# Encapsulando a *NavigationPlanTool* como Serviços *Web*

**Trabalho de Formatura Supervisionado**

**Mauricio Chui Rodrigues**

**Orientador: Prof. Dr. João Eduardo Ferreira**

**Instituto de Matemática e Estatística - Universidade de São Paulo**

*Aos meus pais, June e Marcos.*

# Agradecimentos

Ao meu orientador, Prof. Dr. João Eduardo Ferreira, pela oportunidade oferecida para a participação em um projeto de Iniciação Científica e pela atenção e interesse demonstrados durante todo o trabalho.

Aos meus pais, June e Marcos, e minhas irmãs, Cristine e Debora, por me apoiarem incondicionalmente durante a graduação e em todos os demais momentos da minha vida.

Aos doutorandos Kelly Rosa Braghetto e Marcos Eduardo Bolelli Broinizi pelas importantes contribuições e também pela paciência e atenção dispensadas a este trabalho.

A todos os meus amigos do IME-USP, companheiros de todas as horas e com os quais passei ótimos momentos nos últimos anos. Sem a presença deles, a graduação certamente não seria a mesma.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio financeiro a esta pesquisa.

# Índice

# Índice de Figuras

# Índice de Tabelas

# Siglas

| | |
|---|---|
| **ACP** | Algebra of Communicating Processes |
| **APB** | Álgebra de Processos Básica |
| **API** | Application Programming Interface |
| **BMP** | Bean-Managed Persistence |
| **CMP** | Container-Managed Persistence |
| **CORBA** | Common Object Request Broker Architecture |
| **EJB** | Enterprise JavaBeans |
| **HTML** | HyperText Markup Language |
| **IDE** | Integrated Development Environment |
| **IIOP** | Internet Inter-ORB Protocol |
| **JAXB** | Java Architecture for XML Binding |
| **JAX-RPC** | Java API for XML-Based RPC |
| **JAX-WS** | Java API for XML-Based Web Services |
| **JAR** | Java Archive |
| **J2SE** | Java 2 Platform, Standard Edition |
| **J2EE** | Java 2 Platform, Enterprise Edition |
| **JDBC** | Java Database Connectivity |
| **JDK** | Java Development Kit |
| **JMS** | Java Message Service |
| **JNDI** | Java Naming and Directory Interface |
| **JPA** | Java Persistence API |
| **MDB** | Message-Driven Bean |
| **MOM** | Message-Oriented Middleware |
| **NPDL** | Navigation Plan Definition Language |
| **NPWS** | Navigation Plan Web Services |
| **PN** | Plano de Navegação |
| **POJO** | Plain Old Java Object |
| **RMI** | Remote Method Invocation |
| **RMI-IIOP** | Remote Method Invocation over the Internet Inter-ORB Protocol |
| **RPC** | Remote Procedure Call |

| | |
|---|---|
| **SEI** | Service Endpoint Interface |
| **SFSB** | Stateful Session Bean |
| **SGBD** | Sistema Gerenciador de Banco de Dados |
| **SLSB** | Stateless Session Bean |
| **SOA** | Service-Oriented Architecture |
| **SOAP** | Simple Object Access Protocol |
| **SPI** | Service Provider Interface |
| **SQL** | Structured Query Language |
| **XML** | Extensible Markup Language |
| **WSDL** | Web Services Description Language |

# Capítulo 1

# Introdução

Um processo de negócio é um conjunto de um ou mais procedimentos que leva à realização de um objetivo de negócio. Sua execução apresenta condições bem definidas para início e fim e pode ser o resultado da combinação de procedimentos manuais e automáticos [30].

O termo *workflow*, por sua vez, refere-se à automação total ou parcial de um processo de negócio, pela qual ocorre o fluxo de informações entre participantes por meio de ações e sob regras procedurais.

A NPDL (*Navigation Plan Definition Language*) [7] é uma linguagem para a definição de processos de negócio. Trata-se de uma alternativa à representação de *workflows* que utiliza a álgebra de processos como formalismo e promove a separação explícita entre o ambiente de especificação e o ambiente de execução de um *workflow*.

A ferramenta *NavigationPlanTool* [6] apresenta o interpretador da NPDL e provê mecanismos para a instanciação e o controle de processos. Feita a especificação de um *workflow* por meio da NPDL, a execução de seus passos é controlada por esta ferramenta, também responsável pela interação com usuários e aplicativos.

## 1.1. Contextualização

A ferramenta *NavigationPlanTool* pode ser aplicada na composição de serviços básicos, o que é particularmente interessante em um ambiente *Web*, no qual os serviços que compõem um processo de validação ou controle de dados podem ser heterogêneos e autônomos.

Implementada na forma de biblioteca de funções, a ferramenta *NavigationPlanTool* foi desenvolvida em Java (*Java 2 Platform, Standard Edition – J2SE 5.0*) [19] e é facilmente integrável a outras aplicações desenvolvidas com esta mesma linguagem de programação. No

entanto, aplicações em outras linguagens necessitam de algum tipo de adaptador para que haja a devida interação com a ferramenta.

A disponibilidade de uma interface padrão independente de linguagens de programação contornaria a inviabilidade de implementar um adaptador para cada diferente linguagem. Eliminadas estas restrições, se os contatos com a ferramenta pudessem ser efetuados de qualquer ponto em uma rede de computadores, a *NavigationPlanTool* poderia enfim ter aplicação efetiva em um ambiente *Web*.

## 1.2. Objetivos

Este trabalho, realizado como Iniciação Científica e financiado pelo CNPq durante o período de dezembro de 2006 a dezembro de 2007, teve como meta principal o encapsulamento das funcionalidades da ferramenta *NavigationPlanTool* como serviços *Web* em um servidor de aplicação J2EE (*Java 2 Platform, Enterprise Edition*).

Mais precisamente, buscou-se a criação de serviços *Web* que delegassem as chamadas recebidas ao núcleo de execução da *NavigationPlanTool* e fossem responsáveis pelas seguintes tarefas: encapsular as definições de processos, monitorar a execução de processos e permitir a instanciação de um novo processo por meio de interfaces *Web*.

## 1.3. Organização do Texto

O Capítulo 2 contém fundamentos para o entendimento da NPDL e da ferramenta *NavigationPlanTool*, bem como suas definições. O Capítulo 3 introduz conceitos sobre sistemas distribuídos e descreve a arquitetura de componente *Enterprise JavaBeans*. No Capítulo 4 estão informações sobre *Web Services* e as tecnologias para o desenvolvimento de serviços *Web* em Java. O Capítulo 5 baseia-se nos anteriores para destacar a realização do trabalho, enquanto o Capítulo 6 apresenta a conclusão.

A parte subjetiva da monografia corresponde ao Capítulo 7, onde estão o relato sobre a experiência com o trabalho, a lista de disciplinas cursadas e consideradas relevantes para sua realização, além de expectativas e planos para o futuro.

# Capítulo 2

# NPDL e a Ferramenta *NavigationPlanTool*

A NPDL foi desenvolvida por Kelly Rosa Braghetto para sua dissertação de mestrado e apresentada em julho de 2006 ao Instituto de Matemática e Estatística da Universidade de São Paulo. A segunda contribuição do trabalho foi a ferramenta *NavigationPlanTool*, que contém o interpretador da NPDL.

## 2.1. Fundamentos

Para um razoável entendimento sobre a NPDL e a *NavigationPlanTool*, é interessante que alguns conceitos, estudados durante a preparação para o trabalho, sejam introduzidos: a Teoria de Processos, o formalismo da álgebra de processos, a definição de plano de navegação e as categorias de padrões de controle de fluxo. Também são necessárias algumas noções básicas sobre o modelo relacional de dados.

### 2.1.1. Teoria de Processos

Sistemas geralmente são caracterizados por processos, dinâmicos e ativos, e dados, estáticos e passivos. O comportamento dos sistemas tende a envolver a execução concorrente de vários processos, de modo que estes trocam dados entre si e influenciam-se uns aos outros [15].

A Teoria de Processos é o estudo voltado principalmente a duas atividades relacionadas a processos: verificação e modelagem [17]. A verificação consiste em provar enunciados sobre processos, já a modelagem refere-se à representação dos processos, geralmente por estruturas matemáticas ou expressões em uma linguagem descritiva de sistemas [6].

Segundo [3] e [15], o comportamento de um processo pode ser representado por um grafo de processo, ou seja, um grafo direcionado cujas arestas, denominadas transições, são rotuladas por nomes de ações e cujos vértices são chamados de estados. Uma ação pode ser qualquer

atividade realizada pelo sistema, já o conjunto de estados é uma abstração de todos os possíveis estados no qual o sistema pode ser encontrado.

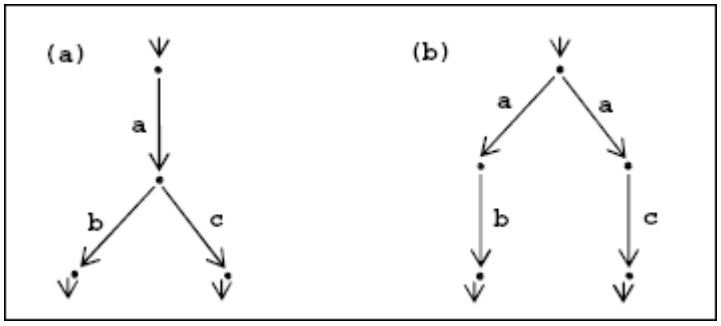



Figura 2.1: Exemplos de grafos de processos (Fonte: [6])

### 2.1.2. Álgebras de Processos

Grafos de processos podem ser representados algebricamente, na forma de termos, visando o raciocínio matemático. Uma álgebra de processos utiliza uma coleção de operadores para tratar a especificação e a manipulação de termos de processos, permitindo que sejam encontradas propriedades indesejáveis de uma especificação de sistema e servindo para a derivação formal das propriedades desejáveis [15].

Toda álgebra de processos é composta por um conjunto de símbolos de ações (eventos), um conjunto de operações e outro de axiomas descrevendo as propriedades dos operadores, porém pode ser estendida com novos operadores para aumentar sua expressividade ou facilitar a especificação do comportamento de sistemas [4].

As definições a seguir foram extraídas de [15].

Uma assinatura Σ é um conjunto finito de símbolos de funções (ou operadores), em que cada símbolo de função *f* apresenta uma aridade *ar(f)* referente a seu número de argumentos. O conjunto T(Σ) de termos sobre uma assinatura Σ é definido como o menor conjunto que satisfaça às seguintes condições: (i) cada variável está em T(Σ); (ii) se $f \in \Sigma$ e $t_1,...,t_{ar(f)} \in T(\Sigma)$, então $f(t_1,...,t_{ar(f)}) \in T(\Sigma)$. A assinatura de uma álgebra de processos consiste em:

- um conjunto finito e não-vazio A de ações atômicas;

- um operador binário "**.**", chamado de composição seqüencial. Sejam dois termos fechados $t_1$ e $t_2$ representando respectivamente processos $p_1$ e $p_2$, então o termo fechado $t_1$ **.** $t_2$ representa o processo que executa $p_1$ e então $p_2$.
- um operador binário "+", chamado de composição alternativa. Sejam dois termos fechados $t_1$ e $t_2$ representando respectivamente processos $p_1$ e $p_2$, então o termo fechado $t_1 + t_2$ representa o processo que executa $p_1$ ou $p_2$.

Um termo fechado é aquele que não possui variáveis. Cada processo finito pode ser representado por um termo de processos básicos, ou seja, um termo fechado composto de ações atômicas e dos operadores acima. A coleção de termos de processos básicos chama-se Álgebra de Processos Básica (APB).

A NPDL baseia-se em uma extensão da álgebra de processos chamada *Algebra of Communicating Processes* (ACP), portanto contém dois conceitos inexistentes na APB: o operador binário "||" da ACP [23], descrito na Seção 2.2.1, e expressões recursivas da ACP *with guarded recursion* [15].

Expressões recursivas são úteis pelo fato de muitos sistemas poderem conter ciclos, de modo que as iterações (finitas ou infinitas) precisam ser representadas. Por exemplo, o processo que executa seqüencialmente as ações *a* e *b* infinitas vezes pode ser representado por $X = a.Y$ e $Y = b.X$ [6].

### 2.1.3. A Arquitetura *RiverFish* e o Plano de Navegação

*RiverFish* [13][14] é uma arquitetura para a representação, o controle e a execução de processos de negócio. Segundo esta arquitetura, processos podem ser classificados como atômicos ou compostos. Processos atômicos são descritos em termos de passos de negócio, enquanto os compostos são obtidos a partir de outros processos de negócio [6]. O principal conceito associado a esta arquitetura é o Plano de Navegação (PN).

