UFSM

TRABALHO DE GRADUAÇÃO

EXPLORE - UMA FERRAMENTA DE *SOFTWARE* PARA EXPERIMENTAÇÃO PRÁTICA COM TRANSAÇÕES DISTRIBUÍDAS EM SISTEMAS BASEADOS EM COMPONENTES

Aluno:
Fábio Ottobeli Machado

Orientador:
Márcia Pasin

Santa Maria, RS, Brasil
2005

EXPLORE - UMA FERRAMENTA DE *SOFTWARE* PARA EXPERIMENTAÇÃO PRÁTICA COM TRANSAÇÕES DISTRIBUÍDAS EM SISTEMAS BASEADOS EM COMPONENTES

Por

Fábio Ottobeli Machado

Trabalho de Graduação apresentado ao Curso de Graduação em Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

Curso de Ciência da Computação

Trabalho de Graduação n° 198

Santa Maria, RS, Brasil

2005

Universidade Federal de Santa Maria

Centro de Tecnologia

Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o

Trabalho de Graduação

Explore - Uma Ferramenta de *Software* para Experimentação Prática com Transações Distribuídas em Sistemas Baseados em Componentes

elaborado por

Fábio Ottobeli Machado

como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA

____

Profa. Marcia Pasin
(Orientador)

____

Prof. Benhur Stein

____

Profa. Iara Augustin

Santa Maria, 14 de Julho de 2005

Agradecimentos

Gostaria de agradecer a toda minha família, que me apoiou e deu muita força durante todos esses anos de curso. Quero agradecer também a Bibiana, que tem sido uma presença constante de amor na minha vida, por todos os momentos felizes que tem me proporcionado e por sempre me amar de forma incondicional. Quero agradecer também a minha orientadora Profa. Marcia Pasin, pela idéia do trabalho e por ter depositado sua confiança em mim. Agradeço também a todos os professores do curso, por terem me oferecido as maiores riquezas que existem, conhecimento e informação. Por fim, agradecer aos amigos e colegas do curso.

SUMÁRIO

Lista de Figuras

Lista de Tabelas

Resumo

1 INTRODUÇÃO 1

1.1 OBJETIVOS GERAIS E ESPECÍFICOS 3
1.2 JUSTIFICATIVA 5

2 REVISÃO BIBLIOGRÁFICA 6

2.1 ARQUITETURA DE COMPONENTES 6
2.2 ARQUITETURAS MULTI-CAMADAS (*MULTI-TIER ARCHITECTURES*) 7
2.2.1 Arquiteturas do lado do servidor 11
2.2.2 Arquitetura J2EE 12
2.2.3 Componentes J2EE – Enterprise Java Beans (EJB) 21
2.2.4. Serviço de transações 31
2.2.5 Transações no EJB 33
2.2.6 JOnAS 40

3 PROJETO DA FERRAMENTA EXPLORE 44

3.1 EXPLORESERVER 45
3.2 EXPLORECLIENT 54
3.2.1 Painel Server-Applications 57
3.2.2 Painel App-Components 57
3.2.3 Painel Component-Beans 58
3.2.4 Painel Bean- Info 58
3.2.5 Painel All-Components 59
3.2.6 Painel All-Beans 59
3.2.7 Painel Info 60
3.2.8 Painel Log-Explore 60
3.3 CASOS DE USO – OPERAÇÕES DOS COMPONENTES DA FERRAMENTA EXPLORE 61

4 IMPLEMENTAÇÃO DA FERRAMENTA EXPLORE 67

4.1 CLASSE PROPRIEDADESBEAN 70
4.2 ESTRATÉGIA DE *PARSE* DOS *BEANS* 73
4.3 COMUNICAÇÃO ENTRE EXPLORESERVER E EXPLORECLIENT 74
4.4 CLASSE EXPLORESERVER 76
4.4.1 Localizar diretórios 76
4.4.2 Receber conexão do ExploreClient 77
4.4.3 Extrair arquivos JAR 78
4.4.4 Extrair descritores 79
4.4.5 Realizar o parse dos descritores 80
4.4.6 Serializar e enviar 82
4.5 CLASSE EXPLORECLIENT 82
4.5.1 Interface gráfica 84
4.5.1.1 Painel Aplicações 87
4.5.1.2 Painel Componentes 88

4.5.1.3 Painel Beans 89
4.5.1.4 Painel Propriedades do *Bean* 90
4.5.1.5 Painel Log do Explore 91
4.5.1.6 Caixa de diálogo "Todos os componentes do servidor JOnAS" 94
4.5.1.7 Caixa de diálogo "Todos os *beans* do servidor JOnAS" 95
4.5.1.8 Barra de Menus 97
4.5.2 Conexão com ExploreServer 98
4.5.3 Processamento de solicitações do usuário 99
5 TRABALHOS FUTUROS 101
6 CONCLUSÃO 104
7 BIBLIOGRAFIA 106

LISTA DE FIGURAS

Figura 2.1 – Aplicação Monolítica – Uma Camada [BOND2002] 7
Figura 2.2 – Aplicação 2-tier [BOND2002]...8
Figura 2.3 – Aplicação 3-tier [BOND2002]...11
Figura 2.4 – Visão geral da arquitetura J2EE [SUN2003a] 15
Figura 2.5 – Arquitetura da plataforma J2EE com multi-tiers [SUN2001a].........16
Figura 2.6 – Arquitetura de um servidor J2EE e seus *container*s [ARMSTRONG2004] 21
Figura 2.7 – Esquema de uma transação com operações encapsuladas................32
Figura 3.1 – Elo de informações de aplicações de um servidor JOnAS 52
Figura 3.2 – Interação básica Usuário-**ExploreClient**-**ExploreServer** 64
Figura 3.3 – Interação Usuário-**ExploreClient** através do botão **allComponentes** 65
Figura 3.4 – Interação Usuário-**ExploreClient** através do botão **allBeans**..........66
Figura 4.1 – Classes da ferramenta Explore...69
Figura 4.2 – Classe **propriedadesBean**...70
Figura 4.3 – Classe **metodoBean** 71
Figura 4.4 – Interface Gráfica **ExploreClient**...85
Figura 4.5 – Menu Opções do **ExploreClient**..86
Figura 4.6 – Painel Aplicações 88
Figura 4.7 – Painel Componentes 89
Figura 4.8 – Painel *Beans*...90
Figura 4.9 – Painel Propriedades do *Bean*..91
Figura 4.10 – Painel Log do Explore 92
Figura 4.11 – Caixa de Diálogo “Todos os componentes do servidor JOnAS”....95
Figura 4.12 – Caixa de Diálogo “Todos os *beans* do servidor JOnAS” 96
Figura 4.13 – Menu Opções do **ExploreClient**..98

LISTA DE TABELAS

TABELA 2.1 – TIPOS DE *ENTERPRISE BEANS* [ARMSTRONG2004]............................ 27

TABELA 2.2 - VALORES DO ELEMENTO TRANS-ATTRIBUTE [BOND2002, MAHAPATRA2000] 37

TABELA 2.3 – ESTADOS DE UMA TRANSAÇÃO [BOND2002].. 39

TABELA 3.1 – ATRIBUTOS GENÉRICOS DE UM *BEAN*... 48

TABELA 3.2 – ATRIBUTOS ESPECÍFICOS DE UM BEAN... 49

TABELA 3.3 – PAINÉIS DO EXPLORECLIENT 56

TABELA 3.4 – CASOS DE USO DA INICIALIZAÇÃO DO EXPLORESERVER 61

TABELA 3.5 – CASOS DE USO DA INICIALIZAÇÃO DO EXPLORECLIENT 63

TABELA 4.1 – PROPRIEDADES DOS *BEANS* CORRESPONDENTES AOS ATRIBUTOS DAS CLASSES 72

TABELA 4.2 – MENSAGENS EXIBIDAS NO PAINEL LOG DO EXPLORE 93

TABELA 4.3 – INTERAÇÃO USUÁRIO-EXPLORECLIENT 100

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Centro de Tecnologia
Universidade Federal de Santa Maria

Explore - Uma Ferramenta de *Software* para Experimentação Prática com Transações Distribuídas em Sistemas Baseados em Componentes

Aluno: Fábio Ottobeli Machado
Orientador: Márcia Pasin
Data e Local da Defesa: Santa Maria, 14 de Julho de 2005.

Este trabalho apresenta o projeto e implementação de uma ferramenta de *software* para o monitoramento de experimentos práticos em ambientes distribuídos orientados a componentes. Esta ferramenta, chamada de Explore, foi desenvolvida utilizando linguagem Java, visando o monitoramento de componentes *Enterprise Java Beans* em um servidor JOnAS.

A Explore reúne informações estáticas sobre os *enterprise beans*, as quais estão configuradas nos descritores de componentes das aplicações J2EE. O trabalho apresenta os seguintes capítulos: uma introdução sobre o assunto abordado; revisão bibliográfica (Arquitetura de componentes; Arquiteturas multi-camadas; tecnologias J2EE e EJB; serviço de transações; servidor JOnAS); o projeto da ferramenta Explore; a implementação da ferramenta Explore; a definição de trabalhos futuros; uma conclusão, abordando os resultados deste trabalho, as dificuldades e os próximos passos no desenvolvimento da ferramenta Explore.

1 Introdução

Tolerância a falhas é uma das características desejáveis nos sistemas computacionais atuais. Em uma operação de compra na Internet, por exemplo, se ocorreu alguma falha durante a execução da aplicação, o usuário precisa ter a garantia de algum retorno especificando que sua operação foi executada com sucesso ou alguma mensagem de erro.

Os sistemas de computação de apenas uma camada, os quais integravam todas as funções em apenas um módulo, apresentavam como aspectos negativos, entre outros, a dificuldade de localizar e recuperar erros. Qualquer alteração no código ou na parte física do sistema, freqüentemente exigia a recompilação de todo o código e eventual indisponibilidade do sistema por certos períodos de tempo. Considerando que muitas vezes essa indisponibilidade do sistema pode trazer grandes prejuízos, surgiu a necessidade de sistemas que apresentassem características desejáveis para um sistema tolerante a falhas, como disponibilidade, recuperação ou mascaramento de erros, facilidade de manutenção e substituição de partes do sistema, preservando a confiabilidade e o desempenho.

Um grande avanço nesse sentido foi a divisão da arquitetura de uma camada para um modelo de três ou mais camadas. Surgiu o conceito da abstração de componentes, a partir dos quais uma aplicação passa a ser dividida em vários sub-sistemas, cada um independente dos outros.

Essa abstração de componentes [VOL2003, SZY1999] possibilita uma estrutura flexível de programação para aplicações em sistemas computacionais. Um dos exemplos promissores que implementa serviços na forma de componentes é a tecnologia J2EE/EJB [SUN2001a]. Esta tecnologia permite que o desenvolvedor de uma aplicação distribuída codifique os serviços específicos (como gerenciar uma conta bancária ou realizar uma compra através da Internet) e configure, através de uma ferramenta gráfica, as propriedades dos serviços não-funcionais como alta disponibilidade, gerenciamento transacional, persistência e segurança. Os serviços não-funcionais são oferecidos através do reuso de componentes de *software*, i.e., código pré-codificado e pré-testado que é inserido automaticamente nas aplicações, de acordo com as especificações do programador. O reuso de componentes, dessa forma, possibilita a construção de *software* com maior rapidez e maior confiabilidade.

Apesar das facilidades para o desenvolvimento de aplicações com componentes, ainda há carência de ferramentas de *software* que auxiliem o desenvolvedor de aplicações distribuídas a detectar e analisar falhas durante a execução dessas aplicações. Faltam ferramentas que facilitem a injeção de falhas e a observação de diferentes cenários de execução.

Este trabalho propõe construir uma ferramenta de *software* que possibilite o monitoramento da execução de experimentos práticos e, posteriormente, a injeção de falhas em sistemas baseados em componentes com serviços transacionais. Essa ferramenta deve ser capaz de obter informações sobre os atributos de execução de um

componente, em uma plataforma distribuída. Como principal resultado esperado está a observação de mecanismos que possibilitem a construção de sistemas computacionais mais robustos. Um sistema computacional robusto, neste contexto, permite a execução de aplicações distribuídas com garantia de serviços com tolerância a falhas. Adicionalmente, este trabalho enfatiza a pesquisa e o desenvolvimento de *software* livre, e independente de plataforma.

1.1 Objetivos gerais e específicos

O objetivo deste trabalho de graduação é o desenvolvimento de uma ferramenta de *software* que possibilite o monitoramento de um servidor que atenda à especificação J2EE da Sun Microsystems [SUN2001a]. O servidor escolhido foi o *Java Open Application Server* (JOnAS), que foi desenvolvido utilizando a linguagem Java. Essa escolha se justifica pelo fato de que o servidor JOnAS é *software* livre, e o consórcio que o mantém (*OBJECT*WEB Consortium) está apoiando e incentivando projetos que visam implementar serviços para esse servidor.

O foco no desenvolvimento desta ferramenta está na criação de um serviço de monitoramento remoto, responsável por recolher informações do servidor e apresentá-las ao administrador do sistema. A tecnologia que constitui-se no centro da arquitetura J2EE é a *Enterprise Java Beans* [SUN2004a], a qual define uma arquitetura de componentes que implementam a lógica de negócios de uma aplicação. Desta forma, este trabalho procura dar os primeiros passos

rumo a um serviço de monitoramento completo de *Enterprise Beans* e dos servidores que ofereçam suporte aos mesmos.

A ferramenta desenvolvida neste trabalho, chamada de **EXPLORE**, deverá estabelecer um elo entre o usuário e as informações das aplicações que o mesmo executa em um servidor. Deve possibilitar o monitoramento de componentes J2EE e *Enterprise Beans* remotos, reunindo informações estáticas e dinâmicas sobre as aplicações executadas em um servidor JOnAS.

Posteriormente, as funcionalidades da ferramenta **EXPLORE** devem ser expandidas, de forma a garantir ao desenvolvedor um controle total sobre sua aplicação. Isso vai possibilitar a criação de ambientes de testes para aplicações J2EE. Nesses ambientes, o desenvolvedor poderá ter em tempo real informações de propriedades pré-configuráveis, controle transacional e controle de execução dos componentes em fase de testes.

O objetivo geral deste trabalho pode ser desdobrado nos seguintes objetivos específicos: o estudo de sistemas orientados a componentes de *software,* o estudo de sistemas transacionais, o estudo de trabalhos correlacionados, o desenvolvimento de um ambiente de experimentação para sistemas transacionais distribuídos, o projeto e o desenvolvimento da ferramenta **EXPLORE**, a exploração e o desenvolvimento e a pesquisa de *software* livre, e independente de plataforma.

1.2 Justificativa

A programação baseada em componentes é um conceito novo, ainda em fase de amadurecimento. O rápido desenvolvimento de aplicações, tolerância a falhas, alta disponibilidade, alta confiabilidade são algumas das tendências previstas no passado, que atualmente vêm despertando grande interesse e se concretizando dentro das instituições.

Sistemas baseados em componentes, no contexto de desenvolvimentos computacionais, se apresentam como uma solução interessante a ser adotada. Várias empresas apresentam seus produtos como soluções capazes de atender ao problema apresentado acima. Entretanto, essas soluções são muitas vezes caras e projetadas para um *hardware* específico. Observa-se também a carência de ferramentas de *software* que possibilitem observar o comportamento destes sistemas, principalmente dos servidores de aplicação. É neste sentido que este trabalho propõe a construção de uma ferramenta gráfica para a observação do comportamento destes servidores.

2 Revisão bibliográfica

2.1 Arquitetura de componentes

Um componente de *software* é um código que implementa um conjunto de interfaces, que são gerenciáveis. Um componente não é uma aplicação por si só, na verdade ele funciona como um módulo que executa uma determinada função específica, o qual pode ser combinado com outros módulos para construir a aplicação em si [VOS1996].

A grande vantagem dos componentes, além da modularização da lógica da aplicação, é a possibilidade de serem reutilizáveis. Isso significa que ao desenvolver um componente que resolva determinado problema, o mesmo pode ser reutilizado sem alterações por qualquer aplicação baseada em componentes que em algum momento precise resolver esse problema. O componente então é considerado como uma caixa-preta, ou seja, basta saber que ele realiza a tarefa e como conectar ele com outros componentes, de forma a obter os resultados esperados.

Tipicamente um componente tem separados uma interface e uma implementação. A interface do componente trata da publicação dos métodos contidos no mesmo, escondendo a implementação do cliente desse componente. A implementação do componente contém o código funcional do mesmo e os dados internos, os quais devem ter seu acesso proibido aos clientes.

Nos primórdios da utilização da lógica de componentes, não existia um acordo entre os fabricantes o que resultava em vários

componentes dependentes de plataforma. A arquitetura de componentes surgiu para definir um conjunto de interfaces entre os componentes e os servidores de aplicação, criando então independência de plataforma para esses componentes e os tornando reutilizáveis. Um dos padrões de arquitetura de componentes do lado do servidor mais utilizados atualmente é a tecnologia *Enterprise Java Beans* (EJB), da *Sun Microsystems* [SUN2004].

2.2 Arquiteturas multi-camadas (*multi-tier architectures*)

Há duas décadas imperavam no mundo da computação os main*frame*s e os computadores pessoais autônomos. Uma aplicação estava contida em uma única máquina, e era comum encontrar aplicações monolíticas, nas quais todas as funcionalidades constituíam um único pedaço de código. Em outras palavras, toda a interface com o usuário, a lógica de negócios e o acesso aos dados encontravam-se juntos no mesmo código (figura 2.1).

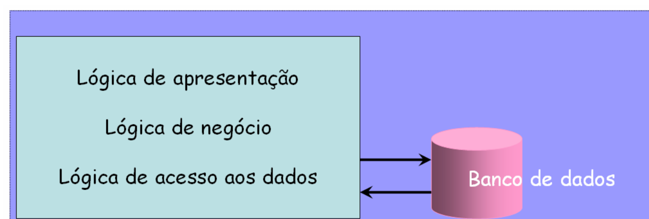

Figura 2.1 – Aplicação Monolítica – Uma Camada [BOND2002]

Isso gerava inúmeros problemas de manutenção e alteração da aplicação. Como todas as funcionalidades encontravam-se agrupadas, uma simples mudança no código de uma dessas funcionalidades freqüentemente implicava em alterações em todo o código, incompatibilidade entre as partes e o surgimento de erros em partes aparentemente não relacionadas.

O primeiro passo na evolução desse sistema de uma camada para os sistemas de hoje, surgiu da necessidade de compartilhar dados entre diferentes máquinas. Apareceram, então, os sistemas de duas camadas (*2 - tier systems*), nos quais a base de dados e toda a lógica necessária para sua manipulação eram armazenadas em uma máquina separada. Dessa forma, criava-se uma independência entre a lógica de acesso dos dados e as outras partes do sistema, proporcionando um certo nível de escalabilidade. Contanto que a interface entre a aplicação cliente e a lógica de acesso aos dados permanecesse inalterada, qualquer alteração poderia ser feita no código de qualquer uma dessas duas camadas sem implicar em alterações na outra (Figura 2.2).

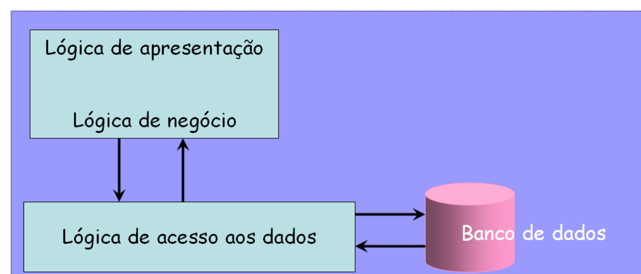

Figura 2.2 – Aplicação 2-tier [BOND2002]

Contudo, no modelo de duas camadas, o cliente ainda está sobrecarregado de código. Toda a lógica de apresentação e de negócios estava agrupada na mesma máquina. Alterações no código exigiam a recompilação dessas duas camadas lógicas (assim chamadas porque não possuem separação física), o que representava um enorme desperdício de tempo. Além disso, existiam problemas de gerenciamento – todos os clientes precisavam atualizar todo o *software* sempre que alguma mudança era aplicada ao mesmo.

A popularização da Internet passou a exigir a separação entre a apresentação e a lógica de negócios, de forma que os resultados possam ser apresentados tanto em uma interface gráfica (*graphical user interface* – GUI) comum quanto em um navegador (*browser*) em formato HTML (*Hypertext Markup Language*) [BOND2002]. A camada lógica do cliente passou a ser separada em duas camadas físicas, nas quais a lógica de apresentação continua rodando no cliente (em um navegador ou em uma interface gráfica) e a lógica de negócios passa a rodar em um servidor Web ou outra máquina com a qual este servidor se comunica. Isso desdobrou o modelo de duas camadas em outro de três ou mais camadas (*n-tier systems* ou *multi-tier systems*), que se tornou um modelo padrão para a Internet (Figura 2.3).

