```
1: Inicialização:
2:      recv_s ← 0                          { mantém as requisições recebidas }
3:      hand_s ← 0                          { mantém as requisições manipuladas }
4:      k_s    ← 0

5: when receive (req^j_c) from client
6:      recv_s ← recv_s ∪ { req^j_c }

7: function getInitVal ()
8:      req_cs ← [ select one request in (recv_s - hand_s) ]
9:      ( upd^{k s}_s , res^{k s}_s ) ← handle_s (req_cs , state^{ks - 1}_s )
10:     return ( req_cs , upd^{k s}_s , res^{k s}_s )

11: when recv_s - hand_s ≠ 0
12:     k_s ← k_s + 1                       { resolve o k_s^th consenso DIV }
13:     ( req_cs , upd^{k s}_{s'} , res^{k s}_{s'} ) DIVpropose ( k_s , getInitVal)
14:     send (res^{k s}_{s'}) to c
15:     state^{k s}_s ← apply (upd^{k s}_{s'} , state^{k s - 1}_s )   { atualiza o estado }
16:     hand_s ← hand_s ∪ { req_c }
```

Fig. A.2. Replicação Semi-passiva (réplica *s*).

Na linha 14, a resposta $res^{k\,s}_{s'}$ é enviada ao cliente. Na linha 15, o estado local do servidor s é atualizado de acordo com a mensagem de atualização $upd^{k\,s}_{s'}$. Finalmente, na linha 16, a requisição que foi manipulada é inserida no conjunto $hand_s$.

A função *getInitVal* retorna valores iniciais para cada instância do problema de consenso. Ela é chamada pelo algoritmo de consenso DIV sempre que um processo é o coordenador, i.e., atua como primário no contexto da técnica de replicação semi-passiva. A função seleciona uma requisição de cliente que não foi manipulada ainda (linha 8), manipula a requisição (linha 9) e retorna a requisição selecionada req_{cs}, a mensagem de atualização $upd^{k\,s}_{s}$, resultante da manipulação da requisição, bem como a mensagem de resposta correspondente $res^{k\,s}_{s}$ (linha 10).

ANEXO C – SERVIÇO DE MENSAGENS ADOTADO PELA J2EE

O serviço de mensagens adotado pela J2EE é o definido pelo JMS (Java *Message Service*), especificado em [SUN, 1999d], que descreve uma comunicação assíncrona entre aplicações corporativas.

1.1. Características

Genericamente, um sistema de mensagens provê facilidades para criação, envio e recebimento de mensagens, definindo uma forma particular de prover tais funcionalidades. A especificação do JMS, por sua vez, provê a união de todas as características dos sistemas de mensagem existentes. Não atende, portanto, a todas as características desses sistemas de mensagens, porém admite as funcionalidades necessárias para implementar sofisticadas aplicações corporativas, maximizando a portabilidade de aplicações que utilizem a API JMS (Java *Messaging Service API*) para o envio e recebimento de mensagens.

Assim, a API JMS visa atender ao serviço de mensagens fornecido por um Provedor JMS, que por vez é uma entidade que implementa o JMS para um produto de mensagens existente.

1.1.1. Não Incluído no JMS

Dentre as atribuições que o JMS não atende, estão [SUN, 1999d]:

- Balanceamento de Carga e Tolerância a Falha: a API JMS não especifica como clientes que cooperem entre si via mensagens apareçam como um único serviço.
- Administração: o JMS não define facilidades para administrar o serviço de mensagens em questão; tal incumbência é realizada via API proprietária de tal serviço.
- Segurança: o JMS não é especifica uma API para controlar a privacidade e integridade das mensagens. Também não é especificado como as assinaturas digitais ou chaves são distribuídas aos cliente. A segurança é uma característica considerada específica do Provedor JMS, e deve portanto ser configurada pelo administrador do serviço via API proprietária – e não através da API JMS pelo clientes.

- Repositório de Tipo de Mensagem: o JMS não define um repositório para armazenar definições de tipo de mensagens, e não define uma linguagem para criar definições de tipos de mensagens.