O PN foi definido formalmente em [13] e estendido em [14] como um conjunto de todos os processos de negócio exigidos para que sejam atingidos os objetivos de negócio em uma aplicação. As quatro definições a seguir são necessárias para complementar sua definição formal:

- Ação simples: conjunto de ações atômicas relacionadas por operadores de seqüência e composição alternativa;
- Ponto de verificação: conjunto de ações atômicas relacionadas por regras restritivas e condicionais;
- Passo de negócio: uma ação simples ou um ponto de verificação;
- Processo de negócio: conjunto de passos de negócio relacionados por operadores básicos de uma álgebra de processos e suas extensões.

Requisições são solicitações de serviços em um sistema de informação. Na arquitetura *RiverFish*, uma instância do PN é um vínculo entre uma requisição feita por um usuário e os passos de negócio que responderão por ela [6]. Cada nova requisição gera uma instância do PN, para que as instâncias sejam usadas no controle do processamento dos passos das requisições.

A possibilidade de reutilizar passos de negócio na definição de processos de negócio, a manutenção dos dados de controle de execução das instâncias de planos e a navegação em bancos de dados relacionais são importantes características da arquitetura *RiverFish* [6].

### 2.1.4. Padrões de Controle de Fluxo

Há vinte padrões de controle de fluxo, definidos em [1], que estabelecem critérios para comparar linguagens para a especificação de *workflows*. Cada linguagem pode ser classificada de acordo com a capacidade de expressar tais padrões por meio de sua sintaxe. Os padrões de controle de fluxo estão divididos em seis categorias [6]:

- Padrões básicos de controle de fluxo: relacionados às construções básicas presentes na maioria das linguagens de modelagem de *workflows*;
- Padrões avançados de ramificação e sincronização: referem-se a controles de fluxo mais avançados e que não estão presentes na categoria anterior;
- Padrões estruturais: representam a flexibilidade requerida pelas linguagens para lidar com paralelismos e múltiplos pontos de entrada e saída em blocos;
- Padrões de múltiplas instâncias: auxiliam na modelagem de casos específicos em que partes de um processo precisam ser instanciadas múltiplas vezes;

- Padrões baseados em estados: estendem a expressividade das linguagens, geralmente ligadas à modelagem de eventos e atividades, com a representação de estados;
- Padrões de cancelamento: modelam o cancelamento de atividades condicionado à ocorrência de um certo evento.

### 2.1.5. O Modelo Relacional de Dados

O modelo relacional, baseado na teoria dos conjuntos e álgebra relacional, é flexível e adequado por solucionar vários problemas encontrados no nível da concepção e implementação da base de dados [10]. Não há caminhos pré-definidos para o acesso aos dados, como ocorria nos modelos de dados precedentes.

A principal estrutura neste modelo é a relação (tabela), composta por um ou mais atributos (campos) que determinam os tipos de dados a serem armazenados. Cada instância do esquema de dados (linha da tabela) é denominada tupla (registro). Todas as estruturas de dados são organizadas em relações [10] e devem estar de acordo com as restrições impostas para evitar aspectos como a repetição e a perda de informações, bem como a incapacidade de representar parte das mesmas.

A figura abaixo apresenta três relações do modelo relacional Aluno - Disciplina, de modo que (a) possui três atributos e duas tuplas, enquanto (b) e (c) possuem dois atributos e três tuplas cada.

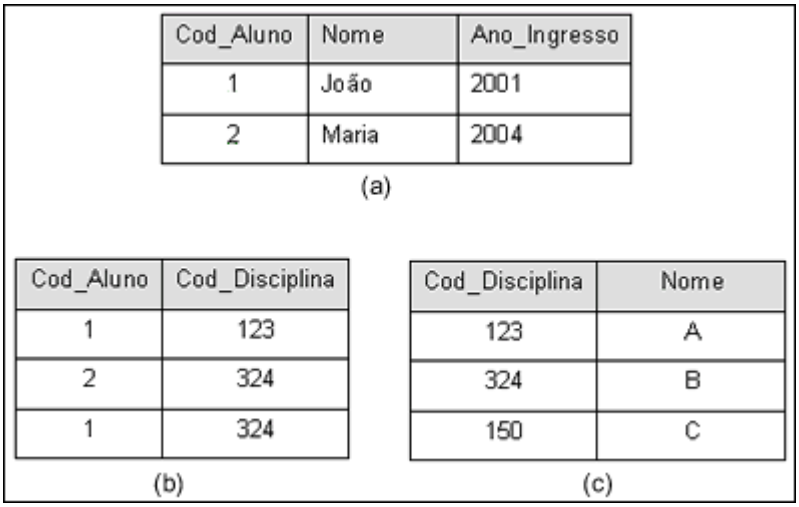



Figura 2.2: Tabelas do modelo relacional Aluno - Disciplina

## 2.2. Definição da NPDL

A NPDL (*Navigation Plan Definition Language*) é uma linguagem de definição de processos de negócio baseada em operadores de álgebra de processos e no conceito de plano de navegação [5][12]. A NPDL permite a especificação de padrões de controle de fluxo e é capaz de expressar alguns aspectos definidos em tempo de execução de processos. Desenvolvida como uma extensão da linguagem SQL (*Structured Query Language*), possibilita o controle de processos de negócio em um modelo relacional de dados.

Processos em NPDL são definidos por expressões algébricas, que podem ser construídas com operadores da NPDL a partir do conjunto de ações atômicas e do conjunto de processos definidos. Uma ação atômica equivale a uma transação que respeita as propriedades ACID (atomicidade, consistência, independência e durabilidade) em um banco de dados [6]: quando executada, ou é totalmente realizada ou é desfeita em caso de problemas na execução [11]. A única forma de comunicação entre ações é por meio do compartilhamento de dados.

### 2.2.1. Operadores da Linguagem

A NPDL contém os seguintes operadores [7]:

a) Composição seqüencial (operador "**.**"): o termo de processo *A.B* significa que inicialmente só o termo *A* está habilitado para execução. O termo *B* estará habilitado somente quando a execução de *A* for terminada;
b) Composição alternativa (operador "+"): o termo de processo *A* + *B* significa que inicialmente ambos os termos *A* e *B* estarão habilitados para execução. A escolha de um deles para execução descarta a execução do outro;
c) Composição paralela (operador "||"): o termo de processo *A* || *B* significa que os termos *A* e *B* podem ser executados paralelamente, dividindo a linha de execução em duas (ou seja, atuando de forma similar a subprocessos);
d) Composição paralela entrelaçada (operador "|*"): o termo de processo *A* |* *B* significa que os termos *A* e *B* podem ser executados em qualquer ordem, mas não paralelamente (i.e., *A.B* + *B.A*);

e) Composição multi-convergente (operador “&”): o termo de processo *A*&*B* significa que o termo *B* será habilitado para execução após o término da execução de cada linha de execução de *A*. Por exemplo, o processo *P* = (*X* || *Y*)&*Z* indica que *Z* será habilitado duas vezes, após o término da execução de *X* e após o término da execução de *Y*;
f) Composição discriminatória (operador “^”): o termo de processo A^B significa que o termo B será habilitado para execução somente após o primeiro término de uma linha de execução de A. Considerando o mesmo exemplo do item anterior, Z seria habilitado uma vez, após a execução de Y, se Y acabar primeiro, ou após a execução de X, caso contrário;
g) Repetição ilimitada (operador “?*”): o termo de processo *A*?* significa que o termo *A* pode ser executado, de forma paralela, uma ou mais vezes;
h) Repetição numericamente limitada (operador “?n”, sendo n um número inteiro positivo): o termo de processo *A*?3 significa que o termo *A* deve ser executado, de forma paralela, três vezes;
i) Repetição limitada por função (operador “?f”, sendo f uma função que devolve um número inteiro positivo): o termo de processo *A*?$f_1$ significa que *A* deve ser executado, de forma paralela, o número de vezes devolvido por $f_1$ em tempo de execução;
j) Execução condicional (operador “%r”, sendo r uma regra, ou seja, uma função que devolve um valor booleano – verdadeiro ou falso): o termo de processo %$r_1$ *A* significa que *A* será habilitado para execução se o retorno de $r_1$ for verdadeiro;
k) Execução condicional negativa (operador “%!r”, sendo r uma regra, ou seja, uma função que devolve um valor booleano – verdadeiro ou falso): o termo de processo %!$r_1$ *A* significa que o termo *A* será habilitado para execução se o retorno da regra $r_1$ for falso.

O comportamento dos operadores para execução condicional (j) e (k) não podem ser representados em álgebra de processos. Os operadores (d), (e), (f), (g), (h) e (i) também não existem em álgebra de processos, porém seus comportamentos podem ser representados com a combinação de operadores existentes. [7]

A tabela com as precedências dos operadores da NPDL está a seguir, juntamente com alguns exemplos de comandos da NPDL. A sintaxe completa dos comandos da NPDL pode ser encontrada em [7].

| | Alta ← Precedência → Baixa | | | | | | |
|---|---|---|---|---|---|---|---|
| **Operador** | *%r*<br>*%!r* | *?**<br>*?n*<br>*?f* | . | \|\| | \|* | ^<br>& | + |

Figura 2.3: Tabela de precedências dos operadores da NPDL (Fonte: [6])

```
CREATE RULE R1 'VerifyPurchasingForm';
CREATE ACTION A1 'ProcessPurchasing';
CREATE ACTION A2 'ApplyDiscornt';
CREATE ACTION A3 'PrintReceipt';
CREATE PROCESS P1 'Purchasing Control Process';
SET P1 = %R1 A1 . ( A2.A3 + A2);
```

Figura 2.4: Exemplos de comandos da NPDL (Fonte: [7])

## 2.3. A Ferramenta *NavigationPlanTool*

O interpretador da NPDL é parte da *NavigationPlanTool*, uma ferramenta que provê mecanismos para a manutenção de ações e processos em bancos de dados relacionais e para o controle de instanciação e execução desses processos [6].

A *NavigationPlanTool* foi desenvolvida em Java (*Java 2 Platform Standard Edition – J2SE 5.0*) [19] como uma biblioteca de funções, portanto é facilmente integrável a outras aplicações em Java. A comunicação com bancos de dados ocorre via JDBC (Java *Database Connectivity*), um conjunto de interfaces para a padronização do acesso, em Java, a bancos de dados relacionais.

Três serviços compõem esta ferramenta: o Interpretador da NPDL, o Serviço de Instanciação de Processos e o Serviço Monitor de Execução de Instâncias de Processo.

### 2.3.1. O Interpretador da NPDL

As funções do interpretador da linguagem são: o recebimento de comandos da entrada; a validação dos comandos por análises léxica, sintática e semântica; e a execução de comandos válidos sobre as relações criadas pelo próprio interpretador em um banco de dados relacional, em que são guardados dados de processos, ações e instâncias.

O interpretador é uma implementação da interface *java.sql.Connection* de Java e recebeu o nome NPDLConnection. Criar a conexão significa preparar o banco de dados para o uso no controle de processos de negócio via *NavigationPlanTool* e, caso não existam no banco, criam-se as relações utilizadas pela ferramenta [6].

Comandos NPDL e SQL podem ser utilizados: comandos NPDL válidos são convertidos em instruções SQL e aplicados sobre as relações no banco de dados. Os comandos NPDL inválidos são repassados diretamente ao banco, responsável pela validação como comandos SQL.

### 2.3.2. O Serviço de Instanciação de Processos

Este serviço disponibiliza funções para a criação de instâncias de processos. Uma instância é uma requisição de um determinado processo e, além de conter uma referência para o processo ao qual está associada, pode também conter informações sobre a data de criação e o usuário que a requisitou [6].

Quando uma aplicação requisita a criação de uma instância, recebe deste serviço um identificador da instância, o qual permite a obtenção de informações da mesma e a interação com o serviço monitor de execução.

### 2.3.3. O Serviço Monitor de Execução de Instâncias de Processo

Responsável por relacionar cada instância de processo a seus dados de execução (plano de navegação), este serviço lida com o início e o monitoramento da execução do plano de navegação das instâncias.

Quando a execução de uma instância é iniciada, o plano de navegação associado a ela é obtido. Cada operação com um passo implica na inserção ou atualização de um registro no *log*.

O serviço monitor de execução oferece as seguintes funções para as aplicações que utilizam a *NavigationPlanTool* [6]:

a) Obtenção dos passos disponíveis para execução: esta função devolve os nomes dos passos que a instância pode executar em seu estado atual. O passo pode ser uma ação, uma regra ou uma função que devolva um valor inteiro;
b) Indicação do início da execução de um passo: esta função recebe da aplicação o nome de um passo e, caso seja válido, cria um registro no *log* para informar sobre seu início. Um identificador do passo é devolvido para a aplicação;
c) Indicação do fim da execução de um passo: o identificador de um passo já iniciado é recebido por esta função e, se for válido, o registro no *log* é atualizado para indicar a finalização do passo;
d) Indicação do cancelamento da execução de um passo: esta função recebe o identificador de um passo já iniciado e, se for válido, o registro no *log* é atualizado para indicar o cancelamento do passo.