Arquiteturas de sistemas multi-camadas possuem as seguintes características [ROM2001]:

1. custos de instalação de *software* são baixos no lado do servidor, pois este é um ambiente controlado;

2. custos de trocas de bancos de dados são baixos, pois não se faz necessária a reconfiguração e/ou reinstalação dos clientes;
3. custos de migração da lógica de negócios são baixos, pois mudanças na mesma não implicam em recompilar e reinstalar a camada cliente;
4. aumento da segurança de partes isoladas da aplicação com o uso de *firewalls* entre as camadas (separando a camada de apresentação da lógica de negócios num sistema Web, por exemplo);
5. recursos podem ser eficientemente enfileirados e reutilizados (um cliente não precisa adquirir um recurso de forma exclusiva e mantê-lo até o fim de sua operação);
6. cada camada pode ser alterada independentemente;
7. quedas de desempenho são localizadas, isto é, se uma das camadas está sobrecarregada, as outras ainda podem funcionar adequadamente;
8. erros críticos são locais à uma camada, de forma que as outras ainda podem funcionar adequadamente contanto que saibam lidar com a falha;
9. desempenho da comunicação tende a sofrer quedas, conforme o sistema for dividido em camadas físicas, pois estas precisam comunicarem-se entre si, aumentando o *overhead* de comunicações no sistema;
10. custos de manutenção são altos, pois a medida que o sistema ganha mais camadas físicas crescem os custos de instalação, atualização e administração de *software*.

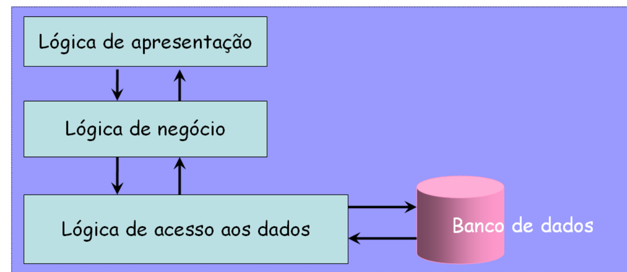

Figura 2.3 – Aplicação 3-tier [BOND2002]

2.2.1 Arquiteturas do lado do servidor

Aplicações no lado do servidor (*server-side*) são construídas para suportar vários usuários realizando inúmeras operações simultaneamente. Dentre as características que esse sistema deve apresentar encontram-se segurança, disponibilidade, escalabilidade e confiabilidade.

Arquiteturas *server-side* baseadas em componentes possuem uma divisão lógica de três camadas, as quais são:

1. Camada de apresentação (*presentation layer*): contém os componentes encarregados de tratar a apresentação dos dados e da interação do usuário com o sistema. É a camada de apresentação a responsável por desenhar uma interface gráfica, a qual interage com o usuário e apresenta as respostas do sistema.
2. Camada de lógica de negócios (*business logic layer*): formada pelos componentes que trabalham juntos para resolver os

problemas de negócios, como as operações envolvidas em uma compra pela Web.

3. Camada de dados (*data layer*): responsável por oferecer armazenamento persistente dos dados à camada de negócios. Contém componentes responsáveis por realizar a comunicação entre os componentes da camada de negócios e as bases de dados.

2.2.2 Arquitetura J2EE

O J2EE (*Java 2 Platform, Enterprise Edition*) [SUN2003a] é uma plataforma orientada a componentes, desenvolvida pela *Sun Microsystems* e por um grupo de empresas da área de software, que possibilita a construção de aplicações distribuídas e reutilizáveis, utilizando-se da linguagem Java. O J2EE foi a resposta da Sun às necessidades do mercado, que exigia uma arquitetura server-side orientada a componentes. A arquitetura J2EE define padrões para produzir aplicações seguras, escaláveis e de alta-disponibilidade. As tecnologias incluídas no J2EE são[ROM2001] [SUN2003]:

1. ***Enterprise* Java *Beans* (EJB)**: define um padrão de componentes *server-side* que implementam a lógica de negócios da aplicação. Define também as interfaces entre os clientes e os serviços, além de oferecer serviços não-funcionais (como persistência e gerenciamento transacional) para os componentes da aplicação. O padrão EJB é o cerne da

plataforma J2EE, fazendo a ligação entre as outras APIs [SUN2004a].

2. **Java *Remote* Method Invocation (RMI) e RMI-IIOP**: o RMI permite a comunicação entre processos e outros serviços relacionados à essa comunicação [SUN1999a]. O RMI-IIOP é uma extensão do RMI e permite a utilização do *Internet Inter-Orb Protocol* (IIOP) [OMG1999] para integração com o CORBA (Common *Object* Request Broker Architecture)[OMG1995]. O RMI-IIOP é a API padrão para comunicação entre objetos distribuídos na plataforma J2EE. [SUN1999b]
3. **Java Naming and Directory Interface (JNDI)**: faz a identificação das localizações dos componentes e outros recursos na rede. [SUN1999c]
4. **Java Database Connectivity (JDBC)**: oferece operações de banco de dados que permitem a conexão do sistema com qualquer banco de dados relacional. [SUN2001b]
5. **Java *Transaction* API (JTA) e Java *Transaction* Service (JTS)**: permitem que os componentes tenham suporte a transações confiáveis. [SUN1999d] [SUN1999e]
6. **Java *Message* Service (JMS)**: permite comunicação assíncrona entre objetos distribuídos, dentro ou fora de um sistema J2EE. É a alternativa para o RMI-IIOP. [SUN2002]
7. **Java Servlets e Java Servers Page (JSP)**: *Servlets* e JSPs são componentes para computação orientada a requisições/respostas, como interação com clientes através de

HTML. JSPs diferem-se de *Servlets* quanto ao fato de não serem código Java puro e ser centrado na apresentação de conteúdo. Na verdade, todo script JSP é compilado em *Servlets*, sendo então o JSP uma abstração de *Servlets* para facilitar a vida dos programadores (que não precisam de grandes conhecimentos em Java para desenvolver JSPs). [SUN2003b]

8. **Java IDL**: implementação Java do CORBA, que permite a integração com outras linguagens de programação e o uso de todas as funcionalidades do CORBA. [OMG2000a] [OMG2000b]
9. **JavaMail**: permite que a aplicação Java envie e-mails através da própria aplicação, independente de plataforma ou protocolo. [SUN2000]
10. **J2EE Connectors Architecture (J2EE CA)**: permitem a integração da aplicação com sistemas de *mainframe* e qualquer outro sistema de informações existente fora do sistema J2EE. [SUN2004b]
11. **The Java API for XML Processing (JAXP)**: o XML (*Extensible Markup Language*) é uma linguagem de marcação para definir metadados que descrevem a estrutura de um conceito, para gerenciamento e marcação de documentos. O XML define DTDs (*Document Type Definitions*) [BOND2002], que são documentos cuja função é especificar a estrutura de um documento XML. Podem aparecer em um documento separado ou no próprio arquivo XML. É utilizada por muitas das

tecnologias J2EE como um descritor de conteúdo. A JAXP é a API oficial para integração do XML com o J2EE. [SUT2004]

12. **The Java Authentication and Authorization Service (JAAS)**: é a API para realizar operações relacionadas à segurança em uma aplicação J2EE. [LAI1999]

Uma visão geral da arquitetura J2EE é mostrada na figura 2.4:

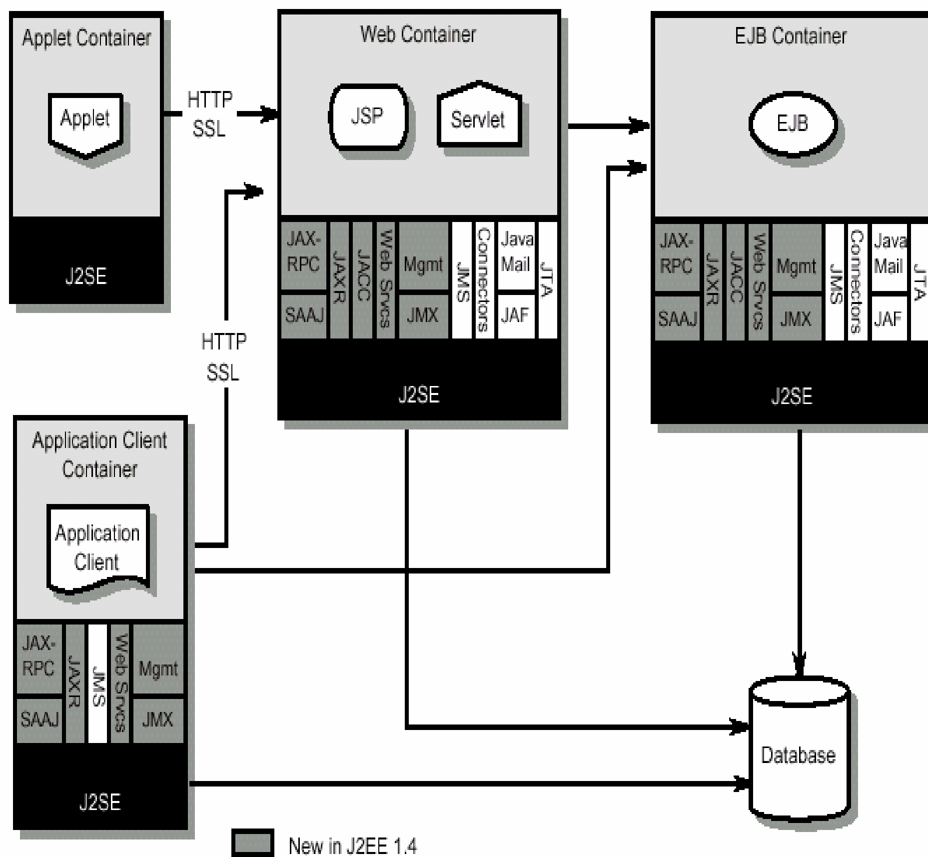

Figura 2.4 – Visão geral da arquitetura J2EE [SUN2003a]

A arquitetura J2EE é composta pelas seguintes camadas (figura 2.5):

1. Cliente: camada que faz a interação com o usuário e exibe os resultados ao mesmo. Os clientes podem ser HTML, applets Java e aplicações Java.
2. Web: responsável por gerar uma lógica de apresentação. Essa camada recebe uma solicitação do cliente e gera uma resposta apropriada. Constitui-se em um *container* Web, no qual rodam *Servlets* e JSPs.
3. Negócios: contém os componentes de negócios, que tratam da lógica de negócios da aplicação. Tipicamente contém um ou mais *containers* EJB, nos quais rodam componentes EJB.
4. EIS: a camada EIS (*Enterprise Information System*) é a responsável pelo armazenamento persistente dos dados da aplicação. Tipicamente é um banco de dados ou um sistema herdado que se comunica com a aplicação J2EE.

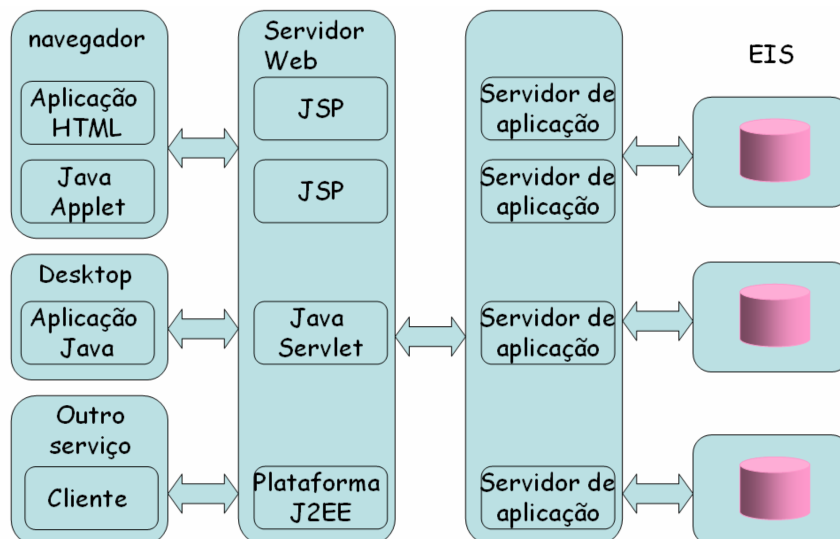

Figura 2.5 – Arquitetura da plataforma J2EE com multi-tiers [SUN2001a]

Os pilares conceituais da arquitetura J2EE são as camadas (*tiers*), os componentes e os *containers*. Sendo o J2EE uma arquitetura *multi-tier*, a aplicação é dividida em três camadas lógicas (apresentação, lógica de negócios e acesso aos dados), que podem corresponder ou não à três camadas físicas. Essas camadas são formadas por componentes, que oferecem serviços à própria camada e às outras. Os componentes existentes em camadas diferentes devem ser fracamente acoplados, garantindo a independência entre os mesmos e entre as camadas. Assim, um componente de negócios em um servidor de aplicação pode oferecer seus serviços a vários tipos de clientes, cada qual com uma camada de apresentação diferente. [SUN2003a] [ROM2001]

Normalmente, no J2EE, os componentes de negócio são encapsulados em EJBs, isto é, são componentes do tipo *Enterprise Java Beans*. Isso porque o EJB provê uma série de serviços não-funcionais, como persistência e gerenciamento do ciclo de vida, aos componentes através de *containers* EJB. Um *container* EJB oferece um ambiente de execução para os *enterprise beans* (componentes EJB) em um servidor EJB (servidor de aplicação que suporta a tecnologia EJB). É o *container* que armazena um *enterprise bean* e repassa a ele todas as chamadas remotas realizadas por clientes. Ele é responsável por gerenciar os *enterprise beans*, interagindo com os mesmos através de métodos de gerenciamento que somente o *container* pode chamar. Um *container* oferece aos *enterprise beans*

uma série de serviços implícitos, relacionados aos serviços de *middleware* de um sistema distribuído. Esses serviços são [ARMSTRONG2004, ROM2001]:

1. Gerenciamento de recursos: numa aplicação J2EE existem vários componentes que acessam diferentes recursos, como *threads, sockets* e conexões à base de dados. É papel do *container* EJB gerenciar apropriadamente o acesso a esses recursos por parte dos *enterprise beans* nele contidos.
2. Gerenciamento de ciclo-de-vida: mediante requisições de clientes, um *container* EJB deve instanciar, destruir e reutilizar apropriadamente os *enterprise beans*. Esse processo deve ser dinâmico e é chamado de gerenciamento do ciclo-de-vida de um *bean*.
3. Gerenciamento de estado: aplicações clientes de um sistema J2EE normalmente demoram um certo tempo entre as requisições de métodos dos *beans*, sendo esse tempo gasto em processamento do cliente (processamento da apresentação dos dados, por exemplo). Durante esse tempo, no qual o componente não é efetivamente utilizado, o *container* pode reutilizar o componente para servir outros clientes. Se o *bean* não guarda estado (*stateless*), o *container* pode reutilizá-lo dinamicamente para servir outros clientes. Isso economiza recursos, já que poucas instâncias de um *bean* podem servir vários clientes. Já se o *bean* guarda estado (*stateful*), o *container* pode reutilizá-lo caso ele passe um longo tempo sem

ser utilizado, contanto que salve em disco o estado do cliente atual (para recuperá-lo quando esse cliente retomar o acesso ao *bean*).

4. Transações: um *container* EJB permite que transações sejam realizadas de forma implícita, automática e segura. Dessa forma, garante o controle de concorrência no acesso à dados compartilhados além da consistência da base de dados.
5. Segurança (Security): *containers* EJB utilizam *Access Control Lists* (ACLs), garantindo a validação dos usuários na execução das tarefas que desejam realizar. Essas ACLs são listas de usuários e direitos de acesso, que permitem a autenticação de um usuário e a autorização das tarefas que esse deseja executar. Os *containers* EJB fazem esse gerenciamento da segurança de forma transparente, eliminando a necessidade do programador de criar código que garanta isso.
6. Persistência: *containers* EJB podem oferecer de forma transparente a persistência dos dados necessários, que são *enterprise beans* que representam dados em um meio de armazenamento.
7. Acesso remoto: *enterprise beans* são desenvolvidos como se fossem componentes *stand-alone*, sem propriedades de rede. Mas, assim que são instalados em um *container* EJB, tornam-se objetos distribuídos e passíveis de replicação em outras camadas. O *container* automaticamente gerencia as questões de rede transformando qualquer componente em um objeto capaz de receber e responder requisições remotas.

8. Transparência de localização: *containers* EJB oferecem transparência de localização, ou seja, não importa em qual máquina um componente está instalado, o código do cliente não precisa especificar o caminho para acessá-lo. Essa tarefa é realizada de forma transparente pelo *container*, que localiza o componente e repassa a ele as requisições do cliente.

Esses serviços que são oferecidos aos componentes pelo *container* liberam o programador para se concentrar apenas na lógica de negócios do componente, sem ter que se preocupar em implementar esses serviços, os quais são complicados e trabalhosos, mas ao mesmo tempo inerentes à um sistema distribuído.

Além do *container* EJB, a arquitetura J2EE define outros *containers* [ARMSTRONG2004]. O relacionamento entre esses *containers* é ilustrado na figura 2.6:

1. *Container* Web (*Web container*): gerencia a execução de componentes web (JSPs e *Servlets*) de uma aplicação J2EE. É executado no servidor J2EE.
2. *Container* de Aplicação de Cliente (*Application Client Container*): gerencia a execução de componentes de aplicação clientes, sendo executado no cliente.
3. *Container* de *Applet* (*Applet Container*): gerencia a execução de *applets*. É formado por um *browser web* e um *plug-in* Java rodando juntos no cliente.

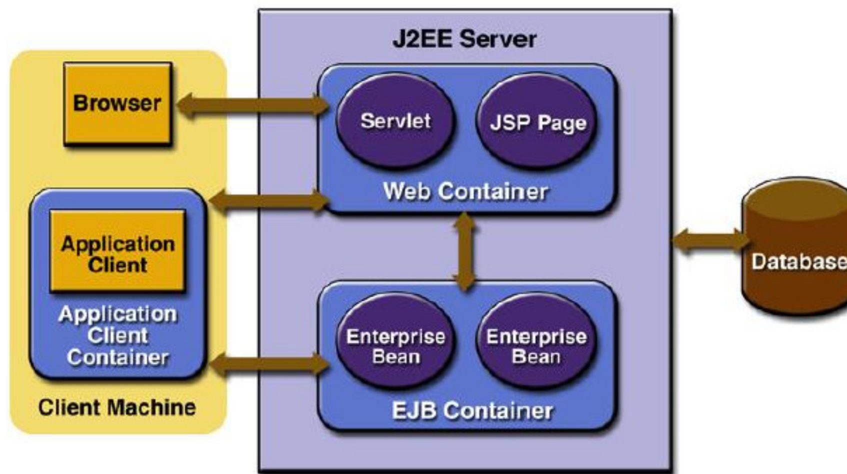

Figura 2.6 – Arquitetura de um servidor J2EE e seus *container*s [ARMSTRONG2004]

2.2.3 Componentes J2EE – *Enterprise* Java *Beans* (EJB)

O *Enterprise Java Beans* (EJB) é o padrão de implementação dos componentes de negócios em sistemas J2EE. Pode-se usar outros tipos de componentes, mas os EJBs (ou *enterprise beans*) beneficiam-se de todas as funcionalidades automaticamente oferecidas pelos *containers*, tornando-se a alternativa mais viável.

A especificação do EJB [SUN2004a] define uma arquitetura para servidores de aplicação, nos quais rodam *containers* EJB, responsáveis por oferecer os serviços não-funcionais da aplicação. Os *beans* (EJBs) representam os componentes de negócios da aplicação, contendo os serviços funcionais da mesma. Esses servidores de aplicação formam, junto com o cliente e o banco de dados (ou outro

meio de armazenamento persistente), uma estrutura de três camadas (3-*tier*).

Enterprise Beans (EJBs)

São componentes *server-side deployable* (instaláveis) distribuídos [ROM2001]:

1. podem ser particionados através de múltiplas camadas;
2. podem ser transacionais;
3. oferecer suporte a multi-usuários com segurança (autenticação e permissão);
4. podem ser instalados em qualquer *container*/servidor compatível com o padrão EJB.

Classe *Enterprise Bean*

Nessa classe estão os detalhes de implementação de um componente de negócios da aplicação, ou seja, contém a codificação do *bean*. Possui uma interface bem definida e obedece certas regras definidas na especificação EJB. Essa especificação define interfaces que a classe *bean* pode implementar. Estas interfaces, por sua vez, forçam a classe *bean* a publicar certos métodos que todos os *beans* devem expor. Esses métodos são definidos no Modelo de Componentes EJB (EJB Components Model). O *container* usa esses métodos para gerenciar o *bean* e alertá-lo de eventos do sistema que possam exigir algum processamento do *bean*.

A interface básica que todos os *beans* devem implementar é a *javax.ejb.EnterpriseBean* cuja forma é:

public interface javax.ejb.EnterpriseBean extends java.io.Serializable

Beans de Sessão e *Beans* de Entidade (tipos de *beans* adiante explicados) tem interfaces mais específicas que estendem essa interface, de maneira que o programador não precisa implementar diretamente a interface *javax.ejb.EnterpriseBean*.

Objeto EJB

Para usar uma instância de uma classe *Enterprise Bean*, o cliente nunca faz uma chamada direta ao método na instância atual do *bean*. Na verdade, a chamada de método é interceptada pelo *container* EJB, que então repassa a mesma para a instância do *bean*. Isso ocorre porque uma classe *Enterprise Bean* não é *network-enabled*, isto é, ela não possui capacidade de trabalhar diretamente com a rede, portanto não pode ser chamada diretamente através da rede. O que ocorre é que o *container* encapsula o *bean* em um objeto *network-enabled*, que, então, recebe as chamadas dos clientes e as repassa para as instâncias dos *beans*.

