Wendel Malta de Assis

Avaliação da tecnologia J2ME no contexto de desenvolvimento de jogos *multiplayers* para celulares

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de Bacharel.

Orientador
Prof. Ricardo Martins de Abreu Silva

Lavras
Minas Gerais - Brasil
Junho de 2003

Wendel Malta de Assis

Avaliação da tecnologia J2ME no contexto de desenvolvimento de jogos *multiplayers* para celulares

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de Bacharel.

Aprovada em *17 de Junho de 2003*

Prof. Guilherme Bastos Alvarenga

Prof. Ricardo Martins de Abreu Silva
(Orientador)

Lavras
Minas Gerais - Brasil

Avaliação da tecnologia J2ME no contexto de desenvolvimento de jogos *multiplayers* para celulares
Resumo

Atualmente, a telefonia celular encontra-se em uma fase de grande expansão, momento em que estão surgindo novas tecnologias de transmissão e aparelhos cada vez mais modernos e com maiores funcionalidades. Estes novos aparelhos permitem o desenvolvimento de aplicações mais complexas, como os jogos.

O objetivo deste trabalho consiste em apresentar as dificuldades e facilidades encontradas ao se utilizar a plataforma J2ME no desenvolvimento de um jogo *multiplayer* para celulares, chamado “Alea Jacta Est´´, e, em seguida, avaliá-la de um modo geral com relação ao desenvolvimento deste tipo de aplicação.

Evaluation of the J2ME technology in the context of development of *multiplayers* games for cellulars
Abstract

Currently, the cellular telephony is on a great expansion phase, contributing to the sprouting of new transmission technologies and moderns devices with more functionalities. These new devices allow the development of complex applications, like games.

The purpose of this work consists of the presentation of difficulties and easiness, found during the use of J2ME platform on the development of a multiplayer game for cell phones, named “Alea Jacta Est“, and after, evaluate it on a general way with the development of this kind of application.

Sumário

1 Introdução **3**

2 Tecnologia Java para o segmento de aplicações *wireless* **9**

2.1 Comunicações *wireless* 10
2.2 Dispositivos móveis embutidos 10
2.3 A tecnologia Java 11
2.3.1 A linguagem de programação Java 12
2.3.2 A plataforma Java 14
2.3.3 A plataforma Java na atualidade 16
2.4 Java 2 Platform, Micro Edition (J2ME) 18
2.4.1 Por que usar J2ME? 18
2.4.2 Um pouco de história 19
2.4.3 Arquitetura 21
2.4.4 J2ME *Connected Limited Device Configuration* (CLDC) . 25
2.4.5 Os *profiles* para o CLDC 30
2.4.6 J2ME *Connected Device Configuration* (CDC) 50
2.4.7 Os *profiles* para o CDC 51
2.4.8 Especificações adotadas pelos celulares 53
2.4.9 Distribuindo a aplicação 53
2.5 O escopo da tecnologia *wireless* em Java 56
2.6 Por que usar a tecnologia Java para o desenvolvimento de aplicações wireless? 57
2.7 Jogos *multiplayers* 58
2.8 Conexões via *sockets* 60
2.9 *Threads* de execução 62

3 Estudo de caso: avaliação da plataforma J2ME na implementação do jogo "Alea Jacta Est" 65
3.1 O jogo "Alea Jacta Est" 66
3.2 Ferramentas utilizadas no desenvolvimento do jogo "AleaJactaEst" 67
3.2.1 Editores de código 67
3.2.2 Emuladores 68
3.2.3 Criação de aplicações 69
3.3 O desenvolvimento do jogo "Alea Jacta Est" 73
3.3.1 A interface 77
3.3.2 A dinâmica do jogo 89
3.3.3 A arquitetura de comunicação 93

4 Resultados e discussões 101
4.1 Vantagens na utilização do *profile* MIDP 1.0 para a construção de jogos *multiplayers* 101
4.2 Desvantagens na utilização do *profile* MIDP 1.0 para a construção de jogos *multiplayers* 103

5 Conclusões 105
5.1 Conclusões 105
5.2 Sugestões para trabalhos futuros 106

Lista de Figuras

1.1 Principais usos da Internet Móvel pelos usuários atuais. 4
1.2 Conteúdos mais procurados pelos usuários de dispositivos móveis. 4
1.3 Previsões de crescimento no número de usuários das telefonias móvel e fixa no Brasil. 5
1.4 Previsões de crescimento do mercado de jogos para celulares. . . . 7

2.1 Processo de compilação e execução de um programa na tecnologia Java. 13
2.2 Um exemplo da portabilidade de Java. 14
2.3 A estrutura da plataforma Java. 16
2.4 A plataforma Java 2 e seus dispositivos alvos. 18
2.5 Pilha de *software* genérica a ser implementada por um dispositivo J2ME. 23
2.6 Classes que fazem parte do *Generic Connection Framework (GCF)*, segundo a especificação CLDC. 27
2.7 Abertura de conexão utilizando o método `open` da classe `Connector`. 28
2.8 Transições de estados em uma MIDlet. 32
2.9 Trecho de código ilustrando o corpo de uma MIDlet. 33
2.10 Classes que fazem parte do pacote `javax.microedition.lcdui`, para manipulação de interfaces. As setas em vermelho indicam uma relação de uso da classe que aponta a seta pela classe alvo dela. A classes que não apresentam nenhuma relação de herança e não são interfaces herdam diretamente da classe `Object` do pacote `java.lang`. 35
2.11 Código que apresenta a utilização de alguns elementos da interface de alto nível. 37

2.12 Resultado da execução, em um emulador de celular, do código da Figura 2.11. Em **a** vemos o celular apresentando as aplicações disponíveis para serem ativadas e em **b** a saída do código. 38
2.13 Modificações feitas no GCF pelo *profile* MIDP versão 1.0. As novas interfaces implementadas encontram-se circuladas de vermelho. 40
2.14 Processo de construção de fundos com a utilização de *tiles* em uma `TiledLayer`. 46
2.15 Processo de construção de imagens de fundo utilizando uma `TiledLayer`. 47
2.16 Hierarquia de classes do pacote `javax.microedition.lcdui.game`. As classes circuladas de azul não fazem parte do pacote de jogos, apenas são extendidas por algumas de suas classes. 48
2.17 Modificações feitas no GCF pelo *profile* MIDP versão 2.0. As novas interfaces incluídas por esta especificação encontram-se circuladas de vermelho. 49
2.18 *Profiles* que fazem parte da configuração CLDC. 51
2.19 Relação entre o CLDC e CDC. 52
2.20 *Profiles* baseados na configuração CDC e suas relações. 53
2.21 Pilha de *software* atualmente implentada pelos aparelhos celulares. 53
2.22 Pilha de *software* a ser implementada pelos aparelhos celulares. . . 54
2.23 Funcionamento de um JAM. 54
2.24 Over-the-air provisioning (OTA). 56
2.25 Arquitetura de comunicação ponto a ponto. 59
2.26 Arquitetura de comunicação baseada em servidor e suas variações. 60
2.27 Requisição de conexão de um cliente a um servidor via *sockets*. . . 61
2.28 Conexão estabelecida entre um cliente e um servidor via *sockets*. . 61
2.29 Modo de execução de uma *thread* única. 62
2.30 Execução de múltiplas *threads* em um único programa. 63

3.1 O ambiente de trabalho do editor JEdit 4.0. 68
3.2 Organização de uma suíte de MIDlets. 72
3.3 *KToolBar*, principal ferramenta do J2ME *Wireless Toolkit*. 73
3.4 Processo de construção e teste de uma MIDlet. 74
3.5 Processo de empacotamento de uma MIDlet. 74
3.6 Classes que fazem parte do jogo “Alea Jacta Est”. 76
3.7 Pilha de especificações J2ME utilizada na implementacao do “Alea Jacta Est”. 76

3.8 Modelo navegacional das interfaces desenvolvidas para o jogo "Alea Jacta Est". A classe `TelaInicial` é corresponde ao início do fluxo. 79
3.9 Interface definida pela classe `TelaInicial`. 80
3.10 Trecho de código enfocando a definição da classe *TelaInicial*. . . . 80
3.11 Trecho de código enfocando a inserção de um `ImageItem` pela classe *TelaInicial*. 81
3.12 Trecho de código que mostra a criação de um `StringItem`. . . . 81
3.13 Trecho de código mostrando como inserir `Commands` em um `Form`. 82
3.14 Interface definida pela classe `TelaIniciarJogo`. 82
3.15 Trecho de código da classe `TelaIniciarJogo`. 83
3.16 Classe `TelaJogo`, na qual toda a interação entre os jogadores ocorre. 84
3.17 *Tiles* utilizados na construção do mapa do jogo. 85
3.18 *Sprites* disponibilizados para a interação com o jogo. Os números entre parênteses correspondem ao valor de ataque, defesa e tempo de construção, respectivamente. 85
3.19 *Tiles* utilizados na construção do mapa do jogo. 87
3.20 Estrutura de classes criadas para utilizar a idéia de *sprite*, *tile* e janela de visualização. 88
3.21 Fluxo de execução criado para a *thread* de manipulação de eventos de teclado no "Alea Jacta Est". 90
3.22 Mensagens trocadas entre o jogador Wendel e o servidor durante o processo de criação de uma cidade com sucesso. 90
3.23 *Loops* de controle de execução definidos pelos *profiles* MIDP 1.0 e 2.0. 92
3.24 Trecho de código que mostra a estrutura tradicional utilizada por uma classe em MIDP 1.0 para um *loop* de jogo. 92
3.25 Trecho de código que mostra a estrutura tradicional utilizada por uma classe em MIDP 2.0 para um *loop* de jogo. 93
3.26 Trecho do código do "Alea Jacta Est" que mostra a abertura de conexões por *sockets*. 95
3.27 Processo de construção de um cabeçalho para uma primitiva de comunicação. 97
3.28 Trecho de código mostrando a maneira como o cliente envia ou recebe mensagens. 98

3.29 Trecho de código mostrando o *loop* onde o jogador permanece até que inicie o seu turno. 99

Lista de Tabelas

3.1 Emuladores disponibilizados pelo J2ME *Wireless Toolkit* e versões de MIDP implementadas por cada um deles. 70
3.2 Características dos emuladores disponibilizados pelo J2ME *Wireless Toolkit*. 71

5.1 Resumo das conclusões obtidas com relação a utilização da plataforma J2ME no desenvolvimento de jogos *multiplayers*. 107

Capítulo 1

Introdução

Atualmente, vivemos em um mundo onde a informação é a principal ferramenta das pessoas e empresas para garantir a sua sobrevivência no mercado. Este novo paradigma iniciou-se com a chegada da Internet, em meados da década de 90. Considerada o maior repositório de informações do mundo, ela permite a troca de conhecimentos de maneira mais rápida e fácil.

Os pequenos dispositivos móveis (como o *Personal Digital Assistant* (PDA) e, recentemente, os celulares mais poderosos) e a comunicação *wireless* formam uma das principais armas das pessoas para que estas se mantenham constantemente informadas, conectadas ao mundo, onde quer que estejam e sem que tenham que carregar muito peso. Portanto, estas tecnologias foram as responsáveis por criar um mercado totalmente novo, o de desenvolvimento de aplicações para os dispositivos móveis. Alguns tipos de aplicações geralmente procuradas são acesso à informações e notícias, leitura de correio eletrônico, transações bancárias e entretenimento. O uso da Internet por estes dispositivos também tem crescido muito nestes últimos tempos, caracterizando o conceito de Internet Móvel (ou Internet sem fio).

A Figura 1.1 apresenta uma pesquisa da Revista Poder [POD2002] sobre os principais usos da Internet sem fio por parte das pessoas. A Figura 1.2 também apresenta uma pesquisa desta mesma revista com relação ao tipo de conteúdo mais desejado pelos consumidores do mercado de dispositivos móveis.

De acordo com a divisão proposta por Monica Pawlan[PAW2002], o mercado para os serviços e aplicações *wireless* é muito grande, podendo ser dividido em dois segmentos:

- *Segmento consumidor:* consiste de jogos, serviços *standalone*, aplicações de

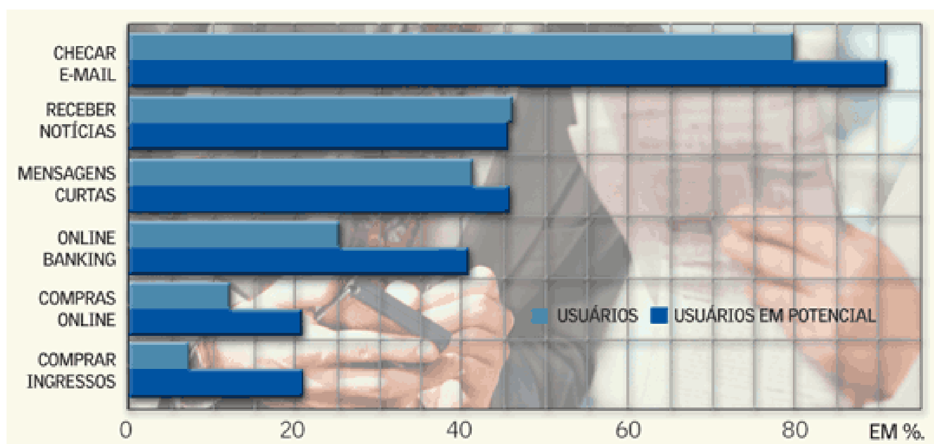

Figura 1.1: Principais usos da Internet Móvel pelos usuários atuais.

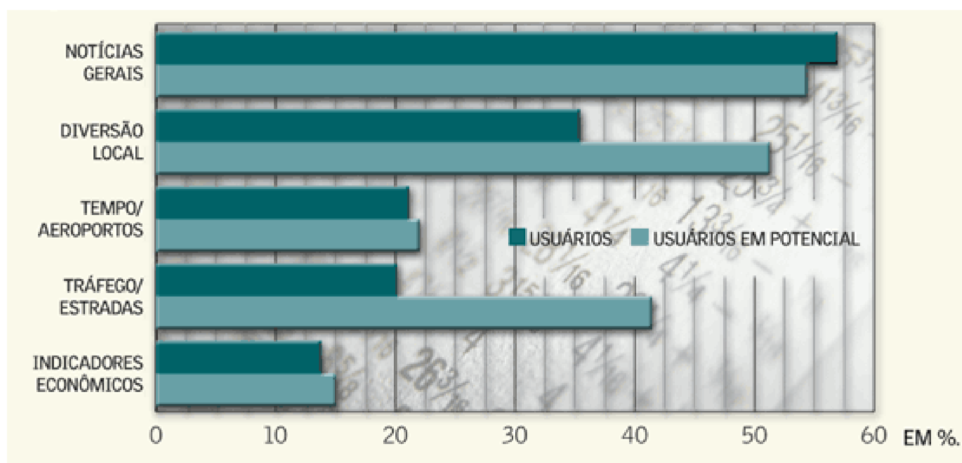

Figura 1.2: Conteúdos mais procurados pelos usuários de dispositivos móveis.

produtividade entre outras aplicações úteis. Este segmento corresponde ao tipo de aplicação que será distribuída pelas prestadoras de serviços celulares.

- *Segmento de negócios:* consiste no acesso as informações da empresa, a qualquer instante, através de aplicações *wireless* padronizadas instaladas em dispositivos móveis.

Este projeto visa atender ao segmento de mercado voltado para o consumidor e também a demanda das pessoas por aplicações de entretenimento em seus dispositivos móveis. Assim, foi avaliada a utilização da plataforma *Java 2 Platform, Micro Edition* (J2ME)[1], da *Sun Microsystems*[2], no desenvolvimento de jogos *multiplayers* para celulares.

Os primeiros celulares possuíam poucos recursos e serviam apenas para a realização e recebimento de chamadas (função básica de todo telefone celular). Os aparelhos atuais, por sua vez, possuem um maior grau de processamento tornando possível a execução de aplicações mais diversificadas, como jogos.

Para se ter uma idéia da situação atual e do alto potencial de crescimento do mercado de dispositivos móveis, em especial dos celulares, os próximos parágrafos irão apresentar alguns números muito interessantes.

Segundo o vice-presidente da Anatel, Antônio Carlos Valente, existem hoje no Brasil, aproximadamente, 32 milhões de celulares em operação [INF2002A]. A Figura 1.3 apresenta um gráfico da Anatel que ilustra as previsões de crescimento no número de usuários de telefonia móvel no país com relação a telefonia fixa.

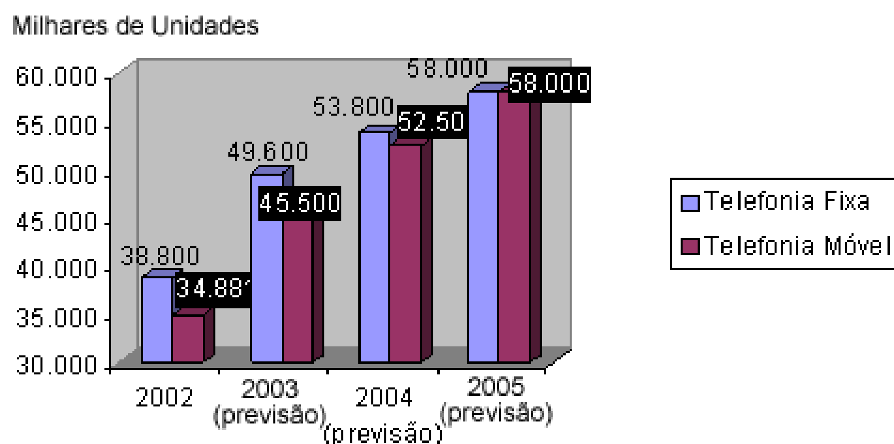

Figura 1.3: Previsões de crescimento no número de usuários das telefonias móvel e fixa no Brasil.

De acordo com dados da *Frost & Sullivan*, em 2000, o Brasil movimentou

[1]No restante desta monografia, quando se for realizar uma referência a plataforma *Java 2 Platform Micro Edition*, se utilizará a sigla J2ME.

[2]Por várias vezes, no restante desta monografia, será utilizada a abreviação *Sun* quando se for fazer uma referência a *Sun Microsystems*.

mais de 15 bilhões de dólares com seu mercado *wireless* [IBI2002].

Segundo os dados publicados pela revista InfoExame[INF2001], a Finlândia está entre os mercados de telefonia celular mais desenvolvidos do mundo, quase 80% da população têm celular, parcialmente por causa do sucesso da *Nokia*,companhia finlandesa e maior fabricante de aparelhos celulares do mundo.

Em 2001, a telefonia celular ultrapassou a telefonia fixa na América Latina com relação ao número de usuários. De acordo com um estudo da *Pyramid Research*[INF2002B], a penetração da telefonia celular chegou a 17%, enquanto a da telefonia fixa ficou em 16,9%. Nos próximos anos, a *Pyramid* acredita que a diferença entre estes índices deva aumentar ainda mais, principalmente com a entrada das novas operadoras e dos serviços 2,5G e 3G. Em 2006, a penetração do telefone celular já deverá ter ultrapassado os 30%, enquanto a da telefonia fixa ainda deverá estar na faixa dos 20%.