### 2.3.4. A Estrutura Relacional de Dados

Quando a *NavigationPlanTool* conecta-se pela primeira vez a um banco de dados relacional, o interpretador da NPDL cria as relações necessárias para a utilização da ferramenta. A prática mais comum é permitir o acesso desta estrutura tanto pela *NavigationPlanTool* quanto por outras aplicações, viabilizando o compartilhamento das definições de processos.

A seguinte descrição das entidades e relacionamentos presentes na estrutura relacional de dados foi extraída de [6].

Um processo é composto por passos. Toda expressão algébrica de processo é convertida em uma expressão equivalente em notação prefixa e então armazenada em banco de dados por meio do relacionamento REL_PLANO_NAVEGAÇÃO.

A entidade INSTÂNCIA_PROCESSO e o relacionamento REL_LOG_INSTÂNCIA estão associados à instanciação e ao controle de execução de processos e são utilizados pelos outros serviços da *NavigationPlanTool*.

Uma instância está sempre associada a um processo por meio do relacionamento REL_INSTANC_PROCESSO. O relacionamento REL_LOG_INSTÂNCIA representa os dados

relativos ao estado da execução dos passos que compõem o plano de navegação de uma instância de processo. Cada elemento de REL_LOG_INSTÂNCIA representa a execução de um passo em uma determinada instância de processo.

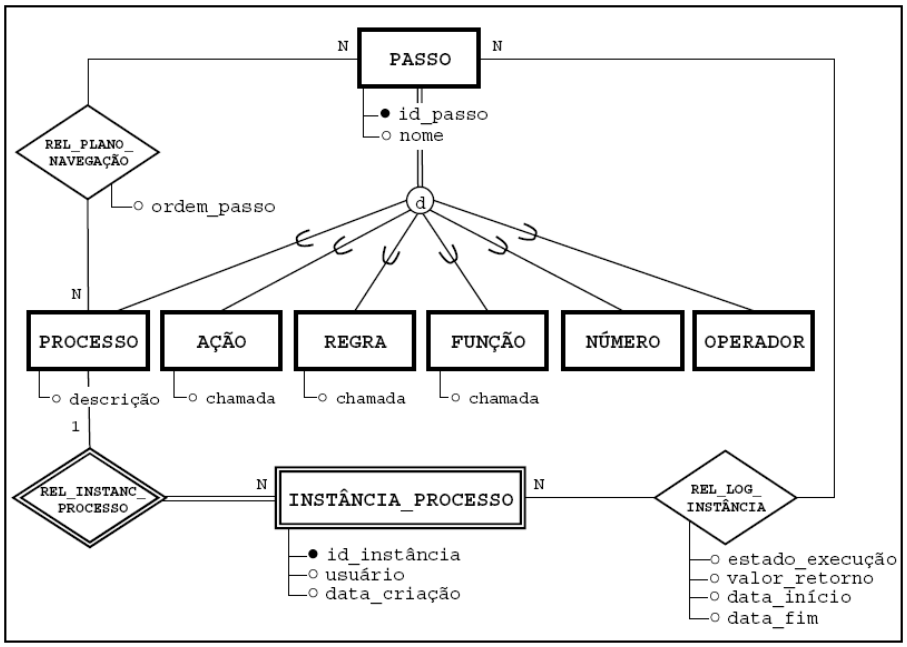



Figura 2.5: Diagrama Entidade-Relacionamento da estrutura criada pelo Interpretador da NPDL (Fonte: [6])

# Capítulo 3

# *Enterprise JavaBeans*

Os estudos para a realização deste trabalho incluíram diversos fundamentos sobre sistemas distribuídos, dentre os quais foram selecionados os conceitos mais relevantes para a exposição nesta monografia.

Apesar da utilização da versão 3.0 de *Enterprise JavaBeans* (EJB) [9] para realizar o trabalho, a versão 2.1 foi a base para toda a preparação. Por apresentarem sensíveis diferenças, serão introduzidas noções sobre a versão 2.1 para, então, serem identificadas as inovações presentes na versão mais recente.

## 3.1. Fundamentos

Sistemas distribuídos, *middleware*, servidores de aplicação e objetos distribuídos são conceitos que devem ser compreendidos antes de prosseguir neste capítulo.

### 3.1.1. Sistemas Distribuídos

Um sistema distribuído é uma coleção de computadores independentes que se apresenta ao usuário como um sistema único e consistente [28]. Trata-se de um sistema que realiza computação distribuída, ou seja, aproveita o potencial de dois ou mais computadores interligados em uma rede para processar uma determinada tarefa.

Uma boa estrutura é necessária para que um sistema distribuído possa ser devidamente desenvolvido e utilizado. Este suporte é obtido por meio do uso de um *middleware*, definido por [2] como uma complexa infra-estrutura de *software* que oferece abstrações para facilitar o desenvolvimento de aplicações distribuídas. Há diversos tipos de *middleware*, cada um com seu conjunto de serviços oferecidos. Um exemplo de serviço tipicamente oferecido por um *middleware* é o controle de transações.

A noção de sistemas distribuídos é bastante útil para o uso de componentes de *software*, unidades binárias de produção, aquisição e aplicação independentes que interagem para formar um sistema funcional [27]. Cada unidade realiza um serviço pré-definido e pode interagir com as demais, de modo que cada componente pode estar em um ponto qualquer de uma rede na realização do processamento de informações.

### 3.1.2. Servidores de Aplicação

Servidor de aplicação é o nome dado à plataforma que disponibiliza um ambiente para a implantação e execução de aplicações. Serviços comuns de *middleware* são geralmente encontrados nestas plataformas, o que possibilita ao desenvolvedor preocupar-se apenas em implementar sua aplicação. Quando pronta, deve ser inserida no ambiente do servidor para que possa usufruir os serviços oferecidos.

Um conjunto de interfaces, chamado arquitetura de componente, permite que um dado componente possa ser utilizado em qualquer servidor de aplicação [26]. Isto cria um alto nível de portabilidade, uma vez que o componente deixa de ser restrito ao servidor para o qual foi desenvolvido.

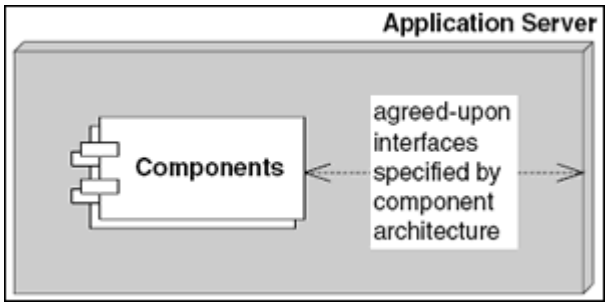



Figura 3.1: Arquitetura de componente (Fonte: [26])

### 3.1.3. Objetos Distribuídos

Um objeto é a representação conceitual de uma entidade em um sistema. Objetos são dotados de estado, definido pelos campos que possuem, e comportamento, definido pelos métodos que apresentam [29]. Um objeto distribuído é aquele que pode ter seus métodos invocados por sistemas remotos.

O procedimento a seguir é realizado quando um cliente (ou aplicação cliente) realiza uma chamada a um objeto distribuído. Tanto o procedimento quanto as observações sobre o mesmo têm [26] como fonte.

1. O cliente realiza a chamada para um objeto situado junto a ele, chamado *stub*. Este tem a função de intermediar a comunicação e age como uma máscara, ocultando do cliente os detalhes da comunicação;
2. O *stub* entra em contato, pela rede, com um *skeleton*, objeto intermediador situado no servidor. O *skeleton* é o responsável pelo recebimento das informações e por sua transmissão ao objeto distribuído;
3. O objeto distribuído consome as informações recebidas para executar o que lhe foi solicitado, devolvendo o controle das operações ao *skeleton*, que o transmite de volta ao *stub*. Por fim, o controle retorna ao cliente, junto com os possíveis resultados da execução.

O *stub* e o objeto distribuído apresentam uma mesma interface, chamada interface remota, o que passa ao cliente a falsa impressão de realizar contato direto e local com o objeto que está no servidor. Teoricamente, pode-se afirmar que existe transparência de distribuição. Na prática, porém, há diferenças no tratamento de erros durante a comunicação com objetos remotos e são encontradas limitações de rede, ou seja, o cliente percebe a existência da distribuição. Portanto é mais apropriado afirmar que existe transparência de localização.

Antes de haver a troca de informações entre *stub* e *skeleton*, é preciso colocá-las em um formato intermediário para transmissão. Segundo [2], este processo é composto por duas etapas, realizadas tanto pelo *stub* quanto pelo *skeleton*: primeiro é feito o empacotamento (*marshalling*), seguido pela “serialização” (*serialization*) dos dados. Para reverter o processo, realizam-se, nesta ordem, a “desserialização” (*deserialization*) e o desempacotamento (*unmarshalling*) dos dados.

O empacotamento consiste no agrupamento dos dados em um formato padrão de mensagem, para que esta possa ser compreendida pelo receptor. “Serialização” é o nome dado à transformação da mensagem em uma seqüência de *bytes* antes do envio pelo canal de comunicação [2].

É interessante notar que um objeto distribuído não é apenas uma implementação de objeto, mas a abstração criada pela interação de *stub*, *skeleton* e objeto.

## 3.2. Tecnologias de Suporte

Três tecnologias destacam-se na utilização de EJB: Java RMI-IIOP (Java *Remote Method Invocation over the Internet Inter-ORB Protocol*), JMS (Java *Message Service*) e JNDI (Java *Naming and Directory Interface*). Por apresentarem muitos detalhes que fogem ao escopo desta monografia, apenas uma breve introdução será feita a cada uma das tecnologias.

### 3.2.1. Java RMI-IIOP

Segundo [2], RPC (*Remote Procedure Call*) é um mecanismo que permite a comunicação entre clientes e servidores por meio de chamadas (ou invocações) remotas de procedimentos.

O RMI (*Remote Method Invocation*) é uma adaptação do RPC orientada a objetos, na qual os procedimentos são métodos presentes em objetos remotos. Toda aplicação que utiliza este tipo de comunicação deve estar de acordo com a API (*Application Programming Interface*) do mecanismo, ou seja, com o conjunto de interfaces que o RMI disponibiliza para o desenvolvimento de aplicações.

Java RMI-IIOP é um mecanismo presente no J2EE para a comunicação em rede. Baseia-se em invocações remotas de métodos sobre o protocolo IIOP (*Inter-ORB Protocol*), desenvolvido inicialmente para suportar a comunicação entre componentes de CORBA (*Common Object Request Broker Architecture*), uma arquitetura da década de 90 para o desenvolvimento de aplicações distribuídas orientadas a objetos.

Para que seja acessível remotamente, um objeto deve estar associado a uma interface remota com informações sobre seus métodos. O servidor J2EE portador do RMI-IIOP é o responsável pela geração de *stubs* e *skeletons*, entre os quais ocorre a comunicação descrita na Seção 3.1.3.

### 3.2.2. Java *Message Service*

JMS é um mecanismo voltado a sistemas baseados em troca de mensagens [26] cuja API é mais complexa do que a presente no RMI-IIOP. Uma camada intermediária é responsável por

interceptar as mensagens enviadas e direcioná-las a seus destinatários. Esta camada, denominada MOM (*message-oriented middleware*), geralmente acumula outras funções, como a de armazenar mensagens até que os respectivos destinatários estejam disponíveis.

A arquitetura do JMS é composta por um conjunto de interfaces voltado a clientes e por uma SPI (*Service Provider Interface*), que padroniza o acesso a serviços proprietários de um fornecedor qualquer. O JMS promove, portanto, a separação total entre as aplicações clientes e a implementação dos serviços disponíveis.

Há dois modelos principais para a troca de mensagens: um é o *publish/subscribe*, em que produtores enviam mensagens para um tópico lido por todos os receptores interessados; outro é o ponto-a-ponto, em que produtores enviam mensagens para uma fila e o primeiro receptor a acessá-la é o único a obter a primeira mensagem da fila naquele momento.

A comunicação por troca de mensagens é assíncrona, o que significa que uma aplicação envia uma mensagem e não precisa bloquear suas operações enquanto aguarda a resposta. Embora esta e outras vantagens sejam proporcionadas pela MOM, a eficiência na comunicação é menor do que a oferecida por RMI-IIOP, dado que as mensagens não seguem diretamente para os destinatários. Surgem, ainda, grandes preocupações em torno da camada intermediária, por ser essencial para a comunicação entre quaisquer aplicações.