Essa interceptação de pedidos permite ao *container* realizar de forma automática alguns gerenciamentos necessários, como lógica de transações, segurança, instanciamento, etc.

Logo, o *container* funciona como uma camada de indireção entre o código do cliente e o *bean*, que se manifesta através de um

único objeto *network-aware*, o objeto EJB. Esse objeto sabe realizar a lógica intermediária que o *container* EJB precisa, antes de servir uma chamada de método através de uma instância de uma classe *bean*. O objeto EJB expõe todos os métodos de negócios que o próprio *bean* expõe e realiza as operações de *middleware* necessárias para repassar as requisições dos clientes para os *beans* apropriados. Sua codificação é específica para cada *container*, sendo que cada desenvolvedor de servidores EJB cria seu próprio objeto EJB [ROM2001, SUN2004a].

Partes de um *bean*

Um *bean* é composto por partes:

1. classe ou objeto EJB: arquivo em linguagem Java que implementa os serviços funcionais da aplicação. Contém métodos de negócios criados pelo programador e métodos padronizados na especificação EJB, permitindo o gerenciamento pelo *container*.
2. Interface *Home* (*Home Interface*): interface especificada pelo desenvolvedor da classe *bean*, define métodos para criar, destruir e encontrar *beans* (objetos EJB que referenciam eles). Portanto, publica métodos que gerenciam o ciclo de vida do *bean* e métodos para reaver instâncias do mesmo, usados pela aplicação cliente. Além disso, a interface *home* também é responsável por informar ao *container* quais parâmetros são necessários na inicialização de um *bean*. O *container* cria um

objeto *home* (*Home Object*), que implementa essa interface criada pelo programador. O objeto *home* funciona como uma fábrica para objetos EJB (EJB *objects factory*). Para conseguir uma referência para um objeto EJB, o cliente deve solicitá-la ao objeto *home*. Este é responsável por instanciar objetos EJB, encontrar objetos EJB existentes (apenas para *beans* de entidade) e remover objetos EJB. Objetos *home* são proprietários e específicos para cada *container* [ROM2001].

3. Interface Remota (*Remote* Interface): aplicações clientes chamam métodos em objetos EJB ao invés de nos próprios *beans*. Os objetos EJB, então, clonam os métodos de negócio que um *bean* expõe. Essa cópia dos métodos se dá através da interface remota, que contém a publicação de todos os métodos funcionais do *bean*. A interface remota deve conter alguns métodos padrão da especificação EJB e todos os métodos de negócio do *bean* correspondente a ela. Como a interface remota publica métodos que podem ser acessados por clientes remotos, todos os métodos devem lançar a exceção **java.rmi.RemoteException** (exceção remota) [ROM2001].
4. Descritor (*Deployment Descriptor*): arquivo em linguagem XML que deve ser incluído no componente *Enterprise Bean*. Contém as especificações de propriedades dos *beans*, que permitem ao *container* realizar serviços de *middleware* implícitos (automáticos, sem necessidade de codificação por parte do desenvolvedor do *bean*) para o componente. O *container* inspeciona o descritor e oferece aos *beans* do

componente os serviços declarados no arquivo. Descritores especificam: gerenciamento do *bean*, requisitos de ciclo de vida, requisitos de persistência, requisitos de transações e requisitos de segurança.

5. Arquivo de propriedades específicas do *bean*: arquivo em linguagem Java que contém propriedades que podem ser lidas em tempo de execução pelo *bean*, para setar funções específicas do *bean*.
6. Arquivo EJB-jar: é uma entidade que contém as classes *bean*, as interfaces *home*, as interfaces remotas, os descritores e arquivos de propriedades dos *beans*. Após a criação desse arquivo, o *Enterprise Bean* é um componente completo e pronto para instalação em um servidor de aplicação.

Tipos de *Beans*

Existem três tipos de *bean*: sessão, entidade e orientado a mensagens. A tabela 2.1 resume os tipos de *beans* e as suas funções em um sistema.

TABELA 2.1 – Tipos de *Enterprise Beans* [ARMSTRONG2004]

Tipo de *Enterprise Bean*	Propósito
Sessão	Realiza alguma tarefa para um cliente; implementa um serviço Web
Entidade	Representa uma entidade de negócios existente em um meio de armazenamento persistente
Orientado a Mensagens	Atua como um *listener* para o JMS, processando mensagens de forma assíncrona

***Beans* de Sessão (*Session Beans*)**

Beans de sessão contém a implementação de uma série de funções da lógica de negócios, funcionando como uma interface síncrona através da qual o cliente utiliza essas funções. Representam, portanto, processos de negócio a serem executados por um cliente. Fatura e autorização de cartão de crédito, preenchimento de um pedido de compra e cálculos são exemplos de típicas operações realizadas por um *bean* de sessão.

Os *beans* de sessão são objetos que estão na memória e possuem ciclo de vida curto. Existem apenas enquanto durar a execução de uma sessão de um cliente. Não são persistentes e guardam apenas dados temporários. Até podem executar operações de

banco de dados utilizando JDBC, mas os dados obtidos estão sujeitos a perdas perante falhas do sistema, que matam o *bean*. Normalmente, só armazenam dados pertinentes à sessão de um cliente.

Existem dois modelos de *beans* de sessão, com informação de estado conversacional (*stateful session bean*) e sem informação de estado conversacional (*stateless session bean*).

Beans de sessão sem estado não guardam informações do cliente. A cada chamada de método, esse tipo de *bean* de sessão remove todo seu estado. Assim, qualquer *bean* sem estado pode servir qualquer cliente, i.e., *beans* sem estado são facilmente reutilizáveis por múltiplos clientes. Quando da utilização desse tipo de *bean*, um *container* pode criar um pool de *beans* com várias instâncias dele para servir a todos os clientes, economizando memória já que o número de instâncias do *bean* no pool será bem menor que o número de clientes acessando o *bean*.

Beans de sessão com estado guardam informações do cliente, armazenando todo o estado conversacional de uma sessão. Por isso, antes de servir outro cliente com a mesma instância do *bean*, o *container* deve armazenar o estado do cliente atual em um meio de armazenamento persistente (processo chamado de Passivation). Quando o antigo cliente retorna a chamar o *bean*, o *container* deve trazer de volta para a memória o estado desse cliente (processo chamado de Activation). Esse esquema de Passivation/Activation permite ao *container* criar um pool de instâncias de *beans* para um *bean* com estado. Note que o cliente não precisa receber de volta a mesma instância do *bean*, o que importa mesmo é o estado

conversacional armazenado, que deve ser carregado para a instância que ele usará. A decisão sobre qual instância do *bean* deverá ter seu estado armazenado para que outro estado seja carregado depende de cada *container* (normalmente utiliza-se a estratégia LRU – último recentemente usado).

Beans de Entidade (*Entity Beans*)

Modelam dados de negócio da aplicação e são objetos persistentes que representam itens de dados. Informações de um produto, cartão de crédito e estoque são exemplos de dados armazenáveis em um banco de dados que podem ser representados por um *bean* de entidade. O ciclo de vida de *beans* de entidade é longo. Eles são capazes de sobreviver a falhas do sistema, pois os itens de dados são mantidos em um meio de armazenamento persistente e tolerante a falhas.

Beans de entidade são utilizados por *beans* de sessão para obter dados manipuláveis para a lógica de negócios. São uma interface síncrona por meio da qual os clientes acessam seus dados e funcionalidades. Agem como uma representação dinâmica dos dados de negócio armazenados em uma fonte de dados persistente, isto é, são visões dos dados que estão armazenados em um banco de dados. Assim, alterações na instância do *bean* implicam em atualizações automáticas no banco de dados. Essas atualizações ocorrem por meio de dois métodos que apenas o *container* pode chamar: ejbLoad() para

carregar dados e armazenar na instância do *bean* e ejbStore() para salvar a instância atual do *bean* no meio de armazenamento.

Assim como acontece com os *beans* de sessão, o *container* pode criar *pools* de instâncias de *beans* de entidade, para permitir o acesso concorrente aos dados. Esses *pools* são usados para manter em memória instâncias de *beans* de entidade que possam representar diferentes instâncias do mesmo tipo de dados em um meio de armazenamento. Para garantir a consistência dos dados, as instâncias são periodicamente sincronizadas com o meio de armazenamento. O *container* define a freqüência de sincronizações através de transações.

A persistência de *beans* de entidade pode ser de dois tipos: gerenciada pelo próprio *bean* (*Bean Managed Persistence* – BMP) ou gerenciada pelo *container* (*Container Managed Persistence* – CMP).

Nos *beans* de entidade BMP, o desenvolvedor do *bean* define a própria lógica de acesso aos dados. Deve, portanto, implementar os métodos **ejbLoad()** e **ejbStore()**, respectivamente para carregar e armazenar dados no meio de armazenamento.

Quando o *bean* de entidade é do tipo CMP, o desenvolvedor do *bean* não implementa nenhuma lógica de persistência dos dados no *bean*, pois o *container* garante a persistência realizando as operações necessárias. Nos *beans* CMP, os campos que representam informações que devem ser persistentes precisam ser públicos (uso da diretiva *public*), para o *container* gerenciar a persistência dessas informações. Todos os campos que devem ser manipulados pelo *container* devem estar especificados no descritor do *bean*.

***Beans* Orientados a Mensagens (*Message-Driven Beans*):**

São objetos que funcionam como *message-listeners*. São similares aos *beans* de sessão pois também representam ações no sistema, isto é, implementam processos da lógica de negócios. Também não armazenam dados. A diferença é que os *beans* orientados a mensagens só podem ser chamados através de mensagens. Dessa forma, oferecem uma interface assíncrona através da qual os clientes podem interagir com o *bean*.

Beans orientados a mensagens permitem o uso de messaging para acessar um sistema EJB (interagem com o JMS – *Java Message Service*). Cada *bean* desse tipo é associado a uma fila de mensagens particular, sendo que qualquer mensagem que chegar a essa fila é entregue pelo *container* para um instância desse *bean*. Clientes que queiram usar um serviço de um *bean* orientado a mensagens devem enviar uma mensagem para a fila associada ao *bean*.

2.2.4. Serviço de Transações

Uma transação [GRA1993] é uma unidade atômica de trabalho – atômica porque é indivisível (ou todas as operações são realizadas ou então nenhuma é efetuada) e de trabalho porque é composta de operações de modificação em dados de uma meio persistente de armazenamento (fig. 2.7).

Figura 2.7 – Esquema de uma transação com operações encapsuladas

Além disso, uma transação deve possuir isolamento aparente de outras transações (executar como se fosse a única transação no sistema).

Para um meio de armazenamento suportar transações, deve ser capaz de atender cinco propriedades, chamadas de propriedades ACID:

1. Atomicidade: todas as operações agrupadas em uma transação devem ser concluídas com sucesso. Caso ocorra uma falha em uma das operações todas as outras já realizadas devem ser desfeitas. Caso exista na transação alguma operação impossível de ser realizada, nenhuma operação deverá ser efetuada. Dessa forma, uma transação executa por completo ou não é efetuada.
2. Consistência: uma transação deve respeitar as regras de consistência do meio de armazenamento, isto é, após a execução da transação os dados devem continuar em um estado consistente, como estavam antes do início da transação. Um estado consistente satisfaz todas as restrições do esquema de dados do meio de armazenamento.
3. Isolamento: várias transações podem estar sendo executadas concorrentemente. Uma transação deve executar como se nenhuma outra estivesse em execução naquele momento.
4. Durabilidade: após a conclusão de uma transação, os efeitos dela sobre os dados nunca devem ser perdidos.

Uma transação normal, composta por uma série de operações delimitadas por uma demarcação de transação, é chamada de transação simples (*flat transaction*). Mas existe também um outro tipo de modelo, no qual uma transação pode abrigar outras transações. São chamadas de transações aninhadas (*nested transactions*). Nesse caso, as transações contidas dentro de outra transação (chamadas de sub-transações). Caso uma transação aninhada contenha outras duas transações, essas só terão resultados duráveis se a transação que as abriga terminar com sucesso. [MARSH1998]

2.2.5 Transações no EJB

Em uma transação SQL existem dois tipos de comandos, os de demarcação da transação e os de modificação dos dados. O processamento da transação (demarcação da transação) é feito pelo gerenciador de transações e a modificação dos dados é realizada pelo gerenciador de recursos. Um sistema gerenciador de banco de dados (SGBD) realiza essas duas tarefas. Mas no EJB, o gerenciamento da transação é realizado pelo *container* EJB. O gerenciamento de recursos continua a cargo do meio de armazenamento, isto é, o SGBD deve deixar o *container* efetuar o gerenciamento das transações, mas deve continuar gerenciando os recursos (registros compartilháveis do banco de dados). Além disso, um *container* EJB não suporta transações aninhadas, executando apenas transações do tipo *flat*.

Em alguns casos, podemos ter mais de uma transação acessando mais de um recurso, por isso o gerenciador de transações do *container* EJB é chamado de coordenador de transações (*transaction coordinator*). O coordenador de transações do *container* EJB suporta três tipos de demarcações de transação: gerenciada pelo *container*, gerenciada pelo *bean* e demarcação feita pelo cliente.

Demarcação de Transações Gerenciada pelo *Container* (*Container-Managed Transaction Demarcation*)

Quando a demarcação de transações é realizada pelo *container*, a interação do *bean* com o *container* e o meio de armazenamento é feita de forma implícita através da configuração do descritor do *bean*. As informações contidas no descritor indicam quando o *container* EJB deve iniciar e terminar transações.

Quando um *bean* de sessão deseja usar o gerenciamento de transações do *container*, deve sempre especificar no descritor que deseja utilizar esse gerenciamento. Já para os *beans* de entidade isso não se faz necessário, pois eles sempre usam o gerenciamento de transações do *container*.

Todo método publicado na interface remota ou *home* deve estar presente em um elemento *container-transaction* no descritor. Esse elemento está contido no elemento assembly-*descriptor*, que por sua vez está contido em outro elemento chamado ejb-jar. Abaixo os DTDs (definições de estruturas para a declaração de elementos no descritor) de XML desses elementos:

<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?, enterprise-beans, relationships?, assembly-descriptor?, ejb-client-jar?)>

<!ELEMENT assembly-descriptor (security-role*, method-permission*, container-transaction*, exclude-list?)>

<!ELEMENT container-transaction (description?, method+, trans-attribute)>

O elemento *assembly-descriptor* define as informações usadas para criar objetos EJB, através dos quais o cliente interage com o sistema. O elemento *container-transaction* define informações relacionadas às transações. Ele identifica um ou mais métodos e especifica atributos transacionais dos mesmos. O elemento *method* identifica um método publicado na interface remota ou na interface *home*. Segue abaixo o DTD do elemento *method*:

<!ELEMENT method (description?, ejb-name, method-intf?, method-name, method-params?)>

O elemento *method-intf* só é necessário quando um mesmo nome de método define métodos diferentes nas interfaces *home* e remota. O elemento *method-name* indica o nome de um método e deve obrigatoriamente ser especificado. Pode ser usado o valor '*' como

indicativo de que a referência é para todos os métodos do *bean*. O elemento *method-params* é opcional e só é usado para diferenciar versões sobrecarregadas de métodos de mesmo nome.

O mais importante de todos esses elementos é o *trans-attribute* (contido no elemento *container-transaction*). É este elemento que indica as características transacionais a serem asseguradas pelo *container* quando o método é chamado. Os valores possíveis para *trans-attribute* estão na tabela 2.2.

Para abortar uma transação, o *bean* pode chamar o método **setRollbackOnly()**, informando ao *container* para evitar o *commit* (conclusão) da transação. O *bean* não aborta diretamente a transação (todo o controle da mesma é do *container*), apenas indica ao *container* que a transação deve ser abortada.

TABELA 2.2 - Valores do elemento trans-attribute [BOND2002, MAHAPATRA2000]

Elemento	**Significado**
NotSupported	O método acessa um gerenciador de recursos que não suporta um coordenador de transações externo. Qualquer contexto de transação corrente será suspenso pelo tempo da duração da execução do método.
Required	Um contexto transacional é garantido. Um contexto atual será usado será usado, se presente. Caso contrário, será criado um novo contexto.
Supports	Usa um contexto transacional válido, se disponível (age como em Required). Caso contrário usa um contexto não especificado (age como NotSupported).
RequiresNew	Um novo contexto transacional será criado. Qualquer contexto transacional existente que seja válido será suspenso até o fim da execução do método.
Mandatory	Um contexto transacional válido deve existir. Uma exceção será lançada pelo *container*, caso contrário. O contexto transacional existente será usado.
Never	Não deve existir um contexto transacional. Uma exceção será lançada pelo *container*, caso exista. O método é chamado com um contexto transacional não especificado (age como NotSupported).

Demarcação de Transações Gerenciada pelo *Bean* (*Bean-Managed Transaction Demarcation*):

Para um maior controle sobre transações, o *bean* não deve usar o gerenciamento pelo *container*. Neste caso, deve implementar sua própria lógica de controle de transações. Esse tipo de demarcação de transações só é possível para *beans* de sessão.

Se o sistema interage com um gerenciador de banco de dados relacional, o *bean* pode implementar o gerenciamento de transações usando JDBC. Caso contrário, deverá implementar o controle transacional usando a JTA (*Java Transaction API*).

Nesse caso, para iniciar uma transação o *bean* deve chamar o método **getUserTransaction()**, do seu *SessionContext*, obtendo então um contexto transacional. Depois, para iniciar a transação deve usar o método **begin()**. Para completar a transação pode usar ***commit*()** ou **rollback()**. No primeiro caso, a transação é terminada com sucesso e os resultados são armazenados de forma persistente no meio de armazenamento. No segundo caso, a transação é abortada e todas as operações já efetuadas por ela são desfeitas, de forma que os dados voltam ao estado que estavam antes da transação. A qualquer momento pode ser obtido o status da transação utilizando o método **getStatus()**. Os possíveis estados são (tabela 2.3):

TABELA 2.3 – Estados de uma transação [BOND2002]

Constante	**Significado**
STATUS_NO_TRANSACTION	Não existe transação ativa
STATUS_ACTIVE	Uma transação está ativa e pode ser usada
STATUS_MARKED_ROLLBACK	Uma transação está ativa mas está marcada para ser desfeita (*rollback*). Qualquer tentativa de dar um *commit* na transação resultará em uma exceção **javax.transaction.RollbackException**.

O descritor de um *bean* que utiliza gerenciamento de demarcação de transações pelo *bean* deve indicar o elemento *transaction-type* como *Bean*. Não é necessário nenhum elemento *container-transaction* no elemento *assembly-descriptor*.

Demarcação de Transações pelo Cliente (*Client-Demarcated Transactions*)

Nesse tipo de demarcação de transações, o próprio cliente inicia uma transação e propaga ela aos *enterprise beans*. Para isso, o cliente deve obter um contexto *UserTransaction* através do JNDI (*Java Naming Directory Interface*).

2.2.6 JOnAS

O JOnAS (*Java Open Application Server*) [OBJ2004] é um servidor de aplicação, escrito na linguagem Java, que atende às especificações do J2EE. É um projeto da *ObjectWeb*, comunidade que desenvolve componentes de *middleware* sobre a licença *open source*. O servidor JOnAS pode ser usado como um *container* EJB (para executar componentes EJB), um *container* Web (para executar JSPs e Servlets) ou como um servidor J2EE (executa componentes EJB e Web).

O JOnAS apresenta os seguintes serviços [OBJ2004] [CECCHET2003]:

1. Serviço de comunicação e nomes (*Communication and Naming Service*): permite o uso de diferentes ambientes de processamento distribuído , entre os quais estão o RMI, o *Jeremie* e o RMI-IIOP. Esse serviço oferece suporte a mais de um desses ambientes em tempo de execução, de forma que mudanças no protocolo de comunicação não exigem a reinstalação dos componentes. A API JNDI também é utilizada para oferecer aos componentes os serviços de localização de objetos remotos e referências de recursos.
2. Serviço de *container* EJB (*EJB Container Service*): este serviço permite ao servidor JOnAS atuar como um *container* para os componentes EJB. Permite que componentes sejam instalados e executados, oferecendo todos os serviços implícitos que um

container EJB deve prover aos *beans*, como persistência, controle transacional e segurança. Os arquivos EJB-jar (componentes EJB) podem ser carregados na inicialização do servidor ou em tempo de execução.

3. Serviço de *container* Web (*Web Container Service*): este serviço permite a execução de um servidor Servlet/JSP na mesma máquina virtual Java (JVM - *Java Virtual Machine*) do JOnAS e carrega aplicações Web nesse servidor. Atualmente podem ser usados o *Tomcat* ou o *Jetty* como servidores *Servlet*/JSP. São então considerados como *web container*s, que permitem aos componentes web o acesso a recursos do sistema e componentes EJB. Os Servlets e JSPs, instalados no *web container*, são responsáveis por definir a lógica de apresentação dos dados na arquitetura J2EE.
4. Serviço EAR (*EAR Service*): é encarregado de carregar aplicações J2EE completas. Essas aplicações completas são arquivos EAR (*Enterprise Application Archive*) [BOND2002], que contém arquivos EJB-jar (que são entregues ao *container* EJB) e/ou arquivos war (que são entregues ao *container* Web).
5. Serviço de Transações (*Transaction Service*): oferece gerenciamento de transações para os componentes EJB. É capaz de lidar com vários recursos simultaneamente, através de conexões JDBC, mensagens e Adaptadores de Recursos (Conectores Java). Utiliza um monitor de transações desenvolvido pela *ObjectWeb* (JOTM), que pode estar distribuído em vários servidores JOnAS, permitindo que uma

transação envolva componentes localizados em servidores diferentes. Suporta transações gerenciadas por *beans* e transações gerenciadas pelo *container*.