Uma pesquisa realizada pela *Jupiter Media Metrix*[IBI2002] revelou que o número de usuários de Internet sem fio nos EUA saltaria dos 4,1 milhões, em 2001, para 96 milhões em 2005. Estendendo para a América Latina, teria-se aproximadamente 50 milhões de usuários acessando a Internet por telefone celular até 2005, o que tornaria o uso das redes sem fio tão disseminado na região quanto o acesso baseado em PCs.

Os dados publicados no site da IBiznet[IBI2002] mostram que, na União Européia, em 2010, a previsão é de que 75% das pessoas entre 15 e 50 anos carreguem consigo um dispositivo móvel. A venda de telefones móveis com acesso à Internet deve passar de um bilhão de unidades por ano em 2004. Ainda de acordo com esta pesquisa, o Japão, em 2005, terá 96 milhões de aparelhos celulares, dos quais 94 milhões terão acesso aos serviços de Internet móvel.

Em conjunto com o crescimento do número de usuários de dispositivos móveis, aparece o mercado de jogos para estes dispositivos, em especial para os celulares. As operadoras do mundo inteiro já faturam 436 milhões de dólares ao ano com esse mercado e esperam crescer a 9,3 bilhões de dólares até 2008, de acordo com John Smedley, diretor de operações da *Sony Online Entertainment*.

Segundo David Fox[FOX2002], em dezembro de 2002, executivos de alto escalão de companhias como *Sony*, *Disney*, *Sega*, *THQ* e *Newscorp* estavam reunidos no *Mobile Entertainment Summit* em Las Vegas, Nevada. O objetivo era discutir o presente e o futuro dos jogos *wireless*. *Frost & Sullivan* aposta que o mercado de jogos móveis movimentará 9,3 bilhões de dólares até 2008. *Ovum* acredita em 4,4 bilhões até 2006. *In-Stat/MDR*, entretanto, prevê que os jogos serão “somente” 2,8 bilhões de dólares até 2006. Como visto, as estimativas apresentadas pelos parti-

cipantes variaram grandemente, mas isto não significa que este mercado não crescerá, uma vez que a pior previsão apresentada ainda revela aumento de mercado. O gráfico, elaborado pelo autor, apresentado pela Figura 1.4 ilustra as previsões de crescimento esperadas no mercado de jogos para celulares nos próximos anos.

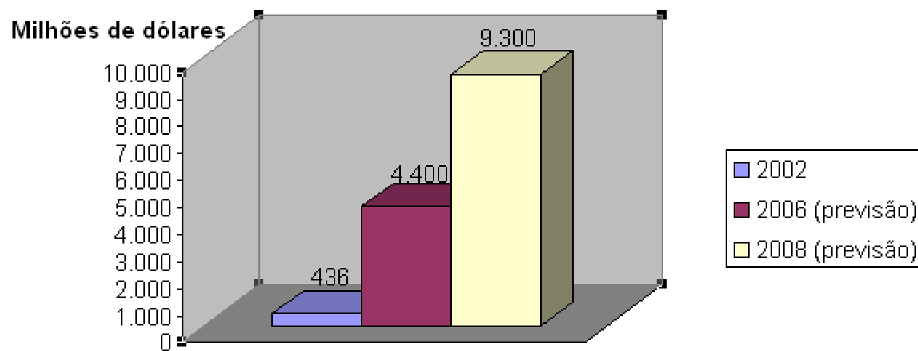

Figura 1.4: Previsões de crescimento do mercado de jogos para celulares.

Acompanhando este mercado promissor, várias empresas anunciaram o desenvolvimento de tecnologias que permitem a criação de aplicações para os dispositivos móveis. Assim, surgiram nomes no mercado como WAP, J2ME, BREW, i-mode (celulares japoneses), dentre vários outros. Cada uma dessas tecnologias visa explorar ao máximo esse mercado de dispositivos móveis com capacidade de comunicação *wireless*.

A tecnologia J2ME vem se destacando devido a sua alta portabilidade, o que permite o desenvolvimento de aplicações para um grande número de diferentes dispositivos sem maiores problemas, e pela sua grande aceitação no mercado atual. Vários fabricantes como a *Nokia*, *Siemens* e *Motorola* já produzem aparelhos com suporte a esta tecnologia.

Para facilitar a análise da tecnologia J2ME foi proposto o desenvolvimento de um jogo modelo. Assim, iniciou-se a construção de um jogo de estratégia *multiplayer* denominado "Alea Jacta Est". Durante o desenvolvimento deste jogo foi possível relacionar as dificuldades e facilidades encontradas na utilização da tecnologia J2ME para a sua implementação e também verificar os pontos negativos e positivos desta tecnologia no desenvolvimento de jogos *multiplayers* para celulares.

Este trabalho encontra-se altamente relacionado com o projeto *"Alea Jacta Est" Um protótipo de jogo de estratégia interativo multiplayer para dispositivos*

móveis, desenvolvido pelo aluno Diego Mello da Silva do curso de Ciência da Computação, da Universidade Federal de Lavras. Deve-se ressaltar que os dois projetos são complementares, sendo que o jogo modelo "Alea Jacta Est" está melhor documentado na monografia do Diego, enquanto os detalhes da implementação e a análise da tecnologia J2ME utilizada no seu desenvolvimento encontram-se neste documento.

Esta monografia está organizada em cinco capítulos, como descrito nos tópicos abaixo:

- O **capítulo 1** corresponde a Introdução ao tema.
- O **capítulo 2** descreve, de modo completo, as principais tecnologias utilizadas neste trabalho.
- O **capítulo 3** apresenta o jogo modelo proposto para servir de base para a análise da tecnologia J2ME, bem como as etapas que foram necessárias para o seu desenvolvimento. Também serão apresentadas as diretrizes seguidas para a realização da avaliação da tecnologia e algumas análises iniciais.
- O **capítulo 4** apresenta os resultados encontrados de uma maneira mais clara e uma discussão dos mesmos.
- O **capítulo 5** apresenta as conclusões finais sobre a utilização da tecnologia J2ME para o desenvolvimento de jogos *multiplayers* para celulares e recomendações para trabalhos futuros.

Capítulo 2

Tecnologia Java para o segmento de aplicações *wireless*

Este capítulo tem por objetivo principal introduzir o leitor as tecnologias *wireless* utilizadas no projeto. Alguns outros conceitos, necessários para o entendimento do jogo modelo construído, como *sockets*, *threads* de execução e jogos *multiplayers* também serão apresentados.

A seção 2.1 e a seção 2.2 tem como objetivo apresentar de forma breve os conceitos básicos relacionados a comunicação *wireless* e aos dispositivos móveis embutidos, respectivamente. A seção 2.3 apresenta as características principais da tecnologia Java, permitindo uma melhor compreensão da plataforma J2ME e da tecnologia de *sockets* e de *threads* de execução em Java. A seção 2.4 apresenta a plataforma J2ME em grandes detalhes, pois esta é a tecnologia alvo de análise deste projeto. A seção 2.5 apresenta algumas considerações que procuram mostrar que quando falamos sobre *wireless* em Java não estamos falando apenas da plataforma J2ME. A seção 2.6 mostra os motivos que levam as pessoas a optar por Java para o desenvolvimento de soluções móveis, utilizando-a de ponta a ponta no projeto. A seção 2.7 apresenta os conceitos relacionados a jogos *multiplayers*, focando-se nas maneiras de se implementá-los. As próximas seções tem o objetivo de apresentar conceitos necessários ao entendimento do funcionamento do jogo modelo construído. Assim, a seção 2.8 apresenta os conceitos básicos de *sockets* e a seção 2.9 fala sobre *threads* de execução de um programa.

2.1 Comunicações *wireless*

O termo comunicações *wireless* pode ser definido como o conjunto de todas as formas de comunicação que não utilizam fios (ou qualquer outro sistema físico) para se propagarem.

Segundo a própria *Sun Microsystems*[SUN2003A], o campo de atuação da tecnologia de comunicação *wireless* é enorme, sendo que ela está presente nas transmissões de rádio e televisão passando pelos *pagers*, telefones móveis e nas comunicações via satélite. Outra face desta tecnologia que está crescendo rapidamente são as redes locais sem-fio (LANs *wireless* ou WLANs).

Conceitualmente, comunicações *wireless* podem ser dividas em dois tipos, área local e grande área. Um dispositivo local é similar aos chaveiros de alarme de carros com botões que travam e destravam o carro, um telefone sem fio de 900MHz, um brinquedo de controle remoto ou uma rede *Bluetooth*. Todos estes dispositivos operam sobre pequenas distâncias, tipicamente alguns poucos metros [SUN2003A].

Dispositivos *wireless* de grande área operam efetivamente sobre uma enorme área. Um *pager* ou um telefone móvel (celular) são bons exemplos. Estes dispositivos conseguem operar a grandes distâncias devido a uma rede mais elaborada. Um celular não tem muito mais poder que o controle remoto de um brinquedo. Ele possui uma rede de antenas celulares cuidadosamente instaladas que permite com que o telefone celular continue funcionando se ele estiver no limite de pelo menos uma torre [SUN2003A].

2.2 Dispositivos móveis embutidos

Produtos de consumo baseados em microprocessadores, como os alarmes, máquinas de café, televisões, ar condicionados e telefones são chamados de dispositivos embutidos (ou dispositivos embarcados) pelo fato de eles possuírem pequenos computadores dentro deles com uma função muito bem delimitada. Os celulares tradicionais são exemplos de dispositivos embutidos. Entretanto, os aparelhos mais novos já se assemelham aos computadores *deskop*, ou seja, permitem a execução de aplicações variadas.

Dispositivos embutidos de consumo e que não estão conectados a parede por um fio, mas ao invés disso usam tecnologia *wireless* para se comunicar são chamados de dispositivos móveis embutidos. Entre os vários tipos destes dispositivos podemos citar o *handheld*, telefones celulares, *pagers* e todos os tipos de *PDAs*

[PAW2002].

A maioria destes dispositivos pode se comunicar através de *links wireless*. Estes *links* podem ser estabelecidos através de luz infravermelha (IR, do inglês *infrared*) ou, mais freqüentemente, ondas de rádio (RF, do inglês *radio frequency*) [JON2002].

Um exemplo comum de dispositivo móvel que utiliza IR para estabelecer um *link* de comunicação é o controle remoto do aparelho de som ou da televisão. Os telefones celulares e dispositivos *wireless* usam RF para se comunicar e utilizam um sistema mais elaborado, baseado em antenas, que não será detalhado neste projeto [JON2002].

2.3 A tecnologia Java

A tecnologia Java surgiu como um produto indireto de um pequeno projeto, conhecido por *Green Project*, iniciado por Patrick Naughton, Mike Sheridan e James Gosling da *Sun Microsystems* em 1991. A equipe do *Green Project*, chamada de *Green Team*, era composta por 13 pessoas e foi incumbida pela *Sun* de antecipar e planejar o futuro da computação. A conclusão inicial do grupo foi que havia uma tendência que consistia da convergência de dispositivos controlados digitalmente e computadores [BYO2002][SUN2002A].

Em 1992, o grupo apresentou um trabalho *demo*, um *handheld* capaz de controlar o sistema de entretenimento de uma casa através de uma interface por toque na tela e animada. A linguagem de programação utilizada neste dispositivo foi criada por James Gosling. Gosling chamou a nova linguagem de *Oak*, devido a árvore de frente a sua janela (*Oak* em português significa carvalho) [BYO2002].

Apesar do dispositivo criado pelo grupo não ter feito grande sucesso no mercado, sendo considerado por muitos um produto a frente de seu tempo, a nova linguagem criada por Gosling destacou-se rapidamente. O *Oak* era uma linguagem orientada a objeto e capaz de trabalhar sobre redes de maneira distribuída. Além disso, possuía uma grande preocupação com a segurança, já implementando criptografia e procedimentos de autenticação [BYO2002].

Depois de várias tentativas de popularizar o *Oak* no mercado de dispositivos embutidos, o grupo finalmente conseguiu enxergar que, pelo menos na época, este não era o mercado ideal para a sua nova tecnologia, mas sim a Internet, que estava começando a ganhar destaque e também possuía as caracterísicas de redes distribuídas tão visadas pela equipe.

Em 1995, o *Oak* foi renomeado para Java e um novo trabalho *demo*, um na-

vegador chamado de WebRunner, foi desenvolvido. Ele foi apresentado por Gosling no Hollywood-Meets-Silicon-Valley juntamente com dois exemplos de *applets* animadas, aplicativos Java que executam em um navegador *web*.

O lançamento oficial ocorreu em 23 de maio de 1995 durante a conferência *SunWorld*.

Nesta época, a tecnologia Java fugiu um pouco de sua idéia original, que era trabalhar com dispositivos embutidos, e utilizou a *web* para ganhar popularidade. Entretanto, atualmente, sabe-se que seu campo de atuação é bem maior do que apenas o desenvolvimento de *applets*.

As pessoas geralmente se referem a tecnologia Java no sentido único da linguagem de programação Java, mas isto não está correto. Na verdade, trata-se de uma linguagem de programação e de uma plataforma. As próximas subseções irão explicar melhor estes dois conceitos.

2.3.1 A linguagem de programação Java

Segundo James Gosling e Henry McGilton[GOS & MCG1996], a linguagem de programação Java é uma linguagem de alto nível que tem como características principais:

- simples;
- orientada a objetos;
- independente de arquitetura;
- portável;
- interpretada;
- distribuída;
- alta-performance;
- *multithreaded*;
- robusta;
- dinâmica;
- segura.

Na maioria das linguagens de programação, pode-se compilar ou interpretar o código fonte para permitir a execução em um computador. A linguagem de programação Java é diferente neste aspecto, realizando as duas etapas. Na etapa de compilação é feita uma tradução do programa (código fonte) em uma linguagem intermediária chamada *bytecodes*, que são códigos independentes de máquina interpretados na plataforma Java. O interpretador tem a função de analisar e executar cada *bytecode* Java. Assim, a compilação é realizada apenas uma vez, enquanto que a interpretação é realizada sempre que o programa é executado. A Figura 2.1, ilustra este processo.

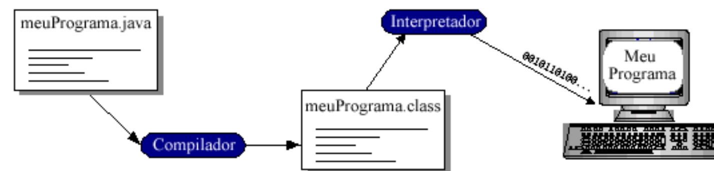

Figura 2.1: Processo de compilação e execução de um programa na tecnologia Java.

Pode-se pensar nos *bytecodes* Java como instruções em código de máquina para a *Java Virtual Machine*, em português Máquina Virtual Java. Todo interpretador Java, não importando se está integrado a um ambiente de desenvolvimento ou em um navegador que pode executar *applets*, é uma implementação desta máquina [CAM & WAL2002].

Como um programa Java é interpretado ao ser executado, isto o torna um pouco mais lento quando comparado com linguagens de programação que possuem compiladores que geram códigos nativos, isto é, códigos para uma plataforma específica. Entretanto, a *Sun Microsystems* tem melhorado cada vez mais os seus interpretadores, através de tecnologias como *just-in-time compiler*, para garantir performance sem perder portabilidade. Esta tentativa de manter a portabilidade e garantir eficiência é a principal dificuldade enfrentada por Java.

É através desta estrutura que Java garante a sua promessa de independência de plataforma. Os *bytecodes* Java realmente tornam a tão famosa frase "escreva uma vez, execute em qualquer lugar" (em inglês, *"write once, run anywhere"*) possível.

A Figura 2.2, de Mary Campione e Kathy Walrath[CAM & WAL2002], apresenta de forma clara a questão da portabilidade em Java. Pode-se compilar o programa em *bytecodes* em qualquer plataforma que possua o compilador Java e depois executá-los em qualquer implementação de máquina virtual Java. Isto significa que desde que um computador possua uma máquina virtual Java insta-

lada, o mesmo programa escrito na linguagem de programação Java pode executar em plataformas tão diferentes quanto *Windows* 2000, *Solaris* ou *iMac*.

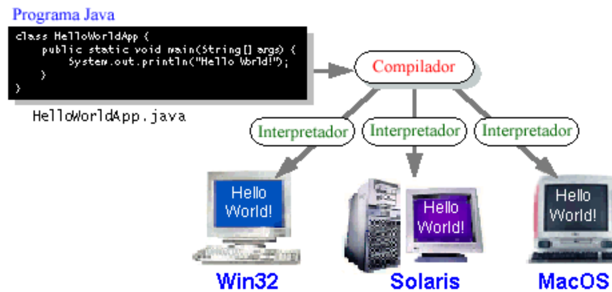

Figura 2.2: Um exemplo da portabilidade de Java.

É impossível falar da linguagem de programação Java sem compará-la com a linguagem de programação *C++*, a qual ela apresenta semelhança sintática. O objetivo aqui não é o de julgar as duas linguagens e dizer qual é a melhor, mas apenas apresentar as suas diferenças principais. A linguagem Java aloca e desaloca memória automaticamente quando o programa cria e destrói objetos, enquanto programadores que usam *C++* devem alocar e desalocar explicitamente a memória utilizada. Em Java não se utilizam ponteiros, mas apenas referências a objetos para resolver o problema da alocação dinâmica de memória. Outra diferença fundamental entre as duas linguagens é que Java inclui um mecanismo de herança conhecido como *interface*, que substitui o conceito de herança múltipla em *C++* [SUN2003A].

A sintaxe e a estrutura da linguagem Java também são muito simples, tornando fácil e rápida a construção de programas e a sua compreensão.

A especificação da linguagem Java pode ser encontrada no livro *The Java Language Specification* [GOS, JOY, STE & BRA2000].

2.3.2 A plataforma Java

Uma plataforma é o *hardware* ou ambiente de *software* no qual um programa executa. A maioria das plataformas podem ser descritas como uma combinação de sistema operacional e *hardware*. A plataforma Java difere de todas as outras por ser uma plataforma únicamente de *software* que executa no topo das outras plataformas baseadas em *hardware* [CAM & WAL2002].

A plataforma Java tem dois componentes:

- A Máquina Virtual Java (Java VM);

- A Interface de Programação de Aplicações Java (*Java Application Programming Interface* - Java API).

A máquina virtual é a base da plataforma Java e ela é portada para várias plataformas de *hardware*, mantendo assim a característica portabilidade desta tecnologia.

Quando um desenvolvedor Java escreve um programa é como se ele estivesse escrevendo para a máquina virtual e não para uma arquitetura específica. É a máquina virtual a responsável por traduzir o código criado pelo programador em um código compatível com a arquitetura em que o programa será executado. Assim, ela está na camada intermediária entre a aplicação e a plataforma. A máquina virtual também controla tarefas relacionadas com a gerência de memória, fornecendo segurança contra códigos maliciosos, e de múltiplos *threads* de execução do programa.

A especificação dizendo o que uma máquina virtual deve apresentar pode ser encontrada no livro *The Java® Virtual Machine Specification*[LIN & YEL1999].

A API Java é uma grande coleção de componentes de *software* que provê as mais variadas funcionalidades ao programador, como interface gráfica com o usuário, criptografia, conexões entre várias outras. Ela está agrupada em um conjunto de bibliotecas (pacotes) de classes e interfaces relacionadas [SUN2003A] [CAM & WAL2002].

