FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

KLEBER MORO SAMPAIO

ACESSO A BANCO DE DADOS EM DISPOSITIVOS MÓVEIS UTILIZANDO J2ME

MARÍLIA
2006

KLEBER MORO SAMPAIO

ACESSO A BANCO DE DADOS EM DISPOSITIVOS MÓVEIS UTILIZANDO J2ME

Monografia apresentada ao Curso de Ciência da Computação, do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Márcio Eduardo Delamaro

MARÍLIA
2006

SAMPAIO, Kleber Moro.
Acesso a Banco de Dados em Dispositivos Móveis Utilizando J2ME / Kleber Moro Sampaio; orientador: Prof. Dr. Márcio Eduardo Delamaro. Marília, SP: [s.n.], 2006.
65 f.

Monografia (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. Dispositivos móveis 2. J2ME 3. Java

CDD: 005.1151

KLEBER MORO SAMPAIO

ACESSO A BANCO DE DADOS EM DISPOSITIVOS MÓVEIS UTILIZANDO J2ME

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação da UNIVEM / F.E.E.S.R., para a obtenção do Título de Bacharel em Ciência da Computação. Área de Concentração: Engenharia de Software.

Resultado:_____

ORIENTADOR: Prof. Dr. Marcio Eduardo Delamaro

1º EXAMINADOR: _____

2º EXAMINADOR: _____

Marília, 04 de Dezembro de 2006.

Aos meus pais,
que sempre aconselharam e incentivaram minhas decisões.

AGRADECIMENTOS

Primeiramente aos meus pais, Angélica e Edvaldo, ao meu irmão George e à minha namorada Mônica, pelo apoio e compreensão nos momentos difíceis.

Ao professor Márcio Eduardo Delamaro pela orientação e encaminhamento dado a este trabalho.

A todos os professores que contribuíram à minha formação pessoal e profissional.

Ao amigo Everthon, que contribui diretamente na minha formação pessoal e aos amigos que fiz na faculdade, cuja ajuda recíproca foi importante para atingirmos o último ano do curso.

“Educai as crianças e não será necessário punir os homens”.
Pitágoras

SAMPAIO, Kleber Moro. **Acesso a Banco de Dados em Dispositivos Móveis Utilizando J2ME.** 2006. 65 f . Monografia - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

O acesso a banco de dados remotos utilizando uma aplicação J2ME implica na utilização de classes e métodos para prover comunicação com um banco de dados e no desenvolvimento de meios para tratar os dados que serão recebidos do banco. Estas funcionalidades devem ser implementadas de maneira direta pelo programador. Este trabalho reúne em um pacote, as classes e métodos que são necessárias neste tipo de aplicação, escondendo a complexidade que o programador da aplicação J2ME precisaria implementar. Para possibilitar que esta aplicação acesse um SGBD foi desenvolvida uma aplicação servidora em Java que processa os pedidos da aplicação cliente e devolve os resultados.

Palavras-chave: dispositivos móveis, Java, J2ME, banco de dados.

SAMPAIO, Kleber Moro. **Acesso a Banco de Dados em Dispositivos Móveis Utilizando J2ME.** 2006. 65 f. Monografia - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

Database access using a J2ME application involves the use of classes and methods to provide communication with a database and the development of ways to treat the data received from this database. These functionalities must be implemented in a direct way by the programmer. This work joins the classes and methods necessary to this kind of application in a package, hiding the complexity that the J2ME application programmer should implement. To allow this application's access to a DBMS, a server application was developed to process the client application's requests and returns the results.

Keywords: mobile devices, Java, J2ME, database.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API: Application Programming Interface (Interface de programação de aplicação)

CDC: Connected Device Configuration (Configuração de dispositivo conectado)

CLDC: Connected Limited Device Configuration (Configuração de dispositivo limitado conectado)

J2EE: Java 2 Enterprise Edition (Edição empresarial Java 2)

J2ME: Java 2 Micro Edition (Edição micro Java 2)

J2SE: Java 2 Standard Edition (Edição padrão Java 2)

JAD: Java Application Descriptor (Descritor de aplicação Java)

JAR: Java Archive (Arquivo Java)

JCP: Java Community Process (Comunidade de processo Java)

JDBC: Java Database Connectivity (Conectividade Java de banco de dados)

JDK: Java Development Kit (kit de desenvolvimento Java)

JSR: Java Specification Request (Requerimento de especificação Java)

JVM: Java Virtual Machine (Máquina Virtual Java)

KVM: KJava Virtual Machine (Máquina Virtual KJava)

ODBC: Open Database Connectivity (Conectividade aberta à banco de dados)

OEM: Original Equipment Manufacturer (Fabricante original do equipamento).

PDA: Personal Digital Assistant (Auxiliar Digital Pessoal)

RMI: Remote Method Invocation (Invocação de Método Remoto)

SGBD: Sistema gerenciador de banco de dados

SQL: Structured Query Language (Linguagem estruturada de instrução)

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

O uso de computadores tem facilitado e melhorado a vida do homem em muitas áreas. Os dispositivos móveis têm agregado cada vez mais funcionalidades, e como não poderia deixar de acontecer, o acesso a informações armazenadas em banco de dados se faz necessário.

Este trabalho consiste em possibilitar esse tipo de acesso por meio de dispositivos móveis utilizando J2ME e comandos SQL. A API J2ME não permite que esse acesso seja feito de maneira direta (SUN, 2006), por isso foram desenvolvidos métodos que executem comandos de banco de dados em um dispositivo móvel, disponibilizando-os em uma biblioteca, com a ajuda de uma aplicação Java que funcionará como um servidor que controlará esse acesso.

Essas operações ocorrem de maneira transparente ao usuário, ou seja, para o usuário da aplicação do dispositivo móvel, o acesso ao banco de dados parecerá ocorrer de maneira direta. Uma exemplificação pode ser vista na Figura 1.

Para o programador desenvolver soluções que necessitam de acesso à banco de dados em dispositivos móveis, torna-se necessário saber que é preciso controlar a comunicação entre a aplicação do usuário e uma aplicação servidora que é a responsável pela manipulação do banco de dados.

Por isso, o desenvolvimento de classes e métodos que abstraiam essa dificuldade inicial pode tornar essa programação mais prática e objetiva, fazendo com que ela fique parecida com o uso da aplicação, podendo ser ilustrada pela Figura 1 também.

O objetivo deste trabalho é desenvolver essas classes e métodos, e está dividido da seguinte forma: o Capítulo 2 trata um pouco sobre a história e alguns conceitos da linguagem de programação Java, que é a linguagem utilizada para o desenvolvimento dos programas deste trabalho e cita o uso da API JDBC (*Java Database Connectivity*) e como ela interage

com um banco de dados, explica como uma aplicação remota pode se conectar a uma outra aplicação através de *sockets* e cita o uso de *threads*; a plataforma J2ME é apresentada no Capítulo 3; o Capítulo 4 apresenta as classes em forma de diagrama e a implementação da biblioteca desenvolvida neste trabalho; o Capítulo 5 mostra o funcionamento da aplicação servidora usada para acessar o banco de dados e o Capítulo 6 ilustra algumas características da aplicação cliente desenvolvida para servir de exemplo de uso da biblioteca desenvolvida por este trabalho.

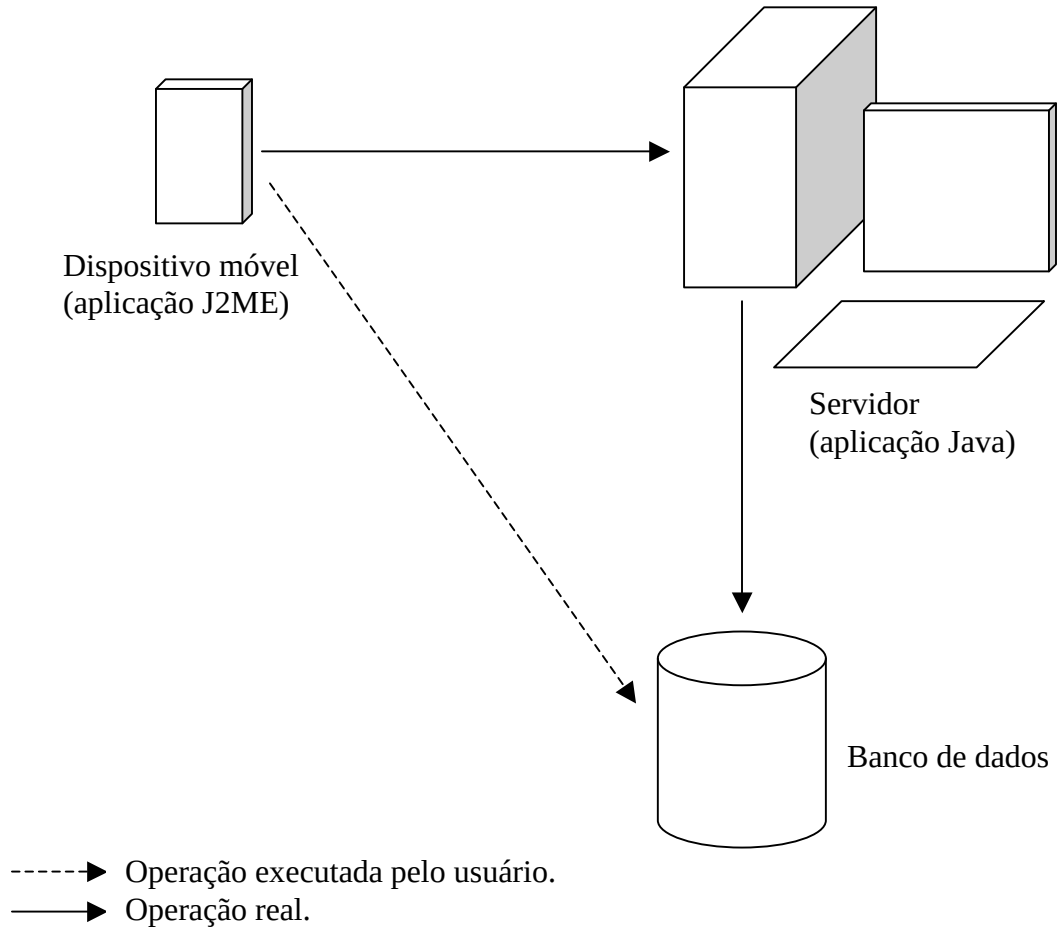

Figura 1- Trajetória de uma requisição

CAPÍTULO 2 – TÉCNICAS E FERRAMENTAS UTILIZADAS

Este capítulo apresenta alguns conceitos importantes sobre a linguagem de programação Java, que permitem que a aplicação desenvolvida seja mais eficaz e segura em relação a linguagens de programação orientadas a procedimentos. Ele ainda apresenta algumas características da API JDBC, que consiste em classes e métodos Java que são implementados pelos desenvolvedores de *drivers* para sistemas gerenciadores de banco de dados (SGBD) e possibilita que qualquer banco que possua *driver* para o JDBC seja acessado de maneira padronizada utilizando a Linguagem Java. Este capítulo ainda apresenta alguns conceitos sobre sobre a comunicação que pode ser feita entre aplicações Java, utilizando *sockets*, e sobre a criação de novas linhas de execução dentro de um mesmo processo utilizando *threads*.

Todos esses conceitos são importantes para o desenvolvimento do trabalho proposto nesta monografia pois a linguagem de programação Java os disponibiliza em forma de classes e métodos e torna seu uso mais acessível.

2.1. Java

Java nasceu da idéia de se criar uma linguagem de programação leve o suficiente para ser colocada em dispositivos de acesso a TV a cabo em 1991 por um grupo de desenvolvimento da Sun. No princípio, foi colocada em comercialização, mas o mercado para estes dispositivos não estava tão bom assim (HORSTMANN e CORNEL, 2000).

Ao mesmo tempo, ela foi desenvolvida para ser usada independente de sistema operacional e de arquitetura. Em meados de 1995, um grupo de desenvolvedores da Sun apresentou um navegador escrito em Java capaz de executar código nele mesmo (HORSTMANN e CORNEL, 2000). Esse tipo de programa que é executado em navegadores

é chamado *applet*. Um problema no seu uso é que os arquivos que são executados precisam ser gravados no computador, havendo, assim, preocupação em desenvolver aplicações pequenas para trafegarem pela internet, mas numa intranet essa tecnologia pode ser usada sem preocupações devido a praticamente não existir problemas com banda (HORSTMANN e CORNEL, 2000).

Soluções desenvolvidas em Java são compiladas e o resultado dessa operação é a geração de um ou mais arquivos chamados *bytecodes* que são interpretados por uma JVM (*Java Virtual Machine*) para que sua execução aconteça. Para cada sistema operacional existe uma JVM específica que faz a ligação de um programa Java ao próprio sistema.