6. Serviço de banco de dados (*Database Service*): cria objetos *Datasource*, um padrão JDBC para gerenciar conexões com um banco de dados. Permite a criação de *pools* de conexões.
7. Serviço de Segurança (*Security Service*): implementa os mecanismos de autenticação para acessar componentes EJB, realizando o mapeamento entre os papéis (*roles*) e as identificações de usuários. Um usuário só pode acessar um componente se estiver em pelo menos um dos papéis do conjunto definido no descritor do componente.
8. Serviço de Mensagens (*Messaging Service*): utiliza uma implementação do JMS (*Java Message Service*) para permitir a utilização de chamadas assíncronas de métodos EJB contidos em *beans* orientados a mensagens.
9. Serviço de Adaptador de Conexão de Recurso J2EE (*J2EE CA Resource Service*): permite a utilização de diferentes EIS (*Enterprise Information System* – banco de dados ou um sistema integrado). Este serviço é responsável por instalar no servidor conectores (adaptadores de recursos), que por sua vez realizam a conexão do servidor de aplicação, dos componentes da aplicação e do EIS.
10. Serviço de Gerenciamento (*Management Service*): é um serviço baseado na JMX (*Java Management eXtensions*), necessário para a administração de um servidor JOnAS através de um

console de administração JOnAS. Um servidor a ser monitorado pelo console deve possuir este serviço rodando.

11. Serviço de Correio Eletrônico (*Mail Service*): este serviço gerencia os recursos necessários para que componentes da aplicação possam enviar mensagens de e-mail através do *JavaMail*.

3 Projeto da Ferramenta Explore

A proposta deste trabalho de graduação é desenvolver uma ferramenta de monitoramento para o servidor JOnAS chamada Explore. O objetivo da ferramenta Explore é permitir, através de uma interface gráfica, que um usuário identifique os *beans* instalados em um servidor JOnAS. A Explore deve listar as aplicações J2EE instaladas, os componentes J2EE, os *beans* que fazem parte destes componentes, e as propriedades estáticas e transacionais desses *beans*. Dessa forma, a Explore pode ser usada para um monitoramento de aplicações em fase de teste, já que através dela o usuário pode verificar se sua aplicação está corretamente instalada e funcionando com as propriedades adequadas. O desenvolvedor pode alterar automaticamente as propriedades de um *bean* em tempo de instalação do *bean*. Essas alterações são apresentadas em modo texto no descritor do *bean* juntamente com outras informações.

O Explore divide-se em dois componentes (*i*) um servidor de monitoramento, chamado **ExploreServer**, que será inicializado como um serviço do servidor JOnAS e (*ii)* uma aplicação gráfica cliente, chamada **ExploreClient**, que permitirá acesso remoto ou local ao serviço de monitoramento.

3.1 ExploreServer

O **ExploreServer** atua como um serviço do servidor JOnAS, sendo, portanto, inicializado junto com outros serviços. Este serviço recebe requisições de informações de um cliente (**ExploreClient**), examina a estrutura de diretórios do JOnAS recolhendo as informações necessárias e as envia ao cliente. Especificamente, as funções do **ExploreServer** são:

1. esperar o **ExploreClient** conectar-se ao serviço;
2. receber as requisições do **ExploreClient**;
3. recolher os dados necessários para o processamento da requisição do **ExploreClient**;
4. retornar o resultado do processamento da requisição ao **ExploreClient**.

O servidor JOnAS armazena todas as aplicações (as quais devem ser executadas em um *container* EJB ou Web) em um diretório chamado "*apps*". Essas aplicações são arquivos EAR (*Enterprise Application Archive*), os quais contém arquivos JAR (componentes EJB) e/ou WAR (componentes Web). Dentro desse diretório existe também outro chamado "*autoload*", no qual são armazenadas as aplicações que são automaticamente instaladas na inicialização do JOnAS. O **ExploreServer** deve encontrar o diretório de instalação do JOnAS e examinar os diretórios *apps* e *autoload*, identificando as aplicações armazenadas. Realizada essa identificação, deve extrair os

arquivos JAR (se existirem) para acessá-los. Em cada arquivo JAR, o **ExploreServer** procura pelo descritor dos *beans* do componente. Esse descritor é um arquivo chamado “ejb-jar.xml”, é único para cada arquivo JAR e está sempre armazenado no sub-diretório “/META-INF” [SUN2001a].

Extraindo o descritor dos *beans* do componente, o **ExploreServer** pode, utilizando um *parser* XML (classe responsável por ler um arquivo XML identificando *tags* e *strings*) [BOND2002], recuperar as propriedades dos *beans*. A especificação EJB 2.0 define que um arquivo JAR é um componente J2EE, e, portanto, deve possuir um descritor de *beans* no qual devem ser configuradas as propriedades de todos os *beans*. Assim, acessando apenas o arquivo “ejb-jar.xml” é possível obter todas propriedades dos *beans* de um componente. Segue abaixo o algoritmo que descreve as ações realizadas pelo **ExploreServer** para obter uma lista dos descritores dos *beans*:

1. Encontrar diretórios “apps” e “apps/autoload” no diretório base de instalação do JOnAS
2. Criar lista de arquivos EAR (LISTA-EAR)
3. Criar lista de arquivos JAR (LISTA-JAR)
4. Criar lista de descritores XML (LISTA-XML)
5. Para cada elemento em LISTA-EAR:
    a. Extrair arquivos JAR
    b. Inserir em LISTA-JAR
6. Para cada elemento em LISTA-JAR:

a. Extrair arquivos "META-INF/ejb-jar.xml", renomeando para "nome-do-jar.xml"
b. Inserir em LISTA-XML

Tendo um lista de descritores de *beans*, o **ExploreServer** pode realizar o *parse* de cada um desses descritores, de forma a obter uma lista de *beans* e atributos dos mesmos:

1. Para cada descritor em LISTA-XML:
   a. Realizar o *parse* do arquivo xml, identificando cada *bean* e seus atributos

Os atributos de um *bean* podem ser genéricos (presentes tanto em *beans* de sessão quanto em *beans* de entidade) ou específicos (inerentes a um tipo de *bean*). A tabela 3.1 lista os atributos genéricos de um *bean*, enquanto a tabela 3.2 lista os atributos específicos.

Todas as propriedades mostradas nas tabelas 3.1 e 3.2 são configuradas no descritor do *bean* dentro de uma *tag* "*enterprise-beans*". Além dessas propriedades, são configuradas no descritor propriedades transacionais dos métodos de um *bean*. Essa configuração é realizada dentro da *tag* "*assembly-descriptor*" do descritor. Dentro dessa *tag*, definem-se elementos "*container-transaction*", nos quais são definidos um ou mais métodos de vários *beans*, que são então associados a um atributo transacional (elemento "*trans-attribute*"). Dessa forma, associado a cada *bean*, o **ExploreServer** cria uma lista de métodos desse *bean*, na qual

armazena-se cada nome de método e seu atributo transacional. Vale lembrar que o elemento método pode ter seu valor setado para "*" no descritor, referenciando que todos os métodos do *bean* tem um mesmo atributo transacional. Nesses casos, a lista de métodos do *bean* possui apenas um elemento, de nome "*" (indicando que são todos os métodos do *bean*) e atributo transacional igual ao valor do elemento "trans-attribute" associado.

TABELA 3.1 – Atributos genéricos de um *bean*

Atributo	**Breve descrição**
ejb-name	*Nickname* do *bean*
home	Nome da interface *home* do *bean*
remote	Nome da interface remota do *bean*
local-home	Nome da interface *home* local
local	Nome da interface local
ejb-class	Nome da classe *Enterprise Bean*
transaction-type	Tipo de demarcação de transações: pelo *bean* ou pelo *container*
reentrant	Se for *true*, o *bean* pode chamar a si mesmo através de outro *bean*

TABELA 3.2 – Atributos específicos de um bean

Tipo de *bean*	**Atributo**	**Breve descrição**
Sessão	*session*-type	Define se o *bean* é conversacional ou não (*stateful* ou *stateless*)
Sessão	*session*-timeout	Tempo de *timeout* em segundos; se um cliente está chamando métodos do *bean*, deve dar *timeout* após expirar esse tempo
Entidade	prim-key-class	Nome da classe de chave primária do *bean* de entidade
Entidade	cmp-field	Especifica quais campos da classe *bean* de entidade são persistentes (apenas para *beans* de entidade com persistência gerenciada pelo *container* – CMP)
Entidade	cmp-version	Versão do gerenciamento de persistência do *bean* pelo *container*

Todo esse processo de localizar e realizar o *parse* de descritores dos componentes EJB pode ser feito no momento em que o **ExploreClient** conecta-se com o **ExploreServer** ou por demanda, conforme o usuário solicitar.

No primeiro caso, cria-se uma lista de estruturas (PROPRIEDADES-BEAN - armazena as propriedades de um *bean*) na qual serão armazenadas todas as propriedades de todos os *beans*.

Os campos que formam a estrutura (PROPRIEDADES-BEAN) responsável por armazenar as propriedades de um *bean* são:

1. Nome da aplicação
2. Nome do componente
3. Nome do *bean*
4. Tipo do *bean* (sessão, entidade ou orientado a mensagens)
5. Nome da interface *home*
6. Nome da interface remota
7. Nome da interface *local-home*
8. Nome da interface *local*
9. Nome da classe *Enterprise Beans*
10. Tipo de transação
11. Atributo *reentrant*
12. Tipo de sessão
13. Tipo de persistência
14. Nome da classe chave primária
15. *Timeout* de sessão
16. Versão CMP
17. Campos CMP (lista com os nomes dos campos "*cmp-field*")
18. Métodos do *bean* (lista de estruturas MÉTODO-BEAN)

Os três primeiros campos são usados para localizar *beans* em uma lista de estruturas PROPRIEDADES-BEAN mediante algum tipo de solicitação. Todos os campos são cadeias de caracteres. Estas são obtidas no processo de *parse* do descritor. As exceções são os campos 1, 2, 17 e 18. Os campos 1 e 2 também são cadeias de caracteres, mas não são obtidos no descritor. São na verdade os nomes do arquivo EAR e do arquivo JAR, respectivamente. Já o campo 17 é uma lista de cadeias de caracteres, sendo que cada cadeia corresponde a um dos elementos "cmp-field" do descritor. O campo 18, por sua vez, é uma lista de estruturas MÉTODO-BEAN, as quais armazenam os nomes de métodos do *bean* e os atributos transacionais associados aos mesmos. Essa estrutura é composta pelos seguintes campos:

1. Nome do *bean*
2. Nome do método
3. Atributo transacional do método

A indexação dessa lista pode ser realizada pelo nome da aplicação (permite obter todos os *beans* de uma aplicação), pelo nome de componente (permite obter todos os *beans* que formam um componente EJB) ou pelo nome de *bean* (permite obter as propriedades de um *bean* específico). Assim, na estrutura "propriedades do *bean*", são adicionados três campos, usados para realizar buscas na lista:

1. nome da aplicação (nome do arquivo EAR);

2. nome do componente (nome do arquivo JAR);
3. nome do *bean* (elemento "*ejb-name*" do descritor).

Através desses três campos, o **ExploreServer** pode retornar ao seu cliente os seguintes tipos de solicitações:

1. lista de aplicações que estão instaladas no servidor JOnAS;
2. lista de componentes que estão instalados no servidor JOnAS;
3. lista de *beans* que estão instalados no servidor JOnAS;
4. quais componentes EJB formam uma determinada aplicação;
5. quais *beans* formam um determinado componente.

Conseqüentemente, cria-se um elo estrutural de informações sobre aplicações de um servidor JOnAS, como mostra a figura 3.1.

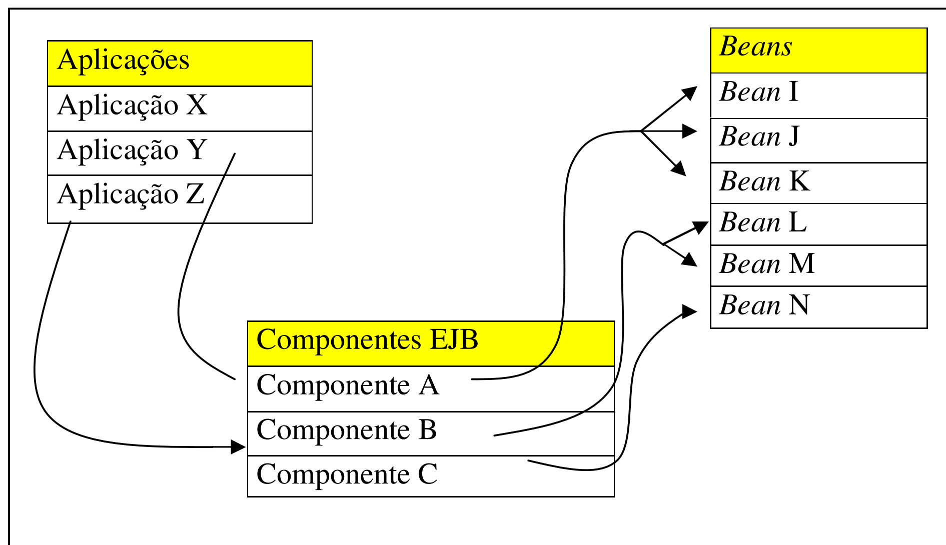

Figura 3.1 – Elo de informações de aplicações de um servidor JOnAS

Na figura 3.1, a aplicação X não possui nenhum componente EJB (é uma aplicação Web – arquivo EAR possui apenas arquivos WAR). A aplicação Y possui um componente EJB, chamado A. Este componente, por sua vez, é formado por três *beans* (I, J e K). Já a aplicação Z possui dois componentes EJB (B e C). O componente B contém os *beans* L e M, enquanto o componente C é formado apenas pelo *bean* N.

Ao final do processo de localizar e obter todas as informações dos descritores, o **ExploreServer** terá uma lista de estruturas PROPRIEDADES-BEAN, na qual estarão armazenadas as propriedades de todos os *beans* do servidor JOnAS. Deve então criar para o **ExploreClient** uma lista contendo uma estrutura APLICAÇÃO, a qual é formada pelo nome da aplicação, uma lista de componentes e uma lista de *beans*, e é utilizada para seleção no **ExploreClient**.

Caso o número de *beans* seja muito grande, o processo de realizar o *parse* de todos os descritores pode consumir muito tempo. Nesse caso, a estratégia de inicializar a estrutura (localizar e realizar o *parse* conforme demanda do usuário) pode apresentar-se como alternativa mais viável. Nessa estratégia, o **ExploreServer** inicialmente localiza os diretórios com os *beans* e cria uma lista de estruturas APLICAÇÃO. Apenas quando o usuário selecionar um componente da aplicação será realizado o *parse* de um descritor. Assim, se o usuário quiser apenas informações de um componente ou

aplicação específicos, economiza-se tempo de computação, já que só será realizado o *parse* de descritores pertencentes aos componentes que forem solicitados.

3.2 ExploreClient

O **ExploreClient** é uma aplicação que implementa uma interface gráfica, responsável por repassar as solicitações do usuário ao **ExploreServer** e apresentar os resultados. Definem-se as seguintes funções:

1. localiza o servidor JOnAS e realiza conexão com o **ExploreServer**;
2. apresenta configuração inicial ao usuário: lista de aplicações do servidor JOnAS e lista de componentes dessas aplicações;
3. recolhe solicitações do usuário e as processa para então solicitar os dados necessários ao **ExploreServer**;
4. apresenta os resultados.

A conexão entre o **ExploreClient** e o serviço implementado pelo **ExploreServer** pode ser local ou remota. O que definirá o tipo de conexão é a informação do arquivo de configuração, que deve ser armazenado junto com a aplicação cliente (**ExploreClient**). Nesse arquivo deve estar o IP da máquina onde está rodando o JOnAS e o **ExploreServer**. Caso o valor do IP esteja setado como *localhost*, a

conexão é local. Caso contrário, o **ExploreClient** deverá utilizar o IP desse arquivo para tentar conectar-se ao serviço.

Realizada a conexão, o **ExploreClient** está pronto para fazer requisições ao serviço de monitoramento. Essas requisições são pedidos de informações de aplicações, componentes e *beans* do servidor JOnAS. Definem-se as seguintes operações:

1. listar componentes de uma aplicação;
2. listar *beans* de um componente;
3. listar propriedades de um *bean;*
4. listar todos os componentes instalados no servidor JOnAS;
5. listar todos os *beans* instalados no servidor JOnAS.

Logo após realizar a conexão com o **ExploreServer**, o **ExploreClient** solicita e lista as aplicações que estão armazenadas no servidor JOnAS. Quando o usuário selecionar uma das aplicações dessa lista, o **ExploreClient** deve solicitar ao **ExploreServer** uma lista dos componentes dessa aplicação. A seleção de um componente da lista deve listar todos os *beans* contidos nele. Finalmente, a seleção de um *bean* da lista deve listar todas as propriedades monitoradas do *bean*.

O **ExploreClient** implementa também dois botões, COMPONENTES e BEANS, os quais solicitam ao **ExploreServer** todos os componentes ou todos os *beans* armazenados no JOnAS, respectivamente.

A interface gráfica é formada por oito painéis, listados na tabela 3.3:

TABELA 3.3 – Painéis do ExploreClient

Nome do Painel	**Descrição**
Server-Applications	Lista as aplicações de um servidor JOnAS
App-Components	Lista os componentes EJB de uma aplicação
Component-Beans	Lista os *enterprise beans* de um componente
Bean-Info	Lista os atributos de um *bean*
All-Components	Lista todos os componentes EJB de um servidor JOnAS
All-Beans	Lista todos os *enterprise beans* de um servidor JOnAS
Info	Lista informações sobre um componente ou *bean* selecionado nos painéis All-Components ou All-*Beans*
Log-Explore	Lista informações sobre o *status* da aplicação Explore

Segue a descrição de cada um dos painéis listados na tabela 3.3.

3.2.1 Painel Server-Applications

Logo após a conexão com o **ExploreServer**, o **ExploreClient** solicita uma lista com todas aplicações armazenadas no servidor JOnAS. É função deste painel exibir esta lista ao usuário e permitir que o mesmo selecione uma aplicação da lista ativando o painel **App-Components**. Caso não seja possível obter a lista de aplicações, o painel **Server-Applications** deve ativar o painel **Log-Explore** exibindo uma mensagem de erro indicando o que ocorreu.

3.2.2 Painel App-Components

A ativação deste painel ocorre através da seleção de um nome de aplicação da lista do painel **Server-Applications**. O painel **App-Components** lista então todos os nomes de componentes que fazem parte da aplicação selecionada, permitindo a seleção de um destes nomes para listar os *beans* do componente (função esta exercida pelo painel **Component-Beans**). Caso não seja possível listar os componentes de uma aplicação, uma mensagem deve informar no painel **Log-Explore** se ocorreu um erro ou se a aplicação selecionada não possui componentes EJB (é uma aplicação Web somente).

3.2.3 Painel Component-Beans

A seleção de um componente no painel anterior implica na listagem de todos os *beans* que formam o componente selecionado. O painel **Component-Beans** exibe então todos os nomes desses *beans*. Novamente, diante de um erro, o mesmo deve ser informado no painel **Log-Explore**.

3.2.4 Painel Bean- Info

A função deste painel é exibir todas as propriedades de um *bean* selecionado no painel **Component-Beans**. Essa listagem de propriedades é padrão, de forma que lista sempre todos os atributos genéricos e específicos de um *bean*. Caso o *bean* não possua algum dos atributos específicos, o painel exibe apenas o nome do atributo sem nenhuma informação associada. Por exemplo, ao selecionar um *bean* de entidade, o campo "Tipo de Sessão" será apresentado com nenhum valor, pois é específico de *beans* de sessão. A exibição de um campo vazio indica que o *bean* não possui este campo no seu descritor. Todas as propriedades listadas neste painel devem ser atualizadas sempre que outro *bean* for selecionado.

3.2.5 Painel All-Components

Este painel somente aparece na interface quando o botão relacionado com o mesmo for ativado (botão "**allComponents**"). Quando isso ocorrer, o **ExploreClient** solicita ao **ExploreServer** uma lista de todos os componentes instalados. Essa lista é então exibida neste painel. Esta lista é atualizada sempre que o botão for reativado. A seleção de um componente da lista ativa o painel **Info**, o qual mostrará o nome do componente e o nome da aplicação do qual ele faz parte. A partir dessa informação o usuário pode localizar o componente e exibir seus *beans* utilizando os três primeiros painéis. Se existir mais de uma cópia do componente (pertencentes à aplicações diferentes) ou outro componente com mesmo nome, o painel exibe várias vezes o mesmo nome. A diferenciação entre esses nomes se dá nas informações associadas a ele, exibidas no painel **Info**.