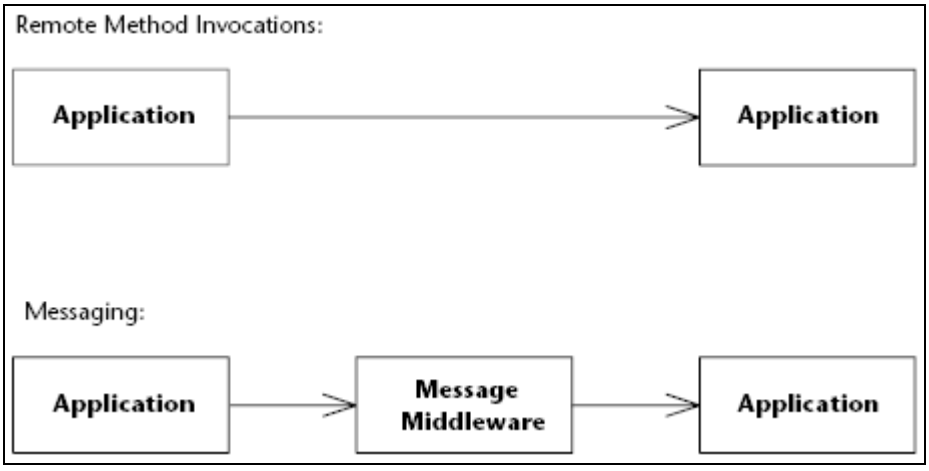



Figura 3.2: RMI x envio de mensagens (Fonte: [26])

### 3.2.3. Java *Naming and Directory Interface*

*Naming service* é um serviço que viabiliza a associação de nomes a objetos e a localização de objetos por seus nomes. *Directory service* é uma extensão do conceito de *naming service* por permitir que atributos de objetos também possam ser aproveitados em tarefas.

O JNDI é um sistema J2EE para clientes baseados em Java interagirem com *naming services* e *directory services*. Sua arquitetura é análoga à do JMS, composta por um conjunto de interfaces e por uma SPI.

Para ter acesso ao *namespace*, conjunto dos nomes existentes no sistema, uma aplicação solicita ao serviço um contexto inicial, ponto de início para explorar o *namespace*. Obtido o contexto, a aplicação pode invocar os métodos para a execução de operações como a listagem de nomes de objetos, a busca de um objeto pelo nome e até mesmo a associação de um objeto a um determinado nome.

## 3.3. Conceitos sobre EJB

*Enterprise JavaBeans* (EJB) é uma arquitetura de componente que simplifica a construção de aplicações distribuídas em Java e visa oferecer portabilidade e reutilização de aplicações por meio de serviços de *middleware* [26].

Os componentes de *software* desenvolvidos com EJB são denominados *enterprise beans*. Baseados no conceito de objetos distribuídos, estes componentes são classificados como *server-side* por precisarem ser implantados em um servidor de aplicação, onde são administrados e aproveitam os serviços oferecidos.

Além da comunicação entre aplicações clientes e *enterprise beans*, um componente pode realizar contato local com outros componentes, o que propicia a divisão de tarefas. As interações locais são muito mais rápidas do que as remotas por não precisarem de *stubs* e *skeletons*, porém os componentes devem estar obrigatoriamente no mesmo servidor.

Até a versão 2.1 de EJB, os *enterprise beans* eram divididos em três grupos: *session beans*, *entity beans* e *message-driven beans*. É interessante conhecer a antiga divisão para que possa ser mais bem compreendida a divisão adotada a partir de EJB 3.0, descrita na Seção 3.5.

### 3.3.1. *Session Beans*

Este grupo é composto por *enterprise beans* que executam ações e cujas instâncias têm o ciclo de vida limitado pela sessão estabelecida pela aplicação cliente. Componentes deste grupo não são persistentes, ou seja, não podem ser armazenados, por exemplo, em bancos de dados.

*Session beans* podem ser classificados como *stateful* ou *stateless*. Os *stateful session beans* (SFSB) suportam a memorização do estado da comunicação com cada cliente, de modo que cada comunicação pode incluir várias requisições. Já os *stateless session beans* (SLSB) não armazenam estado: se um mesmo cliente enviar duas requisições, um SLSB não será capaz de reconhecer que o autor de uma requisição é o mesmo da outra.

Uma vantagem dos SLSB particularmente importante para o trabalho realizado é a possibilidade de exposição como serviços *Web*.

### 3.3.2. *Entity Beans*

*Entity beans* são componentes persistentes que representam entidades do mundo real. Se combinado com o modelo relacional de dados, cada *entity bean* é associado a uma diferente relação no banco de dados, de modo que cada instância de um componente corresponde a uma tupla da respectiva relação. Estes *beans* podem apresentar ciclos de vida muito mais longos do que os de *session beans*, além de resistir a falhas, por exemplo, do servidor de aplicação.

Componentes deste grupo guardam os dados como seus atributos e apresentam métodos tanto para a alteração de dados quanto para o armazenamento e a recuperação dos mesmos em meio persistente. Todo contato com bancos de dados é realizado via JDBC e não pode ser diretamente requisitado por uma aplicação cliente, pois *entity beans* são acessíveis apenas localmente, por meio de outros *beans* no mesmo servidor.

Cada alteração realizada nas informações de um *entity bean* afeta diretamente os dados armazenados. A instanciação de um componente implica na criação de uma tupla na relação associada, enquanto a destruição de uma instância resulta na remoção da respectiva tupla.

Há duas formas de persistir *entity beans*: BMP (*bean-managed persistence*) e CMP (*container-managed persistence*). Com BMP, todo o código de persistência é escrito pelo desenvolvedor da aplicação, o que aumenta consideravelmente o número de linhas de código, porém oferece maior controle sobre o ciclo de vida do *bean*. Com CMP, o servidor é encarregado da persistência e da manutenção do *bean*, resultando em um número menor de linhas de código e em operações otimizadas, porém o desenvolvedor perde o controle da situação.

### 3.3.3. *Message-Driven Beans*

Os *message-driven beans* (MDB) representam a execução de ações e não são persistentes. No entanto, diferente dos *session beans*, baseiam-se em JMS e, portanto, são orientados a mensagens. Isto significa que não há interface para o acesso remoto por clientes, a única forma de entrar em contato com o componente é por meio de mensagens.

As características da comunicação com MDB estão descritas na Seção 3.2.2. Em particular, a opção por componentes deste tipo pressupõe que todas as aplicações clientes são capazes de enviar e receber mensagens apropriadas para o JMS. Neste caso, recomenda-se que a versão do JMS em cada cliente e a versão junto ao MDB sejam do mesmo provedor, para evitar qualquer possível incompatibilidade.

## 3.4. Artefatos em EJB 2.1

A versão 2.1 de EJB exige que toda a estrutura necessária seja explicitamente criada pelo desenvolvedor de um componente. Esta estrutura é representada por um arquivo compactado que contém a classe do *enterprise bean*, sua interface, sua interface *home* e um descritor de implantação. A seguir está uma breve descrição de cada item da estrutura, além dos procedimentos que uma aplicação cliente deve realizar para interagir com um *enterprise bean*.

### 3.4.1. A Classe do *Enterprise Bean*

Esta é a implementação do *enterprise bean*, que varia de acordo com o tipo de componente. Todo componente deve implementar uma determinada interface de acordo com seu tipo (`javax.ejb.SessionBean`, `javax.ejb.EntityBean` ou `javax.ejb.MessageDrivenBean`), de modo que todos os métodos expostos pela interface têm presença obrigatória na implementação, mesmo que nada façam quando chamados.

### 3.4.2. A Interface do *Enterprise Bean*

A comunicação entre clientes e componentes é intermediada por objetos no servidor, denominados EJB *Objects*, que interceptam as chamadas e são responsáveis pela transmissão aos *beans*. Para que um EJB *object* saiba quais métodos de um componente deve clonar, o desenvolvedor precisa especificar uma interface em que declara todos os métodos expostos pela classe do *bean*.

A interface de um componente pode ser local, para acesso por outros componentes, ou remota, para o recebimento de chamadas de clientes, mas é possível que um mesmo *bean* apresente os dois tipos de interface. Toda interface local precisa estender a interface `javax.ejb.EJBLocalObject` e toda interface remota, a interface `javax.ejb.EJBObject`. Ambas são especificadas pelo EJB e seguem regras do Java RMI-IIOP, como restrições na passagem de parâmetros e o lançamento de exceções em caso de falhas.

### 3.4.3. A Interface *Home*

EJB *objects* podem estar em qualquer lugar do servidor. A transparência na localização desses objetos deve-se à existência de fábricas capazes de criar, localizar e destruir EJB *objects*. Cada fábrica, denominada *home object*, é automaticamente gerada pelo servidor e associada a um nome no JNDI. Para que um *home object* identifique o tipo de objeto com o qual deve operar, o desenvolvedor da aplicação deve especificar uma interface *home*.

Toda interface *home* apresenta os métodos expostos por sua fábrica, que devem estar de acordo com o Java RMI-IIOP. Assim, para que uma interface *home* seja definida como local e

opere com EJB *objects* locais, deve estender a interface `javax.ejb.EJBLocalHome`. Interfaces *home* remotas, por sua vez, operam com EJB *objects* remotos e estendem a interface `javax.ejb.EJBHome`.

### 3.4.4. O Descritor de Implantação

Este arquivo, em XML (*Extensible Markup Language*), contém todas as informações necessárias para implantar os *beans* em um servidor de aplicação. No descritor são especificados os componentes que serão disponibilizados, bem como suas classes, interfaces e interfaces *home*, entre outras propriedades.

O descritor pode ser alterado tanto pelo desenvolvedor quanto por quem apenas adquire *beans* previamente desenvolvidos. No último caso, geralmente não há acesso ao código-fonte dos componentes, portanto é interessante que as especificações sejam feitas em um arquivo à parte.

### 3.4.5. O Arquivo EJB-jar

As classes de componentes, todas as interfaces e o descritor devem ser compactados em um arquivo chamado EJB-jar. Feito isso, os *enterprise beans* no arquivo estão prontos para a implantação em um servidor de aplicação.

### 3.4.6. A Aplicação Cliente

O contato com um *enterprise bean* ocorre por meio do envio de requisições, seja via RMI-IIOP (SFSB ou SLSB) ou por meio do envio de mensagens (MDB).

No primeiro caso, a aplicação cliente obtém um contexto inicial no JNDI com as propriedades de acesso ao servidor e procura a interface *home* do *bean*, de acordo com as informações presentes no descritor de implantação. Esta interface é utilizada para solicitar a criação de um objeto por meio do qual a aplicação pode entrar em contato com o componente.

No segundo caso, a aplicação deve obter o contexto inicial no JNDI e localizar a fábrica de conexões JMS e o MDB destinatário. Feito isso, os seguintes passos devem ser realizados: o estabelecimento de uma conexão; a criação de uma sessão, de um produtor de mensagens e de uma mensagem; o preenchimento da mensagem; e o envio da mensagem para o *bean*.

## 3.5. Inovações em EJB 3.0

Tudo o que existia em EJB 2.1 continuou a existir em EJB 3.0 para que não houvesse incompatibilidade com as aplicações já existentes. No entanto, a nova versão da tecnologia trouxe consigo diversas novidades para o desenvolvimento de *enterprise beans*.

Uma importante diferença é o incentivo ao uso de *entities* de JPA (Java *Persistence API*), um conjunto de interfaces cujo objetivo é a padronização da persistência de informações em bancos de dados relacionais. *Entities* são objetos Java que desempenham o mesmo papel dos *entity beans*, portanto pode-se dizer que, a partir de EJB 3.0, os componentes podem ser classificados apenas como *session beans* ou *message-driven beans*.

As demais diferenças visam facilitar o desenvolvimento de componentes. Parte delas é decorrente do uso de anotações Java, recurso oferecido a partir da versão 5.0 do JDK (Java *Development Kit*) para a especificação, no próprio código Java, de propriedades que são avaliadas em tempo de execução. Podem-se citar como principais inovações nos artefatos de EJB:

- Cada *enterprise bean* é um objeto Java qualquer, o chamado POJO (*Plain Old Java Object*). A declaração do tipo de um componente é feita por meio das anotações `@Stateless`, `@Stateful` e `@MessageDriven`, não mais pela implementação de interfaces específicas de EJB. Métodos destas interfaces deixaram, então, de ser obrigatórios na implementação de componentes;
- As interfaces de componentes não precisam estender interfaces de EJB, pois a declaração pode ser feita no código, com as anotações `@Local` e `@Remote`;
- Todas as informações do descritor de implantação podem ser introduzidas no código por meio de anotações Java. Apesar de a tarefa se tornar mais fácil, há a perda de flexibilidade devido ao fato de, geralmente, o desenvolvedor ser o único a ter acesso ao código-fonte;
- A interface *home* foi excluída dos artefatos necessários para o desenvolvimento de *beans*.

A grande diferença para as aplicações clientes está na busca por referências para os *session beans*. Ao invés de procurar no JNDI o nome relacionado ao *bean* para ter acesso à interface *home* e solicitar a criação de um objeto para a realização das chamadas, cada aplicação recebe diretamente este objeto como resultado da busca. Assim, clientes não precisam mais lidar com a fabricação de objetos via interfaces *home*.