Java é uma linguagem de programação orientada a objetos, o que ajuda o desenvolvimento de melhores soluções de software; sua sintaxe é parecida com a de linguagens como C e C++, o que facilita o aprendizado para programadores dessas linguagens, possui mecanismos de segurança de execução e acesso de dados, pode ser usada em ambientes distribuídos e em rede, é portável, a programação de conexões de rede é mais simples que de outras linguagens, assim como a programação concorrente (*multithreading*) (HORSTMANN e CORNEL, 2000).

Em relação às outras linguagens de programação, Java: (1) eliminou a manipulação e alocação direta de memória, sendo que seu coletor de lixo se encarrega do seu gerenciamento e coleta de dados que não são mais necessários aos programas; (2) não requer índices de inserção e remoção em *arrays*, evitando acesso, manipulação ou escrita de dados sobre outros em um vetor (HORSTMANN e CORNEL, 2000).

A API (*Application Programming Interface*) é uma biblioteca (conjunto de classes e métodos) que permite que recursos do computador sejam usados de forma padrão. Nela pode-se encontrar classes e métodos que possibilitam implementar acesso a arquivos, a banco de dados, interfaces gráficas, acesso à rede, segurança, e muitas outras funcionalidades. Para

possibilitar maior interação com os diversos tipos de hardware existentes, foram desenvolvidas quatro plataformas de aplicação do Java: J2EE para aplicações com servidores de muita memória e processamento, J2SE para aplicações em computadores pessoais, J2ME para dispositivos com memória e processamento limitados e JavaCard que é direcionado para uso em *SmartCards* (PAULA, 2006). De acordo com Muchow (2001, p. 2) essa classificação se resume a apenas três: J2EE, J2SE e J2ME, conforme a Figura 2.

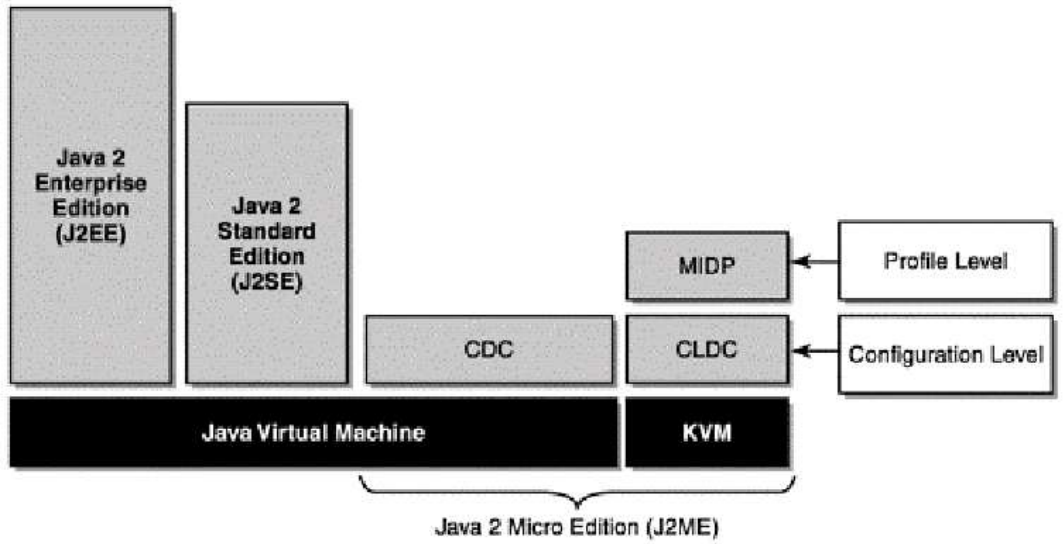

Figura 2 – Plataformas de desenvolvimento Java. Fonte: (HORSTMANN e CORNEL, 2000).

Classe é a base de qualquer programa escrito em Java. Ela contém métodos e atributos que definem o comportamento dos objetos. Uma classe pode ser entendida como um molde padrão, a partir do qual um objeto é criado. Cada objeto pode ter comportamento diferente de um outro objeto, mesmo que eles sejam definidos pela mesma classe, pois no mesmo momento eles podem ter diferentes valores em seus atributos modificados pelos métodos do próprio objeto.

Essa característica ajuda no entendimento de uma propriedade chamada encapsulamento, que consiste em esconder os dados do usuário de um objeto, devendo ser conhecidos apenas os métodos que são necessários para efetuar mudanças no objeto, mas a maneira que eles fazem isso não precisa ser vista pelo usuário. Esta é uma propriedade que

também garante segurança na modificação de objetos e uma boa engenharia de software, pois os métodos só podem alterar os atributos do mesmo objeto em que estão inseridos (HORSTMANN e CORNEL, 2000 ; DEITEL e DEITEL, 2003).

Classes podem ser reutilizadas em novos programas. Elas podem ser oferecidas através de pacotes. Um pacote é composto por um ou mais diretórios que contêm uma ou mais classes. Para definir um pacote, é utilizada a palavra reservada *package* no início do código fonte Java, antes da primeira declaração de classe. Dessa forma também é o procedimento para utilizar um pacote, mas com a diferença de utilizar-se a palavra reservada *import*.

A maneira de prover pacotes por meio de diretórios é uma tentativa da própria Sun Microsystems de padronizar essa utilização. Ela recomenda o uso do domínio do desenvolvedor na Internet ao contrário, ou seja, se um pacote foi desenvolvido pela Sun, ele deve se encontrar em um diretório java, que está dentro de um diretório *sun*, que por sua vez se encontra dentro do diretório *com*, dessa maneira: *\com\sun\java*. O domínio da Sun na Internet é java.sun.com. De acordo com Deitel e Deitel (2003, p. 384), essa iniciativa é um esforço para fornecer nomes únicos para cada pacote.

Herança e polimorfismo são outras características muito importantes para a programação Java. Herança é o reúso de métodos e atributos de uma classe para definir uma nova classe. A classe que herda funcionalidades pode ser chamada de classe derivada, subclasse, ou classe filha, enquanto que a que é herdada pode ser chamada de classe básica, superclasse ou classe pai. A implementação da herança é feita colocando-se a palavra reservada *extends* depois do nome da nova classe e antes do nome da superclasse. Para referir-se a métodos e construtores da superclasse, usa-se a palavra reservada *super*. (HORSTMANN e CORNEL, 2000 ; DEITEL e DEITEL, 2003).

A herança é uma propriedade natural da Linguagem Java e faz parte de cada definição de classe (DEITEL e DEITEL, 2003). Se nenhuma superclasse for explicitamente

definida, a classe *Object* é assumida como tal, por padrão. Esta é a única classe sem nenhuma superclasse (JIA, 2000 ; DEITEL e DEITEL, 2003).

Cada classe não pode ter mais que uma superclasse, pois a Linguagem Java não suporta herança múltipla, mas é possível implementá-la por meio de interfaces (HORSTMANN e CORNEL, 2000 ; JIA, 2000). Interface é uma forma de escrever métodos sem implementá-los. Ela apenas diz o que um método faz, mas não como (HORSTMANN e CORNEL, 2000). Uma classe pode implementar nenhuma ou várias interfaces e deve providenciar a implementação dos seus métodos (JIA, 2000).

Isso quer dizer que uma classe que estende uma interface não herda nenhuma implementação da classe base porque interfaces não têm implementação, mas a interface estendida contém todas as características da interface base, que na verdade são apenas as descrições dos métodos (JIA, 2000). Ainda segundo Jia (2000, p.135), o relacionamento envolvendo interfaces pode ser considerado uma forma fraca de herança.

Polimorfismo é a qualidade do que se apresenta sob formas diversas (POLIMORFISMO). Em Java, essa propriedade é usada para oferecer um serviço de duas ou mais formas diferentes de ser executado. O usuário de um serviço desse tipo não precisa saber dessa característica, ele apenas pede para ser feito e o programa escolhe a melhor forma. De acordo com Horstmann e Cornel (2000, p. 200) a escolha dinâmica sobre qual método deve realizar determinada tarefa sobre um objeto é denominada ligação dinâmica.

Segundo Deitel (2003, p. 374), “[...] herança e polimorfismo são duas tecnologias fundamentais que permitem a verdadeira programação orientada a objetos”.

2.2 JDBC

JDBC é um conjunto de classes e métodos (API) que agem de maneira intermediária

sobre um driver ODBC, formando uma ponte através da qual é feita a comunicação entre uma aplicação Java e um SGBD (JEPSON, 1997).

A API ODBC (*Open Database Connectivity*) foi desenvolvida pela Microsoft antes que a JDBC e tem o mesmo propósito, mas para uma aplicação Java, o uso direto do ODBC não é apropriado pois: (1) foi desenvolvida em C e seu uso pode comprometer a segurança, robustez e a portabilidade; (2) é difícil de entender pois compreende o uso de vários níveis de acesso de dados e de ponteiros; (3) quando ODBC é usado, é necessária sua instalação manual em cada máquina cliente (FISHER, 2003).

O JDBC pode ser simplificado, apenas para seu melhor entendimento como uma tradução do ODBC para uma linguagem de alto nível orientada a objeto (FISHER, 2003).

O maior benefício do uso do JDBC é proporcionar independência de desenvolvedor de banco de dados e, assim como as aplicações desenvolvidas em Java, independência de plataforma, aproveitando que a SQL (*Structured Query Language*) possui muitos comandos padronizados que podem ser utilizados por vários bancos de dados diferentes (JEPSON, 1997). Além disso, pode aproveitar as características da própria Linguagem Java, usando-o num *applet*, por exemplo, na intranet de uma empresa, onde os funcionários podem estar utilizando diferentes sistemas operacionais e acessando diferentes bases de dados (FISHER, 2003).

Observando a API do JDBC, as classes que se encontram no pacote java.sql são todas interfaces e não são implementadas em nenhuma outra classe java. Isto acontece para ficar a cargo do desenvolvedor do driver de cada tipo de banco de dados a implementação dessas classes, possibilitando que o uso dessas classes em um programa Java aconteça de maneira genérica, ampliando a portabilidade do Java e a reutilização de código, além de evitar que um caos aconteça nos laboratórios da Sun, pois estes teriam que desenvolver os *drivers* para cada banco utilizado no Java (SILVEIRA, 2006).

Para desenvolver e/ou utilizar aplicações Java com acesso a banco de dados logo após o lançamento das primeiras versões do JDBC, era necessário instalar suas classes em separado do JDK. Esse pequeno problema foi solucionado com o surgimento do JDK 1.1, que passou a incorporar essas classes por padrão (JEPSON, 1997).

A API JDBC suporta acesso a banco de dados de formas: modelo de duas camadas e de três camadas.

No modelo de acesso de duas camadas a aplicação cliente se comunica diretamente com o banco de dados utilizando um *driver* JDBC como ilustra a Figura 3. O banco pode estar localizado em um outro computador pelo qual o cliente está conectado por meio de uma rede. Essa configuração é chamada de cliente/servidor.

No modelo de acesso de três camadas, ilustrado pela Figura 4, uma requisição do cliente é enviada para uma camada intermediária, e esta a envia ao banco de dados, sendo que o resultado quando é retornado para o cliente também passa por ela. Desta maneira pode-se fazer alguns controles, como verificar se os tipos de consulta ou atualização de dados são adequados a certo usuário (FISHER, 2003).

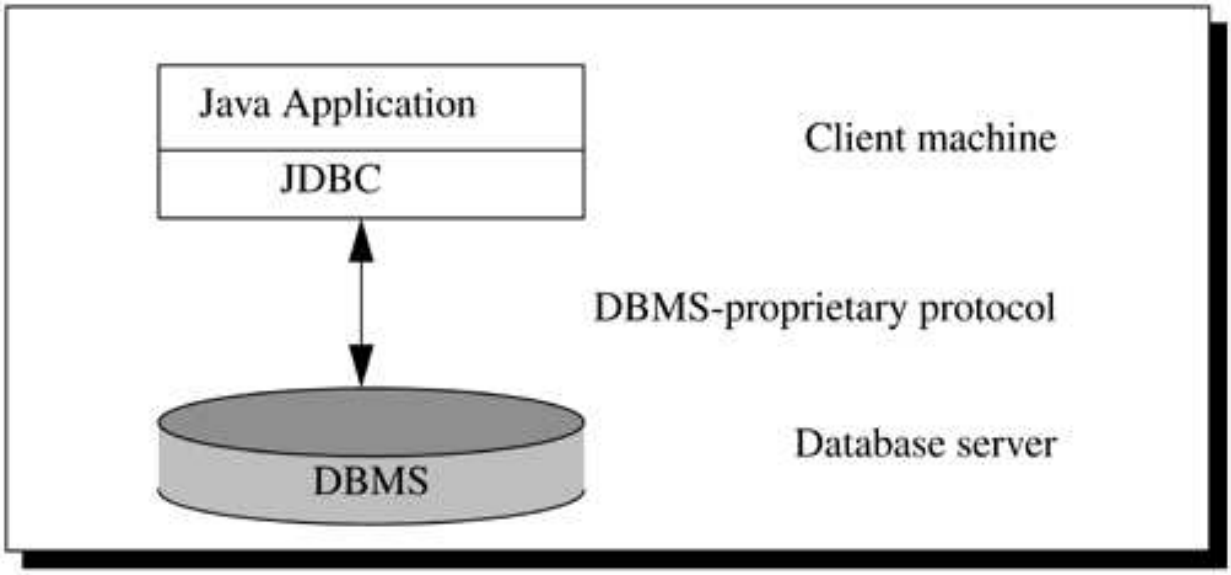

Figura 3 – Modelo de acesso de duas camadas. Fonte: (FISHER, 2003).