3.2.6 Painel All-Beans

Este painel também tem sua ativação por meio de um botão (chamado "**allBeans**"). Quando ativado, o **ExploreClient** solicita uma lista de todos os *beans* armazenados no servidor JOnAS monitorado. A reativação do botão permite a atualização dessa lista. Ao selecionar um *bean* da lista, o painel **Info** é ativado, exibindo o caminho até esse *bean* (a qual componente e aplicação pertence). Novamente, a existência de um mesmo *bean* em vários componentes diferentes

implica na listagem do nome do *bean* mais de uma vez (o número de vezes que será listado será igual ao número de cópias do *bean*), e a diferenciação se dará nas informações do painel **Info**.

3.2.7 Painel Info

Este painel somente é ativado mediante a seleção de um componente ou de um *bean* das listas dos painéis **All-Components** ou **All-Beans**. Se ativado pelo painel **All-Components**, exibe o nome da aplicação do qual o componente faz parte. Mas se for ativado pelo painel **All-Beans**, exibe o nome do componente do qual faz parte e da aplicação que contém esse componente. Essa informação pode ser utilizada pelo usuário para localizar onde está um determinado *bean* e então acessar suas propriedades através dos três primeiros painéis.

3.2.8 Painel Log-Explore

Este painel é responsável por exibir um *log* de operação da ferramenta Explore. Exibe ações e/ou erros do **ExploreClient** e do **ExploreServer**. Também deve exibir outras informações úteis ao usuário, como a indicação de que uma aplicação é totalmente Web. Nesse caso, a aplicação não possui componentes EJB e os painéis ficam vazios.

3.3 Casos de uso – operações dos componentes da ferramenta Explore

Ao iniciar o servidor JOnAS, o **ExploreServer** é automaticamente acionado, como um serviço do servidor. Os casos de uso dessa inicialização estão na tabela 3.4.

TABELA 3.4 – Casos de uso da inicialização do ExploreServer

1	Localiza e obtém uma referência para o diretório apps do servidor JOnAS.
2	Examina conteúdo do diretório apps e de seu sub-diretório autoload.
3	Cria uma lista contendo o nome de cada arquivo EAR nesses dois diretórios.
4	Espera conexão do **ExploreClient**
5	Examina cada arquivo EAR e extrai os arquivos JAR armazenados no mesmo. Cria uma lista de referências contendo todos os arquivos JAR.
6	Para cada arquivo JAR, extrai o arquivo META-INF/ejb-jar.xml, renomeando para "nome-do-JAR.xml". Cria uma lista desses descritores.
7	Se estiver configurado para criar automaticamente todas as estruturas PROPRIEDADES-*BEAN*, realiza o parse de cada um dos arquivos da lista de descritores. Salva os resultados em uma lista de PROPRIEDADES-*BEAN*.

Alternativas aos casos da tabela 3.4:

1. Caso 1: Aborta a operação de inicialização e espera conexão com o **ExploreClient**. Ao conectar, informa este de que não foi possível localizar os diretórios do JOnAS.
2. Caso 5: Se não for possível extrair um ou mais arquivos JAR, informa ao **ExploreClient** o nome de cada um desses arquivos que falharam.
3. Caso 6: Se não for possível extrair os descritores, informa ao **ExploreClient** quais descritores falharam.
4. Caso 7: Se não estiver configurado para inicializar automaticamente as estruturas, aguarda solicitações do usuário para fazê-lo.

Realizadas essas operações iniciais, o **ExploreServer** envia toda a estrutura resultante do parse dos descritores ao **ExploreClient**. Caso esteja configurado para não realizar esse *parse* inicial, aguarda solicitações do usuário para enviar os dados conforme necessário.

A primeira tarefa do **ExploreClient**, ao ser inicializado, é ler o arquivo de configurações para obter o IP do servidor onde estão sendo executados o JOnAS e o **ExploreServer**. Utilizando esse IP, realiza a conexão com o **ExploreServer**. Os casos de uso da inicialização do **ExploreClient** são listados na tabela 3.5.

TABELA 3.5 – Casos de Uso da Inicialização do ExploreClient

1	Lê arquivo de configurações e obtém IP do servidor
2	Conecta com o **ExploreServer**
3	Recebe e armazena a lista de estruturas de propriedades dos *beans*, se já disponível
4	Lista aplicações do servidor JOnAS
5	Espera usuário efetuar alguma ação na interface

Alternativas para os casos de uso da tabela 3.5:

1. Caso 1: Se não encontrar o arquivo de configurações, informa o erro ao usuário e aborta. Caso encontre mas não existir um IP válido, aborta informando o tipo de erro ao usuário.
2. Caso 2: Se não for possível conectar ao serviço do **ExploreServer**, aborta informando o erro ao usuário.
3. Caso 3: Se a lista ainda não estiver disponível, solicita lista de nomes de aplicações ao **ExploreServer**, exibe a lista e espera alguma ação do usuário.

Terminado o processo de inicialização do **ExploreServer** (ver tabela 3.4) e do **ExploreClient** (ver tabela 3.5), as ações efetuadas por cada um destes componentes passam a ser regidas pelo usuário. A figura 3.2 mostra uma interação típica do usuário com a ferramenta Explore.

Nessa figura, o **ExploreClient** conecta ao **ExploreServer**, o qual reúne todas as propriedades dos *beans* em uma lista de estruturas

"PROPRIEDADES-BEAN", enviando essa lista ao **ExploreClient**. Inicia-se então o processo de interação do usuário com o **ExploreClient**. Cada vez que o usuário selecionar um nome em uma das listas dos painéis da interface gráfica, o **ExploreClient** processa o pedido e exibe as informações da aplicação, componente ou *bean* selecionado. O **ExploreServer** só é acionado novamente quando o usuário solicitar nova conexão. Nesse caso, o **ExploreClient** solicita novo envio dos dados ao **ExploreServer**.

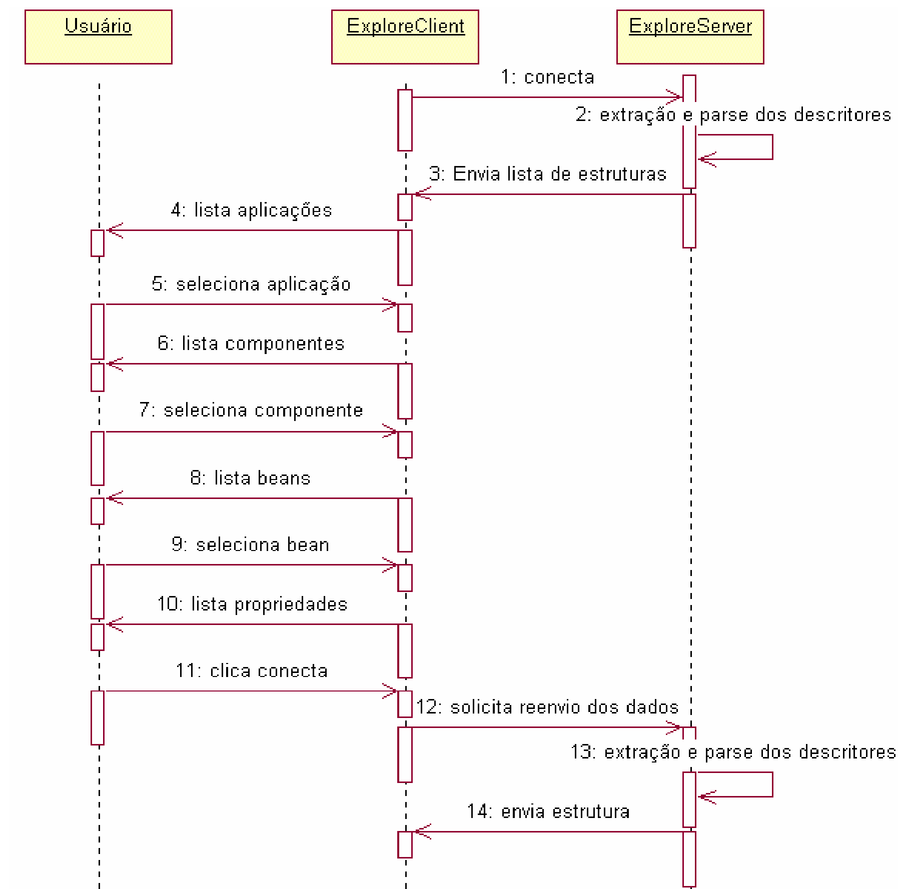

Figura 3.2 – Interação básica Usuário-**ExploreClient**-**ExploreServer**

Já as figuras 3.3 e 3.4 mostram as respostas do **ExploreClient** mediante ações do usuário sobre os botões "**allComponents**" e "**allBeans**".

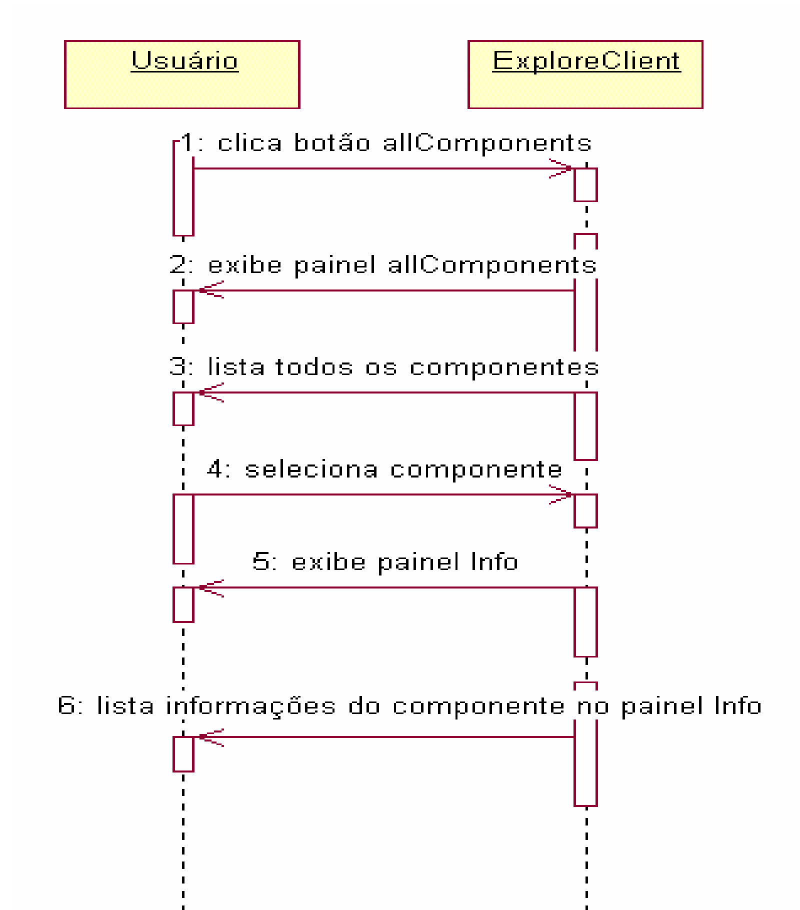

Figura 3.3 – Interação Usuário-**ExploreClient** através do botão **allComponentes**

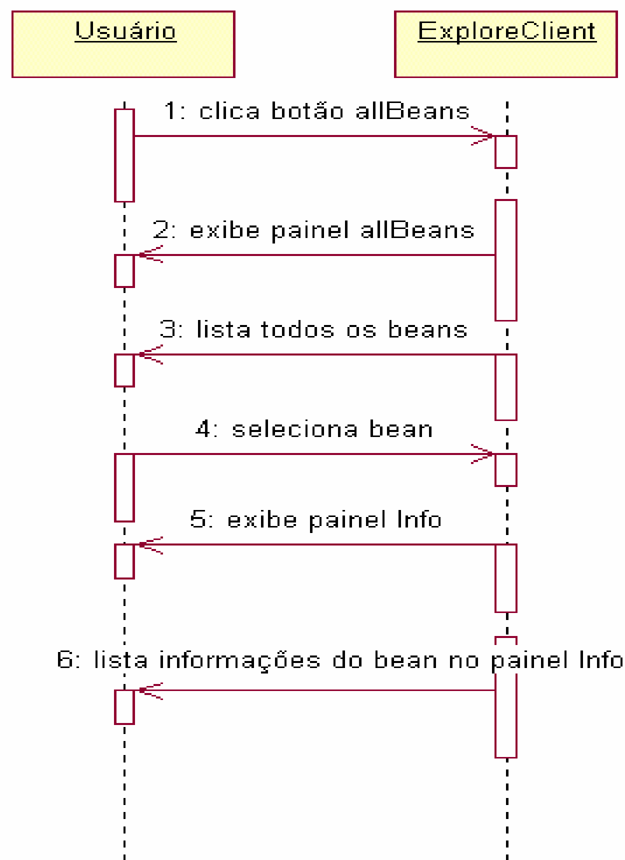

Figura 3.4 – Interação Usuário-**ExploreClient** através do botão **allBeans**

4 Implementação da Ferramenta Explore

A implementação da ferramenta Explore divide-se em duas classes principais, as quais executam as funções de cliente e servidor. Essas classes, por sua vez utilizam outras que ou modelam dados (as propriedades dos beans) ou executam funções auxiliares (extração de arquivos, *parse* de documentos XML, etc).

As duas classes principais são **ExploreServer** e **ExploreClient**. Estas implementam, respectivamente, as funções de servidor e cliente da ferramenta Explore.

A classe **ExploreServer** executa as funções de localizar os descritores dos *beans*, esperar o cliente da ferramenta (**ExploreClient**) conectar-se, realizar o *parse* dos descritores e enviar os dados (propriedades dos *beans*) ao **ExploreClient**. Para isso, utiliza as classes auxiliares **FileCopy**, **JarResources**, **aplicacao** e **leitorXML**.

As classes que modelam dados são **propriedadesBean** e **metodoBean**. Uma instância de **propriedadesBean** armazena as propriedades de um *bean*. Dentre essas propriedades (armazenadas nos descritores dos *beans*), estão os métodos e atributos transacionais dos mesmos, que um *bean* implementa. Para cada método definido no descritor de um *bean*, a ferramenta Explore cria uma instância da classe **metodoBean**, a qual é associada à instância de **propriedadesBean** correspondente.

A extração dos descritores dos *beans*, necessários para obter as propriedades dos mesmos, é realizada com o auxílio das classes **FileCopy** e **JarResources**. **FileCopy** é utilizada para realizar cópias

dos arquivos de aplicação J2EE (arquivos EAR) armazenados em um servidor JOnAS, convertendo-os para arquivos no formato ZIP. Já a classe **JarResources** é utilizada para extrair os arquivos de componentes de aplicação (arquivos JAR) dos arquivos ZIP anteriores, e, em seguida, extrair os descritores dos *beans* (arquivos XML) desses arquivos JAR.

A criação das instâncias de **propriedadesBean**, que modelam as propriedades dos *beans* em um servidor JOnAS, é realizada no momento do *parse* dos descritores dos *beans*. Esse processo é realizado pelas classes auxiliares **LeitorXML** e **XMLHandler**. Enquanto a classe **XMLHandler** é responsável por realizar o *parse* propriamente dito (lê arquivo XML, identifica os beans e as suas propriedades, cria um vetor de objetos **propriedadesBean**), a classe **LeitorXML** é responsável por instanciar um **XMLHandler** e repassar os resultados (vetor de **propriedadesBean**) ao **ExploreServer**.

Durante o *parse* dos descritores, o **ExploreServer** utiliza a classe aplicação para realizar o mapeamento entre os objetos **propriedadesBean** e os nomes dos componentes e aplicações que implementam esse *bean*. Ao final do processo de *parse*, o **ExploreServer** possui um vetor de objetos **propriedadesBean**, representando todas as propriedades dos *beans* armazenados no servidor JOnAS monitorado.

A classe **ExploreClient** é responsável por desenhar uma interface gráfica para o usuário, conectar ao **ExploreServer**, receber o vetor de **propriedadesBean** criado pelo **ExploreServer**, apresentar os resultados ao usuário e receber os pedidos do mesmo. Para apresentar

os dados ao usuário, a classe **ExploreClient** utiliza as classes **propriedadesBean** e **metodoBean**, que modelam as propriedades dos *beans* de um servidor JOnAS.

A figura 4.1 ilustra as classes que formam a ferramenta Explore, e os relacionamentos entre elas (as direções das setas indicam que uma classe da qual parte uma seta utiliza outra, a qual é destino da seta). Nota-se que tanto a classe **ExploreServer** quanto **ExploreClient** utilizam as classes **propriedadesBean** e **metodoBean** para modelar os dados. Porém, **ExploreServer** utiliza também outras classes, as quais auxiliam no desempenho de suas funções.

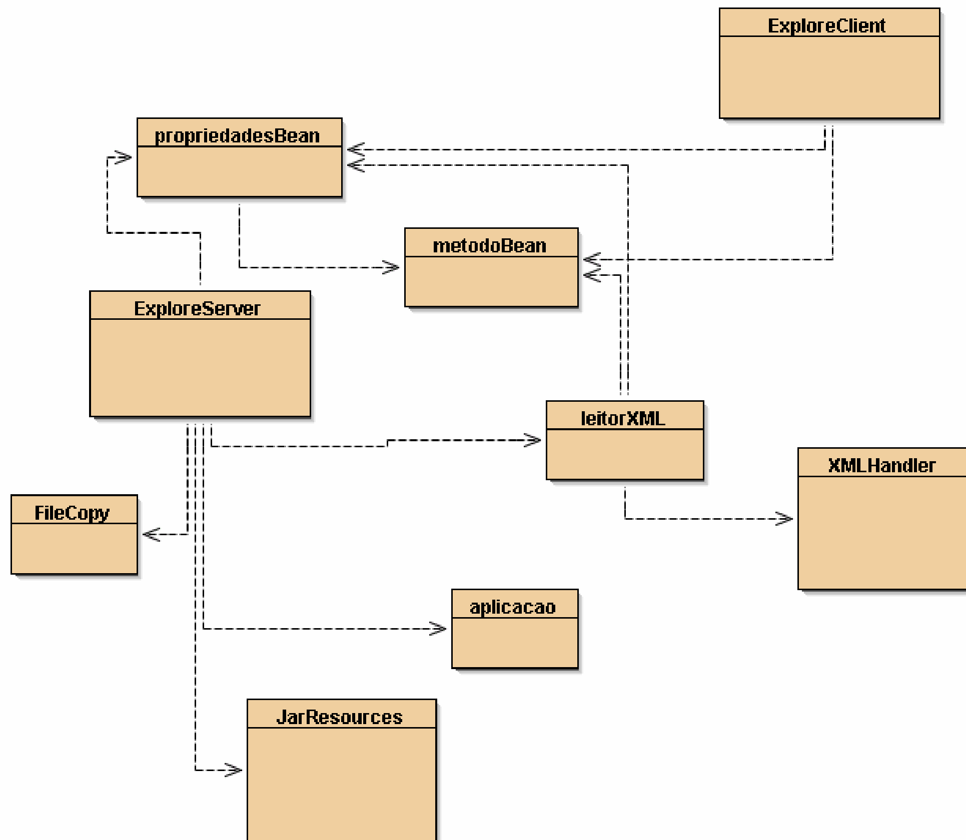

Figura 4.1 – Classes da ferramenta Explore

4.1 Classe PropriedadesBean

O objetivo da ferramenta Explore é reunir informações que descrevem as propriedades de um *enterprise bean*. Estas propriedades, configuradas no descritor de um componente J2EE, são recolhidas e armazenadas pelo **ExploreServer** em um objeto, o qual é então enviado ao **ExploreClient**. A classe que define esse objeto chama-se **propriedadesBean**, a qual implementa a interface *java.io.Serializable*. A implementação dessa interface permite que uma instância da classe **propriedadesBean** seja serializada para transmissão na rede, na forma de um fluxo de bytes.

Os atributos da classe são do tipo String, vetor de Strings e vetor de objetos **metodoBean**, como mostra a figura 4.2.

public class propriedadesBean
public String app public String componente public String tipo public String elementoEjbName public String elemento*Home* public String elemento*Remote* public String elementoLocal*Home* public String elementoLocal public String elementoEjbClass public String elemento*Session* public String elemento*Persistence* public String elementoPK public String elementoReentrant public String elemento*Transaction* public String cmpVersion public String timeout public Vector metodos public Vector cmpFields

Figura 4.2 – Classe **propriedadesBean**

A figura 4.2 mostra todos os atributos da classe **propriedadesBean**. A classe não implementa nenhum método, serve apenas como um repositório de propriedades de um *bean*. Os atributos do tipo String armazenam cadeias de caracteres correspondentes às propriedades encontradas no descritor do *bean*. O atributo **cmpFields** é um vetor de Strings, as quais correspondem aos nomes dos campos de um *bean* de entidade cuja persistência é gerenciada pelo *container*. O atributo "metodos" é um vetor de objetos da classe **metodoBean**. Esta classe define um método de um *enterprise bean*, como mostra a figura 4.3.

public class metodoBean
public String nomeEJB public String nome public String atribTrans

Figura 4.3 – Classe **metodoBean**

As classes **propriedadesBean** e **metodoBean** são, respectivamente, a implementação das estruturas "**PROPRIEDADES-BEAN**" e "**MÉTODO-BEAN**" definidas no projeto da ferramenta Explore. A tabela 4.1 mostra os nomes dos atributos das classes **propriedadesBean** e **metodoBean**, junto com as propriedades do *bean* que esses atributos designam.