# Capítulo 4

# *Web Services*

*Web Services* é uma maneira de expor as funcionalidades de um sistema de informações e torná-las disponíveis por meio de tecnologias *Web* padrões. O uso destas tecnologias reduz a heterogeneidade e visa facilitar a integração de aplicações [2].

O conceito de serviços *Web* surgiu recentemente e é utilizado com grande freqüência. Embora sejam vários os conceitos relacionados e as definições existentes, este capítulo tem como foco as informações relevantes para a realização do trabalho e evita a exposição de detalhes.

## 4.1. Fundamentos

Um serviço pode ser definido como um grupo de componentes relacionados que têm um dado processo de negócio como função [26]. A composição de serviços autônomos viabiliza a reutilização, bastante visada por empresas devido ao aproveitamento de investimentos. O conceito de arquitetura orientada a serviços, em particular, tem grande importância no aumento da reutilização.

### 4.1.1. Arquiteturas Orientadas a Serviços

Arquiteturas orientadas a serviços (*Service-Oriented Architectures*, SOA) são um paradigma para o desenvolvimento de serviços que proporcionam um maior nível de abstração funcional. Em SOA, serviços são entidades autônomas que interagem entre si apesar de possíveis diferenças de implementação ou de plataforma [26]. A noção de integrar estes serviços torna-se cada vez mais importante com o advento de novas tecnologias.

As SOA são complementares às arquiteturas de componente. Enquanto as últimas oferecem reutilização em alto nível, associada ao desenvolvimento de aplicações, as primeiras

cuidam para que o mesmo aconteça em um nível inferior e mais próximo dos protocolos utilizados na comunicação. Portanto é possível que um serviço seja desenvolvido, por exemplo, com EJB.

## 4.2. Definição de *Web Services*

Diversas são as formas de implementação das características do paradigma de SOA. Uma delas, denominada *Web Services*, refere-se a um grupo de tecnologias que utilizam XML como padrão de comunicação. Cada entidade autônoma neste grupo de tecnologias chama-se serviço *Web* (ou *Web service*).

Um serviço *Web* é aquele disponível em uma rede, pública ou privada, e que utiliza um sistema padronizado de troca de mensagens em XML sem depender de sistemas operacionais ou linguagens de programação [8].

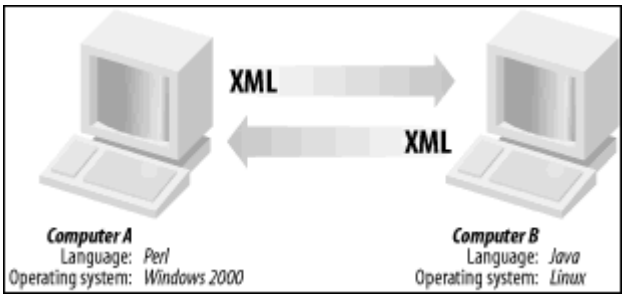



Figura 4.1: Exemplo de serviço *Web* básico (Fonte: [8])

O protocolo SOAP (*Simple Object Access Protocol*) e a linguagem WSDL (*Web Services Description Language*) são partes essenciais de *Web Services*, cuja popularidade deve-se ao suporte de XML, SOAP e WSDL por várias plataformas, o que viabiliza a interoperabilidade de aplicações.

### 4.2.1. SOAP e WSDL

*O* protocolo SOAP baseia-se em XML para a troca de informações entre computadores [8]. Suas mensagens, inteiramente em XML, são independentes de plataformas e linguagens e permitem que aplicações entrem facilmente em contato com serviços *Web* para a realização de chamadas remotas a suas funções.

A WSDL é uma especificação para descrever serviços *Web* em uma gramática comum do padrão XML, representando um contrato entre o solicitante e o provedor de um determinado serviço. Quatro aspectos críticos são retratados pela WSDL [8]:

- A interface do serviço e suas funções disponíveis publicamente;
- Os tipos de dados nas mensagens de pedido e de resposta;
- O protocolo de transporte a ser utilizado;
- O endereço e o nome para localizar o serviço.

## 4.3. Desenvolvendo Serviços *Web* em Java

Duas tecnologias destacam-se no desenvolvimento de serviços *Web* em Java: JAX-RPC (*Java API for XML-Based RPC*) e JAX-WS (*Java API for XML-Based Web Services*).

A JAX-RPC utiliza as versões 1.1 de SOAP e WSDL para suportar a exposição de aplicações como serviços *Web*. As mensagens XML recebidas são convertidas em chamadas Java e repassadas às aplicações, que devolvem os resultados da execução para que sejam convertidos no formato XML e enviados em nova mensagem ao solicitante. Entre outros detalhes técnicos, a JAX-RPC possui seu próprio modelo de mapeamento dos dados para Java 1.4, com algumas limitações quanto ao que pode ou não ser mapeado.

O nome JAX-WS foi adotado a partir do JAX-RPC 2.0, versão compatível com as anteriores apesar de suportar a utilização do SOAP 1.2 e apresentar consideráveis inovações. Uma é o modelo de mapeamento JAXB (*Java Architecture for XML Binding*), mais completo do que o anterior, por prometer a remoção das limitações no mapeamento, e também mais prático, por se tratar de um padrão já reconhecido para o acesso a documentos XML. Outra inovação é o suporte a Java 5.0, o que ampliou os recursos da tecnologia e viabilizou o mapeamento de informações em XML para esta versão do Java.

### 4.3.1. Serviços *Web* com EJB

Conforme descrito na Seção 3.3.1, *stateless session beans* (SLSB) podem ser expostos como serviços *Web*. Para que a JAX-RPC e a JAX-WS possam originar o serviço *Web*

correspondente a um SLSB, é necessária a implementação de uma interface remota diferente da mencionada anteriormente, chamada *Service Endpoint Interface* (SEI). Todo método declarado na SEI pode ser invocado, via mensagens em XML, no serviço *Web* resultante.

A declaração de uma interface como SEI, em EJB 2.1, é possível pelo acréscimo de informações no descritor de implantação. Na versão 3.0, as propriedades são especificadas na classe do *enterprise bean*, com a anotação `@WebService`, e na nova interface, com as anotações `@WebService`, `@SOAPBinding` e `@RemoteBinding`.

# Capítulo 5

# Realização do Trabalho

Este capítulo apresenta todas as atividades realizadas, bem como seus resultados e os produtos obtidos. Para facilitar a compreensão, o capítulo está dividido em seis seções: a primeira relata a preparação para o início do trabalho; a segunda aborda a aplicação dos conhecimentos obtidos para o desenvolvimento do que foi proposto; a terceira e a quarta seções referem-se a propriedades do produto final; a quinta descreve artefatos implementados para complementar o trabalho; e a sexta destaca a produção da documentação.

## 5.1. Estudos Iniciais e Preparação

Boa parte do tempo foi dedicada ao estudo da NPDL, da *NavigationPlanTool* e, principalmente, de EJB e seus fundamentos. A versão 3.0 de EJB havia sido divulgada pouco tempo antes dos estudos, portanto as fontes para pesquisa eram escassas e optou-se por material sobre EJB 2.1. Vários exemplos foram implementados para a assimilação da tecnologia antes de haver as primeiras tentativas com EJB 3.0, de modo que a adaptação à nova versão ocorreu gradativamente.

Desde o início, o servidor de aplicação escolhido para o desenvolvimento foi o JBoss AS (JBoss *Application Server*), uma plataforma de código aberto certificada com o J2EE e amplamente utilizada pela comunidade Java. A versão inicialmente utilizada para o trabalho foi a 4.0.5, última versão estável no período de implementação. Após alguns meses, houve o lançamento da série 4.2 de versões estáveis do JBoss AS, dentre as quais foi escolhida a 4.2.1 para dar continuação ao trabalho.

Os contatos iniciais com o servidor de aplicação, para a implantação de componentes, foram baseados em documentos encontrados na Internet, como os tutoriais disponibilizados pelos próprios desenvolvedores do JBoss AS.

## 5.2. Implementação

Não houve nítida separação entre as fases de implementação e fases de testes. Quando algum erro era detectado durante a execução ou alguma irregularidade era encontrada nos resultados, a busca por uma solução recebia maior atenção do que a continuidade da implementação. Pode-se afirmar o mesmo sobre refatorações no código: quando se identificava uma forma mais eficaz e/ou mais clara de realizar uma determinada tarefa, o código era alterado assim que possível para incorporar a nova forma.

### 5.2.1. Os Grupos de Funções

Selecionadas as funções da *NavigationPlanTool* a serem expostas na interface *Web*, realizou-se um agrupamento das mesmas com base no escopo de cada função. A divisão em grupos proporcionou melhor organização e viabilizou o balanceamento da carga de chamadas entre os futuros componentes, descritos na Seção 5.2.2. Quatro grupos resultaram deste processo:

1. Funções para a execução de comandos NPDL e SQL;
2. Funções relacionadas a processos, como instanciação, remoção de instâncias e consultas a informações sobre os processos existentes;
3. Funções relacionadas a instâncias de processos, como consultas a informações sobre instâncias e obtenção do *log* de execução de uma instância;
4. Funções relacionadas a passos (ações, regras ou funções), como consultas a suas informações e execução de um passo.

### 5.2.2. Expondo as Funcionalidades da *NavigationPlanTool*

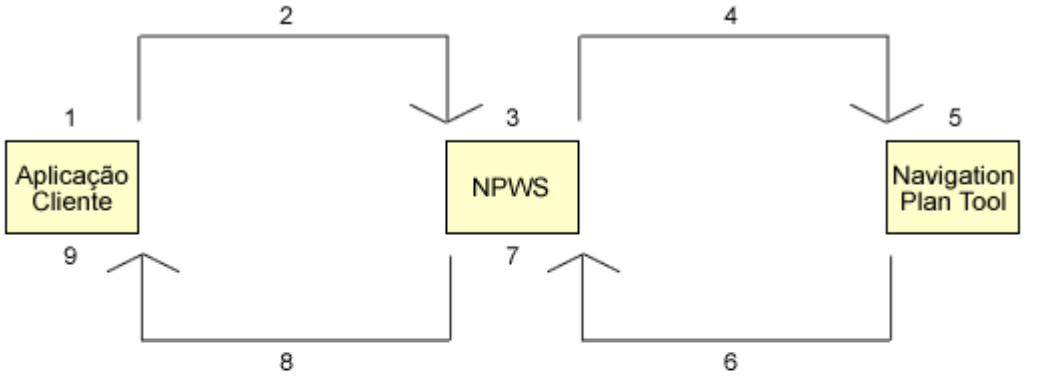
Cada grupo de funções foi implementado por um diferente SLSB com sua respectiva SEI para a exposição como serviço *Web*. Informações sobre as funções, como seus nomes, suas descrições, os parâmetros necessários e a devolução de resultados, estão em [25]. Os componentes e serviços receberam nomes para a identificação, como visto na tabela a seguir:

| Grupo | Nome do componente | Nome do serviço *Web* |
|---|---|---|
| 1 | CommandsBean | Commands |
| 2 | ProcessesBean | Processes |
| 3 | InstancesBean | Instances |
| 4 | StepsBean | Steps |

Tabela 5.1: Nomes dos componentes e de seus respectivos serviços *Web*

Apesar de parte das funções nas interfaces dos componentes serem homônimas a suas correspondentes na *NavigationPlanTool*, cada chamada a uma função de um SLSB não é simplesmente redirecionada à ferramenta.

Quando uma aplicação solicita, via uma mensagem XML, a execução de uma função, o SLSB responsável é acionado para realizar as operações necessárias para o estabelecimento do contato com a *NavigationPlanTool*. Feito isso, o componente invoca a função da ferramenta que corresponde à requisitada pelo cliente e recebe os resultados da execução na linguagem Java. As informações são devidamente formatadas para a conversão em XML e, finalmente, devolvidas à aplicação solicitante.




| |
|---|
| 1. O cliente realiza chamada, em sua linguagem, a uma função exposta pelo NPWS. |
| 2. A chamada é enviada a um serviço do NPWS como uma mensagem em XML. |
| 3. O serviço prepara a comunicação com a ferramenta *NavigationPlanTool*. |
| 4. Uma chamada, em Java, é encaminhada pelo serviço à *NavigationPlanTool*. |
| 5. A função correspondente à solicitada pelo cliente é executada. |
| 6. O resultado da execução é devolvido, em Java, pela ferramenta. |
| 7. O serviço formata devidamente as informações para o envio. |
| 8. As informações são enviadas ao cliente em nova mensagem em XML. |
| 9. O cliente recebe os resultados convertidos em sua linguagem de programação. |

Figura 5.1: Acesso a uma funcionalidade da *NavigationPlanTool* via o NPWS

O trecho de código a seguir exemplifica o que o componente StepsBean precisaria executar se uma aplicação solicitasse a execução da função `getActionInfoById()` exposta em sua SEI. A função `writeStepInfo()`, cujo conteúdo não é relevante para a compreensão do procedimento, não integra a interface do componente por se tratar de uma função auxiliar, responsável por analisar um passo, extrair suas informações, formatá-las adequadamente e devolvê-las em uma nova estrutura de dados.

```
// Criacao de conexao com a fonte de dados
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("nome_da_fonte_de_dados”);
Connection connection = ds.getConnection();
NPDLConnection npdlConn = new NPDLConnection(connection);

// Contato com a NavigationPlanTool (devolucao de objeto)
NPDataAccessor npAccessor = new NPDataAccessor(npdlConn);
Step step = npAccessor.getAction(stepId);

// Fim da conexao e extracao de informacoes
npdlConn.close();
return writeStepInfo(step);
```

Figura 5.2: Linhas de código necessárias para executar a função `getActionInfoById()`

A abstração resultante da preparação feita pelo componente é uma grande vantagem para os desenvolvedores de aplicações por não precisarem se preocupar com detalhes da comunicação, como o estabelecimento de conexões. Caso a *NavigationPlanTool* sofra alguma alteração que leve a uma preparação diferente, apenas o código do componente precisa ser alterado, ao invés do código de todas as aplicações clientes.