1.1.2. Domínios JMS

O produtos de mensagens podem ser classificados genericamente de sistemas *Point-to-Point* (PTP) ou *Publisher-Subscribers* (Pub/Sub), como descrito a seguir [SUN, 1999d]:

- *Point-to-Pointl* (PTP): produtos PTP, ou ponto-a-ponto, são construídos sobre conceitos de filas (*queue*) de mensagens. Cada mensagem é endereçada a uma fila específica, e os clientes extraem as mensagens da(s) fila(s) estabelecida(s) para manter sua próprias mensagens.
- *Publish-Subscribe* (Pub/Sub): os clientes Pub/Sub, ou Publicar/Assinar, endereçam mensagens a um nó em uma hierarquia de conteúdo. Os *publishers* (publicadores) e os *subscribers* (assinantes) são geralmente anônimos e podem dinamicamente publicar ou assinar à uma hierarquia de conteúdo. O sistema de mensagens é que se encarrega de distribuir as mensagens que chegam de um nó de múltiplos *publishers* para múlitplos *subscribers*.

1.2. Arquitetura

Essa seção descreve o ambiente de aplicações baseadas em mensagem e o papel que o JMS representa nesse ambiente.

1.2.1. Aplicação JMS

Uma aplicação JMS é composta das seguintes partes [SUN, 1999d]:

- Clientes JMS: são programas escritos em Java que enviam e recebem mensagens via API JMS.
- Clientes Não-JMS: são programas que usam as APIs nativas do sistema de mensagem em questão, ao invés da API JMS.
- Mensagens: cada aplicação define um conjunto de mensagens que são usadas para comunicação entre clientes.
- Provedor JMS: sistema de mensagens existente que também implementa o JMS, além das funcionalidades de controle e administração necessárias a um serviço de mensagem completamente caracterizado.

1.2.2. Administração

Para prover portabilidade entre diferentes sistemas de mensagens (diferentes tecnologias de mensagens), os clientes JMS são isolados dessas particularidades pelos "Objetos Administraods JMS", criados e customizados pelo administrador do sistema de mensagens, e depois disponibilizados para uso dos clientes. Os clientes então os utilizam através das interfaces JMS, que são portáveis. O administrador cria os "Objetos Administrados JMS" através das facilidades específicas do serviço de mensagem em questão.

Existem dois tipos de "Objetos Administrados JMS" [SUN, 1999d]:

- *Connection Factory*: objeto usado pelo cliente para criar uma conexão com o provedor JMS;
- *Destination*: objeto usado pelo cliente para o destino da mensagem que está enviando, e a fonte da mensagem que está recebendo.

Os "Objetos Administrados JMS" são colocados em um *namespace* JNDI pelo administrador.

1.2.3. Interfaces JMS

o JMS é baseado em um conjunto de conceitos de mensagens comuns. Cada domínio de mensagem JMS – PTP ou Pub/Sub – define seu próprio conjunto customizado de interfaces para esses conceitos, como exibido na Tab. A.1 a seguir.

JMS Parent	Domínio PTP	Domínio Pub / Sub
ConncetionFactory	QueueConncetionFactory	TopicConncetionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

Tab. A.1. JMS - Relacionamento entre as interfaces PTP e Pub/Sub.

Segue uma breve definição desses conceitos JMS:

- *ConncetionFactory*: fábrica de conexão, é um objeto administrado (recebe JNDI) usado pelo cliente para criar uma conexão.
- *Connection*: é uma conexão ativa para um provedor JMS.
- *Destination*: encapsula a identidade de destino de mensagem.

- *Session*: sessão, representa um contexto único de *thread* para envio e recebimento de mensagem.
- *MessageProducer*: produtor de mensagem, é um objeto criado por um *Session* (sessão), usado para enviar mensagens para um destino.
- *MessageConsumer*: consumidor de mensagem, é um objeto criado para receber mensagens enviadas a um destino.