Essa camada pode ser implementada como uma aplicação servidora. Este será o modelo implementado neste estudo.

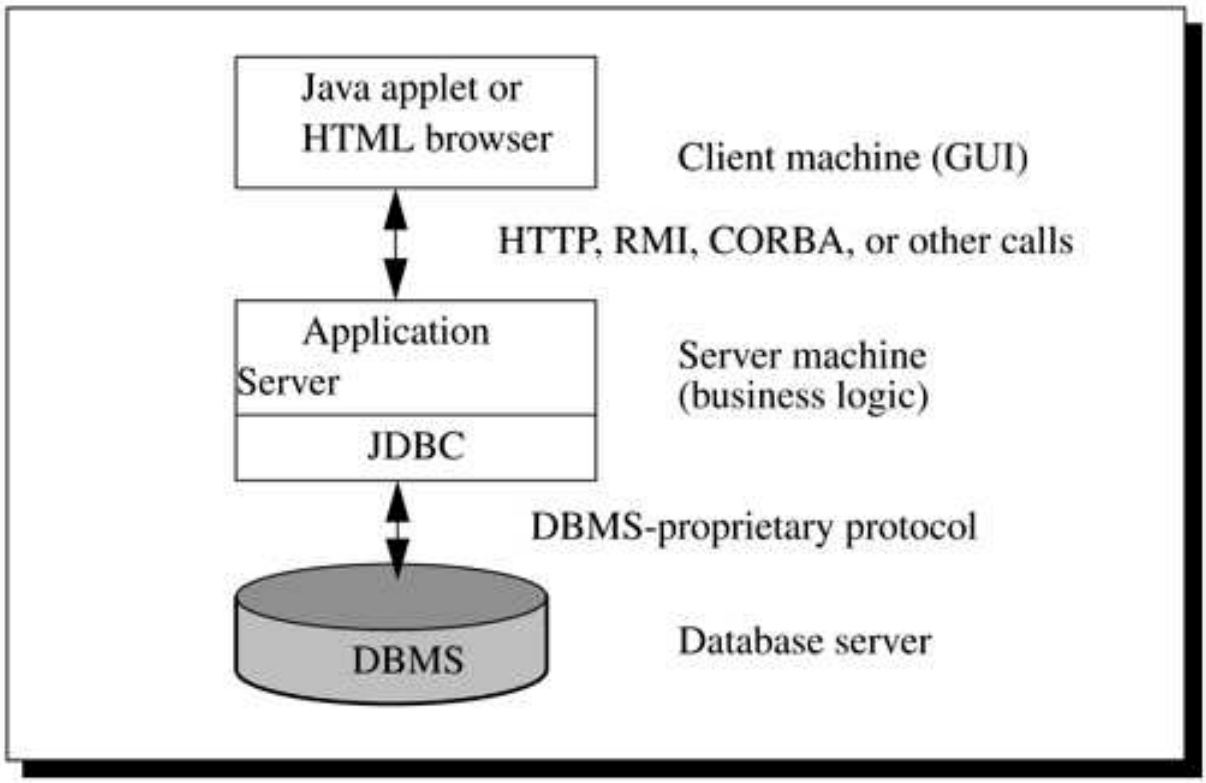

Figura 4 – Modelo de acesso de três camadas. Fonte: (FISHER, 2003).

A Sun Microsystems oferece a estrutura para a utilização do JDBC, como o gerenciador de driver JDBC (*driver manager*) disponível a partir da plataforma Java 2 na forma da classe DriverManager do pacote java.sql, a ponte JDBC-ODBC, e APIs de teste para o JDBC (FISHER, 2003).

A ponte JDBC-ODBC permite que drivers ODBC sejam usados como drivers JDBC, o que abstrai as implementações. Existem quatro tipos de drivers JDBC: (1) ponte JDBC-ODBC com driver ODBC: provê acesso de uma aplicação Java ao banco de dados usando JDBC via o driver ODBC. Essa implementação necessita que arquivos binários ODBC estejam rodando em cada máquina cliente, o que caracteriza um melhor uso em empresas utilizando servidores de três camadas; (2) API nativa parcialmente driver Java: permite que

chamadas JDBC sejam convertidas em chamadas de API para um SGBD específico e também necessita que os arquivos binários estejam nas máquinas clientes; (3) driver JDBC Java puro: permite traduzir chamadas JDBC para um protocolo de rede independente de SGBD, que depois pode ser traduzido para um protocolo de um SGBD específico por um servidor. Dessa forma, uma aplicação cliente pode se conectar a vários SGBDs diferentes e vários clientes podem se conectar ao servidor; (4) Protocolo nativo driver Java puro: converte chamadas JDBC diretamente para um protocolo de rede de um SGBD específico. É indicado para uso em *intranets* empresariais (HORSTMANN e CORNELL, 2002 ; FISHER, 2003).

Os *drivers* de categoria 3 e 4 são os preferidos para utilização da API JDBC, pois oferecem todas as vantagens da Linguagem Java como, por exemplo, instalação automática de um driver JDBC por um applet que precisa usá-lo. Os drivers 1 e 2 são soluções provisórias pois não há driver puramente Java disponíveis (FISHER, 2003).

De maneira breve, a conexão de uma aplicação Java com um banco de dados é feita pela classe *DriverManager*, que gerencia os detalhes de estabelecimento da conexão trabalhando entre o usuário e o *driver* do banco. Esse procedimento inclui duas fases: carregamento do *driver* e a realização da conexão, conforme ilustra a Figura 5 nas linhas 1 e 3 respectivamente.

```
1       Class.forName("sun.jdbc.odbc.jdbcOdbcDriver");
2
3       Connection conexao = DriverManager.getConnection(url,
4             usuário, senha);
```

Figura 5 – Instanciação de um *driver* e criação de uma conexão

O comando que se encontra na linha 1 cria uma instância de um *driver* de banco de dados. O parâmetro usado nessa declaração é o *driver* JDBC usado, no exemplo, do MySQL.

O método *getConnection* ilustrado pela linha 3 da Figura 5, retorna para o objeto

conexao uma conexão aberta para manipulação de comandos no banco. O parâmetro *url* contém pelo menos duas partes: o protocolo indicador (jdbc, por exemplo) e o subprotocolo indicador (o nome do banco, por exemplo). Outras informações podem ser necessárias, dependendo da especificação do fabricante do banco de dados (JEPSON, 1997).

Para que os comandos SQL possam ser executados no banco de dados, um objeto precisa ser criado com o objetivo de ser um mensageiro que utiliza o objeto conexão para se comunicar com o banco. Esse objeto pode ser de três tipos: *Statement*, *PreparedStatement* e *CallableStatement*.

Statements são utilizados quando os comandos SQL que ele executará são modificados várias vezes durante a execução do programa Java. Um exemplo de uso pode ser visualizado na Figura 6.

```
1       Statement consulta_stmt = conexao.createStatement();
2
3       String consulta = “SELECT ...”;
4
5       consulta_stmt.executeQuery(consulta);
```

Figura 6 – Exemplo de utilização da classe *Statement*

Um objeto *PreparedStatement* é voltado para comandos SQL que são executados muitas vezes, pois ele garante melhor performance em relação ao *Statement*. A principal diferença entre eles é que no *PreparedStatement* a instrução SQL deve ser colocada na instanciação do objeto, como pode ser visto na linha 1 da Figura 7. Isto permite que essa instrução seja enviada imediatamente para o SGBD, tornando-a pré-compilada e fazendo com que sua execução seja mais rápida em relação a uma instrução que será compilada mais tarde, durante a execução de um programa (JEPSON, 1997 ; FISHER, 2003):

O objeto criado por meio da classe *PreparedStatement* permite construir uma consulta com pontos de interrogação (?), conforme mostram as linhas de 3 a 6 da Figura 7.

Dessa maneira, a atribuição de valores a esses parâmetros deve ser feita posteriormente na aplicação, com métodos como s*etString()* e s*etInt()* (JEPSON, 1997).

```
1       PreparedStatement consulta_prep =
2            conexao.prepareStatement(“SELECT...” );
3       PreparedStatement insercao_prep =
4            conexao.prepareStatement(“INSERT
5            INTO nome_da_tabela (coluna1, coluna2)
6            VALUES (?, ?) ” );
7
8       insercao_prep.setString(1,”Uma_string”);
9       insercao_prep.setInt(2, 4444);
```

Figura 7 – Exemplo de utilização da classe *PreparedStatement*

Já um objeto criado com a classe *CallableStatement* permite a criação e execução de procedimentos armazenados. Para retornar dados os parâmetros de saída devem ser registrados com o método *registerOutParameter()*, que são dois: a posição do parâmetro de saída e o tipo do dado de acordo com a classe java.sql.Types (JEPSON, 1997).

O procedimento é criado e enviado ao SGBD por um objeto criado pela classe *Statement*, conforme é mostrado na linha 4 da Figura 8, e o objeto criado com a classe *CallableStatement* contém uma chamada para o procedimento, não o próprio procedimento, como mostram as linhas 6 e 7 da mesma figura.

```
1       Statement stmt = conexao.createStatement();
2       String proced_sql = “CREATE PROCEDURE nome_proced”
3              + “num INT, nome VARCHAR(20) OUTPUT ...”
4       stmt.executeUpdate(proced_sql);
5
6       CallableStatement          insercao_call          =
7              conexao.prepareCall(“{call nome_proced}”);
8       insercao_call.setInt(1, 4567);
9       insercao_call.registerOutParameter(2,
10             java.sql.Types.VARCHAR);
11      insercao_call.executeUpdate();
```

Figura 8 – Exemplo de utilização da classe *CallableStatement*

Como no uso do *PreparedStatement*, o parâmetro de entrada é definido pelo método *setInt()*, pois *CallableStatement* é sua subclasse (FISHER, 2003). O procedimento SQL pode ser composto apenas de consultas do tipo SELECT, e sendo assim, o método invocado do objeto *insercao_call* seria *executeQuery()* e o resultado seria armazenado em um objeto *ResultSet*.

O objeto que recebe o resultado de uma consulta SQL (select) é instanciado pela classe *ResultSet*, e pode armazenar qualquer tipo de dado, mas para que a aplicação faça proveito dessas informações, elas devem ser transferidas para objetos ou atributos dos seus devidos tipos utilizando métodos da classe *ResultSet* como *getString*, *getFloat*, *getInt*, etc, como mostra a Figura 9.

```
1       ResultSet resultado = stmt.executeQuery(“SELECT ...”);
2
3       int numero = resultado.getInt(“Codigo”);
```

Figura 9 – Exemplo de uso da classe *ResultSet*

O objeto criado com a classe *ResulSet* possui um cursor que inicialmente está apontando para o inicio da primeira linha de resultados. Dessa forma, o método *next()* desse objeto faz com que o cursor aponte para a próxima linha. Apesar disso, é possível a criação de um objeto que permita que seu cursor ande aleatoriamente por ele, especificando uma das duas constantes no objeto *Statement* que são definidas na classe *ResultSet:* TYPE_SCROLL_INSENSITIVE e TYPE_SCROLL_SENSITIVE, como na Figura 10.

A primeira significa que o objeto *ResultSet* terá um cursor que permite rolagem mas não é sensível a atualização, enquanto que a segunda significa que o objeto terá um cursor que permite rolagem e sensível a alterações.

Ele pode ser usado pelo método *executeQuery()* dos objetos *Statement*,

PreparedStatement ou *CallableStatement*. O parâmetro do método *getInt* é o nome da coluna da tabela no banco de dados, que pode ser também um número: caso a coluna C*odigo* seja a primeira da tabela, ela pode ser representada por *1*, se houverem mais colunas, elas podem ser *2, 3,* e assim por diante.

```
1       Statement stmt =
2           con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);
```

Figura 10 – Uso de uma constante da classe *ResultSet*

2.3 Socket

Para proporcionar a comunicação de duas aplicações Java é necessário construir, por meio de programação, o canal de trocas de informações. Essa construção é facilitada pela abstração de classes Java específicas para este fim. Neste trabalho foram usadas duas aplicações que se comunicam por um canal: uma denominada cliente, que é executada em um dispositivo móvel e uma denominada servidora que é executada em um computador pessoal e espera a solicitação de serviços por parte da aplicação cliente.