Outra importante vantagem decorrente da abstração é a independência de linguagens de programação. Uma determinada aplicação cliente pode ser implementada em uma linguagem qualquer, orientada a objetos ou não, desde que suporte a comunicação via mensagens em XML. No trecho de código acima, objetos Java estão presentes na preparação da chamada e também podem ser devolvidos como resultados da execução de funções. O componente extrai as informações relevantes do objeto para repassá-las formatadas aos clientes, portanto evita que os objetos estejam ao alcance de linguagens incapazes de compreendê-los.

### 5.2.3. O Serviço de Finalização

Uma inovação proporcionada pela realização deste trabalho é a execução de passos encapsulados como serviços *Web*, detalhada na Seção 5.4. Uma vez que tais serviços são entidades autônomas, podem apresentar qualquer tipo de comportamento, inclusive o processamento de informações durante um grande intervalo de tempo. Não seria viável, então, que todas as operações de um componente e de uma aplicação cliente fossem bloqueadas até que houvesse alguma resposta de um certo serviço.

A solução para este problema foi a implementação de EndingBean, um SLSB exposto como o serviço Ending e voltado exclusivamente a chamadas de outros serviços *Web*. Finalizada a execução da lógica correspondente a um passo, um serviço *Web* deve estabelecer contato com Ending para invocar a função `endAction()`, `endRule()` ou `endFunction()`, de acordo com o tipo do passo em questão, e informar os resultados da execução. Não há, portanto, a necessidade de aguardar respostas de serviços: quando há uma resposta, esta é imediatamente notificada.

### 5.2.4. Identificação e Comunicação sobre Erros

A ferramenta *NavigationPlanTool* informa a ocorrência de erros de execução por meio do lançamento de exceções em Java. Cada exceção é um objeto e pode ter seu conteúdo representado na forma de uma *string* (seqüência de caracteres), o que facilita a conversão em XML para o envio a outras aplicações. No entanto, há várias funções expostas pelos serviços *Web* que devolvem tipos diferentes de informação, como números inteiros. A inviabilidade na devolução de *strings* por todas as funções levou à implementação do SLSB ErrorsBean e de dois *entities* de JPA.

O componente ErrorsBean, exposto como o serviço Errors, é o único a disponibilizar uma interface local, além da SEI, para receber chamadas diretas de outros componentes no servidor. Cada chamada local corresponde a um registro de ocorrência de erro, quando algum componente reporta problemas encontrados durante suas operações. Cada chamada remota, por sua vez, é uma requisição, enviada por uma aplicação cliente, de informações sobre um erro.

| Interface local | *Service endpoint interface* (SEI) |
|---|---|
| `reportEndingError()` | `getEndingErrorDescription()` |
| `reportExecutionError()` | `getExecutionErrorDescription()` |

Tabela 5.2: Funções presentes em cada interface do componente ErrorsBean

Os erros foram classificados em dois tipos: os de finalização e os de execução. O primeiro tipo corresponde às exceções encontradas pelo componente EndingBean quando ocorre algo inesperado na finalização de um passo disponível como serviço *Web*. O segundo tipo refere-se a todas as outras exceções, como a lançada se uma aplicação cliente tentar obter informações sobre um processo inexistente. Cada tipo de erro é representado por um *entity* de JPA para o armazenamento e a recuperação de informações em um banco de dados relacional.

Os dois *entities* implementados, EndingError e ExecutionError, são acessíveis apenas localmente por ErrorsBean e configurados com anotações Java e o arquivo "persistence.xml". A implantação de cada um deles em um servidor de aplicação resulta na criação de uma nova relação no banco de dados, além das necessárias para o funcionamento da *NavigationPlanTool*.

Um cliente notificado sobre algum erro de execução pode utilizar o código referente ao erro para solicitar a ErrorsBean que encontre o respectivo *entity* e extraia sua descrição. Não há notificação sobre erros de finalização, portanto, se um passo associado a um serviço *Web* não for finalizado dentro do intervalo esperado de tempo, um cliente deve utilizar o identificador da instância de processo e o identificador do passo no *log* da instância para verificar se algum erro foi registrado no banco.

### 5.2.5. O Arquivo "ds.properties" e a Classe NPWSPropertiesReader

Um servidor de aplicação é geralmente apto a aceitar interações com qualquer sistema gerenciador de banco de dados (SGBD), de modo que o administrador do servidor é o responsável por escolher um SGBD de sua preferência. No entanto, para estabelecer conexões com um banco de dados, os SLSB desenvolvidos procuram seu nome no JNDI. Com o objetivo de manter a flexibilidade oferecida pelos servidores, o nome no JNDI está escrito em um arquivo à parte, chamado "ds.properties", e pode ser facilmente alterado pelo administrador.

Sabe-se, entretanto, que a leitura de arquivos é uma operação custosa, logo é interessante que o número de interações com "ds.properties" seja o menor possível. A classe NPWSPropertiesReader, acessada localmente pelos componentes, é a responsável por limitar esse número. Esta classe utiliza o padrão *Singleton* [16] para impedir que seja instanciada mais de uma vez. Durante a instanciação, o arquivo de propriedades é lido e tem suas informações armazenadas na única instância, de modo a ser efetuada apenas uma leitura do arquivo e todas as consultas serem feitas por meio de chamadas à instância.

### 5.2.6. O Conjunto de Serviços

Os seis SLSB, suas interfaces, os dois *entities*, o arquivo "persistence.xml", o arquivo "ds.properties" e a classe NPWSPropertiesReader foram empacotados em um arquivo do tipo JAR (Java *Archive*), disponível gratuitamente em http://www.ime.usp.br/~chui/npws/.

O conjunto dos seis serviços *Web* resultantes da implantação do JAR em um servidor de aplicações foi nomeado NPWS (*Navigation Plan Web Services*).

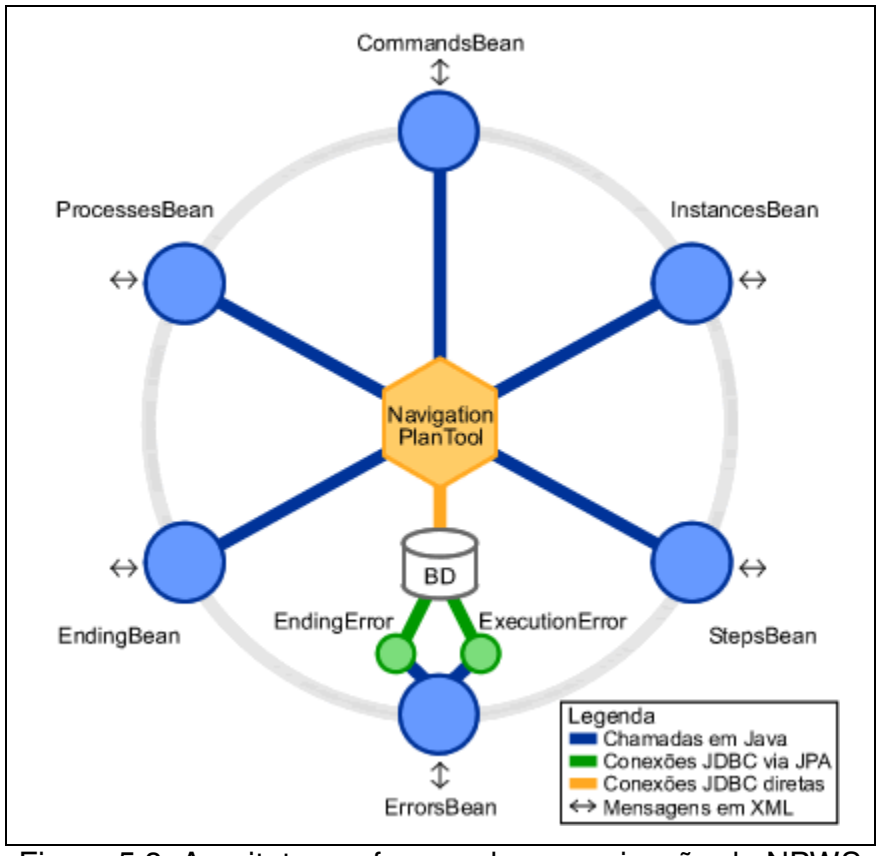


Figura 5.3: Arquitetura e formas de comunicação do NPWS

## 5.3. As Versões do NPWS

Duas versões do NPWS foram desenvolvidas para a realização do trabalho proposto, uma para o JBoss AS 4.0.5 e outra para versões mais recentes deste servidor de aplicação. A única diferença entre as duas implementações está nas estruturas de dados devolvidas às aplicações clientes e no processamento que envolve a criação dessas estruturas.

Durante os testes iniciais com a implementação, quando o servidor em uso era o JBoss AS 4.0.5, encontrou-se um problema no mapeamento de vetores de *strings* em XML para o formato correspondente em Java, o que impedia o recebimento dos dados por aplicações clientes nessa linguagem de programação. Nenhuma solução foi encontrada após um longo tempo, portanto decidiu-se que cada um desses vetores seria representado por uma grande *string*, resultante da concatenação das *strings* do vetor, porém separadas umas das outras pelo caractere '\t'. A responsabilidade por analisar os resultados e gerar os vetores correspondentes passou, então, para as aplicações clientes.

Quando surgiu a série 4.2 de versões estáveis do JBoss AS, o trabalho já havia sido concluído. Realizou-se um teste com o JBoss AS 4.2.1 para avaliar se o problema persistia e o mapeamento dos vetores funcionou corretamente. Optou-se, então, pela implementação de uma versão do NPWS que aproveitasse os novos recursos para produzir os vetores de *strings* e devolvê-los prontos às aplicações.

O motivo para tamanha diferença entre versões do JBoss AS deve-se ao objetivo dos desenvolvedores de preparar o lançamento da versão 5.0, bastante inovadora se comparada à 4.0.5. Em particular, a série 4.2 introduziu, como padrão, a versão 1.2.0 do JBossWS, componente do servidor responsável por serviços *Web*. Esta versão do componente inclui suporte a JAX-RPC e JAX-WS, enquanto a versão 1.0.4, padrão no JBoss AS 4.0.5, suporta apenas JAX-RPC.

Apesar da versão do NPWS para JAX-WS ser recomendada, a antiga não deixou de existir. A decisão foi motivada pela preocupação com usuários da série 4.0 do JBoss AS que não tenham condições de alterar as configurações do servidor ou migrar para uma versão mais recente, uma vez que qualquer mudança pode afetar seriamente o funcionamento de aplicações já implantadas. Neste caso, a versão para JAX-RPC é o único meio de interagir com a *NavigationPlanTool*.

## 5.4. Modos de Execução de Passos

O NPWS viabiliza a execução de passos encapsulados como serviços *Web*, porém nem todo passo precisa ser acessado desta forma. Por exemplo, uma aplicação cliente pode realizar o processamento local de informações ou mesmo requisitar a realização de uma atividade por um ser humano. Assim, é possível que passos sejam executados de forma local para que as aplicações informem ao NPWS os resultados da execução.

Quando uma aplicação cliente executa um passo localmente, a execução deve ser iniciada e finalizada antes de haver o contato com o NPWS. Quando os resultados estiverem prontos, a aplicação deve invocar a função `executeAction()`, `executeRule()` ou `executeFunction()`, todas expostas pelo serviço Steps, para que a *NavigationPlanTool* receba os resultados e atualize as informações no banco de dados.

O caso em que o passo é executado como um serviço *Web* envolve mais cuidado, uma vez que o NPWS se responsabiliza apenas por solicitar o início da execução. A aplicação cliente deve invocar uma das seguintes funções, também expostas pelo serviço Steps: `startActionWS()`, `startRuleWS()` ou `startFunctionWS()`. Assim que a execução do passo é iniciada, a aplicação retoma o controle de suas operações sem precisar esperar pela finalização, a qual é informada posteriormente pelo serviço *Web* ao serviço Ending.