1.2.4. Desenvolvendo um Cliente JMS

Um cliente JMS típico executa o seguinte procedimento de *setup* do JMS:

1. Usa o serviço de JNDI para encontrar um "Objeto Administrado JMS" do tipo *ConnectionFactory*.
2. Usa o serviço de JNDI para encontrar um ou mais "Objetos Administrados JMS" do tipo *Destination*.
3. Usa a *ConnectionFactory* obtida no passo 1 para criar um objeto do tipo *Connection*, que faz uma conexão com o Provedor JMS.
4. Usa o objeto *Connection* obtido no passo 3 para criar um ou mais objetos do tipo *Session*, que habilita uma sessão com o Provedor JMS.
5. Usa os objetos *Session* e *Destination*, criados nos passos 4 e 2 respectivamente, para criar os objetos *MessageProducer* e *MessageConsumer* necessários.
6. Avisa ao objeto *Connection* criado no passo 3 para começar a entrega de mensagens.

1.3. Código Exemplo – Domínio Pub / Sub

O código exibido nas subseções a seguir mostra como enviar e receber mensagens via Domínio Pub / Sub.a

1.3.1. Cliente JMS – *Destionation* tipo *Topic*

```
//-----------------------------------------------------
// Criando uma  TopicConnectionFactory
//-----------------------------------------------------
   TopicConnectionFactory topicConnectionFactory;
   Context messaging = new InitialContext();
   topicConnectionFactory = (TopicConnectionFactory)
   messaging.lookup("TopicConnectionFactory");
```

```
//-------------------------------------------------------
// Criando um objeto administrado "Destination" tipo Topic
//-------------------------------------------------------
   Topic stockTopic;
   stockTopic = (Topic) messaging.lookup("StockTopic");
//-------------------------------------------------------
// Criando uma TopicConnection através de uma TopicConnectionFactory
//-------------------------------------------------------
   TopicConnection topicConnection;
   topicConnection = topicConnectionFactory.createTopicConnection();
//-------------------------------------------------------
// Criando uma TopicSession através de uma TopicConection
//-------------------------------------------------------
   TopicSession session;
   session = topicConnection.createTopicSession(false,
                session.CLIENT_ACKNOWLEDGE);
//-------------------------------------------------------
// Criando um TopicSubscriber (assinante) para o Topic (stockTopic)
//-------------------------------------------------------
   TopicSubscriber subscriber;
   subscriber = session.createSubscriber(stockTopic);

   public class StockListener implements javax.jms.MessageListener {
      void onMessage(Message message) {
            // desempacotando e manipulando as mensagens recebidas.
      }
   }
   StockListener myListener;
   subscriber.setMessageListener(myListener);
//-------------------------------------------------------
// Criando um TopicPublisher
//-------------------------------------------------------
   TopicPublisher publisher;
   publisher = session.createPublisher(stockTopic);
//-------------------------------------------------------
// Iniciando a entrega de mensagens
//-------------------------------------------------------
   topicConnection.start();
```

1.3.2. Mensagem do tipo *MapMessage*

```
//-------------------------------------------------------
// Manipulando uma mensagem do tipo mapeada: MapMessage
//-------------------------------------------------------
   String stockNome;    // o nome da parcela do fundo.
   double stockValor;   // o valor corrente do fundo.
   long   stockTempo;   // o tempo da parcela do fundo.
   double stockDif;     // a variação +/- da parcela do fundo.
   String stockInfo;    // informação sobre o fundo.
   MapMessage message;
   message = session.createMapMessage();
//-------------------------------------------------------
// O estabelecimento (setting) dos campos da mensagem
// pode ser feito em qualquer ordem.
//-------------------------------------------------------
   message.setString("Nome", stockNome);
   message.setDouble("Valor", stockValor);
   message.setLong  ("Tempo", stockTempo);
   message.setDouble("Dif", stockDif);
   message.setString("Info", stockInfo);
```

GLOSSÁRIO

.ear — Arquivo .jar que contém uma aplicação J2EE.

.jar — Java *Archive*: padrão Java que permite a compressão e o armazenamento de um conjunto de arquivos relacionados

.war — Arquivo que contém um módulo Web em uma aplicação J2EE.

Aplicação J2EE — Qualquer unidade onde é possível implantar a funcionalidade J2EE. Pode ser um único (ou múltiplos) bloco(s) empacotado(s) em um arquivo .ear, que é associado com um *deployment descriptor* (descritor de implantação). As aplicações J2EE são tipicamente projetadas para serem distribuídas através de múltiplas camadas de computação.