A classe `Object` do pacote `java.lang` é a classe mãe para todas as outras definidas pela linguagem.

Um pacote da API Java padrão que foi bastante utilizado neste projeto foi o `java.net`. Este pacote, em conjunto com o `java.io`, permite uma manipulação completa de *sockets*, descritos na seção 2.8. Outro pacote utilizado foi o `java.util`, que inclui classes utilitárias (como a manipulação de vetores).

Para construir o jogo foi necessário a utilização de várias *threads* de execução em paralelo e, conseqüentemente, de um modo de sincronização entre elas. A estrutura para a manipulação de *threads* encontra-se disponível no pacote `java.lang`, que é o principal pacote da API Java. *Threads* de execução serão descritas na seção 2.9.

A Figura 2.3 apresenta a estrutura da plataforma Java.

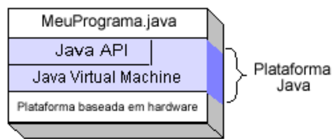

Figura 2.3: A estrutura da plataforma Java.

2.3.3 A plataforma Java na atualidade

A primeira versão de Java liberado pela *Sun Microsystems* foi chamado de Java 1.0.2. O 1.0.2 passou a referir a versão do *Java Development Kit* (JDK) que incluía tudo que era necessário para desenvolver e executar programas Java nas plataformas *Windows* e *Solaris*. Após a versão 1.1, a *Sun* passou a referir-se as versões de Java como *Java Software Development Kit* (JSDK) e a possuir um *run-time* em separado chamado de *Java Run-Time Environment* (JRE).

Segundo Enrique Ortiz[ORT & GIG2001], Java 1.0.2 foi notável por duas coisas:

- Definiu o *Abstract Windowing Toolkit* (AWT), para a criação de interfaces gráficas com o usuário;
- Definiu *applets*, uma maneira na qual os navegadores podem executar aplicativos Java de maneira segura.

A próxima versão foi a Java 1.1, que acrescentou novas características a tecnologia e corrigiu muitos problemas identificados na versão anterior. Ainda de acordo com Enrique Ortiz[ORT & GIG2001], as principais mudanças foram:

- Melhorias no AWT, através do *Swing*. Agora uma interface construída em Java é totalmente independente de plataforma;
- *Remote Method Invocation* (RMI), permitindo que dois programas Java possam "conversar" facilmente via rede;
- Serialização de objetos;
- Compilação *just-in-time*.

Durante o desenvolvimento da versão 1.2 a *Sun* resolveu fazer maiores modificações na maneira como a tecnologia era distribuída e licenciada. Assim, a

versão 1.2 foi chamada de Java 2 e uma divisão da plataforma em edições foi realizada. Esta divisão permite organizar melhor o desenvolvimento e crescimento da tecnologia Java em três grandes áreas:

- Servidores;
- *Desktops*;
- Pequenos Dispositivos.

Cada edição possui todas as ferramentas e suprimentos necessários para o desenvolvimento e execução de aplicativos. Por fim, as edições que fazem parte da atual plataforma Java 2 são:

- *Java 2 Platform, Standard Edition (J2SE)* [JSE2002], cujo alvo são as aplicações *desktop* convencionais;
- *Java 2 Platform, Enterprise Edition (J2EE)* [JEE2002], com ênfase no desenvolvimento de aplicações *server-side* para empresas com necessidade de servir seus consumidores, fornecedores e empregados com soluções sólidas e completas de negócios pela Internet. Projetada para multiusários e aplicações que tem por base a modularização, reutilização e padronização de componentes. Ela é um superconjunto da J2SE e adiciona APIs para a computação do lado do servidor;
- *Java 2 Platform, Micro Edition (J2ME)* [JME2002], é um conjunto de tecnologias e especificações desenvolvidas para pequenos dispositivos como *smart cards*, *pagers* e telefones celulares. Ela usa um subconjunto dos componentes J2SE, tais como subconjuntos das máquinas virtuais e APIs mais fracas. Representa um retorno às origens do Java.

A Figura 2.4 ilustra de maneira didática a relação entre as edições atuais da plataforma Java 2 e os dispositivos que elas buscam atender. Alguns nomes citados na figura serão explicados posteriormente neste capítulo.

As especificações para J2SE, J2EE e J2ME são desenvolvidas de acordo com o *Java Community Process* (JCP). Uma especificação começa seu ciclo de vida como uma *Java Specification Request* (JSR). Um grupo de *experts* consistindo de representantes dos interesses de companhias é formado para criar a especificação. O JSR então irá passar por vários estágios na JCP até ser terminado. Quem continua decidindo tudo é a *Sun Microsystems*, mas as empresas passam a ter acesso a revisões de produtos e podem dar as suas opiniões sobre eles. Maiores informações sobre o JCP podem ser encontradas no site oficial[ORT & GIG2003].

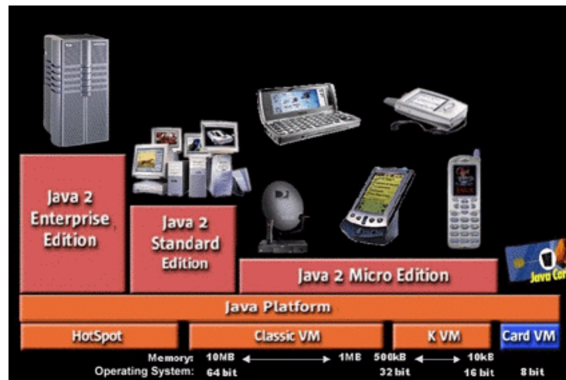

Figura 2.4: A plataforma Java 2 e seus dispositivos alvos.

2.4 Java 2 Platform, Micro Edition (J2ME)

Esta seção tem o objetivo de apresentar detalhes técnicos sobre a plataforma J2ME, focando-se na sua arquitetura e em como utilizá-la para o desenvolvimento de aplicações móveis para celulares. Mas, antes de entrar em tais detalhes, é interessante apresentar algumas justificativas para a sua escolha neste projeto e as tecnologias que surgiram antes dele e que, de certa forma, foram a sua origem.

2.4.1 Por que usar J2ME?

A plataforma J2ME é utilizada para construir aplicações *smart-client*, ou seja, voltadas para o mercado dos dispositivos com restrições de recursos.

Há alternativas ao J2ME, que não fazem parte da plataforma Java, que quase sempre executam mais rápido e tem acesso a maiores capacidades do dispositivo. No entanto, aplicações J2ME são portáveis, de acordo com a idéia principal de Java, e executam de uma maneira relativamente eficiente, apesar de serem interpretados.

Outra justificativa é o fato de J2ME ser considerado um padrão industrial, por ser definido por uma comunidade de desenvolvimento (o JCP) e ser suportado pela maioria dos vendedores de dispositivos.

Segundo Enrique Ortiz[ORT & GIG2001] e John Muchow[MUC2002], as vantagens de se utilizar J2ME podem ser resumidas em:

- O J2ME é uma plataforma portável, o que significa que um programa pode ser executado em aparelhos de diversos fabricantes sem nenhuma alteração;

- J2ME é suportado por vários fabricantes: *Motorola*, *RIM*, *Ericsson*, *Nokia*, *Panasonic*, *Nextel* entre outros;
- Suporte a aplicações *standalone* e baseadas em servidor;
- Alta integração com as outras plataformas Java;
- Utiliza a linguagem Java, a qual é fácil de aprender e utilizar para a escrita de códigos;
- J2ME é baseado em padrões abertos e é definido pelo JCP.

Com a introdução do J2ME, os pequenos dispositivos de consumo, como celulares, podem executar aplicações que não sejam somente aquelas que vem de fábrica.

2.4.2 Um pouco de história

O J2ME não é a primeira aventura da *Sun Microsystems* no mundo dos dispositivos embutidos e dos *handhelds*, mas sim uma volta ao objetivo inicial da empresa ao iniciar o projeto que deu origem ao Java. Ele é o descendente das primeiras plataformas da *Sun* para pequenos dispositivos: a parte *Oak* do projeto *Green* (no início de 1990), *JavaCard* (1996), *PersonalJava* (1997), *EmbeddedJava* (1998) e, mais recentemente, o *Spotless System* e o KVM (1999). Hoje, a plataforma J2ME atinge todos os tipos de dispositivos eletrônicos, dos *low-end* aos *high-end*, com uma única plataforma Java. O *JavaCard* ainda está separado do J2ME, embora altamente relacionado a ele[1].

Em seguida, serão apresentados maiores detalhes das tecnologias anteriores ao J2ME citadas acima.

***Oak* - O projeto *Green*:** o surgimento da tecnologia Java deu-se com o *Oak*, uma linguagem de programação orientada a objetos, independente de máquina e um ambiente para a programação de dispositivos de consumo eletrônicos. O *Oak* foi uma linguagem interpretada e projetada para executar em dispositivos com restrições de recursos. Mas, como já apresentado, seu surgimento aconteceu em um mercado que não estava preparado para seus novos ideais. A *Sun*, por sua vez, não mais dedicou esforços na tecnologia Java para dispositivos eletrônicos até a chegada do *PersonalJava*.

[1] Esta seção foi baseada no artigo de Enrique Ortiz[ORT2002A] sobre J2ME, onde o autor dedica um pouco do seu espaço para falar sobre o mundo *wireless* da *Sun* antes da chegada do J2ME e como se deu o seu surgimento.

*PersonalJava***:** esta nova tecnologia buscou atingir os dispositivos com capacidade de conexão a algum tipo de rede e com interface gráfica, tais como os PDAs e outros dispositivos baseados na web.

O ambiente de aplicações *PersonalJava*, baseado primeiramente nas APIs do JDK1.1 (com poucos pacotes da JDK1.2), requer suporte completo tanto para a especificação da linguagem Java como da máquina virtual Java. A atual e talvez a última versão da especificação do *PersonalJava* é a versão 1.2a.

Esta tecnologia deixará de surgir como uma parte separada da plataforma Java, pois na próxima revisão ela será incluída no *Personal Profile*, uma parte do J2ME que será discutida posteriomente neste capítulo.

*EmbeddedJava***:** o ambiente de aplicações *EmbeddedJava* foi criado com o objetivo de atingir os dispositivos embutidos com funcionalidade dedicada e restrições de memória, tais como dispositivos de controle de processos ou automotivos (como um sistema de localização). Baseado na JDK1.1, este ambiente de execução é muito similar ao ambiente de execução *PersonalJava*, mas ao invés de definir um subconjunto específico da plataforma Java ele permite a companhias ou indivíduos licenciarem a tecnologia para o uso somente dos componentes necessários para um dispositivo em particular. A última versão do *EmbeddedJava* é a versão 1.1.

As especificações *EmbeddedJava* deixam que o licenciado escolha quais são as características da plataforma Java que eles querem dar suporte em seus dispositivos. A idéia é que eles não podem expor estas características para uso por uma terceira parte, somente os seus próprios desenvolvedores podem escrever aplicações que executem neste dispositivo.

A *Sun* descontinuou a especificação *EmbeddedJava* e parou de oferecer suporte para ela, aconselhando àqueles que usam EmbbededJava a procurarem a configuração e *profile* J2ME que mais se adequarem as suas necessidades[2].

*JavaCard***:** a tecnologia *JavaCard* é complementar a do J2ME. Ela adapta a plataforma Java para o uso em *smart cards*, um ambiente muito especializado, com severas restrições de memória e processamento, não adequado para a programação de propósito geral. Um típico dispositivo *JavaCard* tem uma CPU de 8 a 16 bits, de 1 a 5MHz e com memória da ordem de 1KB de RAM e 32KB de memória não volátil (EEPROM ou flash).

Aplicações comuns da tecnologia *JavaCard* (chamadas *applets JavaCard*) incluem identificação digital, armazenamento seguro e cartões *Subscriber Identity Module* (SIM), que podem ser inseridos em diferentes celulares para fazer ligações utilizando a própria conta pessoal. Os celulares GSM utilizam este tipo de

[2]Os conceitos de configuração e *profile* em J2ME serão explicados posteriormente neste capítulo.

cartão para a identificação do cliente.

O *Spotless System* **e a KVM:** A máquina virtual K (KVM), o resultado de um projeto de pesquisa da *Sun* conhecido por *Spotless System*, foi lançada para o público em 1999. Este projeto adaptava a tecnologia Java para dispositivos com sérias restrições de recursos de uma maneira diferente do ambiente de aplicações *PersonalJava*, mas um pouco similar ao ambiente de execução *JavaCard*.

Como descrito no *"The Spotless System"*, um artigo técnico de Antero Taivalsaari, Bill Bush e Doug Simon, o time de pesquisas tinha o seguinte objetivo: "construir a menor máquina virtual Java possível e completa, que deveria suportar um conjunto completo de *bytecodes*, *class loading*, bibliotecas padrões não gráficas e suporte a arquivos de classe básicos para pequenas aplicações". A equipe concentrou-se na portabilidade e tamanho de código ao invés de rapidez na execução.

Quando foi publicada, a KVM foi bem recebida pela comunidade Java. Hoje a Spotless System está envolvida na J2ME *Connected Limited Device Configuration*, que será apresentada na próxima seção.

2.4.3 Arquitetura

A plataforma J2ME não define uma nova linguagem. Ela faz uma adaptação da tecnologia Java existente para *handhelds* e outros dispositivos embutidos.

J2ME mantém a compatibilidade com a J2SE sempre que possível. De fato, ele procura ao mesmo tempo remover partes desnecessárias das outras plataformas e definir novas classes para as extremas limitações dos pequenos dispositivos [ORT2002A].

Ao contrário do J2SE, J2ME não é uma peça de *software* e nem é uma especificação única. Ao invés disso, ele é uma coleção de tecnologias e especificações que foram projetadas para diferentes partes do mercado de pequenos dispositivos. [SUN2003A].

O J2ME possui objetivos diferentes quando comparado com J2SE e J2EE, resultando em uma arquitetura muito diferente. Segundo James White e David Hemphill[WHI & HEM2002], os objetivos chave que guiam esta arquitetura são:

- Proporcionar suporte a uma grande variedade de dispositivos com diferentes capacidades. Estes dispositivos freqüentemente variam em características como interface com o usuário, armazenamento de dados, conectitividade a rede, largura de banda, memória, consumo de energia, segurança e requisitos de desenvolvimento;

- Proporcionar uma arquitetura que possa ser otimizada para dispositivos pequenos e com alta restrição de recursos;
- Foco em dispositivos que podem ser altamente personalizados, freqüentemente utilizados apenas por uma única pessoa;
- Proporcionar conectitividade a rede através de uma grande variedade de serviços e capacidades de acesso. A conectitividade a rede é fundamental para os dipositivos que fazem parte do campo de atuação de J2ME;
- Proporcionar uma maneira de entregar as aplicações e dados sobre uma rede, principalmente *wireless*, para o cliente;
- Proporcionar uma maneira de se desenvolver aplicações para dispositivos sem ter que se preocupar com o modelo e fabricante para o qual se está desenvolvendo;
- Maximizar a atuação de Java entre as várias plataformas possíveis e, ao mesmo tempo, utilizar das vantagens únicas de cada dispositivo.

Para atender a estes objetivos o J2ME foi dividido em configurações e *profiles*[3].

Uma configuração define o ambiente de execução J2ME básico. Este ambiente inclui uma máquina virtuale um conjunto de classes derivadas da plataforma J2SE. Cada configuração está voltada para uma família específica de dispositivos que possuem capacidades similares. Uma configuração, por exemplo, pode ser projetada para dispositivos que tem menos que 512KB de memória e uma conexão intermitente a rede.

Um *profile* é construído sobre uma configuração de maneira a extendê-la através da adição de APIs mais específicas para criar um ambiente completo para a construção de aplicações. Em outras palavras, os *profiles* proporcionam classes que estão voltadas a usuários de um tipo de dispositivo mais específico que o abrangido por uma configuração e cuja algumas funcionalidades (como a interface com o usuário) não foram cobertas por ela. Nem todos os dispositivos suportam todos os *profiles*. No Japão, por exemplo, a NTT DoCoMo lançou um grande número de celulares baseados em CLDC, mas com seu próprio *profile*. Assim, aplicações escritas para estes dispositivos não irão trabalhar com celulares que não foram desenvolvidos pela NTT DoCoMo [ORT & GIG2001].

[3] O termo *profiles*, apesar de ser em inglês, será altamente utilizado neste texto, devido ao seu uso consagrado, ao invés de perfis, sua tradução para o português.

As APIs adicionadas por um *profile* a uma configuração são direcionadas a uma certa classe de dispositivos. Assim, um *profile* para o mercado de telefones celulares é separado de um direcionado para o mercado de organizadores pessoais, mas todos eles trabalham com a mesma configuração de dispositivo móvel.

Embora uma configuração descreva uma máquina virtual e um conjunto básico de APIs, ela não é suficientemente específica para possibilitar a construção de aplicações completas. *Profiles* geralmente incluem APIs para o ciclo de vida da aplicação, interface com o usuário e armazenamento persistente.

Resumindo, um dispositivo que venha a utilizar a tecnologia J2ME deve implementar a pilha de *software* genérica apresentada na Figura 2.5. Esta figura não apresenta as APIs específicas do fabricante, já que estas não fazem parte das configurações e *profiles* padrão. Essas APIs devem ser pensadas como extensões do próprio fabricante aos *profiles* implementados. É claro que um desenvolvedor que resolva optar por utilizar algum pacote desta API estará perdendo portabilidade [ORT2002A].

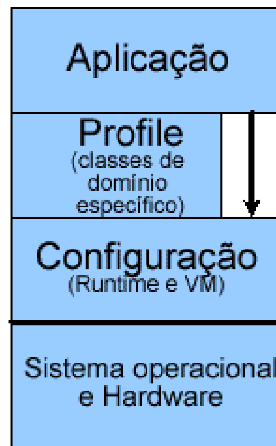

Figura 2.5: Pilha de *software* genérica a ser implementada por um dispositivo J2ME.

Atualmente, duas configurações estão definidas: a *Connected Device Configuration*(CDC)[4] e a *Connected Limited Device Configuration* (CLDC)[5]. Ambas

[4] No restante desta monografia, quando se utilizar a sigla CDC será para indicar uma referência a *Connected Device Configuration*

[5] No restante desta monografia, quando se utilizar a sigla CLDC será para indicar uma referência a *Connected Limited Device Configuration*

objetivam cobrir dispositivos com alguma capacidade de conexão, não importando se é um *link* fixo de alta velocidade ou um *link wireless* de baixa velocidade.

O CLDC[CLC2003] abrange os pequenos dispositivos: telefones celulares, *Personal Digital Assistants* (PDAs) e *pagers* interativos. Estes dispositivos formam uma classe de produtos que possuem limitações sérias de memória, processamento, consumo de energia e largura de banda.

O CDC[CDC2003] foi projetado para dispositivos mais robustos, em termos de memória e poder de processamento, e com melhor conexão a rede. Alguns PDAs (como os da série *Pocket PC* da *Microsoft*) e *laptops* são bons exemplos de dispositivos CDC.

A linha que separa os dispositivos CDC e CLDC não é muito clara, fazendo com que em algumas situações, esta distinção seja feita a critério do fabricante do aparelho. Segundo Enrique Ortiz[ORT & GIG2001], talvez a principal diferença entre o CDC e o CLDC está no fato de que o primeiro exige uma máquina virtual compatível com a implementada pelo J2SE. A *Sun* lançou uma nova máquina virtual Java para a configuração CDC, a Compact VM (CVM), que é mais portável e eficiente que a máquina virtual padrão.