TABELA 4.1 – Propriedades dos *beans* correspondentes aos atributos das classes

Atributo	Propriedade do *bean*
propriedadesBean	
App	Nome da aplicação da qual o *bean* faz parte*
Componente	Nome do componente do qual o *bean* faz parte*
Tipo	Tipo de *bean* (sessão, entidade ou orientado a mensagem)
elementoEjbName	Nome do *bean*
elemento*Home*	Nome da interface *Home*
elemento*Remote*	Nome da interface remota
elementoLocal*Home*	Nome da interface *Home* local
elementoLocal	Nome da interface local
elementoEjbClass	Nome da classe *Enterprise Bean*
elemento*Session*	Tipo de sessão
elemento*Persistence*	Tipo de persistência
ElementoPK	Nome da classe chave-primária
elementoReentrant	Atributo reentrant
elemento*Transaction*	Tipo de demarcação de transações
cmpVersion	Versão utilizada da persistência gerenciada pelo *container*
Timeout	Tempo de *timeout*
CmpFields	Campos persistidos pelo *container*
metodoBean	
NomeEJB	Nome do *bean* que implementa o método
Nome	Nome do método
AtribTrans	Atributo transacional do método

* Essas propriedades não estão no descritor do *bean*. São obtidas a partir dos nomes dos arquivos EAR e JAR que contém o *bean*.

4.2 Estratégia de *parse* dos *beans*

No projeto da ferramenta Explore, são definidas duas estratégias para o *parse* dos descritores dos *beans*. Na primeira, o **ExploreServer** realiza o *parse* de todos os descritores e então envia todos os dados (objetos **propriedadesBean**) para o **ExploreClient**. Na outra estratégia, o *parse* dos descritores é realizado sob demanda do cliente, com o **ExploreServer** enviando apenas os dados dos descritores que o **ExploreClient** solicitar.

Neste trabalho, foi implementada a primeira estratégia. Dessa forma, na execução do cliente, ocorrem basicamente duas trocas de informações:

1. **ExploreClient** conecta e informa estar pronto para receber os dados
2. **ExploreServer** realiza o *parse* dos descritores, cria um vetor de instâncias da classe **propriedadesBean** (resultantes do *parse*) e envia esse vetor

Optou-se por essa estratégia porque ela não sobrecarrega a máquina na qual estão rodando o **ExploreServer** e o servidor JOnAS. Normalmente, o cliente só terá de solicitar que os dados sejam atualizados após longos intervalos de tempo. Isso porque os dados que o **ExploreClient** recebe do **ExploreServer** tornam-se obsoletos apenas quando uma aplicação é atualizada (reinstalada no servido

JOnAS) ou quando uma nova aplicação é instalada (neste caso existem novos descritores cujos dados o **ExploreClient** ainda não possui). Instalações e atualizações de aplicações J2EE certamente não ocorrem com tanta freqüência, de forma que os dados que o **ExploreClient** possui normalmente refletirão o estado atual do sistema.

Ao final do processo de *parse* e envio dos dados, todo o trabalho passa a ser do **ExploreClient**, que deve examinar o vetor para exibir as respostas adequadas às solicitações do usuário.

4.3 Comunicação entre ExploreServer e ExploreClient

A estratégia de *parse* e envio dos dados escolhida apresenta os seguintes efeitos, quanto à comunicação entre o **ExploreServer** e o **ExploreClient**:

1. Quando o **ExploreClient** conectar e informar estar pronto, o **ExploreServer** envia o resultado do *parse* de todos os descritores. A quantidade de dados é possivelmente grande, considerando um servidor JOnAS com várias aplicações instaladas. Isso pode resultar em uma transmissão um pouco demorada de dados entre o **ExploreServer** e o **ExploreClient**.
2. Após o envio dos dados, que se dá no início da execução do **ExploreClient**, só se faz necessária nova transmissão quando o

usuário solicitar uma atualização dos dados. Neste ponto, a comunicação limita-se a curtas trocas de mensagens.

Surge então a necessidade de utilizar um protocolo que garanta confiabilidade na transmissão inicial dos dados. Tem que existir essa garantia da entrega porque os dados são transmitidos na forma de um fluxo de bytes (serialização), sendo reconstruído no **ExploreClient** (deserialição). Se um ou mais bytes forem perdidos na transmissão, o vetor de objetos **propriedadesBean** pode ser corrompido de tal forma que o **ExploreServer** tenha que reenviar todos os dados novamente, o que representaria um custo de tempo proibitivo.

O protocolo escolhido foi o TCP (*Transmission Control Protocol*), que garante a entrega dos dados ao custo de um *overhead* de comunicação [TAN1996]. Esse *overhead* não implica em um tempo maior do que o tempo que seria gasto caso fosse necessário repetir toda a transmissão devido a erros.

A conexão é feita por meio de *sockets* TCP. O **ExploreServer** cria um *SocketServer* e uma *stream* de saída associada ao *socket*. O **ExploreClient** cria um *socket* e o utiliza para conectar-se ao **ExploreServer**. Realizada a conexão, o **ExploreClient** envia uma String "READY" para o **ExploreServer**, indicando estar pronto para receber os dados. O **ExploreServer** então serializa o vetor de **propriedadesBean**, sendo o *array* de bytes resultante da serialização escrito na *stream* de saída. Assim, os bytes são recebidos no **ExploreClient** através de uma *stream* de entrada (associada ao *socket* anteriormente criado) e após são deserializados, reconstruindo o vetor.

4.4 Classe ExploreServer

Esta classe implementa as funções do **ExploreServer**, utilizando como auxiliares as classes **JarResources** [MIT2004], **FileCopy**, **leitorXML** e **XMLHandler**. São funções do **ExploreServer**:

1. Localizar os diretórios "*apps*" e "*autoload*" do servidor JOnAS;
2. Receber conexão do **ExploreClient**
3. Extrair arquivos JAR dos arquivos EAR
4. Extrair os descritores (ejb-jar.xml) dos arquivos JAR
5. Realizar o *parse* dos descritores, criando objetos **propriedadesBean** que armazenam as propriedades configuradas no descritor
6. Serializar os objetos **propriedadesBean** e enviar ao **ExploreClient**

4.4.1 Localizar diretórios

Logo após iniciar a sua execução, o **ExploreServer** deve localizar os diretórios "*apps*" e "*autoload*", nos quais estão armazenadas as aplicações instaladas no servidor JOnAS. A forma utilizada é obter a propriedade **"jonas.root"** do sistema. Para o servidor JOnAS funcionar, é necessário que essa propriedade esteja configurada com o caminho para o diretório de instalação do servidor.

Dentro desse diretório encontra-se o diretório "*apps*", que por sua vez contém o diretório "*autoload*". O **ExploreServer** copia o valor da propriedade **"jonas.root"** para o atributo **"JonasRoot"**, que é utilizado para referenciar o caminho dos arquivos EAR.

Além disso, para criar os nomes de caminho e ao mesmo tempo garantir a portabilidade, é necessário obter outra propriedade do sistema, chamada **"file.separator"**. Essa propriedade é copiada para o atributo **"fileSeparator"**, que é então utilizado para separar os nomes de diretórios ao referenciar um caminho de diretório.

4.4.2 Receber conexão do ExploreClient

Neste ponto, o **ExploreServer** cria um *ServerSocket* e bloqueia esperando pelo **ExploreClient**. Quando este se conectar e enviar a string "READY", o **ExploreClient** deve iniciar o trabalho de obter os dados para envio. O primeiro passo é localizar os arquivos EAR e criar uma lista desses arquivos. Utilizando essa lista, o **ExploreServer** pode agora extrair os arquivos JAR.

Arquivos EAR são na verdade arquivos que usam a mesma compressão que arquivos no formato ZIP. Mas as bibliotecas da linguagem Java que trabalham com arquivos comprimidos (*java.util.zip* e *java.util.jar*) tem problemas para reconhecer e trabalhar com os arquivos EAR. A solução é criar uma cópia de cada um desses arquivos EAR, renomeando para ter a extensão ".zip". Como os formatos EAR e ZIP são compatíveis, esse processo resulta em

arquivos ZIP com o mesmo conteúdo dos arquivos EAR de origem. As cópias são armazenadas em um diretório chamado "ZIPS", filho do diretório da classe **ExploreServer**.

No processo de copiar e renomear os arquivos, é utilizada uma instância da classe **FileCopy**. Essa classe possui um método chamado **"copy"**, que recebe os nomes dos arquivos de origem e destino. O **ExploreServer** cria um arquivo de destino vazio, o qual recebe o resultado da cópia. Agora, o **ExploreServer** deve criar uma lista de referências para esses arquivos ZIP, para então extrair os arquivos JAR.

4.4.3 Extrair arquivos JAR

Utilizando a lista de arquivos ZIP, o **ExploreServer** deve agora extrair os arquivos JAR para o diretório **"componentes-jar"**, que está dentro do diretório da classe **ExploreServer**.

A extração dos arquivos JAR é realizada utilizando a classe **JarResources**. Essa classe cria uma tabela *hash* na qual armazena os bytes de cada um dos arquivos comprimidos em um ZIP ou JAR, indexando pelos nomes dos arquivos. A classe foi alterada pela adição de um atributo (**String *path***) e quatro métodos (***getArquivoJar***, ***getDescritor***, ***setPath*** e ***getJarNames***). O atributo ***"path"*** é utilizado para setar o diretório no qual o arquivo extraído será armazenado. O método **"setPath"** define o valor da variável ***path***. Os métodos **"getArquivoJar"** e **"getDescritor"** são usados para extrair um

arquivo JAR ou um descritor (“META-INF/ejb-jar.xml”), respectivamente. O método **getJarNames** é usado para obter uma lista dos nomes dos arquivos JAR que estão dentro do arquivo comprimido.

Para extrair os arquivos JAR, o **ExploreServer** deve criar uma instância da classe **JarResources** e definir o diretório de destino como sendo **“componentes-jar”**. Então, para cada arquivo ZIP da lista, deve obter uma lista dos arquivos JAR comprimidos dentro dele, usando o método **getJarNames**. Por fim, através do método **getArquivoJar**, extrai cada um dos arquivos JAR, cujos nomes estão na lista de arquivos JAR resultante da aplicação do método anterior.

Terminada essa etapa, o **ExploreServer** cria uma lista de referências para os arquivos JAR extraídos.

4.4.4 Extrair Descritores

O **ExploreServer** extrai todos os descritores para um mesmo diretório, chamado **“descritores”**. Mas todo o descritor tem por padrão o mesmo nome (**ejb-jar.xml**). Portanto, ao extrair os descritores, o **ExploreServer** deve renomeá-los para algum nome conveniente. O padrão escolhido foi renomear os descritores para os nomes dos arquivos JAR, mais a extensão “.xml”.

Novamente, é criada uma instância da classe **JarResources**. O método “**getDescritor**” é chamado passando como parâmetros o nome

original do descritor e o nome do arquivo de destino (“**nome_do_arquivo_JAR.xml**”).

Ao terminar de extrair os descritores, o **ExploreServer** deve criar uma lista de referências para os mesmos.

4.4.5 Realizar o *parse* dos descritores

Um *parser* XML é um componente que faz a leitura de um arquivo XML, realizando todas as tarefas de análise léxica do documento e reconhecimento de padrões estruturais da linguagem. A linguagem Java apresenta a **API JAXP**, para o processamento de arquivos XML. Essa API oferece a possibilidade de criar um *parser* SAX (*Simple API for XML*) ou DOM (*Document Object Model*). Enquanto o *parser* SAX é baseado em eventos (*tag* de início de elemento, *tag* de fim de elemento, etc), o *parser* DOM é baseado na estrutura de árvore. Para realizar o *parse* dos descritores dos *beans* optou-se por implementar um *parser* SAX. Essa escolha foi baseada no fato de que o SAX é a melhor opção quando o objetivo é apenas ler um documento XML e procurar por algumas *tags* [BOND2002].

Foram implementadas duas classes, **XMLHandler** e **leitorXML**. Ambas as classes utilizam as bibliotecas *javax.xml.parsers* e *org.xml.sax*. A classe **XMLHandler** estende a classe **DefaultHandler**, que por sua vez é uma implementação da interface **ContentHandler** da API SAX. Essa classe é a responsável por fazer o *parse* de todo o documento XML, criando as instâncias das

classes **propriedadesBean** e **metodoBean** que recebem as propriedades dos *beans*. Para fazer o *parse*, a classe **XMLHandler** sobrescreve três métodos do **DefaultHandler**: ***startElement***, ***characters*** e ***endElement***. Esses métodos são responsáveis por realizar o processamento necessário quando o *parser* encontra uma *tag* de início (***startElement***), caracteres contidos entre *tags* (***characters***) ou uma *tag* de fim de elemento (***endElement***).

Conforme encontra as *tags* que definem as propriedades que devem ser monitoradas, o *parser* copia os valores para os atributos de uma instância da classe **propriedadesBean**. Ao término do *parse* do documento XML, é retornado um vetor de **propriedadesBean** contendo um objeto para cada *bean* configurado no descritor.

A classe **leitorXML** é responsável por criar um *parser* SAX e uma instância da classe **XMLHandler**. Ela chama então o método "***parse***" do *parser* SAX, passando o objeto **XMLHandler** e o descritor dos *beans*.

Para cada descritor da lista, o **ExploreServer** cria uma instância do **leitorXML** e solicita a realização do *parse* do descritor. No final desse processo, o **ExploreServer** possui um vetor de objetos **propriedadesBean**, com um objeto para cada *bean* instalado no servidor JOnAS.

4.4.6 Serializar e enviar

Para enviar o vetor de **propriedadesBean**, o **ExploreServer** precisa transformá-lo em um fluxo de bytes, para enviar através do *socket* TCP. Para isso, o **ExploreServer** chama o método "**serialize**" passando o vetor como parâmetro. O fluxo de bytes resultante do processo de serialização é escrito na *stream* de saída, sendo enviado ao **ExploreClient**.

4.5 Classe ExploreClient

Esta classe é responsável por desenhar a interface gráfica, receber os dados do **ExploreServer**, processar as solicitações do usuário e apresentar os resultados.

Sua primeira função é desenhar a interface gráfica, através da qual realizar-se-á toda a interação com o usuário.

Após desenhar a interface gráfica, o **ExploreClient** deve localizar o arquivo de configurações, chamado "**ExploreConfig.txt**", no mesmo diretório onde está sendo executado. Esse arquivo deve ter apenas uma linha, contendo o endereço IP do servidor JOnAS monitorado ou "*localhost*", caso o **ExploreClient** seja executado na mesma máquina do servidor JOnAS. Se não encontrar esse arquivo, ou o endereço nele contido seja inválido (por inválido entenda-se um endereço de servidor no qual não esteja sendo executado o

ExploreServer), o **ExploreClient** deve abortar a conexão e informar o ocorrido ao usuário.

Utilizando o endereço IP do arquivo de configurações, o **ExploreClient** deve tentar conectar-se ao **ExploreServer**. Se não tiver sucesso, informa ao usuário e espera que uma nova tentativa seja solicitada pelo mesmo. Caso a conexão seja realizada com sucesso, o **ExploreClient** deve informar ao **ExploreServer** que está pronto para receber os dados. Para isso, utiliza o método **sendMessage** para enviar a string "READY". Após o envio da mensagem, cria uma stream de entrada na qual recebe o vetor de propriedades dos *beans*, através do método **recebePropsBeans**. Esse método cria a stream de entrada utilizando o *socket* de conexão, recebe um fluxo de bytes e reconstrói o vetor de propriedades dos *beans*.

A partir desse momento, o **ExploreClient** está pronto para exibir informações ao usuário, o que faz de acordo com as solicitações deste último. A única informação exibida inicialmente é a lista de aplicações, a partir da qual o usuário começa a interagir com o **ExploreClient** para solicitar as propriedades dos *beans* que deseja consultar. Nesse ínterim, o usuário pode, a qualquer momento, solicitar atualização dos dados ao **ExploreClient** através do menu da interface gráfica. Diante dessa solicitação, o **ExploreClient** conecta novamente ao **ExploreServer** e solicita que este reinicie o processo de localização dos descritores e posterior parse dos mesmos.

4.5.1 Interface Gráfica

A implementação da interface gráfica do **ExploreClient** foi realizada utilizando o padrão *JFC/Swing*. A própria classe **ExploreClient** estende a classe *javax.swing.Frame*, e por isso realiza a criação da interface dentro de seu descritor. O retorno do construtor é um *frame* principal no qual são desenhados os outros componentes da interface (painéis, listas, botões, menus e caixas de diálogo).

A interface gráfica do **ExploreClient** apresenta cinco painéis do tipo *Jpanel*, implementados no *frame* principal para servirem como bases de posicionamento dos outros componentes. A interação com o usuário se dá por meio de listas selecionáveis, botões e um menu. No menu são ativadas duas caixas de diálogo, as quais também são interativas com o usuário (através de listas e botões). A figura 4.4 mostra a visão inicial da interface gráfica do **ExploreClient**, logo após o envio inicial dos dados pelo **ExploreServer**.

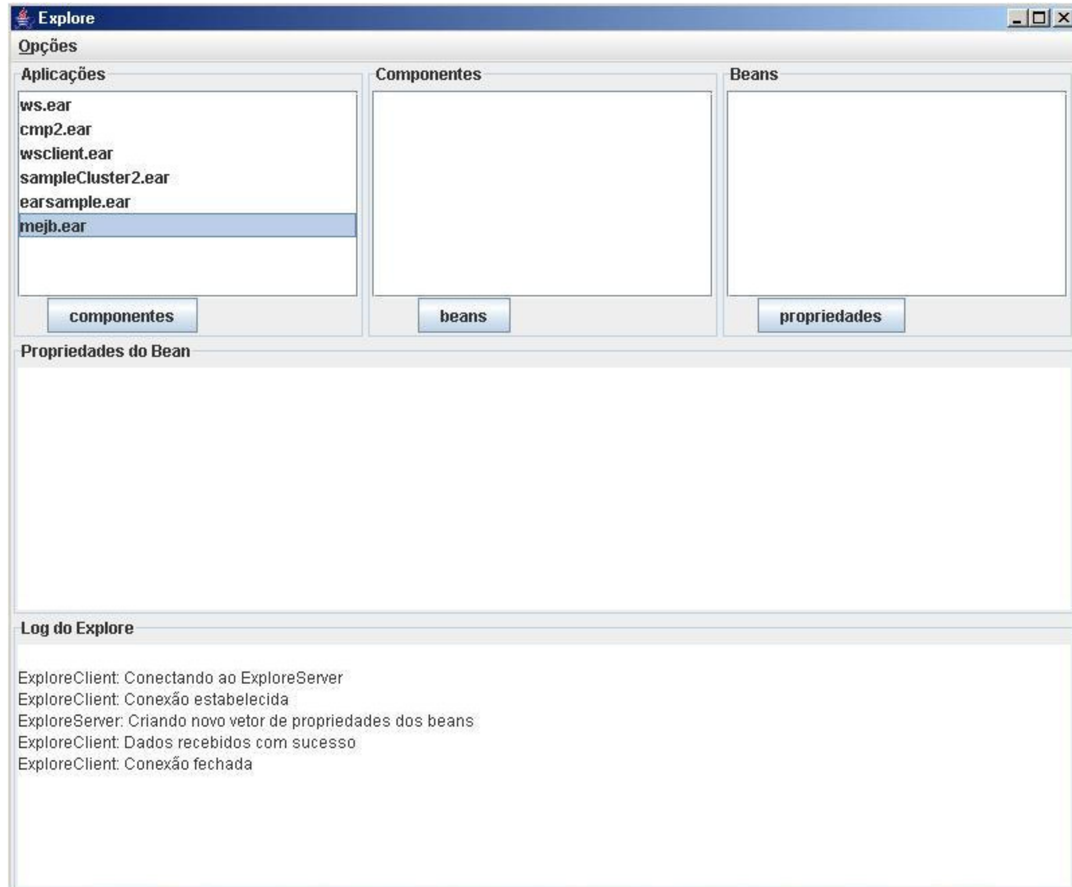

Figura 4.4 – Interface Gráfica **ExploreClient**

Na figura 4.4 observa-se a estrutura de painéis do **ExploreClient**. A parte superior do *frame* é dividida em três painéis, os quais contém uma lista de nomes cada um. Estes podem designar uma aplicação, um componente ou um *bean*, conforme os títulos dos painéis na interface. Os outros dois painéis vistos na figura contém uma área de texto (*JTextArea*) não editável cada um. O texto é não editável para evitar que o usuário acidentalmente altere informações. Todos os cinco painéis base são compostos por um painel de *scroll* (*JScrollPane*), permitindo que tanto as listas como as áreas de texto sejam de tamanho dinâmico, de acordo com a quantidade de

informações (nomes de aplicações, nomes de *beans*, propriedades de *beans*, mensagens de log, etc) recebidas do **ExploreServer**.

Nota-se também que no início da execução do **ExploreClient** a única lista que possui elementos definidos é a de aplicações, pois não precisa de interação com o usuário. A única forma de atualizar a lista de aplicações é através do menu Opções, conforme ilustra a figura 4.5.