Applet — Componente que tipicamente executa em um browser Web, mas não pode executar em uma série de outras aplicações ou dispositivos que suportem o modelo de programação applet - no lado servidor, por exemplo.

Camada — É uma parte bem definida do sistema, delimitada por pacotes ou subsistemas.

Componente — Unidade de software a nível de aplicação, suportada por um *container*. São passíveis de configuração em tempo de implantação. A plataforma J2EE define 4 tipos de componentes: os enterprise beans, os componentes Web, os applets e as aplicações clientes.

Componente JavaBeans™ — Ou apenas bean. Classe Java que pode ser manipulada em uma ferramenta de desenvolvimento visual, e composta dentro de aplicações. Um bean deve aderir a certas propriedades e convenções de interface de eventos.

Componente Web — Um componente que provê serviços em resposta a requisições; pode ser tanto um servlet como um JSP.

***Container* EJB** — *Container* que implementa o contrato do componente EJB da arquitetura J2EE. Esse contrato especifica um ambiente de execução para enterprise beans que inclui segurança, concorrência, transação, implantação, entre outros serviços. Um *container* EJB é provido por um servidor EJB ou por um servidor J2EE.

***Container* Web** — Entidade que implementa o contrato de componente Web da arquitetura J2EE, o qual especifica um ambiente de execução para componentes Web que inclui segurança, concorrência, transação, entre outros serviços. Provê os mesmos serviços que o *container* JSP e uma visão federativa das APIs da plataforma J2EE. É provido por um servidor Web ou um servidor J2EE.

CORBA — Common Object Request Broker Architecture: modelo de objeto distribuído e independente de linguagem, especificado pelo Object Management Group (OMG).

Deployment Descriptor	É o descritor de implantação que define a informação estrutural do enterprise bean e suas dependências externas. A Ferramenta de Implantação de Aplicação (*Application Deployment Tool*) constrói o *deployment descriptor* quando empacota o enterprise bean.
EIS	*Enterprise Information Systems*. Ou Sistemas de Informação Corporativos. Gerlamente são relacionados aos sistemas corporativos legados, compreendendo SGBDs, sistemas ERP, entre outros.
***Entity* Bean**	É um enterprise bean identificado unicamente por uma chave primária e representa uma entidade mantida em um mecanismo de armazenamento persistente, como um banco de dados. Pode representar contas, clientes ou produtos armazenados no banco de dados. É, portanto, persistente e pode ser compartilhado por vários clientes.
Enterprise bean	Ou Enterprise JavaBean™. Um componente que implementa uma tarefa de negócio, ou entidade de negócio, e reside em um *container* EJB; pode ser tanto um *entity* bean ou um *session* bean.
Falha	É o desvio de um serviço entregue, mediante à especificação do sistema. Transição da entrega adequada de serviço para a entrega inadequada de serviço.
Falha por *Crash*	É um tipo de falha onde um elemento de um sistema de informação não responde a mais nenhuma requisição subsequente do cliente.
HTML	*HyperText Markup Language*: linguagem de marcação para documentos de hipertexto da Internet. Permite que certos objetos, como imagens, som, vídeo, campos de formulário, referência via URL, sejam embutidos no documento, além de permitir também uma formatação básica de texto.
HTTP	*HyperText Transfer Protocol*: protocolo da Internet, usado para trazer objetos de hipertexto de hosts remotos. As mensagens de HTTP consistem de requisições do cliente para o servidor, e de respostas do servidor para o cliente.
HTTPS	*Secure Hypertext Transfer Protocol*: protocolo HTTP sobre o protoclo SSL.
IDL	*Interface Definition Language*: linguagem usada para definir interfaces para objetos CORBA remotos.
IIOP	*Internet Inter-ORB Protocol*: um protocolo usado para comunicação entre ORBs CORBA.
Interface - *Home*	É a interface do enterprise bean que define os métodos que permitem ao cliente criar, localizar ou remover um enterprise bean. Para beans tipo *session*, a interface Home define os métodos criar e remover, e para os beans tipos *entity*, define os métodos criar, localizar e remover.