Uma classe utilizada para construir este canal é a *Socket*, que, quando é instanciada em objeto, passa a representar uma abstração para o software de rede que habilita a comunicação de entrada e saída do programa, como um soquete ou conexão (HORSTMANN e CORNELL, 2002). A porta usada para comunicação é um parâmetro para o construtor desse objeto.

Esta classe se encontra no pacote java.net e a manipulação de dados obtidos ou enviados em uma conexão de rede utilizando este pacote é parecida com o manuseio de arquivos ou *streams* (fluxo) (HORSTMANN e CORNELL, 2002).

No caso da criação de uma aplicação servidora, o *socket* que indica uma conexão

com esta aplicação é o *ServerSocket*. Assim como para a classe *Socket*, o parâmetro padrão a ser passado no momento de sua instanciação é a porta de conexão, porém com a diferença de que há o segundo parâmetro opcional: o número de conexões suportadas pelo servidor. Se este parâmetro for omitido, o servidor será capaz de receber apenas 50 conexões simultâneas (Sun, 2006). Um exemplo pode ser visto na Figura 11, a criação de um objeto com a classe *ServerSocket* para a porta 11111 com no máximo 200 conexões simultâneas:

```
1       ServerSocket servidor = new ServerSocket(11111, 200);
```

Figura 11 – Exemplo de uso da classe *ServerSocket*

Para haver conexão, o método *accept()* da classe *ServerSocket* é usado. Ele permite que a aplicação fique a espera de uma conexão na porta pré-definida. Quando uma conexão é estabelecida, este método retorna um objeto *Socket* que pode ser usado para criar objetos que leiam a entrada e escrevam dados na saída para o programa que fez a conexão (HORSTMANN e CORNELL, 2002). Um exemplo pode ser visto na Figura 12.

```
1       DataInputStream entrada = new
2               DataInputStream(socket.getInputStream());
3       DataOutputStream saida = new
4               DataOutputStream(socket.getOutputStream());
```

Figura 12 – Exemplo de instanciação de objetos para comunicação

Para que o servidor possa enviar dados para um possível cliente que efetuou uma conexão, ele deve usar os métodos definidos na classe *DataOutputStream*, como por exemplo *write()*, assim como se ele precisa ler dados que o cliente está lhe enviando, deve utilizar os métodos definidos na classe *DataInputStream*, como por exemplo *read()*. Para a transmissão de dados binários, é necessário o uso do tipo de *Stream* descrito anteriormente. Para o envio

de objetos serializados, pode-se utilizar objetos *ObjectInputStram* e *ObjectOutputStram*. Para texto, *BufferedReader* e *PrintWriter* (HORSTMANN e CORNELL, 2002).

2.4 Thread

Uma *thread* pode ser definida como um fluxo de controle de execução que a faz parecer muito com um processo, pois cada *thread* contém sua própria pilha de execução e variáveis de controle e pode conter variáveis locais definidas pelo programador (TANENBAUM, 2003). Como as *threads* são abertas dentro de um processo, recursos como memória, são compartilhados, fazendo com que uma *thread* seja mais leve que um processo em termos de execução. O fato de ser aberta junto não significa que sua execução será imediata. Esse fato depende muito do hardware e do software utilizado no computador (ARAGÃO, 2006).

Quanto ao hardware, existe um recurso chamado paralelismo, que consiste no uso de 2 ou mais CPUs que podem prover a execução de 2 ou mais processos de maneira simultânea, mas sem garantir que serão executados do começo ao fim, pois pode acontecer de um outro processo ser escalado com maior prioridade. Esse controle depende do software utilizado para gerenciar o hardware do computador, que neste caso é o sistema operacional.

Quanto ao software, o método de controle de execução de processos pode ser dado de duas maneiras: cooperativo ou preemptivo.

No método cooperativo, um processo que detém o uso da CPU permanece desta forma até que ele mesmo decida parar, ou seja, o controle de quanto de processamento esse processo usará depende do programador. O próximo processo que será executado está em uma fila de espera organizada por prioridade. Este método é mais rápido que o peemptivo, mas pode deixar que algum processo nunca chegue a ser executado.

No método preemptivo, um processo recebe uma fatia de tempo para ser executado na CPU. Quando este tempo termina, ele é retirado de execução e todo o seu estado de execução é salvo para que ele possa retornar a execução depois do ponto que parou. Esta atividade é feita pelo sistema operacional e não tem intervenção do usuário. Ela é feita de maneira que este último tenha a sensação de que seus programas estejam sendo executados simultaneamente e no caso da existência de 2 ou mais processadores, esse modelo possibilita o uso de paralelismo (ARAGÃO, 2006), (HORSTMANN e CORNELL, 2000).

Java prevê o uso de *threads* e possibilita sua programação através de sua API. Existem duas maneiras para se construir *threads* em Java: estendendo a classe *Thread* ou implementando a interface *Runnable*.

Quando a classe *Thread* é estendida, e conseqüentemente seu único método *run()* é sobrescrito (SUN, 2006), não é possível estender mais classes. Quando classe *Runnable* é implementada, a programação pode ser mais bem aproveitada, pois se pode implementar outras classes ou estender uma classe na nova *thread* criada (ARAGÃO, 2006).

O uso de *threads* na aplicação servidora é importante, pois dois ou mais clientes podem tentar se conectar ao banco de dados. Nesse caso, a criação de várias *threads* é possível, pois para cada uma é criada uma pilha de dados e, sendo o método de divisão de uma mensagem o mesmo para todas elas, são criadas variáveis locais para manipulação de dados para cada pilha individualmente, o que as impede de entrar em uma condição de corrida (ARAGÃO, 2006).

2.5 Considerações Finais

Como visto, Java é uma linguagem de programação poderosa, que permite o uso de recursos avançados de maneira mais intuitiva e melhora muitos aspectos em relação à

linguagens orientadas a procedimentos.

A forma de utilização de pacotes facilita para os desenvolvedores o compartilhamento de suas classes, de maneira tal que, mesmo que existam outras com os mesmos nomes, elas serão diferentes.

Da mesma maneira, esses conceitos são aplicados às classes da API JDBC, o que proporciona independência de desenvolvimento de *drivers* para um banco de dados qualquer, ou seja, basta que o fabricante implemente as interfaces apresentadas nesta API e forneça a implementação na forma de um *driver* para que as pessoas interessadas na utilização do seu banco de dados possam utilizá-lo.

O uso de classes que proporcionam comunicação entre aplicações remotas é imprescindível, por isso alguns conceitos foram estudados e apresentados neste trabalho. O tipo de dado a ser transportado entre essas aplicações precisa ser conhecido antes do desenvolvimento, pois implica na criação do objeto sobre a conexão e também no tratamento dos dados.

Para o desenvolvimento da aplicação cliente e das bibliotecas propostas neste trabalho foram utilizados conceitos sobre a API Java específica para programação de dispositivos móveis em geral, conhecida como J2ME.

CAPÍTULO 3 – J2ME

J2ME é o resultado do desenvolvimento de uma versão menor do J2SE para que a linguagem Java pudesse ser usada em dispositivos menores que um computador, com menos memória e menor poder de processamento. Foi apresentado em meados de 1999 numa conferência JavaOne promovida pela Sun Microsystems (MUCHOW, 2001).

Como a diversidade de modelos de dispositivos móveis com características diferentes é muito grande, eles foram divididos em dois grupos ou configurações: *Connected Device Configuration* (CDC) e *Connected Limited Device Configuration* (CLDC). Cada configuração consiste em uma maneira diferente de definição de características da linguagem em relação ao hardware (quanto ao uso de memória, recursos de tela, conectividade com redes, poder de processamento e suas limitações) e de bibliotecas para a JVM (MUCHOW, 2001).

A configuração CLDC é destinada ao uso em dispositivos com processadores de 16 bits ou 32 bits e memória limitada entre 160KB e 512KB. Nesta configuração, a máquina virtual Java precisou ser redefinida para se adaptar a estas condições, surgindo assim a Máquina Virtual KJava (KVM).

A configuração CDC é definida para dispositivos com processadores de arquitetura de 32 bits, memória mínima de 2MB e utilizam a JVM na sua versão integral.

Também foram definidas divisões de classes que possibilitam aos desenvolvedores implementarem características encontradas em determinado grupo de dispositivos. Essas divisões são chamadas de perfis ou *profile* e são sete ao todo: (1) *Foundation Profile*: é o centro de todos os perfis usados no CDC, pois contém as classes mais importantes; (2) *Game Profile*: é usado no CDC e contém as classes necessárias para desenvolvimento de jogos para esta configuração; (3) *Mobile Information Device Profile* ou MIDP: é usado na configuração

CLDC e provê classes que possibilitam armazenamento local, comunicação em rede, interação com o usuário, etc, para dispositivos como Palm OS; (4) *PDA Profile* ou PDAP: permite controle de características mais sofisticadas para dispositivos da configuração CLDC, como telas e memória maiores; (5) *Personal Profile*: possibilita o desenvolvimento de aplicações com interfaces complexas para o usuário, como várias janelas. É utilizado no CDC em conjunto com o Foundation Profile; (6) *Personal Basis Profile*: similar ao Personal Profile e também utilizado com o Foundation Profile, mas com classes para o desenvolvimento de aplicações com interfaces mais simples e; (7) *RMI Profile*: contém classes para possibilitar a chamada de métodos remotos (RMI) e é utilizado junto com o Foundation Profile, e conseqüentemente na configuração CDC (MUCHOW, 2001).

O J2ME é organizado segundo a arquitetura definida na Figura 13. A camada base é o sistema operacional, que tem o controle de todas as funcionalidades do dispositivo e disponibiliza seu uso para a máquina virtual Java instalada.

A camada de configuração faz a ligação entre a camada de APIs J2ME, que contém as APIs básicas como as que oferecem entrada e saída (*java.io*), tipos e classes básicas (*java.lang*) e classes de tipo vetor, pilha, e de data e tempo (*java.util*), com a camada da máquina virtual. A camada MIDP contém as APIs de acesso à rede, interface com usuário e armazenamento de dados (javax.microedition) (PAULA, 2006 ; MUCHOW, 2001).

Ainda há as camadas OEM, desenvolvidas pelo fabricante do dispositivo. A base de classes OEM contém as classes utilizadas pelas aplicações OEM, camada esta que se encontra logo acima da base. As aplicações desenvolvidas para a MIDP podem acessar essas camadas, mas isso diminui a portabilidade da aplicação J2ME, pois cada fabricante tem uma implementação diferente para suas classes e aplicações (MUCHOW, 2001).

Essas aplicações MIDP são chamadas MIDlets. Elas devem estender pelo menos uma classe da API disponível para a plataforma J2ME: javax.microedition.midlet.MIDlet.

MIDlets com características em comum podem ser empacotadas, o que garante que elas dividam as mesmas classes. Quando algumas aplicações precisam executar essa classe compartilhada ao mesmo tempo, apenas uma instância é criada naquele dado momento na máquina virtual. Isso pode ser visto como um benefício, pois elas compartilharão também os dados, que podem ser de armazenamento persistente ou preferências do usuário, assim como pode ser uma falha, pois há grande chance de causar condições de corrida na leitura ou escrita de algum dado. Esse compartilhamento não ocorre em MIDlets de pacotes diferentes, pois o nome dessa MIDlet é usado na identificação dos dados associados ao pacote (MUCHOW, 2001).

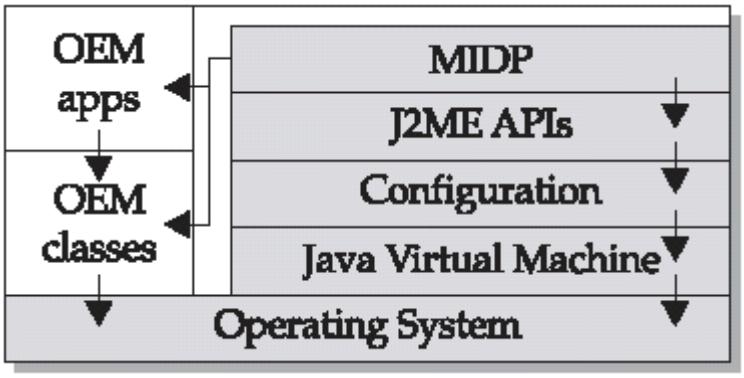

Figura 13 – Camadas da arquitetura J2ME. Fonte: (MUCHOW, 2001)

As aplicações são instaladas, executadas e removidas por um gerenciador que o fabricante do dispositivo móvel fornece. Uma vez instalada, ela pode ter acesso à JVM e à CLDC.

Antes de ser instalada, uma aplicação deve estar contida em um arquivo JAR, assim como todos os arquivos necessários para sua implementação como, por exemplo, classes, e imagens gráficas juntamente com um arquivo *manifest*, que descreve a aplicação e é utilizado pelo gerenciador do fabricante do dispositivo para instalar os arquivos contidos no arquivo JAR. Esse pacote deve conter também um arquivo JAD que acrescenta informações sobre a instalação do arquivo JAR.