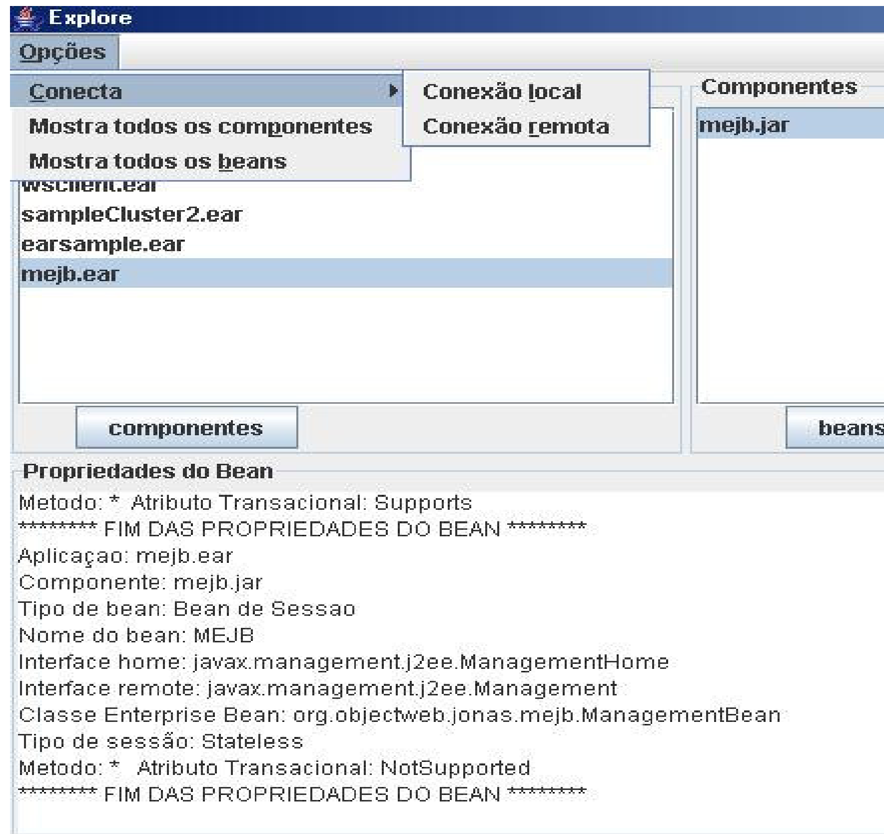

Figura 4.5 – Menu Opções do **ExploreClient**

Selecionando um dos sub-menus "**Conexão Local**" ou "**Conexão Remota**", o **ExploreClient** conecta-se novamente ao **ExploreServer** e solicita um novo vetor de propriedades dos *beans*. Assim, se uma nova aplicação foi instalada no servidor JOnAS durante a execução do **ExploreClient**, ela passará a ter suas

informações (nomes de componentes, nomes de *beans* e propriedades destes) exibidas na interface após a nova conexão.

4.5.1.1 Painel Aplicações

Esse é o primeiro painel da interface e consiste em um *JPanel* de base, no qual são inseridos um painel de *scroll* (*JScrollPane*) e um botão (cujo nome é **componentes**, porque ativa o painel de mesmo nome). Dentro do painel de *scroll*, é inserida uma lista de elementos selecionáveis (*JList*). Este esquema permite que o tamanho da lista não seja limitado ao número de linhas que o painel possa exibir.

Essa lista exibe os nomes das aplicações armazenadas no servidor JOnAS , obtidos através do método **getApplications** do **ExploreClient**, que retorna uma lista de nomes de aplicações e a insere na lista da interface gráfica.

A lista de aplicações permite a seleção de apenas um nome por vez. Ao selecionar um elemento da lista e clicar no botão "**componentes**", é ativado o segundo painel, que mostra os componentes que fazem parte da aplicação selecionada na lista. A figura 4.6 mostra o painel **Aplicações** exibindo os nomes das aplicações que já vem instaladas na versão binária do servidor JOnAS (ws, cmp2, wsclient, sampleCluster2, earsample e mejb).

Figura 4.6 – Painel Aplicações

4.5.1.2 Painel Componentes

É implementado utilizando a mesma estrutura do painel **Aplicações**, onde existe um painel de base, no qual são inseridos um botão (de nome "***beans***") e uma lista (dentro de um painel de *scroll*). A lista desse painel, por sua vez, exibe os nomes dos componentes da aplicação selecionada no painel **Aplicações**. A lista dos nomes dos componentes da aplicação selecionada é obtida através do método **getComponents**, o qual é ativado pelo botão "**componentes**".

Ao selecionar um dos nomes da lista de componentes e clicar no botão, o terceiro painel é ativado, exibindo uma lista dos *beans* que formam o componente selecionado. A figura 4.7 ilustra a interação entre os painéis **Aplicações** e **Componentes**.

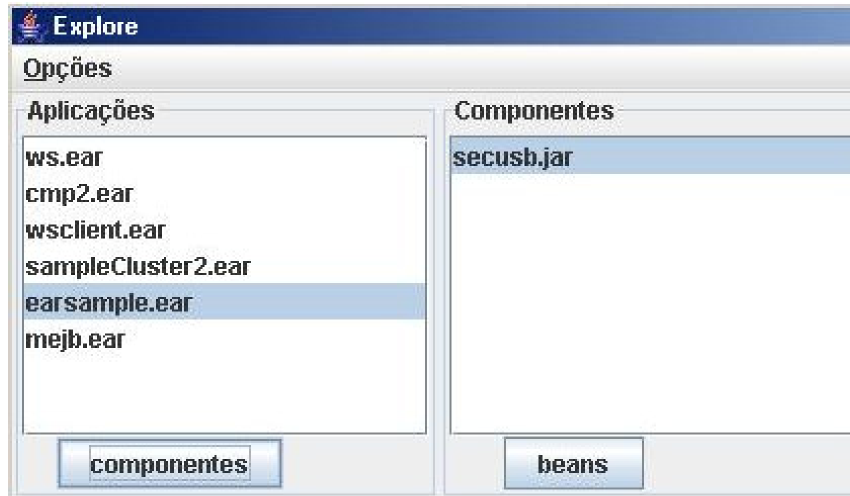

Figura 4.7 – Painel Componentes

Na figura 4.7, a aplicação selecionada (earsample) possui apenas um componente J2EE, de nome “secusb.jar”. Caso uma outra aplicação seja selecionada, a lista do painel **Componentes** permanece inalterada até que o botão “**componentes**” seja reativado. Como o componente “secusb.jar” já está selecionado, se o botão “***beans***” for clicado o painel **Beans** exibirá uma lista dos *beans* implementados em “secusb.jar”.

4.5.1.3 Painel Beans

O painel **Beans** apresenta ao usuário uma lista de nomes de *beans*, os quais fazem parte do componente selecionado. Essa lista de nomes é resultado da aplicação do método **getBeansNames**, ativado pelo botão do painel **Componentes**.

O painel apresenta também um botão “**propriedades**”, o qual aciona o painel “**Propriedades do Bean**”, mostrando as propriedades do *bean* selecionado na lista.

A figura 4.8 mostra o resultado do botão “**beans**” quando o componente selecionado é o “cmp2.jar”. Os nomes de todos os *beans* implementados nesse componente são listados no painel **Beans** (CustomerEJB, AddressEJB, PhoneEJB, CreditCardEJB, CruiseEJB, ShipEJB, ReservationEJB, CabinEJB, TravelAgentEJB e Sequence).

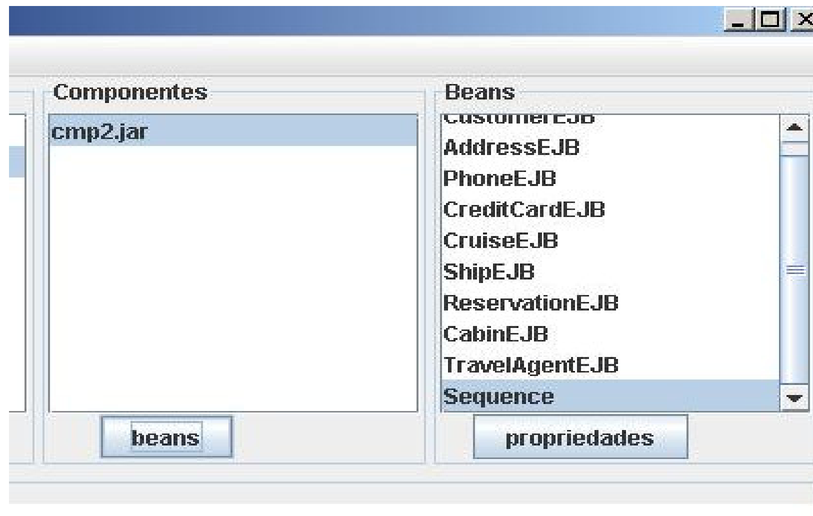

Figura 4.8 – Painel *Beans*

4.5.1.4 Painel Propriedades do *Bean*

É responsabilidade deste painel mostrar as propriedades do *bean* selecionado na lista do painel **Beans**. Considerando o fato de que essas propriedades não tem um tamanho fixo (um *bean* pode definir vários métodos com atributos transacionais diferentes), o painel foi implementado como um *JscrollPane*, que permite deslizá-lo para exibir todas as informações. Dentro desse painel de *scroll*, é inserida uma área de texto não editável. Ao clicar no botão “**propriedades**”, as propriedades do *bean* são impressas nessa área de texto. Ao selecionar outro nome de *bean* e reativar o botão, as novas propriedades serão

inseridas logo abaixo das já existentes. Como separador entre os *beans*, a área de texto utiliza a string "**FIM DAS PROPRIEDADES DO *BEAN***". A figura 4.9 mostra a seleção do *bean* "ShoppingCart" e a impressão de suas propriedades no painel.

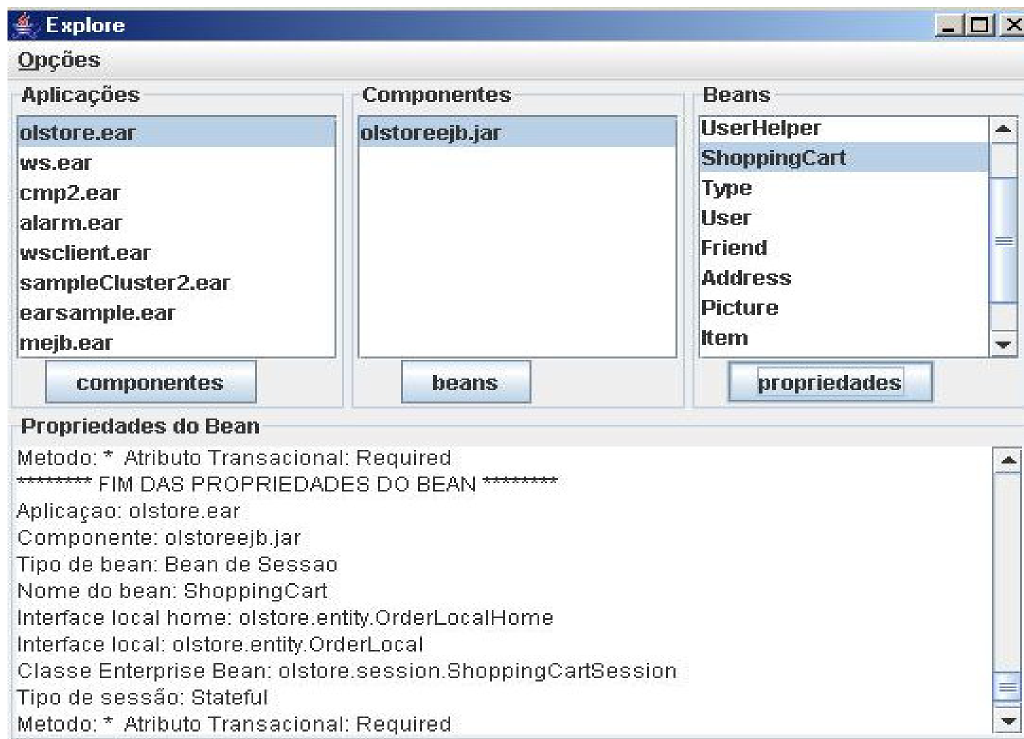

Figura 4.9 – Painel Propriedades do *Bean*

4.5.1.5 Painel Log do Explore

Neste painel são exibidas mensagens de controle da ferramenta Explore. Tais mensagens podem ser geradas pelo **ExploreClient** ou pelo **ExploreServer**, e indicam operações concluídas com sucesso e eventuais erros. Sendo que a duração de uma sessão do **ExploreClient** pode ser longa, o número de mensagens pode crescer rapidamente. Além disso, é interessante que as mensagens antigas não sejam

perdidas. Por isso, esse painel também foi implementado como um *JscrollPane*, de forma a exibir um número ilimitado de mensagens sem que as novas mensagens apaguem as antigas. A tabela 4.2 mostra as possíveis mensagens exibidas nesse painel, junto com seu significado.

A figura 4.10 mostra a saída impressa no painel "**Log do Explore**" durante uma operação normal de troca de dados entre o **ExploreClient** e o **ExploreServer**.

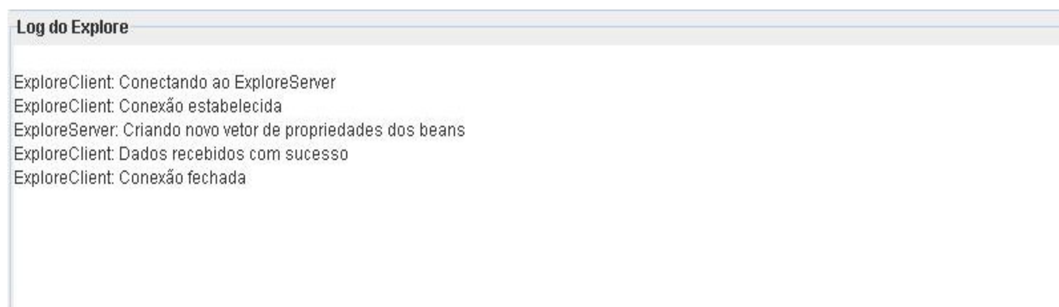

Figura 4.10 – Painel Log do Explore

TABELA 4.2 – Mensagens Exibidas no Painel Log do Explore

Mensagem	Descrição
ExploreServer: Enviou as propriedades dos *beans*	**ExploreServer** serializou e conseguiu enviar o vetor de propriedades dos *beans*
ExploreServer: Criando novo vetor de propriedades dos *beans*	**ExploreServer** vai refazer o parse de todos os descritores para obter um novo vetor de propriedades dos *beans*
ExploreClient: Conectando ao **ExploreServer**	**ExploreClient** iniciou processo de conexão com o **ExploreServer**
ExploreClient: Conexão estabelecida	**ExploreClient** conseguiu com sucesso estabelecer uma conexão com o **ExploreServer**
ExploreClient: Solicitando dados	**ExploreClient** enviou pedido para receber o vetor de propriedades dos *beans*
ExploreClient: Dados recebidos com sucesso	**ExploreClient** recebeu e deserializou o vetor de propriedades dos *beans* com sucesso
ExploreClient: Erro ao receber dados	**ExploreClient** não conseguiu deserializar e reconstruir o vetor de propriedades dos *beans*
ExploreClient: Tentativa de conexão falhou	**ExploreClient** não conseguiu estabelecer uma conexão com o **ExploreServer**
ExploreClient: Não foi possível localizar o arquivo de configurações	**ExploreClient** não encontrou o arquivo de configurações necessário para obter o endereço IP da máquina onde está instalado o JOnAS
ExploreClient: endereço definido no arquivo de configurações é inválido	O endereço IP do arquivo de configurações não é válido, isto é, não corresponde ao endereço de um servidor JOnAS monitorado pelo **ExploreServer**
ExploreClient: Conexão fechada	O **ExploreClient** já recebeu os dados e fechou o *socket* de comunicação com o **ExploreServer**

4.5.1.6 Caixa de diálogo “Todos os componentes do servidor JOnAS”

Esta caixa de diálogo (*Jdialog*) é ativada pelo menu da interface gráfica do **ExploreClient**. Nela é exibida uma lista (*Jlist*) de todos os componentes armazenados no servidor JOnAS, um botão chamado “**mostra informações**” e uma área de texto. A lista de nomes resulta da aplicação do método **mostraAllComponentes**, da classe **ExploreClient**.

Selecionando um nome de componente da lista e ativando o botão da caixa de diálogo, são inseridas na área de texto informações sobre o nome do componente, os nomes dos *beans* que implementa e das aplicações nas quais está inserido. Para separar as informações de um componente de outro, é usada a string “**FIM DAS INFORMAÇÕES DO COMPONENTE nomeComponente**”. A figura 4.11 mostra a caixa de diálogo ativa e algumas informações de componentes que foram selecionados.

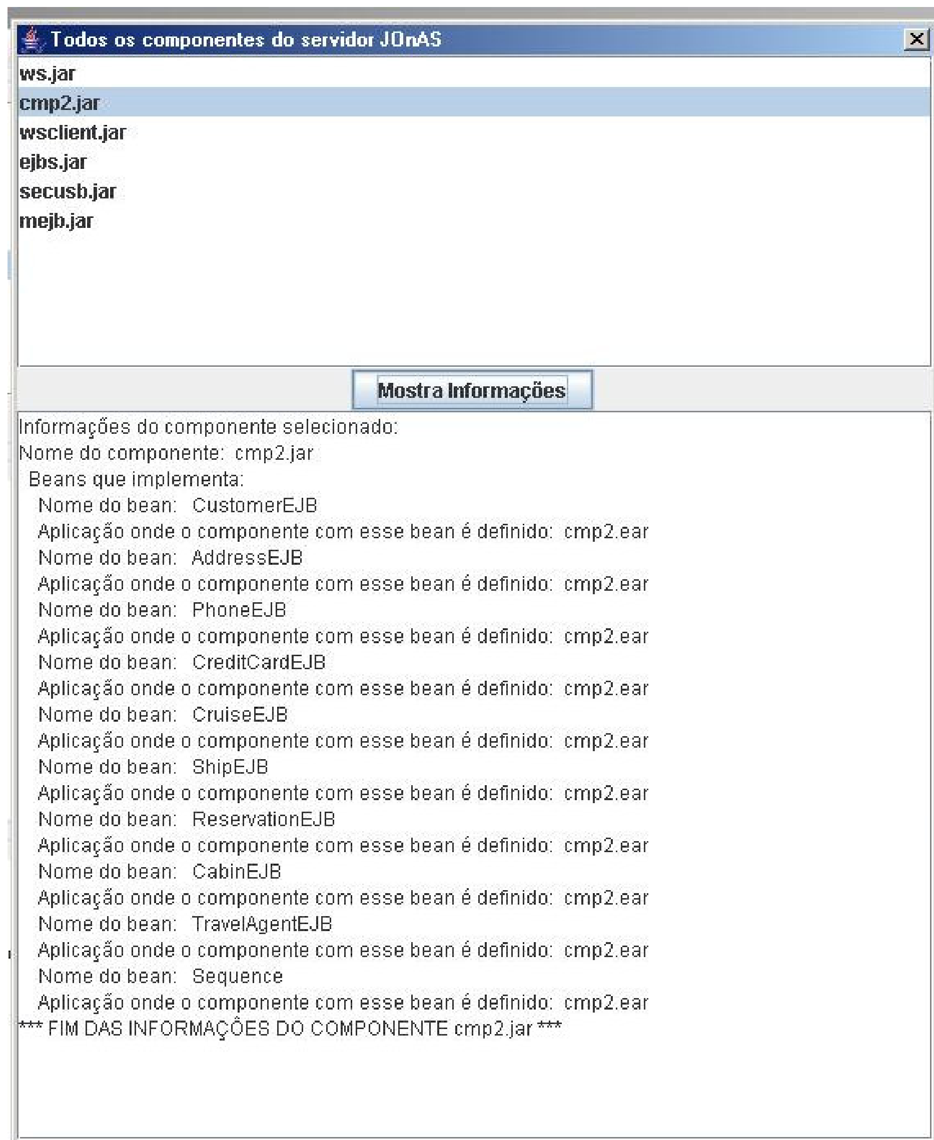

Figura 4.11 – Caixa de Diálogo “Todos os componentes do servidor JOnAS”

4.5.1.7 Caixa de diálogo “Todos os *beans* do servidor JOnAS”

Também é uma *Jdialog* que exibe uma lista (*Jlist*), um botão e uma área de texto. Mas na lista desta caixa de diálogo são exibidos os nomes de todos os *beans* armazenados no servidor JOnAS. A seleção

de um nome de *bean* da lista associada à ativação do botão, insere informações sobre o *bean* na área de texto.

As informações do *bean* que são mostradas são seu nome, o nome do componente que o implementa e a aplicação na qual o componente é utilizado. A figura 4.12 mostra a execução da caixa de diálogo, com algumas informações de *beans* selecionados.