Interface -*Remote*	É a interface do enterprise bean que define os métodos de negócio que o cliente pode chamar.
JAF	JavaBeans *Active Framework*: provê serviços padrão para determinar qual o tipo de uma parte arbitrária de dado, e ativa um componente JavaBeans apropriado para manipular tal dado.
Java™ 2 Platform, Standard Edition (J2SE platform)	O núcleo da plataforma de tecnologia Java (*the Java Core Classes*).
Java™ 2 Platform, Enterprise Edition (J2EE platform)	É um ambiente para desenvolvimento e implantação de aplicações corporativas. A plataforma J2EE consiste de um conjunto de serviços, APIs Enterprise Java e protocolos que provêem funcionalidades para desenvolvimento de aplicações corporativas baseadas em Web e em multi-camada.
Java™ 2 SDK, Enterprise Edition	Implementação da Sun Microsystems para a plataforma J2EE. Essa implementação provê uma definição operacional da plataforma J2EE.
Java IDL	Java *Interface Definition Language*: cria interfaces remotas para suportar comunicação CORBA na plataforma Java. Inclui o compilador IDL-to-Java e um ORB lightweight que suporta IIOP.
JavaMail™	Uma API para enviar e receber correio.
JDBC	Java *Database Connectivity*: API para conectividade independente de banco de dados, entre a plataforma J2EE e diferentes tipos de banco de dados.
JMS	Java *Messaging Service*: API para utilização de sistemas de mensagens corporativos como o IBM MQ Series, TIBCO Rendezvous, entre outros.
JNDI	Java *Naming and Directory Interface*: API que provê funcionalidade de nome e diretório.
JSP	JavaServer Pages™: Tecnologia que expande a Web; que usa modelos de dados (HTML ou XML), elementos customizados, linguagens de *script* e objetos Java do lado servidor para retornar conteúdo dinâmico a um cliente (browser Web).
JTA	Java *Transaction* API: provê uma API de demarcação de transação.
JTS	Java *Transaction Service*: define um serviço de gerenciamento de transação distribuído baseado no CORBA OTS.
Meta-programação	É a arte de desenvolver métodos e programas que lêem, manipulam e/ou escrevem outros programas, através da separação dos mecanismos de gerenciamento do programa da sua funcionalidade.

OLTP — *Online Transaction Processing*: é uma classe de programas que facilita e gerencia aplicações orientadas a transação (tipicamente para entrada de dados e recuperação de transações), com o objetivo de manter a consistência da informação em bancos de dados.

ORB — *Object Request Broker*: uma biblioteca que habilita objetos CORBA a localizar e comunicar-se uns com os outros.

OTS — *Object Transaction Service*: uma definição de interfaces que permitem que objetos CORBA participem de transações.

Pattern — Uma solução comum para um problema freqüente em um dado contexto; especifica um conhecimento coletado das experiências em um certo domínio. Um sistema bem estruturado é repleto de *patterns*: para arquitetura, idioma, projeto, etc. Ele auxilia no entendimento, na especificação, na construção e na documentação de um sistema.

***Pattern* MVC** — *Pattern* Model-View-Controller, ou Modelo-Visualização-Controlador. É uma abstração que visa auxiliar o processo de decompor uma aplicação em componentes lógicos que podem ser arquitetados mais facilmente.

Produto J2EE — Qualquer produto em concordância com a especificação da plataforma J2EE.

Reflexão — Propriedade de programas que atuam sobre si mesmos, permitindo que os programa s mantenham informação sobre si, bem como alterem seu comportamente usando tais informações.

Reificação — *Reification*. Ou materialização. É o processo de converter algo que estava previamente implícito, ou não expresso, em algo explicitamente formulado, que é disponibilizado para manipulação conceitual lógica ou computacional.

RMI — *Remote Method Invocation*: tecnologia que permite que um objeto executado em uma JVM invoque métodos em um objetos que esteja executanto uma outra JVM.

RMI-IIOP — Uma versão da RMI implementada para usar o protocolo CORBA IIOP. RMI sobre IIOP provê interoperabilidade com objetos CORBA implementados em qualquer linguagem se todas as interfaces remotas forem originalmente definidas como interfaces RMI.