Além das configurações e dos *profiles*, existem os chamados pacotes opcionais. Segundo Eric Giguere [GIG2002], um pacote opcional é um conjunto de APIs, mas não chega a ser como um *profile*. Eles não definem um ambiente de execução completo e são sempre utilizados em conjunto com uma configuração ou um *profile*.

Os pacotes opcionais proporcionam suporte a características de dispositivos que não são tão universais a ponto de serem incluídas em um perfil ou tem que ser compartilhadas por diferentes especificações. Estes pacotes também são gerenciados de acordo com as normas do Java Community Process.

Alguns pacote opcionais atualmente disponíveis são:

- **JSR 66: RMI Optional Package.** *Remote method invocation* (RMI), uma característica do J2SE que permite que objetos escritos em Java executando em uma máquina virtual possam chamar métodos Java de objetos que estão executando em outra máquina virtual. Este pacote é um subconjunto da especificação padrão e está disponível para *profiles* baseados em CDC;

- **JSR 120: Wireless Messaging API.** Proporciona o envio e recebimento de mensagens através do serviço *Short Message Service* (SMS);

- **JSR 135: Mobile Media API.** Proporciona a capacidade de reprodução e gravação de conteúdo multimídia;

- **JSR 169: JDBC for CDC/FP.** JDBC é a maneira através da qual as aplicações Java padrões se comunicam com banco de dados relacionais. Este pacote cria uma versão otimizada e reduzida do JDBC padrão para o *profile* CDC.

As próximas subseções apresentam as configurações e *profiles* mais importantes que integram a plataforma J2ME [6].

2.4.4 J2ME *Connected Limited Device Configuration* (CLDC)

O CLDC não requer muitos recursos de sistema, podendo ser executado em dispositivos com 128KB ou mais de memória não volátil (persistente) e 32KB ou mais de memória volátil. Dispositivos CLDC requerem acesso a qualquer tipo de conexão de rede, por isso o nome *connected device*, embora ela possa ser intermitente e de baixa velocidade. Geralmente, os processadores alvos deste tipo de configuração vão de 16 a 32 bits.

O CLDC define requisitos para o ambiente Java. O primeiro é o suporte completo a linguagem Java, exceto por algumas diferenças. Estas diferenças, enumeradas por Enrique Ortiz[ORT & GIG2001], são:

- **Suporte inexistente a pontos flutuantes.** Tipos de ponto flutuante ou constantes não são suportados, assim como nenhuma das classes do núcleo do J2SE que trabalham especificamente com tais valores, como a `java.lang.Float` e a `java.lang.Double`. Em conseqüência, métodos que recebem ou retornam pontos flutuantes também foram retirados;

- **Sem suporte a finalização de objetos.** Para simplificar a tarefa do coletor de lixo automático, o método `finalize` foi removido de `java.lang.Object`. O coletor de lixo irá simplesmente atuar sobre objetos não referenciados;

- **Erros de execução são tratados de uma maneira dependente da implementação.** Erros de execução são exceções lançadas pela máquina virtual e que são subclasses de `java.lang.Error`. O CLDC define somente

[6] As características da configuração CLCD e das duas versões do *profile* MIDP serão apresentadas em maiores detalhes, pois esta organização é a mais adequada para o desenvolvimento de aplicações para celulares. Como o objetivo principal deste projeto é analisar a tecnologia J2ME no desenvolvimento de jogos *multiplayers* para celulares, as análises, discussões e conclusões serão feitas sobre esta organização. Assim, justifica-se a importância de defini-las corretamente e de maneira ampla.

três classes de erros: `java.lang.Error`, `java.lang. OutOfMemoryError` e `java.lang.VirtualMachineError`. Qualquer outra exceção é tratada pela máquina virtual Java de uma maneira dependente da implementação, a qual geralmente representa o término da aplicação.

O segundo requisito é o suporte completo a máquina virtual Java exceto pelas seguintes diferenças, também enumeradas por Enrique Ortiz[ORT & GIG2001]:

- **Suporte inexistente a pontos flutuantes.** Se não existe suporte na linguagem, não há necessidade de se prover suporte na máquina virtual;
- **Sem suporte a finalização de objetos e referências fracas.** Para simplificar a tarefa do coletor de lixo automático;
- **Sem suporte a JNI ou qualquer outra interface de alto nível que depende delas.** Em particular, não há serialização de objetos em CLDC. A máquina virtual pode ter uma interface nativa, mas não é requerido e não deve ser uma interface padrão J2SE;
- **Sem suporte a grupo de *threads*.** *Threads* são suportadas, mas grupos de *threads* não são. A máquina virtual pode escolher implementar *threads* deixando-as ao cuidado do sistema operacional ou por realizar uma própria troca de contexto;
- **Tratamento de erros definidos pela implementação.** De acordo como foi apresentado anteriormente;
- **Verificação de classes é realizada de maneira diferente.** O processo padrão de verificação de classes é computacionalmente muito exigente, por isso, uma alternativa foi definida. O sistema alternativo transfere a maior parte deste processo para um passo de pré-verificação que ocorre no computador *desktop* e não no dispositivo. O aparelho ainda executa uma segunda etapa de pré-verificação, mas muito mais simples que a anterior.

O terceiro requisito é que todas as classes que são escritas ou herdadas de J2SE devem ser subconjuntos das classes da versão J2SE 1.3. Métodos podem ser omitidos, mas nenhum método público novo ou conjunto de dados pode ser adicionado.

O requisito final é que as classes definidas pelo CLDC e seus *profiles* devem estar no pacote `javax.microedition` ou em seus subpacotes. Desta maneira torna-se fácil identificar classes que são específicas do CLDC.

O CLDC inclui classes e interfaces herdadas de três pacotes J2SE:

- `java.lang`;
- `java.io`;
- `java.util`.

Como pode ser notado, a maioria das classes J2SE foram excluídas, inclusive pacotes importantes, como `java.awt` (interface gráfica com o usuário), `java.net` (utilitários para acesso a rede) e `java.sql` (utilização de consultas SQL). Até mesmo nos pacotes que foram incuídos, algumas classes foram retiradas, juntamente com muitos métodos das classes que ficaram.

O pacote `java.io` sofreu uma otimização enorme, pois apenas as suas definições de classe já excedem o limite de espaço definido para dispositivos CLDC. Por isso, apenas as classes mais utilizadas permaneceram na versão da plataforma J2ME, mas ainda tiveram que sofrer algumas modificações em sua estrutura.

Além dos pacotes citados anteriormente, um único pacote foi escrito especialmente para CLDC, o `javax.microedition.io`. As classes que fazem parte deste pacote são ditas parte do *Generic Connection Framework* ou apenas GCF.

O GCF abstrai os conceitos de arquivos, *sockets*, *requests* HTTP e outros mecanismos de entrada de uma maneira simples, voltada para os dispositivos com sérias restrições de recursos. Em outras palavras, o GCF proporciona a mesma funcionalidade das classes `java.io` e `java.net`.

É importante ressaltar que o CLDC apesar de definir o GCF não obriga o suporte para qualquer tipo de protocolo. O suporte a protocolos é realizado a nível de *profile*.

A Figura 2.6 ilustra o conjunto de interfaces que fazem parte do pacote `javax.microedition.io` definidas pela CLDC.

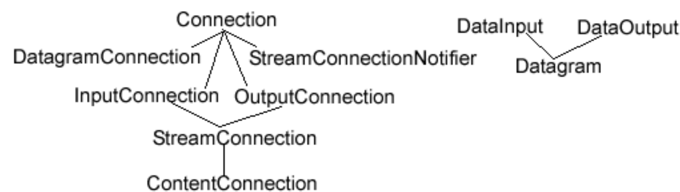

Figura 2.6: Classes que fazem parte do *Generic Connection Framework (GCF)*, segundo a especificação CLDC.

A seguir serão descritas as principais características das interfaces que fazem parte do GCF:

- A interface `Connection` corresponde ao tipo de conexão genérica mais básica existente. Somente o método `close` é definido. Nenhum método *open* é definido nesta classe pelo fato de todo o processo de abertura de objetos do tipo `Connection` ser definido através do método estático `open` da classe `Connector`, também do pacote `javax.microedition.io`. A flexibilidade do GCF encontra-se no fato de se poder utilizar qualquer implementação de protocolo para uma dada conexão, bastando que se carregue as suas definições através do método estático `forName` da classe `Class` do pacote `java.lang`.

  O trecho de código mostrado pela Figura 2.7 ilustra o processo de abertura de conexões.

```
/**
 * Processo de abertura de conexões em J2ME
 *
 *
 * O protótipo do método estático Open da classe Connector é o seguinte:
 *   ->Connector.Open("protocol:address;parameters")
 */

import javax.microedition.io.*;
import javax.microedition.midlet.*;

public class TesteConexao extends MIDlet{

    Connection objeto;

    /**
     * corpo da classe, não importante para o exemplo dado
     *
     */

    //caso haja a necessidade de se carregar algum protocolo
    // deve-se descomentar a próxima linha
    //Class.forName("com.sun.midp.io.j2me.http.Protocol");


    objeto = Connector.open("http://www.algum_web_site.com.br");

    /**
     * corpo da classe, não importante para o exemplo dado
     *
     */


}
```

Figura 2.7: Abertura de conexão utilizando o método `open` da classe `Connector`.

Note que ao se abrir uma conexão através do método `open`, da classe `Connector`, o resultado obtido é um objeto do tipo `Connection`, que deve ser posteriormente convertido, através de um *type casting*, pelo tipo de co-

nexão a ser utilizado. Se houver a necessidade de se carregar o protocolo de comunicação na memória, deve-se descomentar a linha com o comando `forName` e passar para este método o caminho onde se encontra as definição do protocolo desejado.

- A interface `InputConnection` define as características que um fluxo de conexão de entrada deve possuir;
- A interface `OutputConnection` é semelhante a `InputConnection`, com a diferença que ela agora manipula fluxos de saída.
- A interface `StreamConnectionNotifier` define as características que um notificador de conexões deve possuir. Esta classe define apenas um método, que tem a função de ouvir conexões e retornar um objeto que pode ser utilizado para trocar dados entre os elementos conectados. Pode ser utilizado para a construção de aplicações servidoras, ou seja, que precisam escutar por requisições de clientes.
- A interface `DatagramConnection` define as características que uma conexão por datagramas deve possuir, mas a sintaxe para o endereçamento de datagramas é definida apenas a nível de *profile*.
- A interface `StreamConnection` define as características que um protocolo que suporta conexões bi-direcionais deve possuir. Note que ela herda das interfaces que definem um fluxo de entrada e de saída, mas não implementa nenhum método novo.
- A interface `ContentConnection` apresenta alguns métodos para manipular o conteúdo que está passando através do fluxo de comunicação.

Além destas interfaces existem outras classes que são: `DataInput`, `DataOutput` e `Datagram`. As duas primeiras definem os métodos para se converter os dados de um fluxo binário em um tipo primitivo de dados Java, enquanto que a última herda das outras de maneira a prover as abstrações necessárias para um pacote do tipo datagrama.

Estão disponíveis duas máquinas virtuais para esta configuração, a K e a CLDC HotSpot. A máquina virtual K consiste de uma implementação da máquina virtual Java destinada à execução em dispositivos com grandes restrições de recursos (memória e processamento). O K na máquina virtual indica que ela trabalha com uma memória total de pouco mais de cem *kilobytes*, na maioria das vezes, cerca de

128 KB. As especificações dos dispositivos apropriados para trabalhar com esta máquina são: processador de 16/32 bits RISC/CISC com pelo menos de 160KB de memória total, sendo 128 KB para a máquina virtual e o restante destinado às aplicações. Esta máquina foi desenvolvida na linguagem C e objetiva ser completa (de acordo com a especificação CLDC) e tão rápida quanto possível (executando de 30% a 80% mais rápido que a máquina padrão do J2SE). A KVM, como é conhecida esta máquina virtual, é apenas uma implementação de referência da *Sun*, ou seja, ela foi construída apenas para exemplificar a especificação CLDC. Assim, os fabricantes de dispositivos podem escolher portá-la para seus aparelhos ou simplesmente desenvolver sua própria máquina virtual.

A máquina virtual CLDC HotSpot foi projetada para a nova geração de dispositivos móveis que possuem grande memória disponível. As especificações dos dispositivos que podem utilizar esta máquina são: processadores de 32 bits RISC/CISC com memória de 512KB a 1MB.

Atualmente, a versão da configuração CLDC mais utilizada é a 1.0, que possui as características citadas acima. No entanto, encontra-se em processo de desenvolvimento a versão 1.1, que acrescentará características tão importantes quanto o suporte a pontos flutuantes.

2.4.5 Os *profiles* para o CLDC

Estão disponíveis dois *profiles* para esta configuração, o *Mobile Information Device Profile* (MIDP versões 1.0 e 2.0)[7] e o *Personal Digital Assistant Profile* (PDAP). O primeiro é voltado para o mercado de telefones celulares, enquanto que o segundo busca aproveitar melhor as características dos PDAs. A *Sun*, no entanto, possui uma implementação do *profile* MIDP versão 1.0 para o sistema *Palm OS* conhecida por MIDP4Palm[8].

Em MIDP uma aplicação é chamada de MIDlet e as duas versões do *profile* definem o seu ciclo de vida de maneira idêntica. Assim, a primeira coisa a se fazer ao desenvolver MIDlets é herdar a classe `MIDlet` do pacote `javax.microedition.midlet`. Esta classe define os métodos que representam os estados do ciclo de vida de uma aplicação.

A máquina de estados da MIDlet é projetada de maneira a garantir um ambiente de execução de aplicações consistente e próximo das expectativas dos fabri-

[7]Durante o restante da monografia, a sigla MIDP será por várias vezes utilizada como uma referência a *Mobile Information Device Profile*.

[8]Esta seção encontra-se fortemente baseada na documentação da API das duas versões do *profile* MIDP.

cantes e usuários, mais especificamente:

- a latência no processo de inicialização de uma aplicação deve ser curta;
- deve ser possível colocar a aplicação em um estado no qual ela se encontra inativa. Isto permite o atendimento de chamadas enquanto se executa algum programa e a confiança de que ao terminar a ligação o estado da MIDlet será corretamente recuperado;
- a finalização da MIDlet deve ser possível em qualquer momento.

Os estados válidos para uma MIDlet são controlados pelo *application manager software* e podem ser resumidos em:

- **ativo**, a aplicação está executando normalmente;
- **pausado**, a aplicação ainda não começou a executar ou encontra-se em um estado de pausa;
- **destruído**, a aplicação foi terminada.

Uma típica seqüência de execução de uma MIDlet ocorre da seguinte maneira:

1. O *application management software* cria uma nova instância da MIDlet. Nesta situação o construtor padrão da MIDlet é chamado e ela entra no estado de pausa.
2. O *application management software* decide o momento apropriado para iniciar a execução da MIDlet, então ele chama o método `MIDlet.startApp` para colocá-la no estado ativo. Neste ponto a MIDlet coleta os recursos de que necessita para iniciar a sua tarefa.
3. O *application management software* decide que precisa executar alguma tarefa. Assim, ele envia um sinal para a MIDlet parar de executar os seus serviços temporariamente através de uma chamada ao método `MIDlet.pauseApp`. A MIDlet pode então liberar alguns recursos para o sistema. Quando a aplicação puder voltar a executar, uma chamada ao método `MIDlet.startApp` é refeita.

4. O *application management software* determina que a MIDlet não mais é necessária. Assim, ele envia um sinal para ela através do método `MIDlet.destroyApp` dizendo que ela é uma candidata a ser destruída. A MIDlet salva seu estado e libera todos os recursos do sistema que havia alocado para si.

A Figura 2.8 ilustra o processo típico de trocas de estados que uma MIDlet pode sofrer.

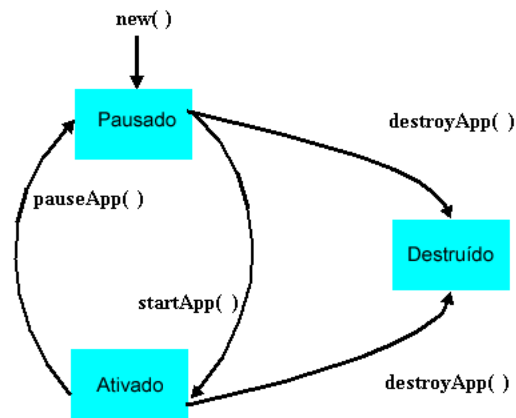

Figura 2.8: Transições de estados em uma MIDlet.

A interface da classe MIDlet define os seguintes métodos:

- `pauseApp`, a MIDlet deve liberar recursos temporários e tornar-se passiva;
- `startApp`, a MIDlet deve requerer todos os recursos necessários e iniciar a sua execução;
- `destroyApp`, a MIDlet deve salvar seu estado e liberar todos os recursos;
- `notifyDestroyed`, a MIDlet notifica o *application management software* que ela deseja ser terminada;
- `notifyPaused`, a MIDlet notifica o *application management software* que ela deseja ser pausada.
- `resumeRequest`, a MIDlet pergunta ao *application management software* se ela pode iniciar novamente a sua execução;

- `getAppProperty`, obtém a propriedade nome da MIDlet.

A figura 2.9 apresenta um trecho de código que ilustra os estados do ciclo de vida de uma MIDlet. Note que todos os métodos referentes ao estado de uma MIDlet devem ser obrigatoriamente implementados.

```
import javax.microedition.midlet.*;

/**
 * Esta MIDlet não executa nenhuma tarefa em especial, serve apenas
 * para mostrar um modelo de classe inicial para uma MIDlet
 *
 * Repare nos métodos startApp, pauseApp e destroyApp
 * para ver como a MIDlet manipula cada transição de estado.
 */
public class TestMIDlet extends MIDlet {
    /**
     * O método startApp deve alocar memória para todos os objetos que
     * serão utilizados e então inciar a execução da MIDlet
     */
    public void startApp() {
        //corpo da função
    }

    /**
     * Ao receber um sinal de pausa a MIDlet deve liberar a memória ocupada
     * por seus objetos. Ao voltar, a thread chamará novamente o método
     * startApp. Caso se queira manter o estado de algum objeto basta não
     * desalocar sua memória neste estado e verificar se o objeto já foi
     * alocado no método startApp antes de se alterar o seu valor
     */
    public void pauseApp() {
        //corpo da função
    }

    /**
     * Destruir a MIDlet deve provocar uma limpeza geral dos recursos
     * alocados e a gravação em meio não volátil dos dados desejados
     */
    public void destroyApp(boolean unconditional) {
        //corpo da função
    }

    /**
     * Um método qualquer que poderia ter sido chamado no startApp.
     */
    void metodo_qualquer() {
        //corpo da função
    }
}
```

Figura 2.9: Trecho de código ilustrando o corpo de uma MIDlet.