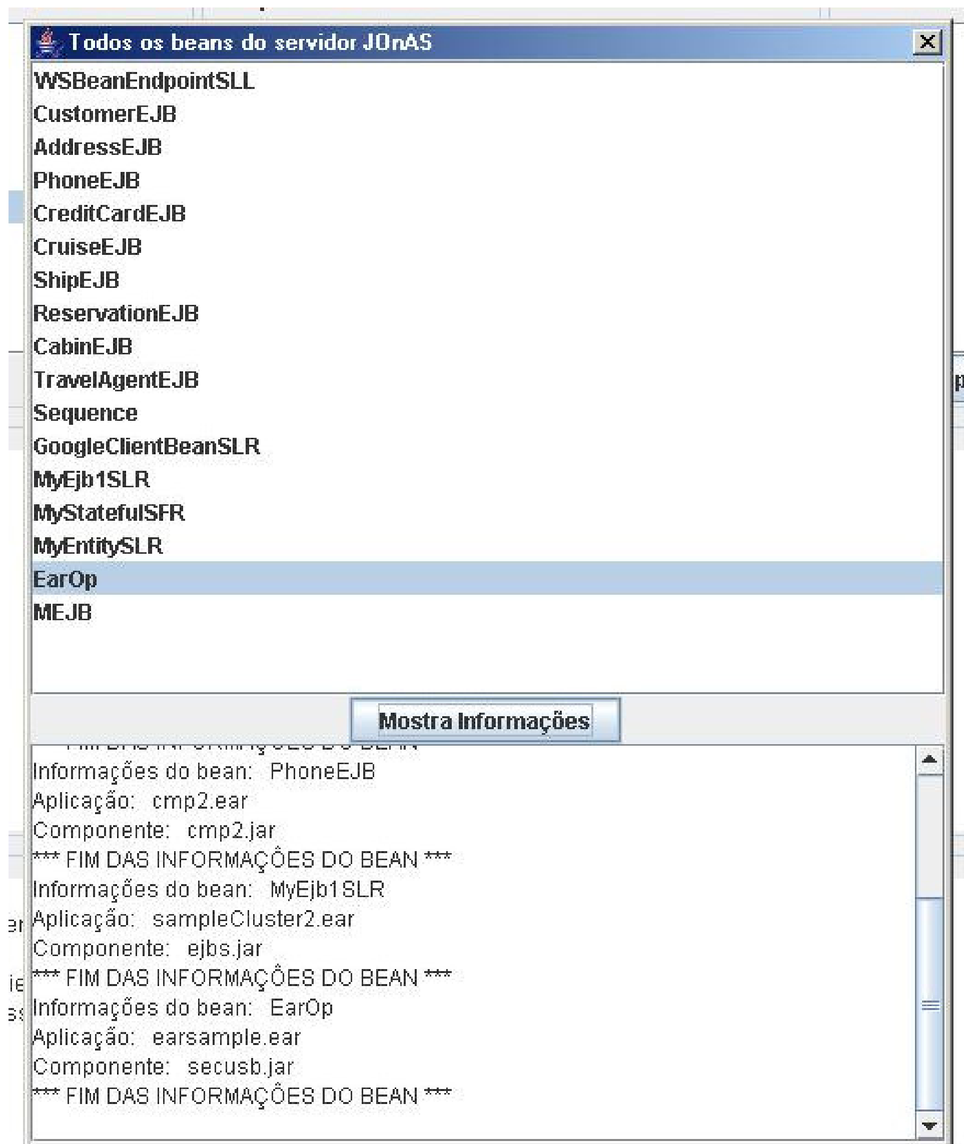

Figura 4.12 – Caixa de Diálogo “Todos os *beans* do servidor JOnAS”

4.5.1.8 Barra de Menus

A barra de menus é formada por um menu principal (**Opções**), o qual contém três itens de menu, (i) **Conecta**, (ii) **Mostra todos os componentes** e (iii) **Mostra todos os beans**.

O item de menu "**Conecta**", possui dois sub-itens, "**Conexão Local**" e "**Conexão Remota**". Selecionando "**Conexão Local**", o **ExploreClient** refaz a conexão com o **ExploreServer** utilizando como endereço IP o valor "*localhost*". Já a seleção do item "**Conexão Remota**" faz o **ExploreClient** reiniciar o processo de ler o arquivo de configurações para obter o endereço IP do servidor JOnAS monitorado, para refazer a conexão. Nos dois casos, o **ExploreClient** solicita novamente o vetor de propriedades dos *beans* ao **ExploreServer**, obtendo dados atualizados.

O item de menu "**Mostra todos os componentes**" ativa a caixa de diálogo "**Todos os componentes do servidor JOnAS**", que exibe uma lista de todos os nomes de componentes instalados no servidor JOnAS.

O último item de menu, chamado "**Mostra todos os beans**", ativa a caixa de diálogo "**Todos os beans do servidor JOnAS**", a qual lista todos os nomes de *beans* instalados no servidor JOnAS.

A figura 4.13 mostra toda a estrutura de menu aberta, de forma que é possível verificar todos os itens do menu e sua forma de organização.

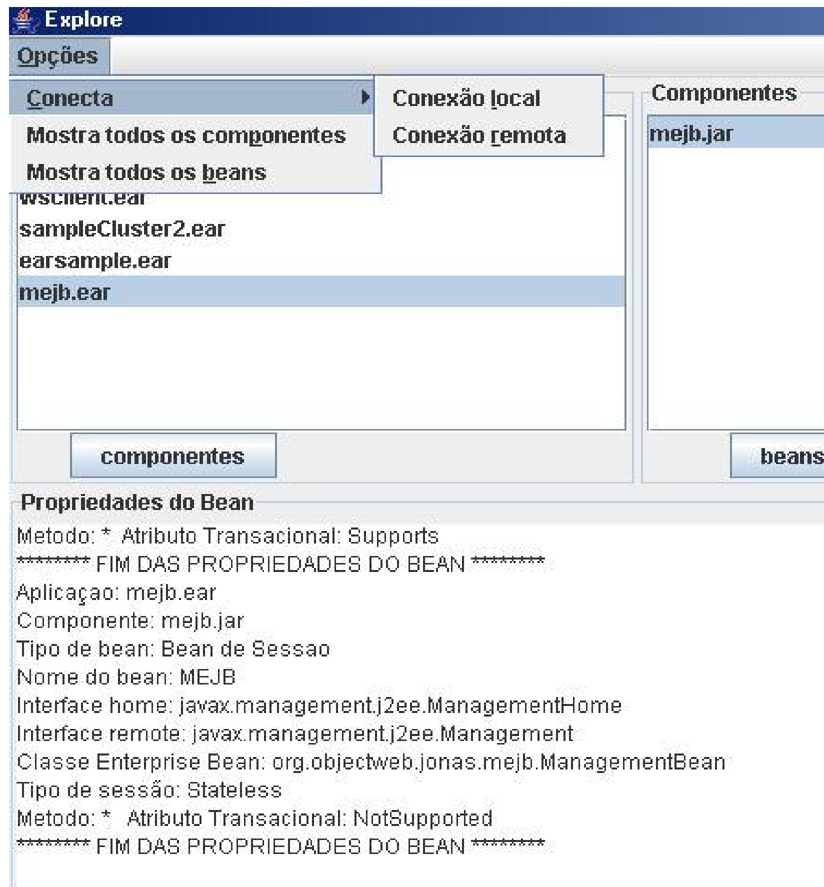

Figura 4.13 – Menu Opções do **ExploreClient**

4.5.2 Conexão com ExploreServer

A conexão com o **ExploreServer** é realizada através de *sockets* TCP. O **ExploreClient** deve criar um *socket*, utilizando o endereço IP do arquivo de configurações e a porta 8000 (porta padrão escolhida para comunicação na ferramenta Explore). Criado esse *socket*, o **ExploreClient** deve enviar a mensagem “READY” para o **ExploreServer**, de forma que este possa iniciar o envio do vetor de propriedades dos *beans*. Após o envio, a conexão é fechada e só é reiniciada quando o usuário solicita ao **ExploreClient** a atualização

dos dados. Essa solicitação de atualização se dá através do item **Conecta** do menu **Opções**, onde o usuário decide se o **ExploreClient** deve ler o arquivo de configurações e tentar uma conexão remota ou deve tentar uma conexão local.

4.5.3 Processamento de solicitações do usuário

A Tabela 4.3 resume os tipos de solicitações do usuário na interface gráfica e como o **ExploreClient** processa essas solicitações.

TABELA 4.3 – Interação Usuário-ExploreClient

Interação do usuário	**Resposta do ExploreClient**
Clica botão componentes	Exibe os componentes da aplicação selecionada
Clica botão *beans*	Exibe os *beans* que estão implementados no componente selecionado
Clica botão propriedades	Exibe as propriedades do *bean* selecionado, provenientes do descritor do *bean*
Clica menu Opções	Abre a lista de itens de menu
Clica Conexão Local	**ExploreClient** tenta criar um *socket* TCP usando como endereço o valor *localhost*. Se conseguir conectar, pede que o **ExploreServer** atualize e envie os dados
Clica Conexão Remota	Tenta ler o arquivo de configurações para obter o IP do *host* onde o JOnAS está executando; utiliza o *socket* para refazer a conexão e solicitar novo envio dos dados
Clica Mostra todos os componentes	Abre caixa de diálogo "Todos os componentes do servidor JOnAS" e exibe uma lista de todos os componentes
Clica Mostra todos os *beans*	Abre caixa de diálogo "Todos os *beans* do servidor JOnAS" e exibe uma lista de todos os *beans*
Clica Mostra Informações	Se a caixa de diálogo for de componentes, exibe informações do componente selecionado. Se não, exibe informações do *bean* selecionado

5 Trabalhos Futuros

Neste trabalho foram implementadas funcionalidades básicas para o desenvolvimento de uma ferramenta de monitoramento que permite realizar testes de aplicações em um servidor J2EE. O serviço de monitoramento da ferramenta Explore ainda é bastante estático, limitando-se a obter propriedades dos descritores de *Enterprise Beans*. Nesses descritores, são configuradas as funcionalidades que o *container* J2EE deve prover aos *beans* em tempo de execução. Mas a leitura dessas propriedades pode ser expandida, de forma a criar um serviço de configuração de descritores através de formulários que seriam criados na interface gráfica. Trocando o *parser* SAX por um *parser* DOM, seria possível alterar as propriedades configuradas em um descritor.

Porém, essa configuração de propriedades seria uma brecha de segurança, pois nos descritores são definidas as políticas de segurança fornecidas aos *beans* pelo *container* J2EE. Dessa forma, seria necessário inserir na ferramenta Explore um sistema de autenticação, permitindo que apenas administradores do sistema tivessem acesso a essas funcionalidades.

O passo seguinte para o desenvolvimento de uma ferramenta de monitoramento mais poderosa seria acrescentar funções de monitoramento dinâmico, que recolhessem informações das aplicações em tempo de execução. O serviço de transações do J2EE certamente é uma das suas funcionalidades mais poderosas. Utilizando alguns dos métodos definidos nas classes do servidor JOnAS, aliado à

inserção de *flags* e outras variáveis de controle, seria possível implementar um serviço de monitoramento de transações em tempo real. Esse serviço seria capaz de recolher informações muito úteis para um desenvolvedor que estivesse testando uma nova aplicação no servidor, pois permitiria ter um total controle sobre as transações de um *container* J2EE, desde solicitar um *rollback* até reiniciar uma transação.

Inicialmente, a ferramenta Explore precisa de melhoramentos em seu sistema de comunicação. Utilizando *threads*, seria possível que o **ExploreServer** atendesse mais de um cliente. Além disso, as trocas de mensagens entre o **ExploreServer** e o **ExploreClient** deve ser ampliada, criando um canal contínuo de troca de informações entre estes. Isso permitiria que ambos soubessem sobre o estado atual do outro. Por exemplo, alterando o código do JOnAS, seria possível avisar o **ExploreServer** sempre que uma nova aplicação fosse instalada no servidor, ou então alterada pela remoção, substituição ou inserção de componentes. O **ExploreServer** então avisaria ao **ExploreClient** que possui uma atualização dos dados, mas enviaria apenas o que foi atualizado. Os dados que o **ExploreClient** já possui, se não sofreram alteração, não precisam ser enviados novamente.

Além disso, o sistema de conexão atual, que utiliza um arquivo de configurações para localizar um servidor JOnAS, seria substituído por um monitoramento dinâmico de servidores. Um painel da interface gráfica listaria os servidores de uma rede, informando quais possuem o JOnAS instalado. O serviço de monitoramento então não se limitaria a um servidor. O **ExploreClient** poderia conectar-se a

vários servidores e monitorá-los de acordo com as solicitações do usuário (seleciona um dos servidores para listar os *beans* em execução ou recolher informações das transações do *container*).

Portanto, são objetivos futuros deste trabalho:

a. Melhorias no sistema de conexão;
b. Sistema de autenticação;
c. Edição de descritores;
d. Monitoramento de *beans* em execução (propriedades, número de instâncias, métodos, etc);
e. Controle Transacional (monitoramento e interação com transações em execução dentro do *container* J2EE).

6 Conclusão

Neste trabalho foi apresentado um estudo da arquitetura J2EE e do servidor JOnAS, e baseado neste foi projetada e implementada uma ferramenta de monitoramento. A pesquisa dos padrões J2EE concentrou-se na tecnologia *Enterprise Java Beans*, que fornece uma arquitetura de componentes e serviços não funcionais para os mesmos. Destes serviços, o de transações recebeu atenção especial, pois este apresenta-se como uma das maiores virtudes da tecnologia EJB.

O resultado desses estudos foi a identificação das propriedades de uma aplicação J2EE e dos serviços que são fornecidos por um *container* que implemente essa arquitetura. Baseado nestes, foi projetada uma ferramenta de monitoramento de aplicações J2EE para o servidor JOnAS.

Enquanto o foco da maioria das ferramentas de monitoramento são análises de desempenho e coleta de informações de sistemas já em fase de funcionamento pleno, o foco da ferramenta Explore é permitir que o desenvolvedor tenha um maior controle ainda na fase de testes da sua aplicação. Dessa forma, busca reunir informações sobre os *enterprise beans* de uma aplicação J2EE, as quais podem ser úteis para o desenvolvedor identificar se o comportamento de seus componentes é o esperado.

A ferramenta Explore ainda não é capaz de realizar o monitoramento de atributos dinâmicos de aplicações J2EE, isto é, informações sobre transações e instâncias dos *beans*. Para incluir essas funções na ferramenta Explore, seria necessário uma maior interação

da ferramenta com o servidor JOnAS, isto é, seria necessário que a Explore fosse informada pelo servidor sobre a execução dos *beans*. A implementação dessas funções esbarrou na falta de documentação disponível sobre o servidor JOnAS, pois tal implementação implicaria em alterar o código-fonte do JOnAS.

Este trabalho é apenas um passo inicial na implementação de uma ferramenta de monitoramento completa, capaz de reunir informações estáticas e dinâmicas de uma aplicação, bem como a alteração de propriedades em tempo de execução.

7 Bibliografia

[ARMSTRONG2004] ARMSTRONG, E.; BALL, J.; BODOFF, S.; CARSON, D.B.; EVANS, I.; GREEN, D.; HAASE, K.; JENDROCK, E. *The J2EE™ 1.4 Tutorial – For Sun Java System Application Server Platform Edition 8.1 2005Q1*. 2004.

[BOND2002] BOND, M.; HAYWOOD, D.; LAW, D.; LONGSHAW, A.; ROXBURGH, P. *Sams Teach Yourself J2EE in 21 days.* Sams Publishing. 2002.

[CECCHET2003] CECCHET, E.; MARGUERITE, J.; EXERTIER, F.; PICONE, J.; SHUE, T. The *JOnAS v.3.2 Tutorial. Object*Web Consortium, 2002-2003.

[ECKEL2003] ECKEL, B. et al. *Thinking in Enterprise Java.* Revision 1.1, 5-06-2003. Disponível em: <http://www.mindview.com>.

[GRA1993] GRAY, J.; REUTER, A. *Transaction processing*: concepts and techniques. San Mateo (CA), USA: Morgan Kaufmann Publishers, Inc., 1993. 1070p.

[LAI1999] LAI, C.; GONG, L.; KOVED, L.; NADALIN, A.; SCHEMERS, R. *User Authentication and Authorization in the Java™ Platform.* Publicado no Proceedings of the 15th Annual Computer Security Applications Conference, Phoenix, AZ. Dezembro, 1999.

[MARSH1998] MARSH, D. A. *Global Transactions – X/Open XA – Resource Managers.* Aurora Information Systems, INC. 02/06/1998.

[MAHAPATRA2000] MAHAPATRA, S. *Transaction Management under J2EE 1.2*. 14/07/2000. Disponível em: < http://www.javaworld.com/javaworld/jw-07-2000/jw-0714-*transaction*-p1.html>.

[MAF1996] MAFFEIS, S. *PIRANHA – A Hunter of Crashed CORBA Objects.* Olsen & Associates, Zurich, 1996.

[OBJ2004] *OBJECT*WEB CONSORTIUM. *Java Open Application Server (JOnAS): a J2EETM Platform.* 2004-06-04.

[OMG1999] *OBJECT* MANAGEMENT GROUP, INC. *Corba/IIOP 2.3.1 Specification*. 07/10/1999.

[OMG1995] *OBJECT* MANAGEMENT GROUP, INC. *The Common Object Request Broker: Architecture and Specification, revision 2.0.* Julho, 1995. Disponível em: <ftp://ftp.omg.org/pub/docs/ptc/96-03-04.pdf>

[OMG2000a] *OBJECT* MANAGEMENT GROUP, INC. *IDL To Java™ Language Mapping Specification*. 08/01/2000. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/2000-01-08.

[OMG2000b] *OBJECT* MANAGEMENT GROUP, INC. *Java™ Language To IDL Mapping Specification*. 06/01/2000. Disponível em: < http://www.omg.org/cgi-bin/doc?ptc/2000-01-06.

[ROM2001] ROMAN, E. et al. *Mastering EJB and the J2EE platform.* Wiley; 2nd edition (December 14, 2001).

[SUN2001a] SUN MICROSYSTEMS, INC. *EJB Specification 2.0.* 2001. Disponível em: <http://java.sun.com/products/ejb/docs.html>.

[SUN2004a] SUN MICROSYSTEMS, INC. *EJB Specification version 3.0.* 24/06/2004. Disponível em: <http://java.sun.com/products/ejb/docs.html>

[SUN2003a] SUN MICROSYSTEMS, INC. *Java™ 2 Platform Enterprise Edition Specification, v.1.4.* 24/11/2003. Disponível em: <http://java.sun.com/2ee/1.4/ download.html#platformspec>

[SUN1999a] SUN MICROSYSTEMS, INC. *Java™ Remote Method Invocation Specification.* Dezembro, 1999. Disponível em: < ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf>

[SUN1999b] SUN MICROSYSTEMS, INC. *RMI-IIOP Programmer´s Guide.* 1999. Disponível em: < http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/rmi_iiop_pg.html>

[SUN1999c] SUN MICROSYSTEMS, INC. *Java™ Naming and Directory Interface 1.2 Specification.* 14/07/1999. Disponível em: http://java.sun.com/j2se/1.3/docs/guide/jndi/spec/jndi/

[SUN2001b] SUN MICROSYSTEMS, INC. *JDBC 3.0 Specification – Final release.* Outubro, 2001. Disponível em: < http://java.sun.com/products/jdbc/download.html>

[SUN1999d] SUN MICROSYSTEMS, INC. *Java™ Transaction API, Version 1.0.1.* 29/04/1999. Disponível em: <http://java.sun.com/products/jta/>

[SUN1999e] SUN MICROSYSTEMS, INC. *Java™ Transaction Service, Version 1.0.* 01/12/1999. Disponível em: < http://java.sun.com/products/jts>

[SUN2002] SUN MICROSYSTEMS, INC. *Java™ Message Service Specification, Version 1.1.* 12/04/2002. Disponível em: < http://java.sun.com/products/jms>

[SUN2003b] SUN MICROSYSTEMS, INC. *Java™ Servlet Specification, Version 2.4.* 24/11/2003. Disponível em: <http://java.sun.com/products/servlet>

[SUN2000] SUN MICROSYSTEMS, INC. *JavaMail™ API Specification Version 1.2.* Setembro, 2000. Disponível em: <http://java.sun.com/products/javamail>

[SUN2004b] SUN MICROSYSTEMS, INC. *J2EE™ Connector Architecture 1.5 Specification – Final Release.* Novembro, 2004. Disponível em: <http://java.sun.com/j2ee/connector>

[SUT2004] SUTTOR, J.; WALSH, N.; KAWAGUCHI, K. *JSR 206 Java™ API for XML Processing (JAXP) 1.3.* Sun Microsystems, 2004.

[VOS1996] VOSS, G. *JavaBeans: Introducing JavaBeans.* November, 1996. Disponível em: <http://java.sun.com/developer/onlineTraining/*Beans*/*Beans*1/>

[SZY1999] Szyperski, C. *Component Software – Beyond Object-Oriented Programming*. [S.l]: Addison-Wesley, 1999.

[VOL2003] Völter, M. *A taxonomy of components*. Journal of *Object* Technology, v.2, n.4, July-Aug. 2003, pp.119-125.

[MIT2004] MITCHELL, J. D.; CHOI, A. *Java Tip 49: How to extract Java resources from JAR and zip archives*. Disponível em: <http://www.javaworld.com>

[TAN1996] TANEMBAUM, A. *Computer Networks*. Prentice-Hall, 3ª Edição, 1996.