Servidor EJB — Software que provê serviços para um *container* EJB. Por exemplo, um *container* EJB tipicamente baseia-se em um gerenciador de transação, que é parte de um servidor EJB, para executar o *two-phase commit* através de todos os gerenciadores de recursos participantes da transação. A arquitetura J2EE assume que *containers* EJB são hospedados em servidores EJB do mesmo fornecedor de software, portanto não especifica o contrato entre essas duas entidades. Um servidor EJB pode hospedar um ou mais *containers* EJB.

Servlets Um programa Java que estende a funcionalidade de um servidor Web, gerando conteúdo dinâmico e interagindo com clientes Web, através do paradigma requisição-resposta.

***Session* Bean** É um enterprise bean criado para um cliente (geralmente existe apenas durante uma sessão cliente / servidor). Pode realizar operações para o cliente (cálculos ou acesso a bancos de dados, etc.). Não é recuperado em caso de falha no sistema. Pode não ter estado (*stateless*) ou manter um estado conversacional através de métodos e transações. Se contiver estado, o *container* EJB gerencia esse estado, no caso do objeto ter que ser removido da memória. No entanto, o mesmo deve gerenciar seus dados persistentes.

SSL *Secure Socket Layer*: protocolo de segurança que provê privacidade nas comunicações feitas na Internet. O protocolo permite que aplicações cliente-servidor comuniquem-se de uma forma que não possam sofrer escuta às escondidas, nem interferências. Os servidores são sempre autenticados; os clientes são opcionalmente autenticados.

Tolerância a Falha Métodos e técnicas que objetivam prover um serviço adequado, concordante com suas especificações, a despeito da ocorrência de falhas.

UML *Unified Modeling Language*: é uma linguagem de modelagem padrão para software – uma linguagem para visualizar, especificar, construir e documentar os artefatos produzidos durante o processo de desenvolvimento de um sistema de software.

URL *Uniform Resource Locator*: padrão para escrever uma referência textual de uma parte de um dado na World Wide Web. Seu formato é do tipo "protocol://host/localinfo"

XML *Extensible Markup Language*: linguagem de marcação que permite definir as *tags* (marcações) necessárias à identificação de dados e texto em documentos XML. Os *deployment descriptors* (descritores de implantação) J2EE são expressos em XML.

REFERÊNCIAS BIBLIOGRÁFICAS

[BOOCH et al., 2000] BOOCH, Gary, JACOBSON, Ivar, RUMBAUGH, James. **UML – Guia do Usuário**. Ed. Campos. Rio de Janeiro, 2000.

[BRAY et al., 1998] BRAY, Tim, PAOLI, Jean, SPERBERG-McQUEEN, C. M. **Extensilble Markup Language (XML) 1.0**. W3c Recommendation. Fev. 1998. [online] http://www.w3.org/TR/1998/REC-xml-19980210

[CHAPPEL, 1996] CHAPPEL, David. **Understanding ActiveX and OLE**. Microsoft Press, EUA, 1996.

[CORNELL & HORSTMANN, 1997] CORNELL, Gary, HORSTMANN, Cay S. **Core Java**. Makron Books. SP, 1997.

[DÉFANO et al., 1998] DÉFANO, X., SCHIPER, A., SEGENT, N. **Semi-Passive Replication**. Proceedings of the 17th IEEE International Symposium on Reliable Distributed Systems, 1998.

[DeLOTTINVILLE, 1994] DeLOTTINVILLE, Paul. **Open OLTP: To Monitor or Not To Monitor**, Datamation, EUA, p. 59-61, Nov. 1994.

[FABRE et al., 1995] FABRE, J., NICOMETTE, V., et Al. **Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming**. Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena, CA, EUA, Jun. 1995.

[GAMA et al., 1995] GAMA, E., HELM, R. et Al. **Design Patterns – Elements of Reusable Object-Oriented Software**. .Addison-Wesley Longman Inc., EUA, 1995.

[GOLM & KLEINÖDER, 1998] GOLM, Michael, KLEINÖDER, Jürgen. **metaXa and the Future of Refletion**. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canadá, Out. 1998.

[GUERRAOUI & SHIPER, 1996] GUERRAOUI, R., SCHIPER, A. **Fault-Tolerance by Replication in Distributed Systems**. International Conference on Reliable Software Technologies, Springer Verlag (LNCS), 1996.