A criação de uma aplicação J2ME é similar a de uma aplicação J2SE comum, a não ser pelo fato da classe precisar estender a classe MIDlet e conter três métodos abstratos: startApp(), pauseApp() e destroyApp(). Esses métodos definem o ciclo de vida da aplicação e são usados pelo gerenciador de aplicações do dispositivo para controlar sua execução, conforme a Figura 14 (MUCHOW, 2001).

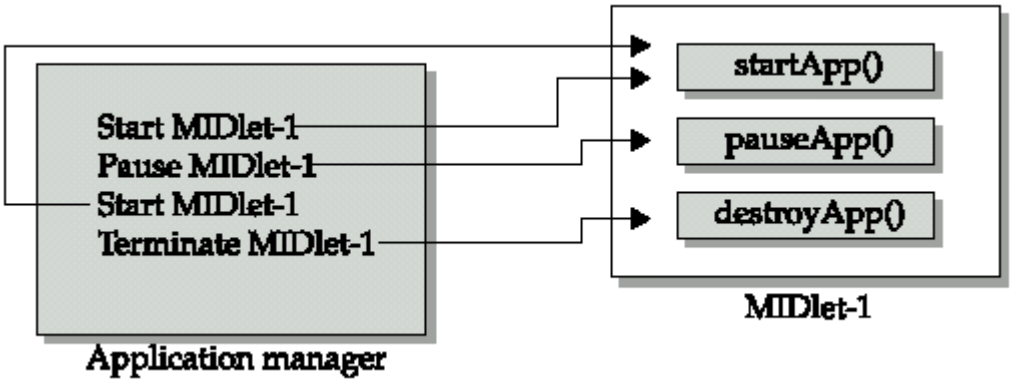

Figura 14 – Ciclo de vida do MIDlet e sua interação com o gerenciador do dispositivo. Fonte: (MUCHOW, 2001).

A interação com o usuário é feita com o monitoramento que um CommandListener oferece a cada tela exibida no dispositivo. Qualquer comando feito pelos botões ativa a execução de uma determinada ação. A forma de interface mais usada com o usuário é o Form. Nele pode-se inserir caixas de texto para colher dados do usuário ou apenas para mostrar dados, itens de escolha, caixas de data e outros tipos. Os dispositivos móveis não têm suporte a recursos como, por exemplo, rodar um SGBD (MUCHOW, 2001), mas há uma API desenvolvida por um grupo do JCP com a cooperação de Jon Ellis (um dos autores de JDBC API Tutorial and Reference), denominada JSR 169: Pacote opcional JDBC para CDC/Foundation Profile, que propõe funcionalidade equivalente ao pacote java.sql do J2SE com um conjunto menor de APIs devido às limitações dos dispositivos móveis (JCP, 2006).

O desenvolvimento das classes e métodos de acesso a um banco de dados proposto por este trabalho visa à configuração CLDC no perfil MIDP.

3.1 Considerações Finais

Para o desenvolvimento de uma aplicação J2ME, algumas características dos dispositivos móveis devem ser levadas em consideração, como quantidade de memória, tamanho da tela e teclas disponíveis. A divisão de configurações em CDC e CLDC e a definição de perfis diferentes melhoram a visão do programador sobre quais classes e métodos podem ser utilizados para determinado dispositivo.

Com o acesso à camada OEM, uma aplicação deixa de ser portável, fazendo com que ela perca uma característica importante da linguagem.

O uso da JSR 169 implica em um problema: como não há possibilidade de um SGBD ser instalado em dispositivos móveis, toda parte de comunicação com uma aplicação remota que acesse o banco de dados e retorne resultados precisa ser desenvolvida. Este trabalho apresenta uma biblioteca que reúne essas funcionalidades para uso no J2ME, disponibilizando-as em um formato parecido com a API JDBC.

Há *drivers* desenvolvidos para acessar SGBDs utilizando as classes e métodos definidos na JSR 169, como por exemplo o Mimer, que é utilizado no acesso do SGBD Mimer SQL (MIMER, 2006). Para um outro produto, o PointBase Micro, há um pequeno SGBD que pode ser instalado em dispositivos móveis (POINTBASE, 2006). Em ambos os casos, se o desenvolvedor da MIDlet precisa utilizar um SGBD robusto instalado remotamente, precisa desenvolver toda comunicação com uma aplicação que se conecte ao banco de dados.

CAPÍTULO 4 – BIBLIOTECA DE ACESSO A BANCO DE DADOS

A linguagem de programação Java facilita em muitos aspectos o desenvolvimento de aplicações devido à abstração e ao encapsulamento, mas conforme aumenta a necessidade de uso de mais recursos da linguagem, maior a complexidade que envolve seu desenvolvimento.

Quando um programador de aplicações que acessam banco de dados em dispositivos móveis utiliza J2ME, ele precisa usar classes e métodos disponíveis nesta API para prover comunicação e controle das informações enviadas e recebidas, pois o banco de dados é remoto.

Para auxiliar estes desenvolvedores, este trabalho reúne em um só pacote algumas classes e métodos com denominações iguais às da API JDBC, com métodos que oferecem a comunicação direta com uma aplicação servidora, sendo que esta realiza o contato com o banco de dados. Desta forma, a aplicação cliente pode enviar requisições, que são processadas pela aplicação servidora, e receber as respostas desse processamento sem que seja necessário criar objetos para comunicação direta, ou seja, o programador pode utilizar a biblioteca desenvolvida neste trabalho da mesma forma que utilizaria a API JDBC, sem se preocupar com a comunicação e com detalhes de tratamento de dados enviados e recebidos.

Neste Capítulo são apresentadas as características para criação das classes da biblioteca proposta por este trabalho, e no seu último tópico, o detalhamento do funcionamento de suas classes e métodos.

4.1 Definição das classes

Conforme observa-se na Figura 15, a aplicação cliente pode instanciar objetos do tipo *Connection*, *Statement* e *ResultSet*.

Quando um objeto *Connection* é criado, são passados por parâmetro a *url* para o

banco de dados, o nome e senha do usuário para o método *getConnection* do próprio objeto. Este método cria uma mensagem com os dados passados por parâmetro e os envia à aplicação servidora, instanciada com a classe *Servidor* na Figura 23, para que ela crie a conexão com o banco de dados localmente pela classe *DriverManager* nativa do JDBC. Cada cliente pode realizar uma ou mais conexões, mas cada conexão possui apenas um cliente. Essa relação procede da mesma maneira para os objetos *Statement* e *ResultSet*.

O objeto instanciado com a classe *Connection* ainda contém o método *createStatement()* que gera um pedido de criação de um objeto da classe *Statement* e o envia também para que a aplicação servidora. O objeto criado com a classe *Statement* é usado para enviar instruções em SQL para o servidor através do seu método *executeQuery()*. Quando este método é invocado, um objeto da classe *ResultSet* precisa ser instanciado para armazenar o retorno da execução dessa instrução. A classe *ResultSet* possui alguns métodos para capturar dados sobre este objeto e para manipular o cursor que aponta para ele. Todas as características citadas acima são representadas na Figura 15.

Para a aplicação servidora atender à requisição de dois ou mais clientes simultaneamente é criado um objeto da classe *ThreadServidor* que, pelo seu método *run()*, cria uma nova linha de execução (*thread*) para cada cliente que tenta estabelecer conexão com o servidor. Além disso, essa aplicação precisa entender a requisição que um cliente a envia. Para isso, utiliza a o metodo *divSentenca()* para tratar a mensagem recebida.

No cliente, uma mensagem é gerada conforme o código de uma operação que é requerida ao servidor. Esses códigos podem ser compostos de um ou mais números inteiros: 0 para a realização de uma conexão com o banco de dados, 1 para a criação de um objeto da classe *Statement*, e assim por diante. Um caracter especial, como um ponto (.), por exemplo, permite que mais informações sejam colocadas em uma mensagem, pois possibilita que a aplicação servidora separe os dados transmitidos e decida o que fazer com esses dados.

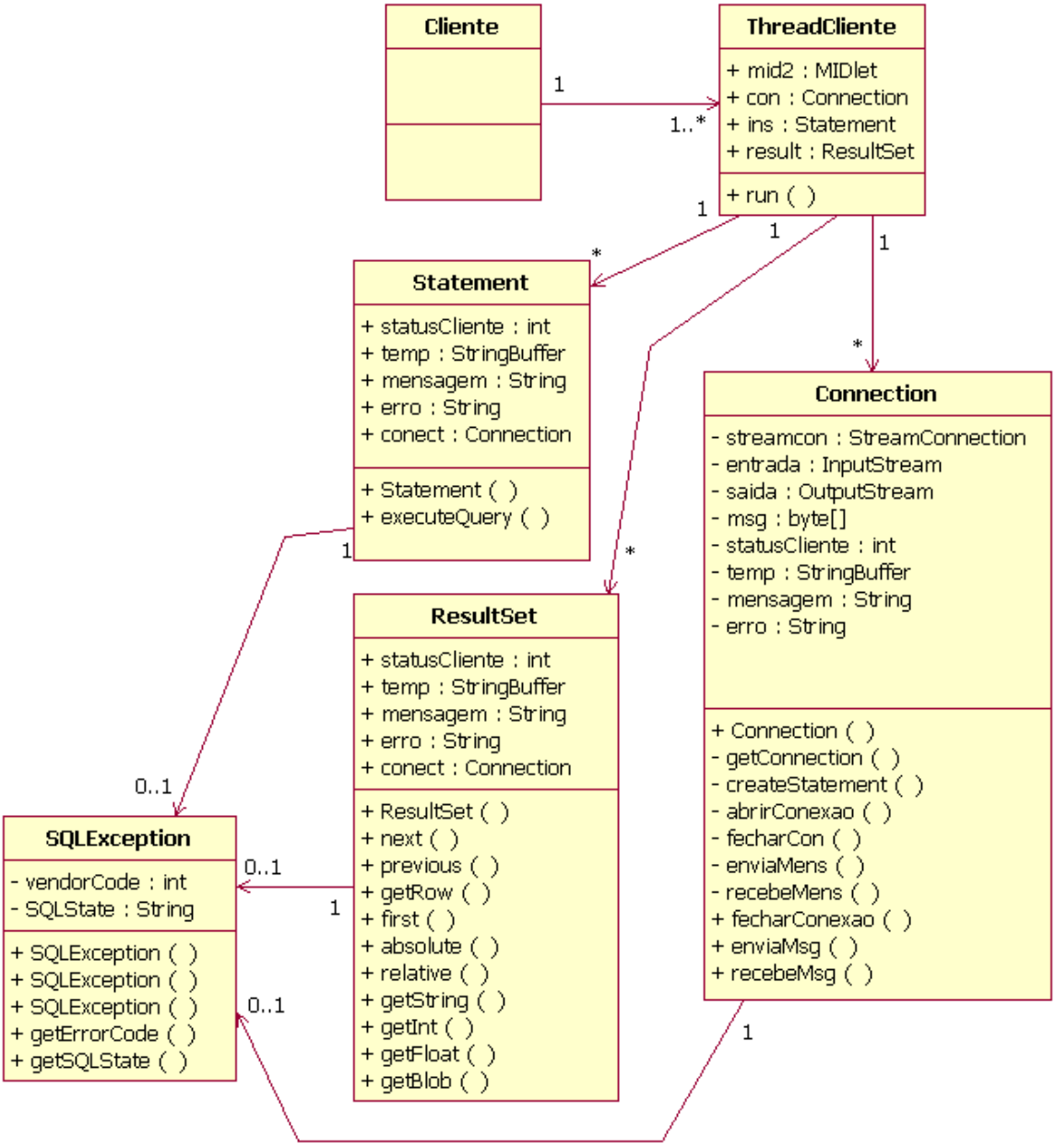

Figura 15 – Diagrama de classes para a biblioteca cliente

4.2 Biblioteca Cliente

A biblioteca desenvolvida neste trabalho visa facilitar o uso de bancos de dados pelo programador de uma aplicação J2ME. As classe e métodos contidos nela e suas funções podem ser melhor visualizados na Tabela 1. Desta forma, ela foi desenvolvida com a idéia de

ter o funcionamento parecido ao máximo com a API JDBC existente para o Java SE.