O **Mobile Information Device Profile versão 1.0**[MID2003] (JSR-37) trabalha com dispositivos celulares. Segundo Enrique Ortiz[ORT2002A], ele é o primeiro e mais maduro *profile* J2ME e é suportado pelos principais fabricantes de dispositivos móveis, como *Motorola*, *Nokia* e *Ericsson*. Os requisitos de *hardware* e *software* para ele são:

- *Hardware*:
  - Tela de tamanho mínimo de 96x54 pixels;

  - Um dos seguintes mecanismos de entrada: teclado de telefone, teclado QWERTY ou toque na tela;
  - Capacidade de comunicação bidirecional;
  - Pelo menos 128KB de memória não volátil para o *software* MIDP, 8KB para armazenamento persistente e 32KB para o ambiente de execução. Esta quantidade é adicional ao CLDC e a qualquer API específica do fabricante.

- *Software*, características que o sistema operacional deve possuir para adotar o MIDP 1.0:
  - Conversão da entrada do usuário nos eventos apropriados;
  - Gerência do ciclo de vida de uma aplicação;
  - Maneiras de se acessar a memória não volátil, de acessar a rede e escrever na tela.

Os itens a seguir, extraídos do livro de Enrique Ortiz[ORT & GIG2001], resumem o que o *profile* MIDP 1.0 pode e não pode fazer:

- **Interface com o usuário**. O MIDP implementa um conjunto simples de classes (definidas em `javax.microedition.lcdui` e quase sempre referidas como LCDUI) para a criação de interfaces em celulares.

  O MIDP 1.0 define dois níveis de interface com o usuário, chamados de interface com o usuário de baixo nível e alto nível.

  A interface com o usuário de alto nível define uma aplicação em termos abstratos. Ao invés de controlar exatamente o que será desenhado na tela e como será feito o mapeamento dos botões pressionados pelo usuário, ela define classes genéricas que são portáveis ao longo de todos os dispositivos MIDP, deixando que cada implementação em particular decida como portar estas classes de maneira adequada ao aparelho. Não se tem acesso direto a tela e nem aos eventos gerados pela entrada do usuário. Este tipo de interface é mais útil para o desenvolvimento de aplicações *enterprise*, como aplicações de *e-commerce*.

  A interface com o usuário de baixo nível, por outro lado, provê acesso direto a tela e aos eventos de entrada. A aplicação pode desenhar na janela através do uso de ferramentas de desenho de primitivas 2D e é notificada quando o usuário realiza alguma entrada. Esta liberdade é útil quando se está desenvolvendo um jogo, mas não garante portabilidade entre os dispositivos

MIDP. Assim, o desenvolvedor deve ser capaz de construir a aplicação da maneira mais portável possível.

Interfaces com o usuário de alto e baixo e nível podem ser utilizadas em uma mesma aplicação, mas nunca ao mesmo tempo;

A Figura 2.10 apresenta as classes que fazem parte do pacote `javax.microedition.lcdui`.

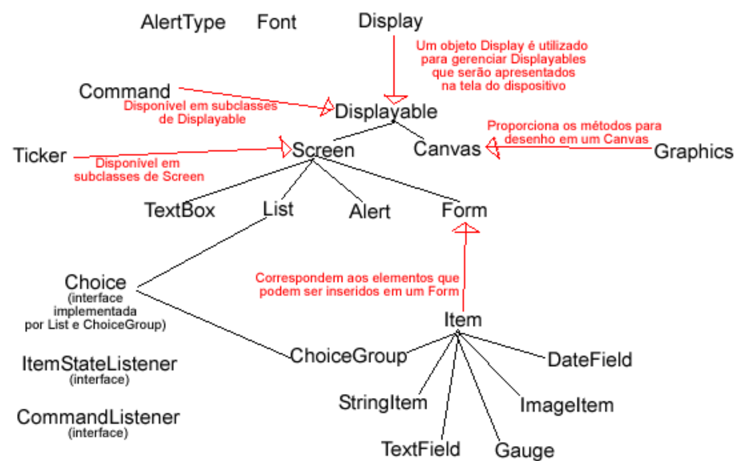

Figura 2.10: Classes que fazem parte do pacote `javax.microedition.lcdui`, para manipulação de interfaces. As setas em vermelho indicam uma relação de uso da classe que aponta a seta pela classe alvo dela. A classes que não apresentam nenhuma relação de herança e não são interfaces herdam diretamente da classe `Object` do pacote `java.lang`.

As interfaces implementadas pelo pacote `javax.microedition.lcdui` são:

- `Choice`, define a API para componentes que implementam seleção a partir de um determinado número de alternativas;
- `CommandListener`, utilizada por aplicações que precisam tratar eventos de alto nível.
- `ItemStateListener`, utilizada para receber eventos que indicam mudanças no estado interno de um item de formulário.

A seguir estão relacionadas as principais características das classes do pacote `javax.microedition.lcdui`:

– `Display`, representa o gerenciador de telas e dispositivos de entrada do sistema;

– `Displayable`, objeto que tem a capacidade de ser colocado em uma tela;

– `Command`, encapsula as propriedades de um comando, mas não o seu comportamento, que é definido pela interface `CommandListener`.

– `Font`, permite a manipulação de fontes;

– `Image`, permite manipular imagens;

– `Screen`, superclasse comum a todos componentes de alto nível.

– `Canvas`, classe básica para a escrita de aplicações que precisam manipular eventos de baixo nível e chamar primitivas gráficas para desenhar na tela;

– `Graphics`, define os métodos para se desenhar em um `Canvas`. Possui suporte a primitivas 2D de desenho;

– `Ticker`, implementa a capacidade de se inserir um texto rolável ao longo da tela;

– `TextBox`, representa uma tela onde o usuário pode inserir e editar textos;

– `List`, é uma tela contendo uma lista com um conjunto de escolhas;

– `Alert` e `AlertType`, proporcionam a capacidade de utilização de uma tela capaz de mostrar ao usuário dados enquanto ele espera pelo resultado de algum processamento;

– `Form`, corresponde a uma tela que contém uma mistura de itens arbitrárias: imagens, campos de textos editáveis ou não, campos de data e grupos de listas;

– `Item`, superclasse dos elementos que podem ser inseridos em um `Form` ou `Alert`;

– `ImageItem`, componente que permite a inserção de uma imagem em um `Form` ou `Alert`;

– `DateField`, componente editável para a apresentação de informações de datas e tempo (calendário);

- `StringItem`, utilizado para apresentar uma `String`;
- `TextField`, campo de texto editável;
- `ChoiceGroup`, implementa a mesma funcionalidade de um `List` para um `Form`;
- `Gauge`, representa uma barra gráfica que indica a completude de uma tarefa;

O trecho de código apresentado pela Figura 2.11 ilustra a utilização de alguns elementos de alto nível do pacote LCDUI.

```
1  import javax.microedition.lcdui.*;
2  import javax.microedition.midlet.*;
3
4  public class HelloMIDlet
5      extends MIDlet
6      implements CommandListener {
7    private Form mMainForm;
8
9    public HelloMIDlet() {
10     mMainForm = new Form("HelloMIDlet");
11     mMainForm.append(new StringItem(null, "Hello, MIDP!"));
12     mMainForm.addCommand(new Command("Exit", Command.EXIT, 0));
13     mMainForm.setCommandListener(this);
14   }
15
16   public void startApp() {
17     Display.getDisplay(this).setCurrent(mMainForm);
18   }
19
20   public void pauseApp() {}
21
22   public void destroyApp(boolean unconditional) {}
23
24   public void commandAction(Command c, Displayable s) {
25     notifyDestroyed();
26   }
27 }
```

Figura 2.11: Código que apresenta a utilização de alguns elementos da interface de alto nível.

Este pequeno código tem a função de apresentar na tela do celular a mensagem *"Hello, MIDP!"*. Note que as linhas 16, 20 e 22 implementam os métodos obrigatórios que correspondem ao estado de uma MIDlet e a linha 2 importa o pacote `javax.microedition.midlet`, que apresenta as definições de uma MIDlet.

A linha 1 importa o pacote `javax.microedition.lcdui`, que é utilizado para a manipulação de interfaces. A linha 6 informa que esta classe será responsável por tratar eventos relacionadas a objetos `Commands`.

A construção da interface ocorre dentro do construtor da MIDlet, a partir da linha 10 até a 13, onde um objeto de nome `mMainForm`, do tipo `Form`, é

instanciado. Após a instanciação, um `Item`, do tipo `StringItem`, contendo a frase a ser exibida no celular é adicionado ao objeto `mMainForm`, seguido por um objeto `Command`. A linha 13 diz a MIDlet qual classe irá responder aos eventos do tipo `Command`. Neste exemplo a própria classe irá tratar este tipo de evento.

Na linha 17, obtém-se uma referência ao objeto `Display` que é utilizada para setar a tela atual para `mMainForm`, que por ser do tipo `Screen` pode ser apresentado no *display* do dispositivo.

A saída deste programa pode ser vista na Figura 2.12, que apresenta a captura de uma tela de celular.

Figura 2.12: Resultado da execução, em um emulador de celular, do código da Figura 2.11. Em **a** vemos o celular apresentando as aplicações disponíveis para serem ativadas e em **b** a saída do código.

- **Persistência de dados**. Qualquer tipo de aplicação requer uma maneira de gravar informações de maneira a poder acessá-las posteriormente. Com as aplicações para dispositivos móveis não é diferente, mas como fazer para armazenar dados em aparelhos MIDP. Existem duas alternativas, a primeira se refere ao armazenamento, através da utilização da rede, em um banco de dados. O problema é o custo de acesso a rede e o fato de ela não estar

sempre disponível. A segunda, e exigida pela especificação MIDP 1.0, é o armazenamento local.

O MIDP 1.0 define um conjunto de classes e interfaces (em `javax. microedition.rms`) chamadas *Record Management System* (RMS) para armazenamento local. O RMS é implementado como um conjunto de registros e torna disponível diversas funções para lidar com estes regitros, inclusive funções de ordenação. O acesso a um registro individual é feito através de índices e a leitura e escrita são operações atômicas que utilizam vetores de *bytes* para trabalhar com os dados. Uma característica importante é que os dados armazenados por uma aplicação não podem ser acessados por outra aplicação;

- **Conectividade** *wireless*. Como já apresentado, o GCF definido pela CLDC não requer suporte a qualquer tipo de protocolo de comunicação, seja ele para acesso a dados locais, a portas seriais ou a rede. O MIDP 1.0, por outro lado, requer suporte ao acesso bidirecional a rede e implementa obrigatoriamente um protocolo.

  O protocolo implementado pelo MIDP 1.0 é um subconjunto do HTTP 1.1. Este protocolo é o mesmo utilizado por um navegador *web* e habilita uma aplicação a se comunicar com qualquer servidor *web* na . O HTTP é um protocolo do tipo *request-response*, mas pode ser utilizado para comunicação bidirecional, com a única restrição de que o cliente deve iniciar a "conversa".

  Os demais protocolos existentes, como datagramas e *sockets*, não são obrigatórios e nem mesmo definidos pela especificação MIDP 1.0, mas isso não impede que os fabricantes de aparelhos os implementem através do GCF ou de bibliotecas próprias. A questão é que como o protocolo HTTP é o único obrigatório, ele é o único que garante a portabilidade;

  A Figura 2.13 apresenta as modificações feitas na estrutura de interfaces do GCF pelo MIDP 1.0.

  Como pode ser notado a única modificação realizada está na presença da classe `HttpConnection` que herda de `ContentConnection` e provê as funcionalidades do protocolo HTTP para os celulares.

- **Suporte a imagens**. O único tipo de imagem aceito pelo MIDP 1.0 é o padrão PNG. Um detalhe é que a implementação de transparência não é obrigatória;

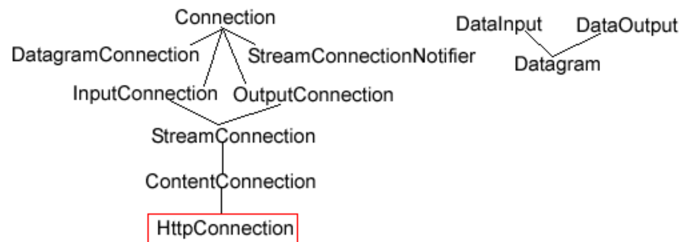

Figura 2.13: Modificações feitas no GCF pelo *profile* MIDP versão 1.0. As novas interfaces implementadas encontram-se circuladas de vermelho.

- **Classes adicionadas do J2SE pelo MIDP 1.0**. As classes `java.lang.IllegalStateException`, `java.util.Timer` e `java.util.TimerTask` foram incluídas de maneira a possibilitar a escrita de código que executa em *threads* de *background* em tempos específicos;

- **APIs de acesso ao sistema**. O MIDP 1.0 não possui classes que permitem o acesso direto as características de um dispositivo, como gerenciamento de energia e telefonia;

- **Segurança**. O MIDP 1.0 não acrescenta nenhum pacote com mecanismos de segurança ao CLDC. Em particular, apesar de suportar o protocolo HTTP não suporta o protocolo HTTPS, que é a versão segura do HTTP.

Resumindo, os pacotes que fazem parte da API do MIDP 1.0, excluídos os implementados pelo CLDC, são:

- `javax.microedition.lcdui`, interface com o usuário;

- `javax.microedition.rms`, armazenamento persistente;

- `javax.microedition.midlet`, ciclo de vida da aplicação;

- `javax.microedition.io`, acesso a rede e inclusão de suporte ao protocolo HTTP.

O **Mobile Information Device Profile versão 2.0**[MID2003] (JSR-118) acrescentou diversas características e melhorias a especificação MIDP 1.0, mas continuou com o objetivo da primeira versão de ser um ambiente de desenvolvimento aberto e desenvolvido em conjunto com diversas empresas.

O MIDP 2.0 objetiva explorar melhor a capacidade dos novos aparelhos celulares que estão chegando no mercado. Segundo Qusay Mahmoud[MAH2003], no MIDP 2.0, um dispositivo deve possuir as seguintes características mínimas:

- Uma tela de tamanho 96x54 e profundidade de 1 bit (preto e branco);
- Um dos seguintes mecanismos de entrada: teclado de telefone, teclado QWERTY ou toque na tela;
- Conectividade a rede: *wireless* e com largura de banda limitada;
- Habilidade de reproduzir tons (suporte a sons);
- 256KB de memória (incluindo CLDC), 8 KB de memória não volátil para armazenamento persistente e 128KB de memória volátil para o ambiente de execução Java.

A especificação do MIDP 2.0 é baseada na do MIDP 1.0 e completamente compatível com ela. Assim, aplicações escritas para MIDP 1.0 irão executar sem problemas nos novos aparelhos que suportam MIDP 2.0. Algumas novas características do MIDP 2.0, enumeradas por [MAH2003], são:

- Comunicação via *sockets* e datagramas;
- Suporte a HTTPS e *secure sockets* (SSL);
- Inclusão do *over-the-air provisioning*, apresentado na seção 3.2;
- APIs específicas para o desenvolvimento de jogos;
- APIs para áudio e vídeo.

Os novos pacotes adicionados pelo MIDP 2.0 foram:

- `javax.microedition.lcdui.games`, voltado ao desenvolvimento de jogos;
- `javax.microedition.pki`, voltado a segurança, permite o uso de certificados para autenticar informações em conexões seguras;
- `javax.microedition.media`, proporciona acesso a funções multimídias;

- `javax.microedition.media.control`, define tipos específicos de controles para serem usados como um *player* multimídia.

O único problema do MIDP 2.0 é que ainda não existem aparelhos que o implementem, apenas emuladores. De acordo com Jonathan Knudsen[KNU2002], o processo para que o MIDP 2.0 comece a ser implementado deve ainda passar pelas seguintes etapas:

1. A especificação torna-se uma versão final após revisão pública (passo em execução);

2. A implementação de referência é lançada;

3. Fabricantes de dispositivos implementam MIDP 2.0;

4. Fabricantes verificam se os dispositivos estão de acordo com as leis governamentais;

5. Dispositivos são lançados.

A estimativa de quando os dispositivos MIDP 2.0 serão encontrados em grande volume no mercado é no início do segundo semestre de 2003. Mas isso não impede que se apresente em detalhes as suas principais novidades com relação a versão anterior:

- **Conexões seguras**. MIDP 2.0 requer que os dispositivos implementem HTTPS, que é basicamente o HTTP sobre *Secure Sockets Layer* (SSL). SSL é um protocolo de *sockets* que criptografa os dados enviados sobre a rede e proporciona métodos de autenticação para os dois lados da comunicação.

  O fato de o MIDP 2.0 obrigar a implementação deste protocolo faz com que se torne possível o desenvolvimento de aplicações que necessitem de segurança, como em transações com cartões de crédito. O suporte a HTTPS é proporcionado através do *CLDC's Generic Connection Framework* no pacote `javax.microedition.io`.

- **Multimídia**. As APIs disponíveis estão relacionadas a um subconjunto de manipulação de sons da *Mobile Media API* (MMAPI). Também há a possibilidade de gravação de áudio. É obrigatório o suporte apenas para arquivos WAV.

- **Melhorias nos objetos de interface de alto nível com o usuário**. As grandes mudanças foram realizadas nas classes `Form` e `Item`. O *layout* de formulários é consideravelmente mais sofisticado que no MIDP 1.0. Os itens antigamente só podiam ser colocados da esquerda para a direita em linhas de cima para baixo. Agora, este *layout* pode ser modificado, mas a implementação é que ainda decide exatamente onde os itens serão colocados e o seu tamanho. A novidade é que agora pode-se definir um tamanho mínimo e preferencial que pode ser escolhido pela aplicação.

  Um novo integrante dos objetos de *layout* no MIDP 2.0 é o `Spacer`, `Item` não navegável que representa um espaço em branco e permite um ajuste mais cuidadoso do *layout*. O seu funcionamento é idêntico as imagens espaçadoras que são utilizadas na confecção de *sites*. No item `ChoiceGroup` foi incluído o tipo POPUP.

  Comandos podem ser adicionados a qualquer `Item` em MIDP 2.0, não apenas a um `Displayable`.

  Um novo `Item`, chamado `CustomItem`, foi implementado. Este novo item auxilia na personalização dos controles, permitindo a criação de novos itens pelo próprio desenvolvedor. O conceito é similar ao de um `Canvas`, o `CustomItem` permite o desenho de qualquer nova forma e a resposta a eventos de interface com o usuário.

- **A API de jogos (GAME API)**. Este pacote será muito detalhado devido a sua importância no desenvolvimento de jogos e também para tornar possível uma comparação com a implementação do jogo modelo que foi desenvolvido sem as novas técnicas disponibilizadas por este pacote.

  Esta foi uma das grandes mudanças no MIDP 2.0. A *Sun* e as empresas que ajudaram no desenvolvimento dessa nova especificação reconheceram a importância do mercado de jogos para celulares e deram especial atenção na construção de um pacote específico para a sua criação. Este novo pacote possibilita a automatização de vários processos necessários a fase de desenvolvimento de um jogo e é especialmente útil para jogos 2D baseados em *tiles*.

  A idéia básica por trás deste novo pacote está no fato de o conteúdo de uma tela poder ser composto de diferentes camadas. Assim, pode-se colocar qualquer conteúdo em uma camada e ir encaixando-as uma sobre a outra para formar o efeito desejado. As camadas podem ser manipuladas independentemente.