[JACOBSON et al., 1999] JACOBSON, Ivar, BOOCH, Gary, RUNBAUGH, James. **The Unified Software Development Process**. Addison-Wesley Longman Inc., EUA, 1999.

[KICZALES et al., 1993] KICZALES, Gregor, ASHLEY, J. Michael, et Al. **Metaobject Protocols: Why We Want Them, and What Else They Can Do**, publicado no *Object-Oriented Programming: The CLOS Prospective*, págs. 101-118, Andreas Paepcke, Ed., MIT Press, Cambridge, MA, EUA, 1993.

[KRAMER, 1996] KRAMER, Mitchell I.. **Microsoft Transaction Server: A General Purpose Intrastruture for Multitier Applications**. Patricia Seybold Group. EUA, Nov. 1996.

[LAPRIE, 1990], LAPRIE, J. C. **Dependability: Basic Concepts and Associated Terminology**. LAAS-CNRS, Toulouse, 1990.

[LAU, 1996] LAU Cheuk Lung. **Implementação de Técnicas de Replicação de Componentes de Software sobre Plataforma Aberta CORBA**. Dissertação submetida à UFSC para obtenção de grau de Mestre em Engenharia Elétrica. Florianópolis, Mai. 1996.

[LAU et al., 1999] LAU Cheuk Lung, Fraga, Joni da Silva, FARINES, Jean-Marie. **CosNamingFT – Um Serviço de Nomes Tolerntes a Fahas em Conformidade com o Padrão OMG**. Anais do XVII Simpósio Brasileiro de Redes de Computadores, p. 549-564. UFBA, Salvador, Mai. 1999.

[LAU & MAZIERO, 1999] LAU Cheuk Lung, MAZIERO, Carlos Alberto. **MetaFT – A Reflective Approach Using Group Support to Implement Replication Tecniques in CORBA**. XIX IEEE International Conference of the Chilean Computer Science Society (SCCC´99), Talca, Chile, Nov. 1999.

[LEMAY et al., 1996] LEMEY, Laura, PERKINS, Charles L., MORRISON, Michael. **Teach Yourself Java in 21 Days – Professional Reference Edition**. Ed. Sams, EUA, Nov. 1996.

[MAES, 1987] MAES, Pattie. **Concepts and Experiments in Computacional Reflection**. OOPSLA'87 Proceedings, EUA, Out. 1987.

[MELEWS, 2000] MELEWS, Deborah. **CORBA and EJB Team UP in Data Center**. Application Development Trends Magazine, EUA, Jul. 2000 [online] http://www.adtmag\Pub\article.asp?ArticleID=881.

[MICROSOFT, 2000a] MICROSOFT CORPORATION. **Visual Studio.NET**. EUA, Novez. 2000. [online] http://msdn.microsoft.com/vstudio.

[MICROSOFT, 2000b] MICROSOFT CORPORATION. **Visual Studio for Applications (VSA) "Customize Distributed Applications" – White Paper**. EUA, Dez. 2000. [online] http://msdn.microsoft.com/vstudio/vsa/vsaindepth.doc.

[MICROSOFT, 2001] MICROSOFT CORPORATION. **Visual Studio for Applications Frequently Asked Questions**. EUA, Jan. 2001. [online] http://msdn.microsoft.com/vstudio/vsa/qa.asp.

[OLIVA, 1998] OLIVA, Alexandre. **Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java™**. Dissertação submetida à UNICAMP para obtenção de grau de Mestre em Ciência da Computação. Campinas, Ago. 1998.

[OLIVEIRA et al., 1998] OLIVEIRA, R. P., NASCIMENTO, M. E., MELO, A. C. **Replicação de Objetos Utilizando o Padrão CORBA**. II Workshop em Sistemas Distribuídos - WoSiD'98, p.17-24, PUC-PR, Curitiba, Jun. 1998.

[OMG, 1999] OBJECT MANAGEMENT GROUP. **CORBA Components**. OMG Document, *orbos/99-02-05*, EUA, Fev. 1999.