Classes	**Métodos**	**Acesso**	**Função**
Connection		Público	comunicação com aplicação servidora e com o SGBD
	Connection()	Público	construtor
	getConnection()	Público	criação de um objeto da classe *Connection*
	createStatement()	Público	criação de objeto da classe *Statement*
	abrirConexao()	Privado	Conexão com aplicação servidora
	fecharCon()	Privado	fecha conexão com aplicação servidora
	enviaMens()	Privado	envia mensagens para a aplicação servidora
	recebeMens()	Privado	recebe mensagens da aplicação servidora
	fecharConexao()	Público	fecha conexão com aplicação servidora
	EnviaMsg()	Público	envia mensagens para a aplicação servidora
	recebeMsg()	Público	recebe mensagens da aplicação servidora
Statement			envio de *queries*
	Statement()	Público	construtor
	executeQuery()	Público	cria objeto da classe *ResultSet* e envia *query* SQL para a aplicação servidora
	executeUpdate()	Público	envia *query* de modificação
ResultSet			manipulação de resultados
	ResultSet()	Público	construtor
	next()	Público	movimenta o cursor de um objeto da classe *ResultSet* para a próxima linha
	previous()	Público	movimenta o cursor de um objeto da classe *ResultSet* para a linha anterior
	getRow()	Público	retorna o número da linha para a qual o crusor aponta atualmente
	first()	Público	Movimenta o cursor para a primeira linha em um objeto da classe *ResulSet*.
	absolute()	Público	Movimenta o cursor para a linha dada como parâmetro no objeto da classe ResulSet
	relative()	Público	Movimenta o cursor em relação à linha atual no objeto ResulSet.
	getString()	Público	retorna a *String* armazenada em determinada coluna dada como parâmetro
	getInt()	Público	retorna o *int* armazenado em determinada coluna dada como parâmetro
	getFloat()	Público	retorna o *float* armazenado em determinada coluna dada como parâmetro
	getBlob()	Público	retorna o *Blob* armazenado em determinada coluna dada como parâmetro
SQLException			manipulação de erros
	SQLException()	Público	construtor
	getErrorCode()	Público	retorna código do erro apresentado
	getSQLState()	Público	retorna descrição do erro apresentado
Codigo			constantes de comunicação

Tabela 1 – Descrição das classes e métodos desenvolvidos

O nome do pacote que contém as classes dessa biblioteca é javaxtend.sql. A intenção era utilizar o nome javax.sql, pois javax significa que este pacote é uma extensão, ou seja, pacote que não é fornecido junto com o núcleo Java, mas neste caso este nome já é definido pela API, então não pode ser utilizado (HORSTMANN e CORNELL, 2002).

A classe *Connection* possui os seguintes métodos privados usados para comunicação com a aplicação servidora: *abrirConexao(), fecharCon(), enviaMens()* e *recebeMens()*. Possui também um método para a criação do objeto *Connection* do banco de dados: *getConnection()*, que tenta criar uma conexão entre a aplicação cliente e a servidora através do método *abrirConexao()*. Caso ele consiga, o atributo statusCliente recebe o valor de uma constante definida na classe *Codigo* deste mesmo pacote, indicando se a conexão ocorreu ou não. Caso sim, é enviada à aplicação servidora, através do método *enviaMens()*, de acordo com a linha 12 da Figura 16, uma mensagem contendo o nome do banco de dados, do usuário e a senha para acesso, de acordo com os parâmetros passados pela aplicação cliente. Cada um desses dados são separados por um sinal de ponto(.), o que possibilita que toda essa informação seja enviada de uma só vez para a aplicação servidora e permite que esta separe as informações e tome a decisão sobre o uso dos dados, conforme pode-se oberservar nas linhas 8 e 9 da Figura 16.

Após o envio, a resposta do servidor é aguardada pelo método *recebeMens()*, na linha 16 desta mesma Figura. Caso a aplicação servidora consiga criar um objeto da classe *Connection*, um novo objeto desta mesma classe é retornado pelo método *getConnection()* da biblioteca, conforme mostra a linha 23 da Figura 16. Caso contrário, é lançada uma nova exceção SQL pela classe *SQLException*.

O método para criação de um objeto *Statement* se encontra na classe *Connection* desta biblioteca, assim como na classe homônima da API JDBC. Nela é testado o conteúdo do atributo statusCliente: se seu valor indicar que a conexão com o servidor e o objeto

Connection no servidor foram efetuados com sucesso, o objeto *Statement* pode ser criado, senão é lançada uma nova exceção *SQLException*, como mostra a Figura 17.

```
1   public static Connection getConnection(String url, String bd,
2      String usuario, String senha) throws SQLException
3   {  abrirConexao(url);
4
5      if (statusCliente == Codigo.CONEXAO_SERV_OK)
6      {
7             temp = new
8       StringBuffer().append(bd).append(".").append(usuario).app
9       end(".").append(senha);
10            mensagem = new String(temp);
11
12            enviaMens(mensagem);
13
14            temp.delete(0, temp.length());
15
16            mensagem = new String(recebeMens());
17            statusCliente = (Integer.parseInt(mensagem));
18            System.out.println("Resposta do servidor: " +
19         statusCliente);
20
21            if (statusCliente == Codigo.CONNECTION_OK)
22            {      System.out.println("Objeto Connection criado");
23                   return new Connection(statusCliente);
24            }
25            else
26            {      erro = new String("Erro no banco de dados");
27                   throw new SQLException(erro, statusCliente);
28            }
29     }
30     else
31     {      erro = new String("Nao ha conexao com o servidor");
32            throw new SQLException(erro, statusCliente);
33     }
34   }
```

Figura 16 – Método *getConnection()*

O método *abrirConexao()*, como já citado anteriormente, efetua uma conexão para comunicação com a aplicação servidora, utilizando a interface *StreamConnection* que define como uma conexão por *streams* deve funcionar [SUN, 2006]. Para que um objeto desta classe seja criado, utilizou-se neste caso, o método *open()* da classe *Connector*, usando dois parâmetros:o endereço ou url da aplicaçao servidora e a habillitação do modo escrita e leitura para os *streams* da conexão, como pode ser observado nas linhas 4 e 5 da Figura 18.

```
1    public Statement createStatement() throws SQLException
2    {    if (statusCliente == Codigo.STATEMENT_OK)
3         {    return new Statement(this, statusCliente);
4         }
5         else
6         {    System.out.println("Erro: Nao foi criado o objeto
7                    conexao");
8              throw new SQLException(erro, statusCliente);
9         }
10   }
```

Figura 17 – Método *createStatement()*

```
1    private static void abrirConexao(String url)
2    {   //Cria stream (fluxo) de comunicacao
3       try
4       {     streamcon = (StreamConnection)Connector.open(url,
5           Connector.READ_WRITE);
6             saida = streamcon.openOutputStream();
7             entrada = streamcon.openInputStream();
8             statusCliente = Codigo.CONEXAO_SERV_OK;
9       }
10      catch (IOException ioe)
11      {}
12   }
```

Figura 18 – Método *abrirConexao()*

Há também na classe *Connection* o método *fecharCon()*, que é utilizado para avisar à aplicação servidora que o cliente deseja fechar a conexão. Se o servidor aceitar a requisição, é devolvido um código que corresponde que esta ação foi executada, avisando a biblioteca sobre este estado. Para isto este método utiliza os métodos *enviaMens()* e *recebeMens()*.

Os métodos *enviaMens()* e *recebeMens()* realizam seu trabalho de enviar e receber mensagens em duas etapas: para o *enviaMens()*, primeiramente é enviado o tamanho da mensagem, depois a mensagem em si, como mostra a Figura 19. O mesmo ocorre no método *recebeMens()*: primeiro é recebido o tamanho da mensagem, depois é definido um *buffer* com o tamanho exato da mensagem, que a recebe logo em seguida. Para este método existe uma particularidade: enquanto o método *enviaMens()* manipula *strings*, este manipula um *array* do tipo *byte*. Esta característica permite que o cliente receba qualquer tipo de dados, mas foi

planejada principalmente para possibiltar que a aplicação cliente receba imagens, conforme ilustra a Figura 20. O tratamento sobre o tipo de dado recebido é feito pelos métodos *get* da classe *ResultSet*.

```
1    private static void enviaMens(String mensagem)
2    {   String tam;
3        try
4        {    tam = Integer.toString(mensagem.length()) + "?";
5             saida.write(tam.getBytes(), 0, tam.length());
6             saida.flush();
7
8             saida.write(mensagem.getBytes(), 0,
9                mensagem.length());
10            saida.flush();
11       }
12       catch (IOException ioe) {}
14   }
```

Figura 19 – Método *enviaMens()*

```
1      private static byte[] recebeMens()
2      {      int tam = 0;
3      int ch;
4             StringBuffer st = new StringBuffer();
5             try
6             {      while ((ch = entrada.read()) != -1)
7                    {      if ((char)ch == '?')
8                           {      break;
9                           }
10                          st.append((char)ch);
11                   }
12                   tam = Integer.parseInt(new String(st));
13
14                   msg = new byte[tam];
15                   entrada.read(msg, 0, tam);
16            }
17            catch (NullPointerException npe) {}
18            catch (IOException ioe)     {}
19
20            return msg;
21     }
```

Figura 20 – Método *recebeMens()*

Quanto às classes *Statement* e *ResultSet*, seus construtores são definidos com a instanciação de um objeto *Connection* para cada uma delas, ou seja, os métodos de envio, recebimento e finalização da conexão podem ser invocados nessas classes através do atributo

que recebe o objeto da classe *Connection* passado por parâmetro. Esta caracteristica é mais bem observada na Figura 21. Essas classes devem usar os métodos públicos deste objeto que se encarregam de invocar os métodos privados. Os métodos públicos da classe *Connection* são: *fecharConexao()*, *enviaMsg()* e *recebeMsg()*.

```
1       public Statement(Connection c, int status)
2       {       conect = c;
3               this.statusCliente = status;
4       }
```

Figura 21 – Construtor da classe *Statement*

```
1   public class Codigo
2   {  /*************************************************************
3      *  Status de comunicação do servidor com a aplicação cliente:
4      *************************************************************/
5      public static final int CONEXAO_SERV = 0;
6      public static final int CONNECTION = 1;
7      public static final int STATEMENT = 2;
8      public static final int RESULTSET = 3;
9      public static final int NEXT = 4;
10     public static final int PREVIOUS = 5;
11     public static final int GET_ROW = 6;
12     public static final int FIRST = 7;
13     public static final int ABSOLUTE = 8;
14     public static final int RELATIVE = 9;
15     public static final int GET_STRING = 10;
16     public static final int GET_INT = 11;
17     public static final int GET_FLOAT = 12;
18     public static final int GET_BLOB = 13;
19     public static final int FECHAR = 14;
20     public static final int EXECUTE_UPDATE = 15;
21
22     /*************************************************************
23     *  Tabela de status de confirmação no cliente:
24     *************************************************************/
25     public static final int CONEXAO_SERV_OK = 30;
26     public static final int CONNECTION_OK = 31;
27     public static final int STATEMENT_OK = 32;
28     public static final int RESULTSET_OK = 33;
29     public static final int EXECUTE_UPDATE_OK = 34;
30     public static final int ERRO = 35;
31     public static final int FECHADO = 36;
32  }
```

Figura 22 – Classe Codigo

Na classe *Statement*, há o método *executeQuery()* que envia para a aplicação servidora o código da requisição, indicando a criação de um objeto *ResultSet* junto com a *query* a ser executada, separada do código pelo sinal de ponto (.). Da mesma forma que acontece na classe *Connection*, se a resposta da aplicação servidora for positiva, de acordo com a tabela de constantes da classe Codigo, um novo objeto *ResultSet* é retornado pela biblioteca, senão uma nova exceção *SQLException* é lançada. A tabela de constantes é mostrada na Figura 22.

Na classe *ResultSet* se encontram os métodos para movimentação do cursor sobre um objeto da classe *ResultSet*, como *next(), previous(), first(), absolute(), relative(),* e métodos para resgatar informações, como *getStrin(), getInt(), getFloat() e getBlob()*. Todos se baseiam no mesmo princípio utilizado pelos métodos das outras classes desta biblioteca: enviar requisições e receber a resposta do servidor. No caso, os métodos que resgatam informações sobre o objeto instanciado pela classe *ResultSet* possuem ainda a função de modificar o tipo das informações recebidas, ou seja, a mensagem recebida pelo método *getFloat()*, por exemplo, precisa ser recebida como *String*, para este ser transformado em *Float* e retornado para a aplicação cliente. Para melhor entendimento, este processo está ilustrado na Figura 23.

```
1       public float getFloat(int index) throws SQLException
2       {       temp = new
3          StringBuffer().append(Codigo.GET_FLOAT).append(".").
4          append(index);
5
6               mensagem = new String(temp);
7               temp.delete(0, temp.length());
8
9               conect.enviaMsg(mensagem);
10
11              mensagem = new String(conect.recebeMsg());
12              return Float.parseFloat(mensagem);
13      }
```

Figura 23 – Método *getFloat()*

CAPÍTULO 5 – APLICAÇÃO SERVIDORA

A aplicação cliente precisa se comunicar com uma aplicação que tenha acesso ao banco de dados de maneira remota. Neste trabalho, a aplicação que realiza a tarefa de executar instruções e enviar os resultados para o cliente é denominada aplicação servidora e as classes que foram desenvolvidas para este fim são apresentadas na Figura 24. No decorrer deste Capítulo é apresentado o funcionamento da aplicação servidora.