Dispositivos *wireless* tem poder de processamento mínimo, de tal maneira que esta API procura melhorar a performance ao diminuir a quantidade de processamento feito em Java. Esta abordagem também adiciona o benefício de ser capaz de reduzir o tamanho da aplicação. A API está estruturada de uma maneira a proporcionar ao fabricante de dispositivos uma liberdade considerável para a sua implementação, permitindo o uso extensivo de código nativo, aceleração de *hardware* e formatos de dados de imagem específicos de dispositivo quando necessário.

A API é composta de cinco classes que estão descritas a seguir, de acordo com as características apresentadas pela *Sun Microsystems*[SUN2002C]:

- **GameCanvas**. Esta classe é uma subclasse de `Canvas` no pacote `javax.microedition.lcdui` e porporciona a funcionalidade básica de uma tela para um jogo. Em adição aos métodos herdados de `Canvas` há a possibilidade de se consultar o estado atual das teclas do jogo e sincronizar o envio de gráficos para a tela. Estas características foram implementadas de maneira a simplificar o desenvolvimento de jogos e provocar uma melhora de performance.

  Outra característica implementada está relacionada com a possibilidade de se obter o objeto de desenho `Graphics` de qualquer ponto do programa. Assim, não há mais a necessidade de se repintar a tela apenas através do método `paint`;

- **Layer**. Esta classe representa um elemento visual em um jogo. É uma classe abstrata que forma a base para o desenvolvimento de camadas (as classes `Sprite` e `TiledLayer` a extendem). Ela também proporciona atributos básicos, como localização, tamanho e visibilidade. Pode ser herdada para o desenvolvimento de classes que representam camadas de uma maneira personalizada;

- **LayerManager**. Para jogos que empregam diversos objetos da classe `Layer`, o `LayerManager` simplifica o desenvolvimento através de uma automatização do processo de formação de imagens, renderizando as regiões corretas de cada `Layer` na ordem apropriada.

  O `LayerManager` mantém uma lista ordenada com as `Layers` que lhe foram passadas. Estas camadas possuem um índice que indica a sua posição no eixo z. Assim, um índice 0 indica maior proximidade com o usuário. As camadas podem ter sua posição trocada livremente pelo desenvolvedor.

A ordem das camadas em um `LayerManager` indica a seqüência em que elas serão renderizadas.

A classe *LayerManager* proporciona várias características que podem ser utilizadas para controlar como as camadas são renderizadas na tela. Uma delas é a presença de uma janela de visualização que controla o tamanho da região visível pelo usuário e tem a sua posição relativa ao sistema de coordenadas do objeto *LayerManager*. Ao trocar a posição da janela de visualização obtém-se um efeito de rolagem de tela. Pode-se definir o tamanho da janela de visualização e sua posição.

O método `paint` desta classe é utilizado quando se desejar pintar a tela. Ele pede como parâmetro uma localização (x,y) que indica a partir de onde no *display* do dispositivo do usuário se iniciará a renderização da janela de visualização. Quando necessário o método `paint` realiza o recorte das imagens;

– **Sprite**. Representa uma camada animada capaz de apresentar vários quadros gráficos. Cada quadro possui um objeto `Image` relacionado e todos são de mesmo tamanho. Para prover animação seqüêncial dos quadros pode-se definir uma seqüência de animação arbitrária. A classe `Sprite` ainda proporciona métodos para a realização de transformações e detecção de colisão, simplificando em muito a implementação da lógica de um jogo.

  Os *sprites* são os elementos de interação do jogo, ou seja, são eles que podem ter suas posições alteradas pelo jogador durante uma partida.

– **TiledLayer**. Esta classe habilita o desenvolvedor a criar grandes áreas de conteúdo gráfico de maneira fácil, a partir de um objeto `Image` de referência. Ela faz com que não seja necessário carregar uma grande imagem de fundo, permitindo que se verifique os elementos da figura que se repetem e, assim, carregá-los apenas uma vez.

  Esta classe define uma matriz de células onde cada uma pode apresentar um entre os vários *tiles* que fazem parte de um único objeto `Image`.

  As células da tabela também podem ser preenchidas com *tiles* animados, onde os dados da imagens são carregados rapidamente. Esta abordagem permite a animação de uma grande região, como o movimento das águas em um lago, sem exigir muito tempo de processamento para renderizar a imagem, pois seu carregamento em memória ocorre apenas uma vez.

O objetivo desta classe é facilitar o processo de criação de fundos de tela roláveis quando se está desenvolvendo jogos 2D.

Os *tiles* são elementos de tela estáticos e que são utilizados para preencher as células de um objeto `TiledLayer`. Uma imagem é quebrada em uma série de *tiles* de mesmo tamanho, especificado pelo desenvolvedor. Esta imagem, chamada de imagem fonte, fornece os *tiles* que serão utilizados para a construção dos mais variados arranjos de camadas, dependendo da situação do jogo, como mostra a Figura 2.14.

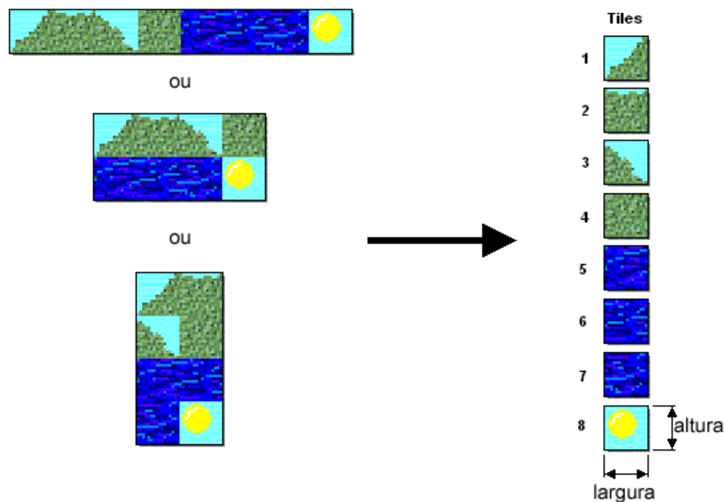

Figura 2.14: Processo de construção de fundos com a utilização de *tiles* em uma `TiledLayer`.

Cada *tile* possui um único identificador, sendo que o localizado no canto superior esquerdo da imagem fonte recebe o número 1 e os seguintes são numerados consecutivamente (linha a linha). Este *tiles* recebem a denominação de estáticos pelo fato de haver um *link* fixo entre ele e a imagem associada.

Um *tile* estático é criado quando um objeto *TiledLayer* é instanciado, mas pode ser atualizado através de uma chamada ao método `setStaticTileSet()`.

Tiles animados são criados com o método `createAnimatedTile()`, o qual retorna o índice a ser usado para fazer referência a este novo *tile*.

Este índice é negativo e começa a partir do -1. Cada *tile* animado faz referência a um estático que corresponde a figura inicial que será apresentada.

As células na matriz de um objeto `TiledLayer` possuem todas o mesmo tamanho, sendo que o número de linhas e colunas de cada célula é especificado no construtor. O tamanho das células é o mesmo tamanho dos *tiles*.

O conteúdo de cada célula é especificado pelo índice do *tile*. Um índice 0 indica que a célula está vazia, portanto nada será desenhado em seu lugar. Por padrão todas as células contém o índice 0.

O conteúdo é alterado através dos métodos `setCell()` e `fillCells()`. Várias células podem conter o mesmo *tile*, entretanto, uma única célula não pode conter mais de um *tile*.

A Figura 2.15 ilustra como construir um fundo utilizando uma *TiledLayer*.

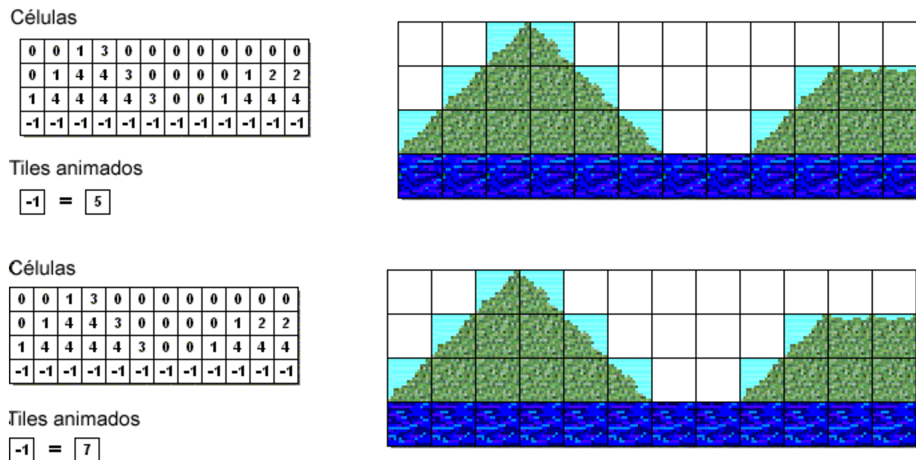

Figura 2.15: Processo de construção de imagens de fundo utilizando uma `TiledLayer`.

Neste exemplo, a área da água é preenchida com um *tile* animado de índice -1, o qual faz uma referência inicial ao *tile* estático 5. A área inteira da região de água pode ser animada com a mudança do tile estático associado através do método `setAnimatedTile(-1, 7)`.

Uma `TiledLayer` pode ser renderizada manualmente através de uma

chamada ao método `paint` ou automaticamente usando um objeto `LayerManager`.

A Figura 2.16 apresenta a hierarquia de classes do pacote de desenvolvimento de jogos disponibilizado pelo MIDP 2.0.

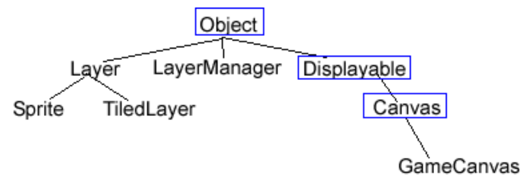

Figura 2.16: Hierarquia de classes do pacote `javax.microedition.lcdui.game`. As classes circuladas de azul não fazem parte do pacote de jogos, apenas são extendidas por algumas de suas classes.

Métodos que modificam o estado de objetos `Layer`, `LayerManager`, `Sprite` e `TiledLayer`, geralmente, não tem seu efeito visualizado imediatamente. Ao invés disso, eles são simplesmente armazenados dentro do objeto e usados nas próximas chamadas ao método `paint()`. Esta abordagem é adequada para aplicações de jogo, onde há um ciclo de jogo dentro do qual objetos são atualizados e onde a tela é repintada no final do ciclo.

- **Imagens RGB**. Os dados de uma imagem são representados através de um inteiro de 32 bits por pixel, onde cada 8 bits são para o nível de transparência, vermelho, verde e azul;

- **Permissões e assinatura de código**. O MIDP 2.0 voltou-se muito para a área de segurança, que no caso do MIDP 1.0 foi deixada um tanto de lado. Uma das novidades foi em relação ao conceito de código confiável e não confiável. Códigos não confiáveis não podem abrir nenhum tipo de conexão sem a autorização do usuário, enquanto que códigos confiáveis são assinados digitalmente pelo desenvolvedor de maneira que o dispositivo consiga reconhecer esta assinatura e dar total liberdade a aplicação de executar.

- **Vários protocolos de comunicação**. O MIDP 2.0 define outros protocolos de comunicação além do HTTP, como datagrama, *socket* e *socket server*. A especificação não requer que se implemente-os, faz apenas uma recomendação e disponibiliza a API de suporte necessária.

Através desta abordagem o fabricante que for implementar algum protocolo além do HTTP e HTTPS não precisa mais criar a sua própria API, pode-se utilizar das interfaces definidas pelo próprio *profile*. Com isso, garante-se a portabilidade das aplicações.

A Figura 2.17 apresenta a hierarquia de interfaces do GCF implementadas pelo MIDP 2.0;

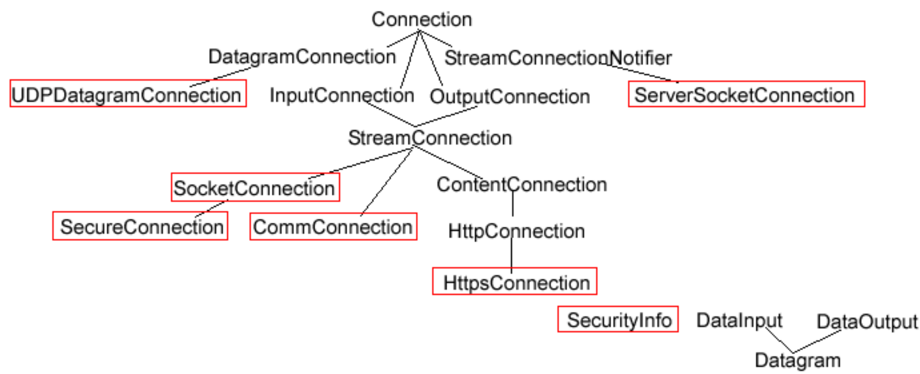

Figura 2.17: Modificações feitas no GCF pelo *profile* MIDP versão 2.0. As novas interfaces incluídas por esta especificação encontram-se circuladas de vermelho.

A seguir encontra-se uma descrição das classes que não faziam parte da versão anterior do *profile* MIDP:

- `CommConnection`, define uma conexão com uma porta serial;
- `HttpsConnection`, define os métodos e constantes necessários para estabelecer uma conexão segura via rede;
- `SecureConnection`, define um fluxo de conexão por *sockets* seguro;
- `SecurityInfo`, define os métodos necessários para se accessar informações sobre uma rede segura;
- `ServerSocketConnection`, define um servidor baseado em *sockets*;
- `SocketConnection`, define uma conexão por *sockets*;
- `UDPDatagramConnection`, esta interface define uma conexão por datagramas não confiável.

Para este projeto foi utilizada a interface `SocketConnection` para estabelecer uma conexão com o servidor. A utilização da API do MIDP 2.0 restringiu-se a esta classe.

- **Push Registry**. Este novo sistema permite que um servidor inicie remotamente uma aplicação, devidamente registrada, no aparelho do usuário;
- **Compartilhamento de dados**. Aplicações em MIDP 2.0 podem compartilhar seus arquivos de registro.

O *Personal Digital Assistant Profile* estende as capacidades do MIDP 1.0 (por isso, deve ser construído sobre este *profile*) e requer o CLDC 1.1. Ela proporciona APIs para a interface com o usuário e o armazenamento de dados para dispositivos como o *handheld*, *Personal Digital Assistants* e *Palm Pilots*, com as seguintes características:

- Não menos que 1000KB de memória total (ROM + RAM) disponível para o ambiente de execução Java e bibliotecas e não mais que 16MB;
- Potência limitada;
- Tipicamente operado a bateria;
- Acesso bidirecional a rede, intermitente e com largura de banda limitada;
- Interfaces com o usuário com variados graus de sofisticação, mas tendo *displays* com uma resolução mínima de pelo menos 128x128 pixels, um dispositivo de apontamento e uma entrada de caracteres.

O PDA é o primeiro *profile* baseado na CLDC criado especificamente para dispositivos PDAs. Esta especificação foi desenvolvida com o objetivo de aproveitar melhor alguns recursos comuns a todos os PDAs, como acesso a agenda de endereços e a listas de coisas a fazer. Os PDAs também tender a possuir mais capacidades que os simples dispositivos baseados no MIDP.

A relação entre o MIDP e o PDAP é ilustrada na Figura 2.18, retirada do artigo de Enrique Ortiz[ORT2002A].

2.4.6 J2ME *Connected Device Configuration* (CDC)

A especificação J2ME *Connected Device Configuration* (CDC), JSR36, é baseada na especificação da máquina virtual Java clássica. Ela define um ambiente

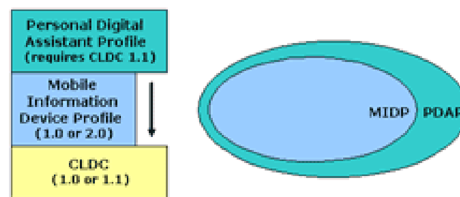

Figura 2.18: *Profiles* que fazem parte da configuração CLDC.

de execução completo que inclui toda a funcionalidade de um ambiente para o *desktop*.

Esta configuração é direcionada a dipositivos mais robustos, com grande capacidade de acesso a rede, *wireless* ou não, com pelo menos poucos megabytes de memória disponível e que podem se conectar a Internet ou a outros dispositivos. As características exigidas por esta configuração são:

- ROM disponível de pelo menos 512KB;
- RAM disponível de pelo menos 256KB;
- Processadores de 32 bits;
- Conectividade a algum tipo de rede;
- Suporte para uma implementação completa da máquina virtual Java, como a definida em [LIN & YEL1999].

Apesar da especificação exigir apenas 256KB de RAM, o ideal é que este valor seja de no mínimo 2MB.

O CDC é um superconjunto do CLDC, como mostra a Figura 2.19. Assim, o CDC inclui todas as classes definidas pelo CLDC, outras do núcleo do J2SE e mais algumas novas, não existentes nem no J2SE.

2.4.7 Os *profiles* para o CDC

Os *profiles* disponíveis para o CDC são o *Foundation Profile (FP)*, o *Personal Basis Profile (PBP)* e o *Personal Profile (PP)*.

A especificação **J2ME** *Foundation Profile*, JSR46, é adequada para dispositivos que necessitam de um amplo suporte a plataforma Java, mas não necessitam de interface gráfica. Ele estende o CDC adicionando novas classes J2SE e também proporciona a base para que outros *profiles* possam ser construídos sobre ele,

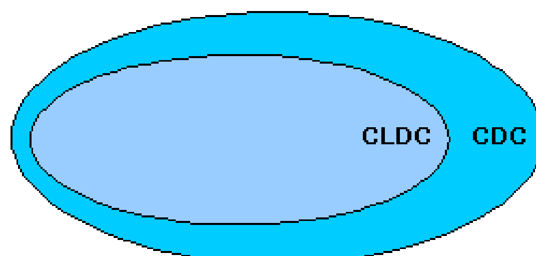

Figura 2.19: Relação entre o CLDC e CDC.

adicionando uma interface gráfica com o usuário e outros comportamentos. Ele é adequado a dispositivos com as seguintes características:

- ROM disponível de no mínimo 1MB;
- RAM disponível de no mínimo 512KB;
- Conectividade a algum tipo de rede;
- Nenhuma interface gráfica com o usuário.

O **J2ME** *Personal Basis Profile* é construído sobre o *Foundation Profile*. Este *profile* define um subconjunto otimizado da classe AWT, utilizada para a construção de interfaces gráficas.

A proposta de especificação **J2ME** *Personal Profile*, JSR62, reempacota o *PersonalJava[tm] Application Environment* para assim prover a especificação J2ME para dispositivos que necessitam de um alto grau de conectividade a Internet. Os dispositivos que venham a implementar este *profile* devem possuir as seguintes características:

- ROM disponível de no mínimo 2.5MB;
- RAM disponível de no mínimo 1MB;
- Conectividade de alto nível com algum tipo de rede;
- Interface gráfica com o usuário (GUI) com um alto grau de fidelidade a *web* e capacidade de executar applets.

O J2ME *Personal Profile* será compatível com o *PersonalJava* Application Environment especificações 1.1.x e 1.2.x.

A relação entre estes *profiles* é ilustrada pela Figura 2.20.