[ORFALI & HARKEY, 1996] ORFALI, Robert, HARKEY, Dan, EDWARDS, Jeri. **The Essential Distributed Objects Survivel Guide**. John Wiley & Sons Inc. EUA, 1996.

[ORFALI & HARKEY, 1998] ORFALI, Robert, HARKEY, Dan. **Client/Server Programming with Java and CORBA - Second Edition**. John Wiley & Sons Inc. EUA, 1998.

[RUMBAUGH et al., 1994] RUMBAUGH, James, BLAHA, Michael, et Al. **Modelagem e Projetos Baseados em Objetos**. Ed. Campos. Rio de Janeiro, 1994.

[SOBEL & FRIEDMAN, 1998] SOBEL, Jonathan, FRIEDMAN, Daniel P. **An Introduction to Reflection-Oriented Programming**. Reflection'96, San Francisco, CA, EUA, Abr. 1996. [online] http://www.cs.indiana.edu/hyplan/jsobel/rop.html

[SUN, 1997a] SUN MICROSYSTEMS. **JavaBeans™ - Specification**. Sun Microsystems, Inc., EUA, Jul. 1997. [online] http://java.sun.com/products/javabeans

[SUN, 1997b] SUN MICROSYSTEMS. **Java™ Core Reflection API**, Sun Microsystems, Inc., EUA, 1997. [online] http://java.sun.com/products/jdk/1.3/docs/guide/reflection/index.html

[SUN, 1999a] SUN MICROSYSTEMS. **Enterprise JavaBeans™ Technology**. Sun Microsystems, Inc., EUA, Jun. 1999. [online] http://java.sun.com/products/ejb

[SUN, 1999b] SUN MICROSYSTEMS. **Enterprise JavaBeans™ Specification, v1.1.**, Sun Microsystems, Inc., EUA, Dez. 1999. [online] http://java.sun.com/products/ejb

[SUN, 1999c] SUN MICROSYSTEMS. **Developing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, v1.0.**, Sun Microsystems, Inc., EUA, Dez. 1999. [online] http://java.sun.com/j2ee

[SUN, 1999d] SUN MICROSYSTEMS. **Java™ Message Service, v1.0.2.**, Sun Microsystems, Inc., EUA, Nov. 1999. [online] http://java.sun.com/products/jms

[SUN, 1999e] SUN MICROSYSTEMS. **Java™ 2 Platform Enterprise Edition Specification, v1.2.**, Sun Microsystems, Inc., EUA, Dez. 1999. [online] http://java.sun.com/j2ee

[SUN, 1999f] SUN MICROSYSTEMS. **J2EE Platform Role Guide**, Sun Microsystems, Inc., EUA, Dez. 1999. [online] http://java.sun.com/j2ee

[SUN, 1999g] SUN MICROSYSTEMS. **The J2EE Application Programing Model**, Sun Microsystems, Inc., EUA, Sep. 1999. [online] http://java.sun.com/j2ee

[SUN, 2000] SUN MICROSYSTEMS. **Designing Enterprise Applications with the Java™ 2 platform, Enterprise Edition**, Sun Microsystems, Inc., EUA, Mar. 2000. [online] http://java.sun.com/j2ee

[SZYPERSKI, 1998] SZYPERSKI, Clemens. **Component Software: Beyond Object-Oriented Programming**. Addison-Wesley Pub., EUA, 1998.

[TATSUBORI & SHIGERU, 1998] TATSUBORI, Michiaki, SHIGERU, Shiba. **Programming Support of Design Patterns with Compile-time Refection**. OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canadá, Out. 1998.

[TATSUBORI, 1999] TATSUBORI, Michiaki. **An Extension Mechanism for the Java Language**. Dissertação de Mestrado em Engenharia, Universidade de Tsukuba, Ibaraki, Japão. Fev. 1999. [online] http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/papers/mich_thesis99.pdf

[THOMAS, 1998] THOMAS, Anne. **Enterprise JavaBeans™ Technology - Server Component Model for the Java™ Platform**. Patricia Seybold Group, EUA, Dez. 1998.

[THOMAS, 1999] THOMAS, Anne. **Java™ 2 Platform, Enterprise Edition - Ensuring Consistency, Portability, and Interoperability**. Patricia Seybold Group, EUA, Jun. 1999.