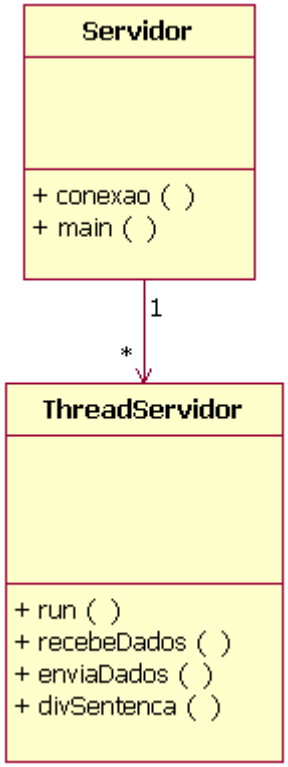

Figura 24 – Diagrama de classes para a aplicação servidora

Esta aplicação fornece o acesso a um banco de dados para a aplicação cliente e realiza também o controle de comunicação. Para que um cliente possa se conectar à aplicação servidora e estabelecer um canal de comunicação, uma porta é especificada e é usada na criação do objeto *socketServidor* da classe *ServerSocket*.

O método *Conexao()* é chamado logo após o carregamento do *driver* do banco de

dados pelo método *main()* da classe Servidor. Conforme pode-se observar na Figura 25, este *driver* é definido por uma linha de texto, especificamente a linha 2, contida no arquivo config.txt e para utilizar outro, basta mudar o nome do *driver* no referido arquivo. Para os testes utilizou-se o driver do banco de dados MySQL.

```
1       porta = 60500
2       driver = sun.jdbc.odbc.JdbcOdbcDriver
3       url_parcial = jdbc:odbc:
```

Figura 25 – Arquivo config.txt

O objeto socketServidor através do método *accept()* obriga que a aplicação aguarde, em um *loop* infinito, algum contato na porta determinada, de acordo com a linha 4 da Figura 26. Assim que este contato acontece, uma *thread* é criada a fim de prover execução simultânea de serviços do servidor tendo como parâmetro um objeto *socket* que contém a conexão, conforme mostra a linha 6 da mesma Figura. O loop *while* da linha 2 possibilita que o servidor aguarde a tentativa de conexão de outra aplicação cliente enquanto a conexão anterior continua sendo atendida.

```
1       System.out.println("Aguardando conexao na porta " + PORTA);
2       while(true)
3       {       try
4               {       Socket novaConexao = socketServidor.accept();
5                       System.out.println("Conexao estabelecida");
6                       ThreadServidor ts = new
7                               ThreadServidor(novaConexao);
8                       new Thread(ts).start();
9               }
10              catch (IOException ioe)
11              {       System.err.println("Conexao rejeitada");
12                      ioe.printStackTrace();
13              }
14      }
```

Figura 26 – Aplicação servidora aguarda conexão

A linha 8 da Figura 26, que chama o método *start()* da classe *Thread*, inicia a execução da *thread* ts criada devido à conexão de um cliente ao servidor. Dentro da classe *ThreadServidor* são definidos os objetos que enviarão e receberão dados através do objeto *socket*: *DataInputStream* e *DataOutputStream*. Também são definidas constantes que indicam o status das requisições do cliente e são utilizadas para comunicá-lo que tal tarefa foi realizada. Estas constantes são as mesmas utilizadas na classe *Codigo* da biblioteca cliente. O seu transporte, no caso um número inteiro, é mais eficaz que o envio de uma palavra que indica a requisição de um cliente ou a comunicação do acontecimento de algum erro. Por isso, para que as duas aplicações (cliente e servidora) possam se comunicar através de códigos, os valores das constantes devem ser os mesmos nas duas aplicações.

No início do método *run()* da classe *ThreadServidor*, conforme mostra a Figura 27, são declarados alguns atributos para auxiliar esta comunicação. Elas são usadas principalmente pelo método *divSentenca()*, cujo objetivo é dividir uma mensagem recebida do cliente. É necessária esta divisão pois o cliente pode enviar o código da requisição desejada em conjunto com outras informações, como por exemplo, uma requisição para criação de um objeto da classe C*onnection* (código “1” da tabela de constantes), juntamente com o nome do banco de dados, o usuário e a senha, separados por um sinal de ponto (.). O método *divSentenca()* divide a mensagem quando encontra o primeiro sinal de ponto (.) da sentença, armazenando as duas partes resultantes em um vetor. Assim, se a mensagem possui mais que um sinal de ponto(.) , ela deve ser dividida mais de uma vez. Esse método pode ser observado na Figura 28.

Na prática, o código para criação de um objeto instanciado pela classe C*onnection* não precisa ser utilizado, pois a primeira ação a ser feita logo após a conexão de um cliente é a criação deste objeto, não há outra requisição que ele possa fazer antes. Quando este objeto é criado, o atributo statusCliente recebe o código que condiz com o estado atual da requisição

de acordo com as constantes definidas, ou seja, o atributo statusCliente indica se a requisição de conexão na aplicação servidora e a criação do objeto *Connection* foram efetuadas com sucesso. Conforme mostra a Figura 29 na linha 2, o atributo *conexao* é um objeto da classe *Connection*. Caso a aplicação servidora não consiga estabelecer contato com o banco de dados usando a url, usuário e senha que a aplicação cliente enviou, o atributo conexaoAtiva recebe o valor falso, de acordo com a linha 17, o que faz com que a *thread* termine e, conseqüentemente, o cliente não poderá fazer requisicões nesta conexão. Por isso, o status é enviado ao cliente, como mostram as linhas 14 e 24 da mesma Figura. Também é criado um objeto da classe *Statement* nas linhas 6, 7 e 8, pois todas as outras requisições dependem da intermediação deste objeto.

```
1       public void run()
2       {       boolean conexaoAtiva = true;
3
4               String prim_parte = null;
5               String seg_parte = null;
6               String confirmacao = null;
7               Vector sentenca = null;
8               String temp = null;
```

Figura 27 – Declaração de atributos auxiliares à comunicação

```
1       private Vector divSentenca(String original)
2       {       Vector<String> sentenca = new Vector<String>();
3               int ponto = 0;
4
5               sentenca.removeAllElements();
6
7               ponto = original.indexOf(".");
8               if (ponto < 0) //nao tem ponto
9               {       sentenca.addElement(original.substring(0));
10                      return sentenca;
11              }
12
13              sentenca.addElement(original.substring(0, ponto));
14              sentenca.addElement(original.substring(ponto + 1));
15
16              return sentenca;
17      }
```

Figura 28 – Método *divSentenca()*

A partir deste ponto, a *thread* entra em *loop* infinito enquanto houver conexão ativa com o cliente, por meio da qual ele pode fazer qualquer tipo de requisição, dentre as previstas pelas constantes citadas anteriormente e pelas classes e métodos definidos na biblioteca cliente.

```
1       try
2       {       conexao = DriverManager.getConnection(urlStr, usuario,
3                       senha);
4               confirmacao = "Objeto Connection criado";
5
6               stmt =
7       conexao.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
8       ResultSet.CONCUR_UPDATABLE);
9
10              System.out.println("Objeto Statement criado");
11              statusCliente = STATEMENT_OK;
12              System.out.println(confirmacao);
13
14              enviaDados(Integer.toString(statusCliente).getBytes());
15      }
16      catch (SQLException e)
17      {       conexaoAtiva = false;
18              conexao = null;
19              statusCliente = e.getErrorCode();
20
21              confirmacao = "Erro na comunicacao com o banco de dados";
22              System.out.println(confirmacao + statusCliente);
23
24              enviaDados(Integer.toString(statusCliente).getBytes());
25      }
```

Figura 29 – Criação do objeto da classe *Connection* e envio do status ao cliente

O controle sobre a possibilidade da execução de determinada requisição, como por exemplo, se é possível a criação de um objeto da classe *ResultSet* e a realização de uma consulta, é feito pela biblioteca cliente através do atributo *statusCliente*, para evitar tráfego desnecessário. No caso, a requisicão nem sairá da aplicação cliente se os objetos instanciados pelas classes *Connection* e *Statement* não foram ainda criados.

As outras requisições se resumem a comparar o código da requisição enviado pelo cliente com alguma das constantes equivalentes da aplicação servidora, separar a mensagem

recebida através do método *divSentenca()*, realizar a requisição e enviar o resultado de sucesso ou de erro ao cliente, como mostra a Figura 30, num exemplo sobre o pedido de movimentação do cursor que aponta para o objeto *resultado* criado com a classe *ResultSet*.

```
1       else if (codigoRequisicao == NEXT)
2       {       try
3               {       if (resultado.next())
4                       {       temp = new String("true");
5                               System.out.println("outra linha: "+temp);
6                       }
7                       else
8                       temp = new String("false");
9
10                      enviaDados(temp.getBytes());
11              }
12              catch (SQLException e)
13              {       statusCliente = e.getErrorCode();
14
15                      confirmacao = "Erro na comunicacao com o banco de
16                              dados ";
17
18                      System.out.println(confirmacao + statusCliente);
19
20              enviaDados(Integer.toString(statusCliente).getBytes());
21              }
22      }
```

Figura 30 – Exemplo de atendimento de requisições

CAPÍTULO 6 – APLICAÇÃO CLIENTE

Neste Capítulo é apresentada uma aplicação desenvolvida a título de exemplo para testar as funcionalidades da biblioteca desenvolvida. Essa aplicação, denominada aplicação cliente, utiliza alguns recursos da API J2ME para ilustrar algumas soluções que podem ser usadas por um desenvolvedor.

A primeira característica a ser notada nesta aplicação é a declaração de importação do pacote que contém as classes e métodos desenvolvidas neste trabalho, por meio da cláusula *import* da Linguagem Java, mostrada na linha 1 da Figura 31.

Logo após a importação dos pacotes necessários segue a declaração da classe Cliente estendendo a classe MIDlet que se encontra no pacote javax.microedition.midlet e implementando a classe CommandListener do pacote javax.microedition.lcdui, de acordo com as linhas 7, 8 e 9 da mesma Figura (SUN, 2006). A criação de uma constante que contém o caminho para que a aplicação cliente possa se conectar à aplicação servidora também é realizado, como mostram as linhas 15 e 16.

```
1       import javaxtend.sql.*;
2
3       import javax.microedition.midlet.*;
4       import javax.microedition.lcdui.*;
5       import java.io.*;
6
7       public class Cliente extends
8               javax.microedition.midlet.MIDlet implements
9               CommandListener
10      {
11              Form inForm  = new Form("Teste de biblioteca:
12                      javaxtend");
13              Form outForm = new Form("Resultado");
14
15              private static final String url =
16                      "socket://lost.univem.edu.br:60500";
17
18              Connection con = null;
19              Statement ins = null;
```

Figura 31 – Início da aplicação cliente

Pode-se observar ainda nas linhas 11, 12 e 13 da Figura 31, a criação de dois atributos que correspondem a formulários definidos pela classe *Form*, que pode ser usado para inserção de qualquer tipo de dado para ser mostrado na tela do dispositivo móvel. Esses dois formulários são os únicos usados pela aplicação de exemplo, mas por meio deles pode-se modificar e criar várias telas (SUN, 2006). Nesta mesma Figura ainda pode ser vista a declaração dos objetos globais das classes *Connection* e *Statement*, que podem ser utilizados em outras classes dentro da MIDlet.

A implementação do método *startApp()* dá continuidade ao código da aplicação cliente. Neste método se encontram os atributos e objetos que são instanciados no primeiro momento de execução da aplicação, ou seja, logo no seu início, fazendo com que a MIDlet entre em estado ativo (SUN, 2006). Nele é criada a tela inicial da aplicação, que contém os campos de texto para o preenchimento por parte do usuário, com os dados básicos para a realização da conexão à aplicação servidora e ao banco de dados. Esses dados são o nome do banco ao qual se deseja acessar, o nome do usuário cadastrado no banco de dados e sua senha, conforme mostram as linhas 4 à 9 da Figura 32.

Nesta Figura pode ser observada também nas linhas 14 à 18, a criação de comandos que são atribuídos aos formulários e representam a ação ou escolha que o usuário pode fazer sobre determinada tela, como por exemplo, para a tela de inserção dos dados para conexão é atribuído apenas o comando “OK”, como mostra a Figura 33.