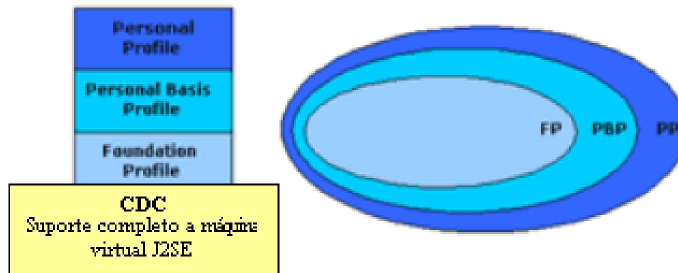

Figura 2.20: *Profiles* baseados na configuração CDC e suas relações.

2.4.8 Especificações adotadas pelos celulares

Como já apresentado, os telefones celulares tên características que vão de encontro as especificações da configuração CLDC. Assim, eles irão implementar os *profiles* existentes para esta configuração, mais especificamente o *profile* MIDP. Com isso, as análises realizadas no prósimo capítulo serão realizadas exclusivamente sobre esta organização.

A Figura 2.21 apresenta a organização atualmente implementada pelos aparelhos celulares.

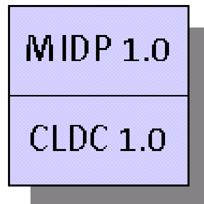

Figura 2.21: Pilha de *software* atualmente implentada pelos aparelhos celulares.

A Figura 2.22 apresenta a organização futura a ser implementada pelos aparelhos celulares. Em meados do segundo semestre de 2003 acredita-se que já estarão disponíveis no mercado dispositivos com MIDP 2.0, mas ainda com a versão 1.0 da configuração CLDC.

2.4.9 Distribuindo a aplicação

Esta seção foi baseada em um artigo introdutório de J2ME de BillDay, que pode ser encontrado no seu site oficial[DAY2001].

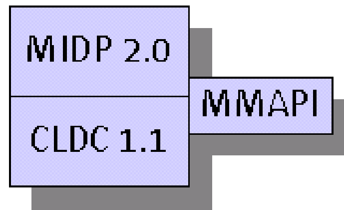

Figura 2.22: Pilha de *software* a ser implementada pelos aparelhos celulares.

Uma vez que a aplicação está pronta para ser distribuída é necessário um mecanismo que a carregue para o dispositivo desejado. Esse mecanismo é denominado *Java Application Manager* (JAM).

A especificação MIDP descreve claramente as características que o dispositivo deve possuir para tratar esta situação. A especificação dá um certo grau de flexibilidade para os fabricantes definindo apenas o que o mecanismo a ser criado deve ter, não entrando em detalhes sobre como deve ser feita a sua implementação.

O JAM deve ser capaz de copiar um código através de uma conexão *wireless*, saber onde ele deve ser armazenado e como o usuário deve iniciar a aplicação. Estes passos precisam ser feitos de maneira altamente eficiente, de modo a economizar tempo e dinheiro do usuário.

A Figura 2.23 demonstra um esquema de como o JAM deve trabalhar.

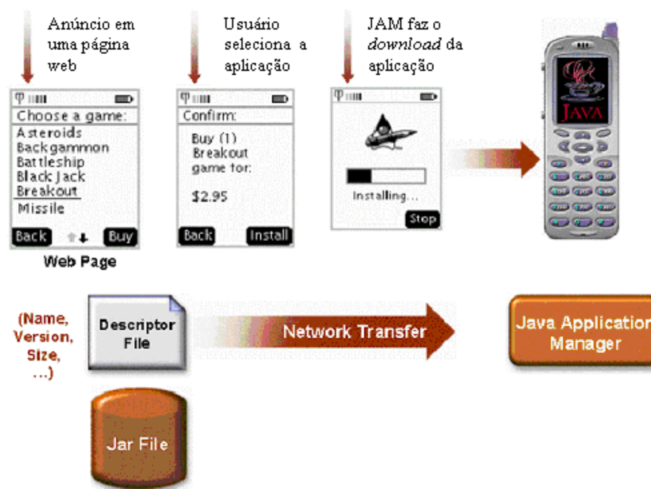

Figura 2.23: Funcionamento de um JAM.

Neste cenário, um consumidor vai a uma página *web*, ou página wap. A página lista as aplicações disponíveis para cópia.

Se o consumidor quer comprar sua aplicação, ele a seleciona da página *web* usando seu aparelho de telefone móvel. Ao selecionar a aplicação inicia-se automaticamente o *download* do arquivo *.jad*, da ordem de dezenas de bytes, da rede para o aparelho. Como este arquivo é muito pequeno ele é copiado rapidamente e de maneira barata através da conexão *wireless*.

O arquivo JAD tem a função de dizer algumas coisas básicas ao consumidor, como:

- A versão da aplicação a ser instalada, evitando assim que se faça o download de uma versão igual ou menor a uma já instalada no aparelho;
- Tamanho da aplicação atual, se o usuário tem somente 2KB de espaço disponível e a aplicação tem 6KB ele pode mostrar uma janela dizendo que o usuário não tem espaço suficiente para armazenar aquela aplicação. Isto é bom para o consumidor porque ele não perde tempo e dinheiro na sua conexão *wireless* copiando uma aplicação para a qual ele não tem memória suficiente.

Uma vez que o consumidor está pronto para fazer o download da aplicação e o JAM confirmou que há espaço suficiente, a cópia é iniciada. O JAM irá salvar a aplicação no dispositivo e então apresentar um menu de seleção para que o usuário possa iniciá-la.

Segundo a especificação MIDP, a abilidade de realizar o *download* e instalação de conteúdo, por demanda, sobre uma rede *wireless* é conhecida pelo termo *over-the-air provisioning (OTA)*.

Da perspectiva dos clientes móveis, como já foi apresentado, o conceito de OTA está relacionado como a maneira com que o dispositivo se comporta na operação de encontrar uma aplicação interessante na *web*, iniciar o *download* via rede *wireless*, instalá-la e deixá-la pronta para o uso.

Segunto Enrique Ortiz[ORT2002B], os participantes deste processo são:

- Dispositivo Cliente, que deve possuir uma aplicação que torne possível a busca de aplicações na rede;
- A rede, que é qualquer rede *wireless*;
- Servidor de *downloads*, que é um servidor visível na rede onde se está executando um aplicativo *web server* que tem acesso a um repósitório de conteúdo. É o responsável por apresentar ao cliente as aplicações disponíveis;

- O repositório de conteúdo, onde permanecem todas as suítes de MIDlets que estão disponíveis para *download*.

A Figura 2.24 apresenta a estrutura do sistema OTA.

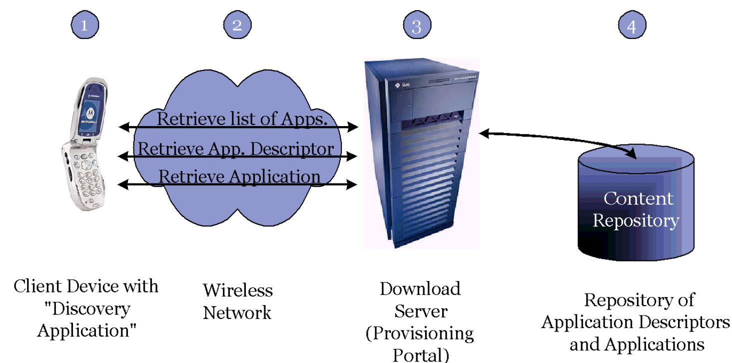

Figura 2.24: Over-the-air provisioning (OTA).

A primeira versão da especificação MIDP OTA era somente uma prática recomendada. Atualmente, muitos dispositivos suportam a recomendação do MIDP 1.0. O MIDP 2.0 tornou a sua implementação obrigatória.

2.5 O escopo da tecnologia *wireless* em Java

A *Sun Microsystems*[SUN2003A] conceitua a tecnologia *wireless* Java como a interseção de dois grandes mundos, o da comunicação de dados *wireless* e o da plataforma Java. Segundo a própria *Sun* a tecnologia *wireless* Java abrange partes do J2ME, *PersonalJava*, J2SE e J2EE. Assim, alguns conceitos devem então ser aqui esclarecidos:

- Tecnologia *wireless* Java e J2ME não são a mesma coisa. J2ME incorpora mais que apenas dispositivos *wireless*. Enquanto algumas partes do J2ME são explicitamente projetadas para dispositivos *wireless*, outras partes não são. Dispositivos CDC tem por padrão conexões *Ethernet*. Por sua vez, a tecnologia Java *wireless* não está confinada apenas ao J2ME. Pode-se ter um *laptop* ou *palmtop*, executando aplicações J2SE, conectado a outros computadores via uma rede local sem fio;

- MIDP não é tudo do J2ME. MIDP é apenas o primeiro perfil terminado e o primeiro a ser incorporado aos dispositivos. Algumas pessoas assumem que sempre que estão falando sobre MIDP estão falando sobre J2ME. J2ME tem muitas outras especificações, sendo que o MIDP foi apenas o primeiro da fila;

- MIDP não é toda a tecnologia *wireless* Java. A plataforma Java oferece uma enorme variedade de escolhas para os programadores *wireless*, como o *PersonalJava* e o J2SE em dispositivos *wireless*.

2.6 Por que usar a tecnologia Java para o desenvolvimento de aplicações wireless?

A tecnologia Java, como já foi dito anteriormente, está presente em uma grande gama de dispositivos, desde cartões de crédito, passando por máquinas *desktop* até aplicações para servidores. Assim, é possível interligar todos estes dispositivos, fazendo com que eles trabalhem juntos, através do uso de uma única linguagem.

A *Sun Microsystems*[SUN2003A] apresenta três grandes vantagens na utilização da tecnologia Java e da plataforma J2ME para o desenvolvimento de projetos:

- A plataforma Java é segura. Códigos Java sempre executam dentro dos limites da máquina virtual que, por sua vez, provê um ambiente seguro para a execução de código "abaixado" da Internet. Uma aplicação binária poderia travar o aparelho ou até mesmo obrigar-nos a reiniciá-lo. No caso de uma aplicação Java, o máximo que poderia ocorrer é algum problema com a máquina virtual, sem afetar o funcionamento do aparelho;

- A linguagem Java é excelente e permite uma programação robusta. Algumas características, como o coletor de lixo automático, são muito eficientes na prevenção de *bugs* comuns em outras linguagens de programação. A sintaxe de Java torna fácil a escrita de programas e a sua compreensão;

- Portabilidade é uma grande vitória para a tecnologia Java *wireless*. Um único executável pode ser executado em múltiplos dispositivos. Por exemplo, quando se escreve um MIDlet, uma aplicação MIDP, ela irá executar em qualquer dispositivo que implementa a especificação MIDP. Até mesmo se uma aplicação Java faz uso de APIs específicas de um certo fabricante, as aplicações escritas em Java são mais fáceis de serem modificadas em comparação àquelas feitas em *C* ou *C++*.

2.7 Jogos *multiplayers*

Jogo *multiplayer* é o nome dado ao estilo de jogo no qual os jogadores não interagem apenas com personagens manipulados pelo programa, mas também com pessoas reais.

Apesar dos estudos em Inteligência Artificial estarem se desenvolvendo em uma velocidade incrível nos últimos anos e de estar cada vez mais presente nos novos jogos, muito ainda falta para se construir um personagem que se comporte igual a um humano de verdade. Na grande maioria das vezes, após uma certa ambientação com o jogo, o movimento dos personagens gerenciados pelo computador torna-se previsível. Por isso, a possibilidade de jogar com pessoas reais foi bem recebida entre os consumidores de jogos eletrônicos, uma vez que o comportamento de um indivíduo é realmente imprevisível.

O fato de se reunir com os amigos e realizar disputas entre várias pessoas ao mesmo tempo também é mais divertido, permitindo uma fuga do tradicional um a um nos imposto pelos jogos mais antigos.

O advento dos jogos *multiplayers* se deu com os computadores *desktop* e com a popularização da Internet, que permitiu a disputa de partidas com qualquer pessoa no mundo.

No entanto, este problema com a velocidade de conexão foi bem resolvido com a popularização das redes locais. Através delas várias pessoas passaram a se reunir com os amigos para montar a sua própria rede e disputar suas partidas. Este tipo de encontro é conhecido pelo nome de *LAN Party*. Os jogos *multiplayers* também popularizaram outro termo, as *LAN Houses*. *LAN Houses* são lojas que possuem vários computadores interligados em rede e vários jogos de maneira que, através do pagamento de uma taxa por tempo de uso dos computadores, as pessoas possam disputar partidas.

O início desta seção já deve ter deixado claro que a implementação de jogos neste estilo não é tão simples. Atualmente, são várias as pesquisas que vêem sendo desenvolvidas para solucionar problemas conhecidos destes tipos de jogos. Uma das frentes de pesquisa nesta área está relacionada com a arquitetura de comunicação entre os clientes de um jogo. Outros tópicos mais avançados correspondem a como compensar o atraso na troca de dados em uma rede muito lenta, evitando que se renderize uma imagem com atraso em relação a outro jogador que possua uma conexão mais rápida.

O tipo de arquitetura a ser utilizada no desenvolvimento do jogo modelo proposto por este trabalho foi uma decisão importante. Assim, o restante desta seção irá apresentar uma breve descrição dos modelos de arquitetura de comunicação

mais conhecidos [9]:

- **Topologia** *Peer-to-Peer* **(ponto-a-ponto)**. Em um projeto de sistema ponto a ponto, existem diferentes estações de trabalho interconectadas de maneira que cada uma delas pode enviar mensagens diretamente a qualquer ponto da rede. Aqui podem ocorrer três tipos de comunicação entre os pontos da rede:

  - **Multicast**. Neste tipo de comunicação as mensagens são enviadas a grupos de pontos dentro da rede ou até mesmo para a rede toda.
  - **Unicast**. Permite a comunicação apenas entre dois pontos de cada vez.

  Neste tipo de topologia os clientes, geralmente, mantém informações completas do estado atual do jogo. Esta característica cria um problema em potencial: o fato de ela não ser escalável, ou seja, quanto maior for o número de clientes maior vai ser o número de atualizações a serem feitas e conseqüentemente o *overhead* de mensagens será enorme. Assim, ela é freqüentemente utilizada quando o número de clientes é reduzido.

  A Figura 2.25 ilustra este tipo de arquitetura.

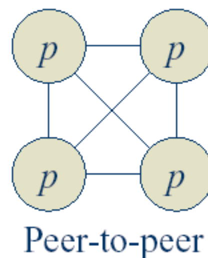

Figura 2.25: Arquitetura de comunicação ponto a ponto.

- **Topologia baseada em servidor**. Neste tipo de projeto os clientes não trocam mensagens entre si, apenas as enviam a um servidor. O servidor é o responsável por controlar a lógica do jogo, deixando apenas a renderização e outros trabalhos mais simples por conta dos clientes. Nesta arquitetura podem ocorrer duas situações, uma onde existe uma rede de servidores(*Server-Network*) e a outra na qual um único servidor (*Client/Server*) é o responsável por todo o controle do jogo.

[9] O modelo utilizado por este projeto será melhor detalhado no próximo capítulo.

Neste tipo de topologia deve-se evitar sobrecarregar o servidor, visto que ele será o elemento crítico da rede. O ideal é otimizar a freqüência com que as mensagens são enviadas. Os problemas ocorrem quando o servidor não consegue responder a tempo todas as requisições vindas dos clientes.

A Figura 2.26 ilustra os modelos de arquitetura baseados em servidor.

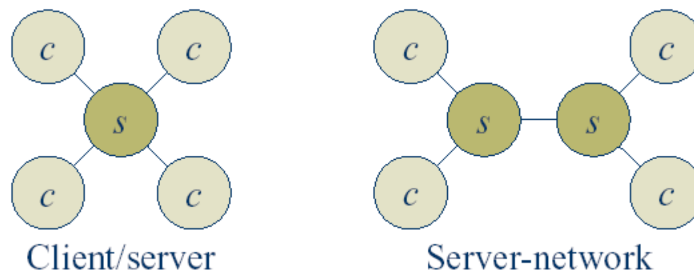

Figura 2.26: Arquitetura de comunicação baseada em servidor e suas variações.

2.8 Conexões via *sockets*

Um *socket* é um ponto final de um *link* de comunicação bi-direcional entre dois programas executando em uma rede. Um *socket* é limitado a um número de porta de tal maneira que o protocolo TCP pode identificar a aplicação para a qual os dados são destinados e enviá-los corretamente[10].

Os passos para se estabelecer uma conexão via *sockets* entre dois computadores são:

1. Deve-se estabelecer um servidor que ficará esperando por requisições de conexões por parte dos clientes. Normalmente, um servidor executa sobre um computador específico e define um *socket* que está limitado a um número de porta específico. O servidor apenas espera, escutando através do *socket*, por uma requisição de conexão de um cliente qualquer;

2. Depois de estabelecido o servidor, qualquer cliente que conheça o nome de domínio (*hostname*) do computador e o número de porta pela qual o servidor espera por conexões pode tentar fazer uma conexão;

[10]Esta seção foi escrita com base no capítulo de *sockets* do Java Tutorial[SUN2002E].

3. Se tudo ocorrer bem, o servidor aceita a conexão e obtém um novo *socket* com uma porta diferente. Esta etapa de criação de um novo *socket* é necessária para permitir que a porta atual fique livre para continuar a receber requisições de outros clientes enquanto atende as necessidades do atual cliente conectado;

4. Do lado do cliente, se a conexão é aceita, um *socket* é criado e passa a ser utilizado para realizar a comunicação com o servidor. Note que o *socket* do lado do cliente não é limitado ao número de porta usado para conectar-se ao servidor. Ao invés disso, ao cliente é atribuído um número de porta local, ou seja, da sua própria máquina;

5. O cliente e o servidor podem agora comunicar-se livremente, apenas escrevendo e lendo de seus *sockets*.

A Figura 2.27 e a Figura 2.28 representam, respectivamente, um cliente tentando realizar uma conexão com um servidor e uma conexão já estabelecida entre dois computadores.

Figura 2.27: Requisição de conexão de um cliente a um servidor via *sockets*.

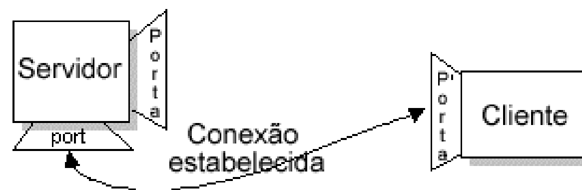

Figura 2.28: Conexão estabelecida entre um cliente e um servidor via *sockets*.

O jogo desenvolvido para a análise da plataforma J2ME utiliza *sockets* como protocolo de comunicação entre os celulares e um servidor, criado utilizando a plataforma J2SE.

Para desenvolver o servidor no qual os celulares se conectam para jogar foram utilizados os pacotes `java.net` e `java.io` da API do J2SE. O primeiro pacote oferece uma classe, `Socket`, que implementa o lado cliente de uma conexão bidirecional entre um programa Java e outro programa na rede. Para estabelecer um *socket* servidor, capaz de escutar por requisições de clientes, foi utilizada a classe `ServerSocket`.

O pacote `java.io` é utilizado para obter os fluxos de comunicação com o cliente através do *socket* criado em uma conexão aceita pelo servidor.

2.9 *Threads* de execução

A maioria dos programadores estão familiarizados com a escrita de programas seqüênciais, como aplicações que ordenam listas ou encontram os 100 primeiros números primos. Nestes tipos de programas, tudo tem um começo, uma seqüência de execução e um final [11].