### 5.4.1. A Detecção do Modo de Execução

A detecção do modo de execução ocorre por meio da análise da chamada de execução de um passo, atribuída geralmente na criação do mesmo. Se a chamada contiver o prefixo “WS::”, o passo em questão é considerado encapsulado; se não contiver este prefixo, o passo é considerado como executável localmente.

O restante da chamada de um passo encapsulado deve obrigatoriamente apresentar três informações sobre o respectivo serviço *Web*, nesta ordem: endereço do arquivo WSDL, *namespace* em que está localizado o serviço e nome do serviço. Cada informação deve estar separada das demais pelo caractere “!”. O NPWS valida o formato da chamada e, se estiver correto, promove uma tentativa de comunicação.

Se não possuir o prefixo “WS::”, a chamada de execução é isenta de validação, pois a responsabilidade por toda a execução do passo é da aplicação cliente.

### 5.4.2. Restrições na Criação de Serviços *Web*

Qualquer serviço *Web* associado à execução de um passo deve obedecer a duas importantes restrições:

- O serviço precisa expor uma função `execute()`, que não recebe parâmetros e devolve um valor inteiro. Esta função, invocada pelo NPWS, deve iniciar a execução do passo e devolver imediatamente o valor 0 para notificar o sucesso na operação;
- Assim que a execução terminar, o serviço deve invocar uma das funções do serviço Ending, descritas na Seção 5.2.3, para transmitir os resultados ao NPWS.

Serviços de amostra foram implementados para testes com o NPWS visando este tipo de execução. Informações sobre estes serviços estão na próxima seção.

## 5.5. Artefatos Opcionais

Implementaram-se alguns artefatos para testes com o NPWS: três serviços *Web* de amostra e três aplicações de demonstração. Todos os arquivos estão disponíveis em http://www.ime.usp.br/~chui/npws/, porém são opcionais por não afetarem o funcionamento do NPWS nem o desenvolvimento de aplicações clientes.

### 5.5.1. Serviços *Web* de Amostra

Os três serviços *Web* foram desenvolvidos de forma similar ao NPWS, com a exposição de SLSB e em duas versões, uma para JAX-RPC e outra para JAX-WS.

SampleActionBean, SampleRuleBean e SampleFunctionBean são componentes que representam, respectivamente, uma ação, uma regra e uma função. Cada SLSB apresenta uma SEI, onde está exposta a função `execute()`, responsável por criar e iniciar uma *thread* e então

devolver o valor 0 como indício de sucesso. A *thread* realiza a execução da lógica correspondente ao passo e finalmente estabelece contato com o NPWS para informar os resultados da execução.

| Nome do componente | Nome do serviço *Web* | Tipo de Passo |
|---|---|---|
| SampleActionBean | SampleAction | Ação |
| SampleRuleBean | SampleRule | Regra |
| SampleFunctionBean | SampleFunction | Função |

Tabela 5.3: Informações sobre os serviços *Web* de amostra

A execução relacionada a cada SLSB consiste no seguinte: a *thread* criada aguarda cinco segundos e devolve resultados pré-definidos. A regra devolve sempre o valor 1, que significa *true* (verdadeiro), a função devolve o valor 3 e a ação não tem valor de retorno.

### 5.5.2. Aplicações de Demonstração

Três aplicações de demonstração foram implementadas com HTML (*HyperText Markup Language*) e a tecnologia *Java Servlet*. Operáveis manualmente por usuários, porém cada uma com um escopo diferente, as aplicações podem ser acessadas como quaisquer páginas dinâmicas encontradas na Internet, além de serem facilmente configuráveis para lidar com as diferentes versões do NPWS (de forma mutuamente exclusiva). A seguir está uma breve descrição das aplicações, bem como a imagem da interface gráfica de cada uma.

A primeira aplicação é voltada exclusivamente à execução de comandos NPDL e SQL, de forma a possibilitar, entre outras opções, a criação e a definição de passos e processos. O usuário digita os comandos em uma caixa de texto e solicita a execução dos mesmos para receber os resultados em outra caixa de texto.

A segunda aplicação é uma área de controle para processos definidos com a *NavigationPlanTool*, seja por meio do NPWS ou por aplicações no servidor que eventualmente realizem contato direto com a ferramenta. O usuário pode visualizar as listas e informações de todos os processos, passos e instâncias, bem como filtrar o conteúdo de cada lista de acordo com as opções oferecidas. Entre as funcionalidades desta aplicação, destaca-se a instanciação de processos.

A terceira aplicação, uma extensão da segunda, tem como aspecto mais importante a execução das instâncias de processos: o usuário é informado sobre os passos disponíveis de uma instância de processo, escolhe o que deseja executar e, após a execução, recebe as novas opções. Este ciclo é realizado até que a execução da instância termine. Sempre que um usuário seleciona um passo, esta aplicação também trata de informá-lo sobre o respectivo tipo de execução, para que o usuário esteja ciente sobre o que deve ocorrer se optar pela execução.

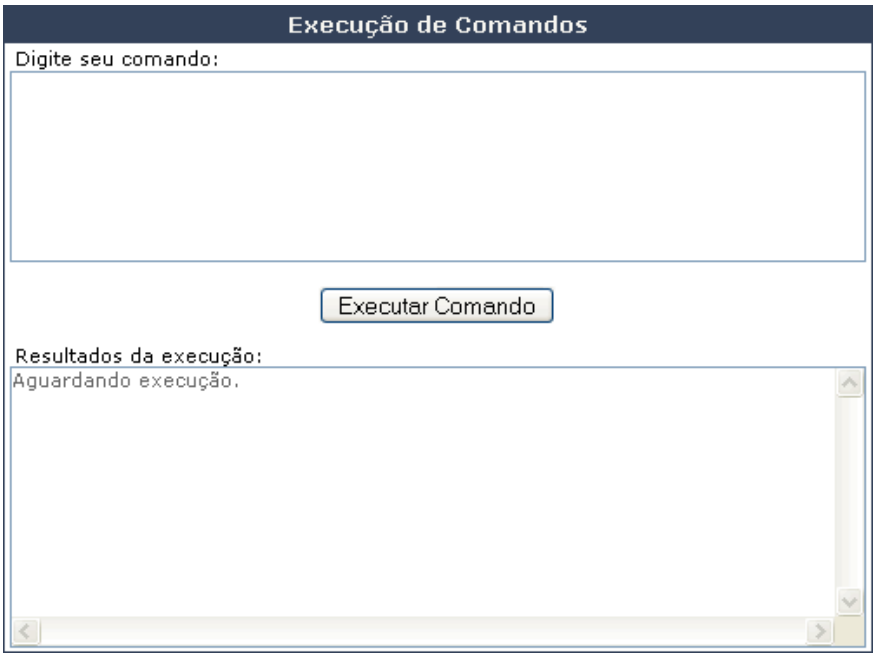



Figura 5.4: Interface gráfica da aplicação para a execução de comandos

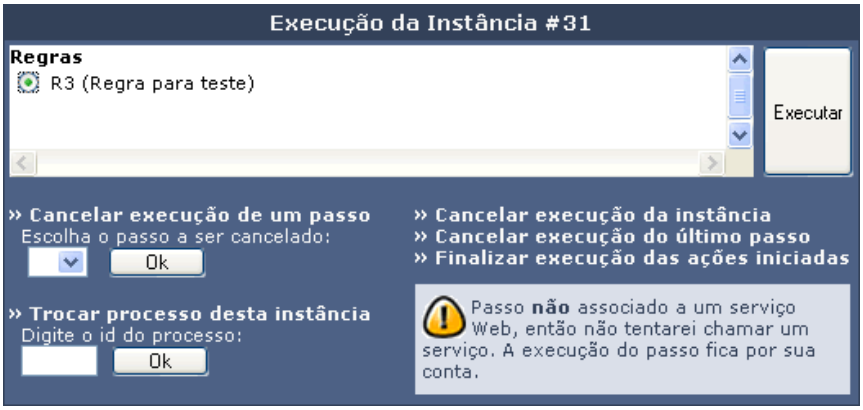



Figura 5.5: Interface gráfica da aplicação para a execução de instâncias de processos


Figura 5.6: Interface gráfica da aplicação para o controle de processos, instâncias e passos

## 5.6. Documentação

Dois manuais, sujeitos a modificações, foram escritos logo após o desenvolvimento do NPWS e dos artefatos opcionais. Quando um leitor encontra alguma dificuldade para compreender ou colocar em prática o que está presente em algum dos manuais, recomenda-se que entre em contato e exponha suas críticas. A intenção é tornar os manuais cada vez mais intuitivos para os futuros usuários.

O Manual de Configuração do Servidor [24] contém todas as informações para preparar o ambiente necessário para a utilização do NPWS e dos artefatos opcionais. Seu conteúdo é relevante tanto para administradores de servidores de aplicação quanto para usuários que nunca tiveram contato com servidores e têm interesse em vivenciar tal experiência.

O Manual de Uso [25] tem como público-alvo os desenvolvedores de aplicações que desejam utilizar as funcionalidades expostas pelo NPWS. Este documento descreve aspectos relacionados à compatibilidade com os serviços e à execução de passos, apresenta um exemplo simples de uso dos serviços e contém sugestões para a implementação de aplicações clientes. Uma seção do manual particularmente importante é a lista com informações detalhadas sobre todas as funções expostas pelos serviços desenvolvidos.

Usuários simpatizantes de Javadoc [20] ainda podem optar pela documentação gerada em páginas HTML e disponível para acesso *online*. Neste caso, as informações variam de acordo com a versão retratada do NPWS.

# Capítulo 6

# Conclusão

A integração de aplicações é um problema existente há bastante tempo e para o qual já foram propostas várias soluções. No entanto, toda solução, por mais sucesso que tenha obtido, sempre apresentou algum tipo de restrição. Os serviços *Web* são a mais recente tentativa de dar um passo além do que é possível com as formas convencionais de *middleware*.

O encapsulamento da ferramenta *NavigationPlanTool* com o NPWS cria uma camada de abstração que possibilita o acesso às funcionalidades da ferramenta de forma distribuída, por aplicações em qualquer linguagem de programação que aceite comunicação via mensagens em XML. As antigas restrições na integração de aplicações clientes com a ferramenta, portanto, deixaram de existir.

O NPWS, por permitir que cada passo em um processo de negócio tenha sua execução associada a um serviço *Web*, ainda viabiliza a colaboração entre serviços em um ambiente heterogêneo para que um objetivo de negócio possa ser atingido. Assim, proporciona maior aproveitamento do potencial oferecido pela NPDL e pela *NavigationPlanTool* no controle de processos de negócio em um modelo relacional de dados.

# Capítulo 7

# Avaliação Subjetiva

Este capítulo da monografia é pessoal e nele descreverei minha experiência ao realizar o trabalho de Iniciação Científica durante um ano.

## 7.1. Desafios e Frustrações

Não faltaram desafios durante a realização do trabalho. Durante a preparação, estudar e compreender centenas de páginas sobre EJB e seus conceitos associados foi cansativo, porém gratificante. O maior desafio foi a superação de todos os problemas encontrados durante a implementação, a qual deveria estar concluída até junho de 2007, prazo máximo para a submissão de trabalhos ao XXII Simpósio Brasileiro de Banco de Dados (SBBD 2007).

O resumo submetido teve como autores os doutorandos Kelly Rosa Braghetto e Marcos Eduardo Bolelli Broinizi, o Prof. Dr. João Eduardo Ferreira e eu, todos do IME-USP. Na ocasião, estavam implementadas apenas as primeiras versões dos artefatos opcionais e do NPWS. Infelizmente o resumo não foi aceito, o que impediu a apresentação do trabalho na Sessão de Demos do simpósio.

### 7.1.1. Problemas de Compatibilidade

O primeiro problema de compatibilidade surgiu no início do desenvolvimento de exemplos com EJB 2.1 e envolveu o Eclipse, ambiente escolhido para a programação, e o JBoss AS. Visando a integração de ambos, instalei o *plug-in* JBoss IDE (JBoss *Integrated Development Environment*) para Eclipse, porém a última versão do *plug-in* havia sido projetada para Eclipse 3.1 e apresentava conflitos com versões mais recentes, o que impedia a integração. A única solução encontrada foi obter e utilizar o Eclipse 3.1.

O segundo problema de compatibilidade surgiu durante os testes com um serviço *Web* simples que eu havia implementado. Parte das bibliotecas do JBoss AS 4.0.5 entravam em conflito com outras do Java 6, de modo a inviabilizar a comunicação com o serviço. Vários usuários já haviam se deparado com o mesmo problema, inclusive estudantes do IME-USP, portanto não foi difícil descobrir que o uso do Java 5 era a solução.

### 7.1.2. O Problema no Mapeamento de Dados

O maior obstáculo para a realização do trabalho dentro do prazo estipulado foi, sem dúvida, o problema no mapeamento de vetores de *strings* em XML para Java. Os testes mostravam que apenas este tipo complexo gerava erros, pois vetores de inteiros longos eram corretamente mapeados.