Assim que este comando é ativado pelo usuário, a aplicação permite que o objeto tC, instanciado com a classe *ThreadConexao*, conforme mostram as linhas 25 e 26 da Figura 32, entre em execução. O uso de *threads* na criaçao da conexão e comunicação com a aplicação servidora é muito importante pois evita que algum atributo seja acessado e/ou modificado simultaneamente por dois métodos na mesma *thread*, mais especificamente o *commandAction()* da classe MIDlet, como mostra a Figura 34, o compilador J2ME advertindo

sobre o problema.

```
1       public void startApp()
2       {       display = Display.getDisplay(this);
3
4               banco = new TextField("Banco", null, 25,
5                       TextField.ANY);
6               usuario = new TextField("Usuario",null,25,
7                       TextField.ANY);
8               senha = new TextField("Senha",null,25,
9                       TextField.PASSWORD);
10              inForm.append(banco);
11              inForm.append(usuario);
12              inForm.append(senha);
13
14              cmdOK = new Command("OK",Command.SCREEN,1);
15              cmdSair = new Command("Sair",Command.EXIT,1);
16              cmdContinuar = new Command("Continuar",
17                      Command.SCREEN,1);
18              cmdVoltar = new Command("Voltar",Command.BACK,1);
19
20              inForm.addCommand(cmdOK);
21              inForm.setCommandListener(this);
22              outForm.setCommandListener(this);
23              display.setCurrent(inForm);
24
25              ThreadConexao tC = new ThreadConexao(this);
26              new Thread(tC).start();
27      }
```

Figura 32 – Método *startApp()*

Figura 33 – Tela de inserção de dados para conexão

Na classe *ThreadConexao* se encontra um exemplo de uso direto da biblioteca desenvolvida neste trabalho, como mostra a Figura 35. Esta classe é usada apenas para a conexão e criação do objeto *ins* instanciado pela classe *Statement*, e no caso deste último não é preciso enviar uma requisição, já que ele é instanciado na aplicação servidora automaticamente após a criação do objeto *con*, pela classe *Connetion*.

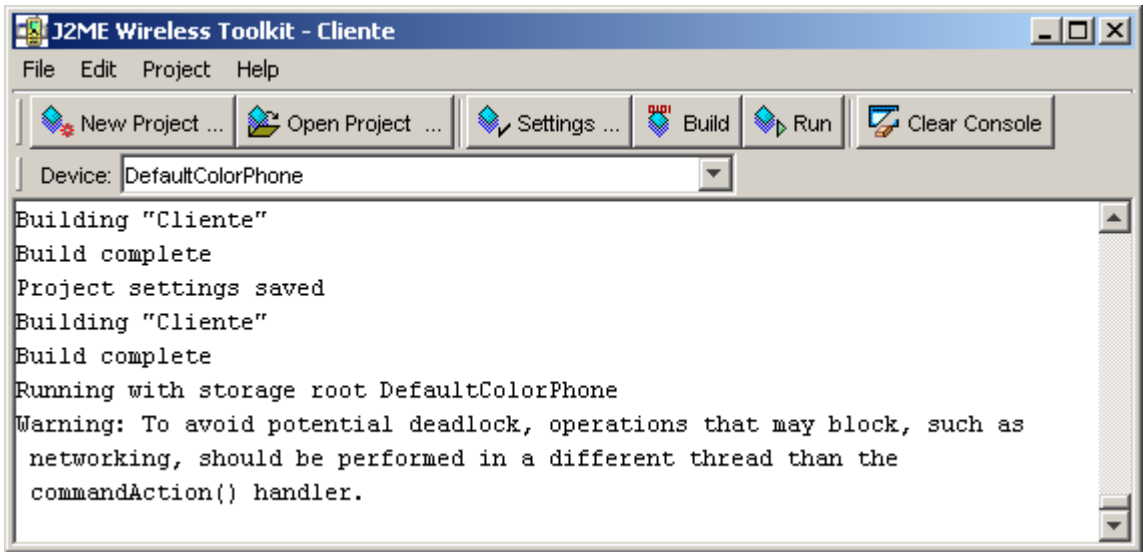

Figura 34 – Advertência sobre a falta do uso de *threads*

```
1       try
2       {       con = Connection.getConnection(url,
3                       banco.getString(), usuario.getString(),
4                       senha.getString());
5
6               ins = con.createStatement();
7
8               outForm.append("Conexao realizada");
9               outForm.addCommand(cmdContinuar);
10      }
11      catch (SQLException e)
12      {       System.out.println("Erro: " + e.getSQLState() + " " +
13                      e.getErrorCode());
14
15              outForm.append("Erro: " + e.getSQLState() + " " +
16                      Integer.toString(e.getErrorCode()));
17              outForm.append("\n");
17      outForm.addCommand(cmdSair);
18      }
```

Figura 35 – Conexão por meio da biblioteca desenvolvida

Pode ser observado na Figura 35 que o método *getConnetion()* foi colocado na classe *Connection*, diferente da API JDBC, na qual este método se encontra na classe *DriverManager*. Há esta diferença pois a classe *DriverManager* é usada para gerenciar funções do *driver* usado para acessar o banco de dados, característica esta que não é necessária na biblioteca desenvolvida neste trabalho (SUN, 2006).

Como dito anteriormente, para exemplificação do uso desta biblioteca, a aplicação cliente possui mais duas *threads*: *ThreadConsulta* e *ThreadInsere*. Essencialmente, o método *run()* da classe *ThreadConsulta* é implementado com o código que realiza uma consulta no banco de dados definido pela conexão, de acordo com a Figura 36. No banco de dados foi criada uma tabela chamada “cliente” que contém apenas duas colunas: uma denominada “numero” do tipo inteiro e outra chamada “nome” do tipo varchar.

```
1       try
2       {       result = ins.executeQuery("select * from cliente");
3
4               while (result.next())
5               {       numero = result.getInt(1);
6                       System.out.println("numero: " + numero);
7
8                       nome = result.getString(2);
9                       System.out.println("nome: " + nome);
10
11                      outForm.append(Integer.toString(numero));
12                      outForm.append(nome);
13
14                      outForm.append("\n");
15              }
16      }
17      catch (SQLException e)
18      {       System.out.println("Erro: " + e.getSQLState() + " " +
19                      e.getErrorCode());
20
21              outForm.append("Erro: " + e.getSQLState() + " " +
22                      Integer.toString(e.getErrorCode()));
23
24              outForm.append("\n");
25      }
```

Figura 36 – Consulta por meio da biblioteca desenvolvida

Para um objeto instanciado com a classe *ThreadInsere* cabe a inserção de dados

numa tabela do banco de dados da conexão atual. Como pode ser visto nas Figuras 36 e 37, a tabela usada de exemplo é denominada “cliente” e a linguagem SQL é usada normalmente.

No caso de uma conexão realizada com sucesso pelo objeto *con* instanciado pela *ThreadConexao*, a aplicação cliente emite a saída ilustrada na Figura 38. O acontecimento de algum erro também é comunicado, como por exemplo, a aplicação servidora pode não ser encontrada ou a conexão ou criação de um dos objetos pode não ser possível.

```
1       try
2       {       ins.executeUpdate("insert into cliente values (90,
3                       'Teste')");
4
5               outForm.append("Comando efetuado com sucesso");
6               outForm.append("\n");
7       }
8       catch (SQLException e)
9       {       System.out.println("Erro: " + e.getSQLState() + " " +
10                      e.getErrorCode());
11
12              outForm.append("Erro: " + e.getSQLState() + " " +
13                      Integer.toString(e.getErrorCode()));
14
15              outForm.append("\n");
16      }
```

Figura 37 – Inserção de dados no banco por meio da biblioteca desenvolvida

Através de um menu pode-se escolher o que a aplicação cliente faz, no caso, consultar a tabela cliente ou inserir dados nela e através de um formulário, o que gera resultados que podem ser mostrados na tela do dispositivo, conforme mostram as Figuras 39 e 40.

Figura 38 – Resultado de uma conexão

Figura 39 – Resultado de uma consulta

Figura 40 – Resultado de uma inserção

CAPÍTULO 7 – CONCLUSÕES

Java é uma linguagem de programação que possui soluções prontas para serem usadas em forma de classes e métodos e estende suas características para o desenvolvimento empresarial, dispositivos móveis, computadores pessoais, etc. Ela ainda possui vasta documentação e suporte contínuo de várias empresas.

Já para o J2ME, apesar da flexibilidade que a linguagem oferece, há falta de bibliotecas prontas para serem usadas, o que acaba dificultando a implementação de funcionalidades que cumpram um papel específico, como a API JDBC, que permite acesso a diversos bancos de dados de forma padronizada.

O principal objetivo deste trabalho foi atingido, pois para o programador de uma aplicação J2ME que necessite de acesso a banco de dados basta programar como se estivesse utilizando a API JDBC.

Sobre a JSR 169 ficam duas considerações: dispositivos móveis não possibilitam que um SGBD robusto seja instalado neles mesmos e se o programador resolver utilizá-la para acessar um SGBD, necessitará desenvolver toda parte de comunicação com uma aplicação que tenha acesso ao banco de dados. Neste caso, o uso da biblioteca e da aplicação servidora desenvolvidos neste trabalho é indicado, principalmente por esconder essa complexidade do programador.

No caso da utilização da biblioteca desenvolvida neste trabalho, o desenvolvedor da MIDlet precisa implementar as classes e métodos desta biblioteca dentro de *threads*, pois há comunicação remota durante seu uso. Houve a tentativa de criar *threads* automaticamente dentro da biblioteca, mas uma MIDlet possui partes que são tratadas como *threads* pela KVM, o que torna complexa a criação de novas *threads* dentro das que já existem (SUN, 2006).

Algumas melhorias e aperfeiçoamentos podem ser feitos a fim de tornar a biblioteca e a aplicação servidora mais eficazes, como o teste de consulta de imagens do banco de dados

e carregamento na aplicação cliente por meio do método *getBlob()* que já se encontra na biblioteca. Fica também a sugestão de desenvolvimento de outras classes e métodos similares à API JDBC.

REFERÊNCIAS

ARAGÃO, Raphael D.. Java Threads. **Revista MundoJava**. Curitiba, ano 3, número 16, p. 72-80, 2006.

DEITEL, H. M; DEITEL, P. J. **Java como programar.** 4ª ed., Tradução: Carlos Arthur Lang Lisboa. Porto Alegre: Bookman, 2003. Título original: Java How to Program, 4th Edition.

FISHER, Maydene; ELLIS, Jon; BRUCE, Jonathan. **JDBC API Tutorial and Reference**. 3ª ed. Santa Clara: Addison Wesley, 2003.

HORSTMANN, Cay S; CORNELL, Gary. **Core Java 2: Volume I – Fundamentals**. 5ª ed. Palo Alto: Prentice Hall, 2000.

_____. **Core Java 2: Volume II - Advanced Features**. Palo Alto: Prentice Hall, 2002.

JEPSON, Brian. **Programando Banco de Dados em Java**. Tradução Miguel Luis Cabrera. São Paulo: Makron Books, 1997. Titulo original: Java Database Programming.

JCP. **JSR 169: JDBC Optional Package for CDC/Foundation Profile**. Disponível em: <http://www.jcp.org/en/jsr/detail?id=169>. Acessado em 4 jun. 2006.

JIA, Xiaoping. **Object-Oriented Software Development Using Java**. [S.l.]. Addison Wesley, 2000.

KEOGH, James. J2ME: **The Complete Reference**. McGraw-Hill/Osborne, 2003.

MIMER. **Java programming for mobile phones with Mimer SQL**. Disponível em: http://developer.mimer.com/howto/howto_43.htm. Acessado em 08 dez 2006.

MUCHOW, John W.. **Core J2ME Technology & MIDP**. Prentice Hall, 2001.

PAULA, Gustavo Eliano de. Java ME vs. Java SE. **Revista MundoJava**. Curitiba, ano 3, número 16, p. 26-33, 2006.

POINTBASE. **PointBase Micro**. Disponível em: http://www.pointbase.com/products/micro.aspx. Acessado em 08 dez 2006.

POLIMORFISMO. In: FERNANDES, Francisco; LUFT, Celso Pedro; Guimarães, F. Marques. **Dicionário Brasileiro Globo**. São Paulo: Globo, 1995.

SILVEIRA, Paulo. **Introdução ao JDBC**. GUJ – Grupo de Usuários Java. [S.l.: s.n., s.d.]. Disponível em: <http://www.guj.com.br/java.tutorial.artigo.7.1.guj>. Acessado em 19 mar. 2006.

Sun Microsystems, Inc. **User's Guide Wireless Toolkit Version 1.0.4 Java™ 2 Platform**, Micro Edition. Palo Alto, CA, EUA, 2002.

_____. **Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification.** Disponível em: http://java.sun.com/j2se/1.4.2/docs/api/. Acessado em 20 ago 2006.

_____. **Java™ 2 Platform Standard Edition 5.0 API Specification.** Disponível em: http://java.sun.com/j2se/1.5.0/docs/api/. Acessado em 16 set 2006.

_____. **Java ME Technology APIs & Docs**. Disponível em: http://java.sun.com/javame/reference/apis.jsp. Acessado em 22 nov 2006.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 2ª ed. São Paulo: Pearson Prentice Hall, 2003.

TREMBLETT, Paul. **Instant Wireless Java with J2ME**. Berkley, CA: McGraw-Hill/Osborne, 2002.