O projeto entrou em estado de espera por cerca de dois meses. Tentei solucionar o problema sozinho, pesquisei o que poderia estar ocorrendo, mas não havia progresso. Então me registrei como membro do fórum de discussões da comunidade JBoss, onde encontrei usuários com o mesmo problema e recebi algumas sugestões que não surtiram o efeito esperado. Após pedir auxílio a pessoas com mais experiência no assunto e perceber que não sabiam o que fazer, eu decidi alterar a estrutura de dados para *strings* concatenadas, o que sabia que funcionaria e viabilizaria a continuação do trabalho.

Este problema me incomodou tanto que, mesmo com a solução improvisada, esperei pelas novas versões do JBoss AS para que pudesse testar novamente o mapeamento. Como imaginava, o mapeamento funcionou com as novas versões, então optei pelo desenvolvimento das novas versões do NPWS e dos serviços de amostra e adaptei as aplicações de demonstração para serem compatíveis com as duas versões da implementação.

## 7.2. Alegrias e Satisfações

Dois foram os principais momentos de alegria relacionados à realização deste trabalho: o primeiro ocorreu quando o NPWS foi concluído, depois de todos os problemas superados; o segundo, quando cursei Tópicos de Sistemas de Computação (MAC0434), no segundo semestre

de 2007. A disciplina, voltada a sistemas de *middleware*, me fez perceber o quanto eu havia aprendido com os estudos para a realização do trabalho.

Fiquei feliz, também, pelo NPWS ter sido utilizado de forma distribuída por vários alunos da disciplina de Laboratório e Modelagem de Banco de Dados (MAC0439-5861), lecionada pelo Prof. Dr. João Eduardo Ferreira durante o segundo semestre de 2007.

Pôsteres sobre o NPWS foram apresentados em dois simpósios, o III Simpósio de Iniciação Científica e Pós-Graduação do IME-USP e o XV Simpósio Internacional de Iniciação Científica da Universidade de São Paulo, ambos em novembro de 2007. Em particular, o segundo simpósio me deixou bastante satisfeito por eu ter encontrado, na sessão de computação, outros estudantes que realizaram trabalhos com serviços *Web* e demonstraram grande interesse no meu trabalho.

## 7.3. Disciplinas Relevantes

As disciplinas cursadas e consideradas por mim como relevantes para a realização deste trabalho estão a seguir, bem como a razão pela qual cada uma consta na lista.

### 7.3.1. MAC0110 - Introdução à Computação

Esta foi a disciplina em que tive realmente meus primeiros contatos com linguagens de programação, pois ingressei com apenas algum conhecimento sobre programação de sites. A linguagem adotada para a disciplina foi Java e, portanto, houve a introdução de noções sobre orientação a objetos, as quais seriam mais bem compreendidas posteriormente.

### 7.3.2. MAC0122 - Princípios de Desenvolvimento de Algoritmos

Imagino que esta seja a disciplina mais importante da graduação para vários alunos, não só para mim, por introduzir conceitos básicos sobre algoritmos. Aprendi a utilizar arquivos como entrada e saída de dados, as estruturas de dados fundamentais, os tipos de alocação de memória, a identificação e a correção de falhas em programas, etc. Enfim, foi a base para as demais disciplinas do curso e para qualquer implementação que eu tenha realizado.

### 7.3.3. MAC0242 - Laboratório de Programação II

Esta disciplina tem particular importância, não só por eu ter voltado a programar em Java e compreendido melhor a orientação a objetos, mas pelo fato de um exercício-programa meu e de André Shoji Asato ter agradado ao Prof. Dr. João Eduardo Ferreira, que lecionava a disciplina na ocasião. O professor conversou conosco sobre Iniciação Científica e, dentre as propostas, escolhi a que resultou neste trabalho.

### 7.3.4. MAC0426 - Sistemas de Bancos de Dados

Nesta disciplina são introduzidos vários conceitos sobre bancos de dados e, em particular, sobre bancos de dados relacionais. O modelo relacional de dados é essencial para a *NavigationPlanTool* e a NPDL, as quais são fundamentais para o meu trabalho. Portanto esta disciplina também foi relevante para a sua realização.

### 7.3.5. MAC0342 - Laboratório de Programação Extrema

Adquiri nesta disciplina o hábito de refatorar o código sempre que eu estiver descontente com seu estado, noção empregada na implementação realizada. Além disso, como utilizamos Java para o projeto na disciplina, conheci detalhes da linguagem úteis para o trabalho, como a configuração de uma aplicação por meio de um arquivo de propriedades.

### 7.3.6. MAC0441 - Programação Orientada a Objetos

Conheci os padrões de *design* (*design patterns*), entre eles o *Singleton*, importante para a eficiência do que implementei por viabilizar que os arquivos de propriedades sejam lidos apenas uma vez na configuração de aplicações. Se não fosse por este padrão, talvez eu encontrasse algum problema com o excesso de instâncias de leitores de arquivos.

### 7.3.7. MAC0438 - Programação Concorrente

Como optei por realizar configurações utilizando instâncias únicas, surgiram preocupações com o acesso concorrente, que poderia resultar na criação simultânea de duas instâncias ou na consulta de informações quando a instância ainda não estivesse completa. A solução foi retirada de uma das aulas desta disciplina e nunca apresentou falhas durante os testes.

### 7.3.8. MAC5861 - Modelagem de Banco de Dados

Esta disciplina foi cursada no último semestre da graduação, porém optei por cursá-la como uma disciplina da pós-graduação. Durante a preparação do ambiente para que os alunos pudessem utilizar o NPWS, identifiquei e corrigi erros pontuais nas aplicações de demonstração.

## 7.4. Expectativas e Planos para o Futuro

Considero muito interessantes os conhecimentos que adquiri e apliquei durante a realização deste trabalho. Em particular, gostei bastante de lidar com EJB e serviços *Web* e de conhecer a representação e o controle de *workflows* em bancos de dados relacionais por meio da NPDL e da *NavigationPlanTool*.

É provável que eu continue o desenvolvimento do trabalho para realizar alguns aprimoramentos. A possibilidade de implementação do NPWS como *MBeans* [18][21][22] vem sendo estudada, o que resultaria na disponibilidade dos serviços para quaisquer outros no JBoss AS, com transparência similar aos serviços de persistência e controle de transações. Também tenho interesse em criar uma área destinada ao trabalho em meu site, pois atualmente os arquivos estão soltos em diretórios, sem qualquer página relacionada.

Pretendo ingressar no programa de mestrado do IME-USP logo após o término da graduação. Bancos de dados e sistemas distribuídos são áreas que me atraem bastante, com destaque para a primeira. Para atuar em qualquer uma das duas áreas, imagino que seja interessante conhecer as novas tendências e os trabalhos que vêm sendo desenvolvidos nos últimos anos por quem atua na área.

# Referências Bibliográficas


[1] AALST, W. M. P. van der; HOFSTEDE, A. H. M. ter; KIEPUSZEWSKI, B.; BARROS, A. P. Workflow Patterns. Distributed and Parallel Databases, v. 14, n. 1, p. 5-51, jul. 2003.

[2] ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. Web Services Concepts, Architectures and Applications. Springer-Verlag, 2004. 398 p.

[3] BAETEN, J. C. M.; BASTEN, T. Partial-Order Process Algebra (and its Relation to Petri-Nets). In: BERGSTRA, J. A.; PONSE, A.; SMOLKA, S. A. Handbook of process algebra. Amsterdã: Elsevier Science Inc., 2001. p. 769-872.

[4] BERGSTRA, J. A.; PONSE, A.; SMOLKA, S. A. Handbook of process algebra. Amsterdã: Elsevier Science Inc., 2001. 1356 p.

[5] BRAGHETTO, K. R.; FERREIRA, J. E.; PU, C. Using Control-Flow Patterns for Specifying Business Processes in Cooperative Environments. In Proceedings of the 2007 ACM Symposium on Applied Computing, Seoul, Korea, March 11-15, 2007, pages 1234–1241. ACM, 2007.

[6] BRAGHETTO, K. R. Padrões de Fluxos de Processos em Banco de Dados Relacionais. Dissertação de Mestrado, Instituto de Matemática e Estatística - Universidade de São Paulo, 2006.

[7] BRAGHETTO, K. R.; TAKAI, O. K.; FERREIRA, J. E.; PU, C. Controlling Processes in Relational Data Model with NPDL and NavigationPlanTool. 2006. 15 p. (Relatório técnico, Departamento de Ciência da Computação, Universidade de São Paulo) Disponível em: <http://www.ime.usp.br/~jef/kelly/NPDL.pdf>. Acesso em: 24 nov 2007.

[8] CERAMI, Ethan. Web Services Essentials. O'Reilly, 2002. 304 p.

[9] EJB. Especificação disponível em: <http://java.sun.com/products/ejb/>. Acesso em: 24 nov 2007.

[10] ELMASRI, E. R.; NAVATHE, S. Fundamentals of Database Systems. 3.ed. Addison-Wesley, 2001. 1000 p.

[11] FERREIRA, J. E.; FINGER, M. Controle de Concorrência e Distribuição de Dados: Teoria Clássica, suas Limitações e Extensões Modernas. In: Escola de Computação, 12., 2000, São Paulo. Livro-texto… São Paulo: Instituto de Matemática e Estatística, USP, 2000. p. 188.

[12] FERREIRA, J. E.; TAKAI, O. K.; BRAGHETTO, K. R.; PU, C. Large Scale Order Processing through Navigation Plan Concept. In: IEEE International Conference on Services Computing (SCC 2006), 2006, Chicago, p. 297-300.

[13] FERREIRA, J. E.; TAKAI, O. K.; PU, C. Integration of Business Processes with Autonomous Information Systems: A Case Study in Government Services. In: International IEEE Conference on E-Commerce Technology, 7., 2005, Munique, p. 471-478.

[14] FERREIRA, J. E.; TAKAI, O. K.; PU, C. Integration of Collaborative Information System in Internet Applications using RiverFish Architecture. Aceito para International Conference on Collaborative Computing: Networking, Applications and Work Sharing, 1., 2005, San Jose.

[15] FOKKINK, W. J. Introduction to Process Algebra: Texts in Theoretical Computer Science. Berlin: Springer-Verlag New York, Inc., 2000. 163 p.

[16] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. 416 p.

[17] GLABBEEK, R. J. van. The linear-time – branching time spectrum I. The semantics of concrete, sequential processes. In: BERGSTRA, J. A., PONSE, A., SMOLKA, S. A. Handbook of process algebra. Amsterdã: Elsevier Science Inc., 2001. p. 3-99.

[18] HALLOWAY, S. D. Component Development for the Java Platform. Addison-Wesley, 2002. 334 p.

[19] JAVA 2 Platform Standard Edition - J2SE 5.0. Disponível em: <http://java.sun.com/j2se/1.5.0>. Acesso em: 24 nov 2007.

[20] JAVADOC Tool. Disponível em: <http://java.sun.com/j2se/javadoc>. Acesso em: 24 nov 2007.

[21] LINDFORS, J.; FLEURY, M. JMX: Managing J2EE with Java Management Extensions. Sams Publishing, 2002. 394 p.

[22] MARINESCU, F. EJB Design Patterns - Advanced Patterns, Processes and Idioms. Wiley, 2002. 288 p.

[23] MILNER, R. A Calculus of Communicating Systems. Secaucus: Springer-Verlag New York, Inc., 1982. 260 p.

[24] NAVIGATION Plan Web Services – Manual de Configuração do Servidor. Disponível em: <http://www.ime.usp.br/~chui/npws/manuais/manual_servidor.pdf>. Acesso em: 24 nov 2007.

[25] NAVIGATION Plan Web Services – Manual de Uso. Disponível em: <http://www.ime.usp.br/~chui/npws/manuais/manual_cliente.pdf>. Acesso em: 24 nov 2007.

[26] ROMAN, E.; SRIGANESH, R. P.; BROSE, G. Mastering Enterprise JavaBeans. 3.ed. Wiley, 2004. 839 p.

[27] SZYPERSKI, C.; GRUNTZ, D.; MURER, S. Component Software: Beyond Object-Oriented Programming. 2.ed. Addison-Wesley/ACM Press, 2002. 589 p.

[28] TANENBAUM, A. S.; STEEN, M. van. Distributed Systems: Principles and Paradigms. Amsterdã: Prentice Hall, 2002. 803 p.

[29] TAPPER, J.; TALBOT, J.; HAFFNER, R. Object-Oriented Programming with ActionScript 2.0. New Riders, 2004. 504 p.

[30] WORKFLOW Management Coalition, Terminology & Glossary. Winchester: Workflow Management Coalition, 1999. 65 p. (Relatório técnico, WFMC-TC-1011) Disponível em: <http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf>. Acesso em: 24 nov 2007.