Uma *thread* é similar aos programas seqüênciais que a maioria dos programadores desenvolvem. Para ser mais claro, um programa deste tipo corresponde a uma *thread* de execução iniciada pelo sistema operacional, por exemplo. Assim, uma única *thread* também possui um começo, uma seqüência e um final. Entretanto, uma *thread* não pode executar a si própria. Ao invés disso, ela executa dentro de um programa. A Figura 2.29 apresenta esta relação de um modo mais claro.

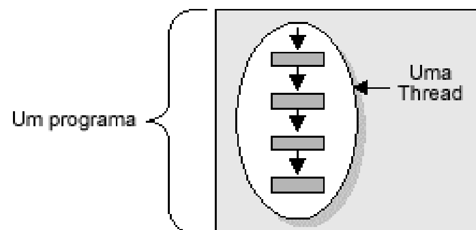

Figura 2.29: Modo de execução de uma *thread* única.

Resumindo, uma *thread* é um fluxo seqüêncial único dentro de um programa.

Como pode ser notado, não há nada de novo no conceito de uma única *thread*. A novidade está no fato de que se pode utilizar múltiplas *threads* de execução no

[11]Esta seção foi escrita com base no capítulo de *threads* do Java Tutorial[SUN2002E].

mesmo programa, ao mesmo tempo e realizando tarefas diferentes. Esta característica torna possível a construção de aplicações muito mais complexas. Este esquema é ilustrado na Figura 2.30

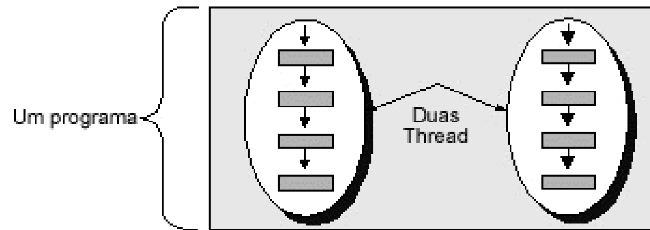

Figura 2.30: Execução de múltiplas *threads* em um único programa.

O servidor do jogo modelo desenvolvido faz uso de *threads* múltiplas para manter várias seções de jogos abertas simultaneamente, enquanto que o cliente as utiliza para poder trocar dados com o servidor sem atrapalhar o fluxo de execução normal da MIDlet.

Capítulo 3

Estudo de caso: avaliação da plataforma J2ME na implementação do jogo "Alea Jacta Est"

Este capítulo tem como objetivo principal apresentar como o jogo modelo desenvolvido para a análise da plataforma J2ME foi implementado. Esta apresentação será feita de acordo com três perspectivas: interface, dinâmica do jogo e arquitetura de comunicação utilizada. Neste capítulo também estão relacionadas as ferramentas necessárias para o desenvolvimento do projeto, as dificuldades e facilidades encontradas na implementação do jogo e algumas análises iniciais da tecnologia.

Para atingir os objetivos propostos foi realizada a divisão do capítulo em três seções: a seção 3.1 faz uma breve descrição do jogo modelo desenvolvido; a seção 3.2 discute todas as ferramentas utilizadas no projeto e a seção 3.3 apresenta a maneira como o jogo foi implementado, ressaltando as dificuldades e facilidades encontradas, e algumas análises iniciais da plataforma J2ME, seguindo as perspectivas propostas no parágrafo anterior.

3.1 O jogo “Alea Jacta Est”

“Alea Jacta Est” é um jogo de estratégia *multiplayer* em que cada participante representa um general romano. O objetivo é expandir o seu prórpio império e conquistar todos os demais generais. A vitória sobre os generais é conseguida através de disputas entre as unidades militares de cada jogador e também entre estas unidades e as cidades inimigas.

“Alea Jacta Est” faz referência a uma célebre frase pronunciada pelo general romano Júlio César e que traduzida do latim significa “A sorte está lançada”. Júlio César a pronunciou quando entrou em Roma para atacar Pompeu, que havia se rebelado contra as leis impostas pelo Senado e se declarado imperador, e viu quão fraco era o seu exército em comparação com o do inimigo. Pelo fato de a ação do jogo ocorrer na época do grande Império Romano esta frase foi escolhida para ser seu título.

A lógica do jogo é simples: após criada uma cidade, inicia-se a construção de novas unidades militares ou de outras unidades construtoras. Com isso, pode-se montar um verdadeiro exército e iniciar uma guerra, promover a paz ou construir novas cidades visando uma expansão cada vez maior do império. Assim, uma partida sempre é diferente da outra.

Algumas opções de unidades militares a serem construídas são: cohorts, pretorianos, falanges, bigas, catapultas e trirremes. Estas unidades são as reponsáveis pela maior parte da interação do jogo.

O jogo é disputado em turnos, isto é, todos os jogadores, na sua vez, devem executar uma ação para cada unidade militar ou cidade que possui antes que outro possa iniciar sua jogada. O número de jogadores simultâneos em uma partida é determinado pelo jogador que iniciou a sessão de jogo.

“Alea Jacta Est” traz para as pequenas telas dos celulares os grandes sucessos em jogos de estratégica lançados para computadores *desktop*, como *Age of Empires*[MIC2003] da *Microsoft* ou *Civilization* [MIP2003] da *Microprose*.

Maiores informações sobre o “Alea Jacta Est” podem ser encontradas no projeto *“Alea Jacta Est” Um protótipo de jogo de estratégia interativo multiplayer para dispositivos móveis*, do aluno Diego Mello da Silva do curso de Ciência da Computação, da Universidade Federal de Lavras.

3.2 Ferramentas utilizadas no desenvolvimento do jogo "AleaJactaEst"

Atualmente, existem várias ferramentas que auxiliam no desenvolvimento de aplicações utilizando a tecnologia J2ME. O objetivo principal desta seção é apresentar aquelas que foram utilizadas neste projeto, incluindo editores de texto para a escrita dos códigos, emuladores para a realização dos testes e ambientes para a compilação das aplicações.

3.2.1 Editores de código

Um editor de códigos é utilizado para escrever o código das aplicações a serem desenvolvidas. Ele pode ser um simples editor de texto, como o *notepad* do *Windows* ou o *joe* do *Linux*, ou estar incorporado a um grande ambiente de desenvolvimento integrado (IDE, do inglês *Integrated Development Environment*), como o *Sun One Studio*[ONE2003] da *Sun Microsystems* ou o *JBuilder*[JBU2003] da *Borland*.

Java não exige que se use nenhum editor em especial, permitindo ao desenvolvedor utilizar aquele que mais lhe agrada.

Alguns editores gratuitos e famosos para se escrever aplicações em Java são o *JCreator LE*[JCR2003], o *Sun One Studio* e o *JEdit*[JED2003]. Segue uma breve descrição de cada uma destas ferramentas:

- O **Sun One Studio**, como o próprio nome já diz, é distribuído pela *Sun* e é um ambiente totalmente integrado para o desenvolvimento de aplicações Java em qualquer plataforma. Está disponível em versões para *Linux* e *Windows*.

- O **JCreator LE** é uma versão gratuita de um ambiente mais poderoso chamado *JCreator Pro*. Está disponível somente para *Windows*.

- O **JEdit** é um editor muito famoso pela quantidade de *plug-ins* que oferece e também por ser totalmente escrito em Java, o que garante a sua portabilidade.

Não é fácil determinar critérios que permitam avaliar qual é o melhor editor a ser utilizado por um projeto, na maioria das vezes esta escolha é uma questão pessoal. Neste projeto, o *JEdit* foi o escolhido devido a sua grande facilidade de

uso, a alta funcionalidade apresentada, o fato de ser portável e por já ser utilizado freqüêntemente pelo autor do projeto.

A Figura 3.1 apresenta a janela de trabalho do JEdit.

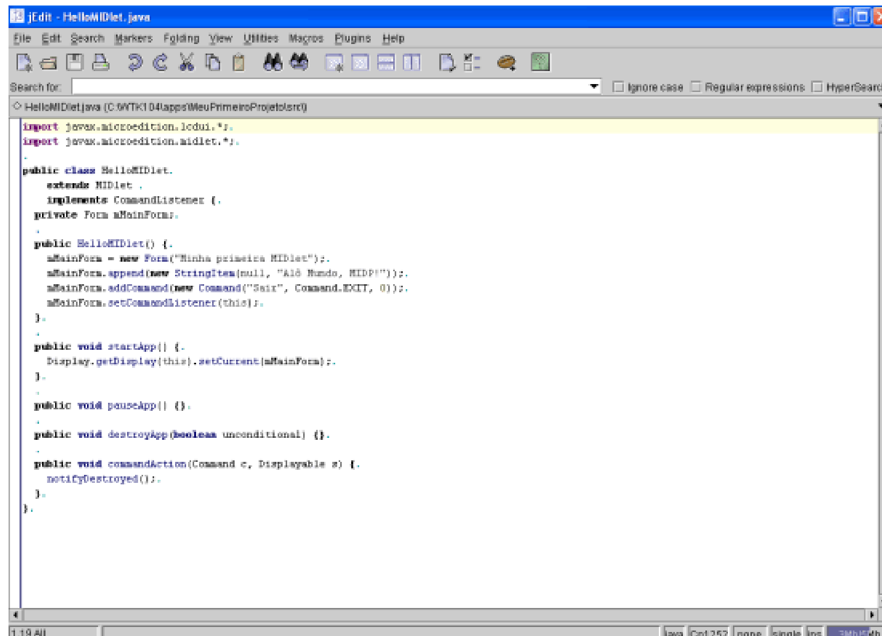

Figura 3.1: O ambiente de trabalho do editor JEdit 4.0.

3.2.2 Emuladores

Antes de iniciarmos esta seção é interessante apresentar alguns conceitos:

- Um **emulador** é um programa, ou dispositivo, que permite que um equipamento, um sistema operacional ou um programa, emule outro.

- **Emular** é comportar-se (programa ou equipamento) como outro, aceitando as mesmas entradas e produzindo as mesmas saídas, ainda que não com a mesma velocidade ou pelos mesmos processos.

Um emulador é necessário para que seja possível testar as aplicações construídas em inúmeros dispositivos sem que seja necessário comprar todos os aparelhos.

Para a realização deste projeto foram utilizados os emuladores fornecidos pela *Palm* (POSE - *PalmOS Emulator*) e pela *Sun* (integrado ao J2ME *Wireless Toolkit*). Estas ferramentas estão melhor descritas a seguir:

- **POSE**[POS2003]: o *PalmOS Emulator* é um *software* que emula o *hardware* de vários modelos de PDAs da **Palm**. Ele cria *handhelds* virtuais que podem ser utilizados no *Windows*, *Mac OS* ou *Unix* para o teste de aplicações. Para utilizar o emulador é necessário obter as ROMs dos dispositivos que se deseja emular.

- **J2ME** *Wireless Toolkit*[SUN2002D]: esta ferramenta inclui emuladores de vários dispositivos compatíveis com os *profiles* MIDP 1.0 e MIDP 2.0. Está disponível para *Windows*, *Solaris* e *Linux* e pode ser integrado a vários ambientes de desenvolvimento, como o *Sun One Studio* e o *JBuilder*.

Atualmente, o *J2ME Wireless Toolkit* está disponível em duas versões: a versão 1.0.4_01 implementa o *profile* MIDP 1.0, enquanto que a versão 2.0 Beta 2 implementa o MIDP 2.0. Os emuladores disponibilizados estes *toolkits* estão relacionados na Tabela 3.1. As características de cada um deles podem ser encontradas na Tabela 3.2[1].

As características dos aparelhos que podem ser emulados pelo POSE variam enormemente, devido a grande diversidade de dispositivos da *Palm* para os quais ele apresenta suporte. Entretanto, possuem geralmente um *display* de 160x160, resolução indo de preto e branco a 16 bits, entrada realizada por *Graffiti* e *hard buttons*, nenhum *soft button* e teclas especiais, como MENU e HOME.

O emulador da *Palm* foi utilizado apenas com a intenção de verificar a real portabilidade de J2ME, pois o foco deste projeto está em avaliar a plataforma no contexto de desenvolvimento de jogos *multiplayers* para celulares.

Um dos objetivos futuros deste trabalho é executar o jogo desenvolvido em emuladores de celulares reais, como os disponibilizados pela *Nokia*.

3.2.3 Criação de aplicações

Antes de se iniciar uma discussão sobre o processo de criação de uma aplicação é necessário apresentar uma definição para o termo MIDlet, que será freqüentemente citado. Uma MIDlet é uma aplicação feita segundo o *profile* MIDP. Caso fosse utilizado o *profile* PDA seria uma PDAlet.

[1] As tabelas foram compiladas a partir dos manuais do *Wireless Toolkit*, distribuídos pela *Sun Microsystems*[SUN2002B][SUN2003B].

Tabela 3.1: Emuladores disponibilizados pelo J2ME *Wireless Toolkit* e versões de MIDP implementadas por cada um deles.

Emulador	**MIDP**	**Descrição**
DefaultColorPhone	1.0 e 2.0	Telefone genérico com um display colorido.
DefaultGrayPhone	1.0 e 2.0	Telefone genérico com um display em tons de cinza.
MinimumPhone	1.0	Telefone genérico com um display de capacidade limitada.
RIMJavaHandheld	1.0	RIM do Research In Motion Ltd.
Motorola_i85s	1.0	Motorola i85s da Motorola, Inc.
PalmOS_Device	1.0	Palm OS Personal Digital Assistant da Palm, Inc (A emulação usa o POSE.).
MediaControlSkin	2.0	Telefone genérico com controles de áudio e vídeo
QwertyDevice	2.0	Dispositivo *handheld* genérico que usa o estilo de teclado QWERTY.

Um outro termo utilizado será a suíte de MIDlet. Uma suíte de MIDlet, segundo a *Sun Microsystems*[SUN2002B], é um agrupamento de MIDlets que podem compartilhar recursos em tempo de execução. Mais formalmente, uma suíte de MIDlet inclui:

Tabela 3.2: Características dos emuladores disponibilizados pelo J2ME *Wireless Toolkit*.

Dispositivo	**Display**	**Resolução**	**Mecanismos de Entrada**	**Número de *Soft Buttons***	**Teclas Especiais**
DefaultColorPhone	180x208	256 cores	ITU-T keypad	2	
DefaultGrayPhone	180x208	256 tons de cinza	ITU-T keypad	2	
MinimumPhone	96x54	preto e branco	ITU-T keypad	0	BACK, MENU
RIMJavaHandheld	198x202	preto e branco	teclado QWERTY	0	MENU
Motorola_i85s	111x100	preto e branco	ITU-T keypad	2	MENU
MediaControlSkin	180x208	256 cores	ITU-T keypad	2	
QwertyDevice	640x240	256 cores	teclado QWERTY	2	MENU, Shift, Ctrl, Char

- Um arquivo descritor de aplicações (*Java Application Descriptor* - JAD): apresenta um conjunto pré-definido de atributos que podem ser utilizados pelo *software* gerenciador de aplicações para identificar e instalar uma MIDlet. Pode-se definir atributos próprios além dos padronizados;
- Um arquivo *Java Archive* (JAR), que por sua vez possui:
  - Classes Java de cada MIDlet na suíte;
  - Classes Java compartilhadas entre as MIDlets;
  - Arquivos extras usados pelas MIDlets;

– Um arquivo manifesto descrevendo o conteúdo do arquivo JAR (parecido com o arquivo JAD).

A Figura 3.2 apresenta a organização dos componentes citados acima:

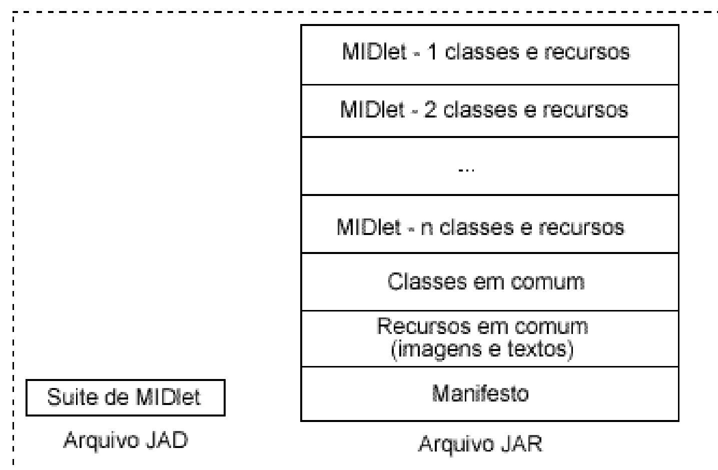

Figura 3.2: Organização de uma suíte de MIDlets.

Para compilar, distribuir e realizar testes de performance mais avançados a ferramenta J2ME *Wireless Toolkit*, o kit de desenvolvimento *wireless* da *Sun*, será novamente utilizada.

O *Wireless Toolkit* não é apenas um emulador, mas um conjunto de ferramentas que torna fácil o processo de criação, distribuição e teste de performance de aplicações MIDP (a sua interface gráfica automatiza uma série de processos) [SUN2002B].

O *KToolBar*[2], incluído no J2ME *Wireless Toolkit*, é um ambiente de desenvolvimento mínimo. Ele possui uma interface para compilação, empacotamento, teste e execução de aplicações MIDP. As únicas ferramentas extras necessárias são um depurador e um editor de códigos [SUN2002B].

A Figura 3.3 apresenta a ferramenta *KToolBar*, parte do J2ME *Wireless Toolkit*.

Ao compilar uma MIDlet, todos os arquivos *.java* no diretório de arquivos fontes do projeto atual são compilados utilizando o compilador do J2SE. Esta

[2]Uma explicação detalhada sobre como utilizar o J2ME *Wireless Toolkit* e sua ferramenta KToolBar pode ser encontrada nas duas versões dos manuais User's Guide Wireless Toolkit[SUN2002B][SUN2003B] distribuídos pela *Sun* Microsystems.

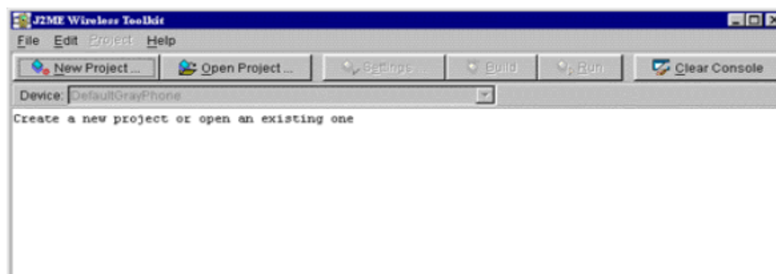

Figura 3.3: *KToolBar*, principal ferramenta do J2ME *Wireless Toolkit*.

não é uma compilação normal, devido ao fato de os arquivos deverem ser compilados em um ambiente MIDP ao invés de em um ambiente J2SE. Para entender a dificuldade deste processo basta imaginar uma MIDlet que use a classe `java.lang.System`, ela possui diferentes APIs no J2SE e no MIDP. Assim, ao compilar o projeto, deve-se ser capaz de analisar quais classes são permitidas [KNU & NOU2002].

Após a compilação, as classes MIDP devem ser pré-verificadas antes de serem executadas em um dispositivo. Isto também ocorre no J2SE, onde há um verificador de *bytecodes* que checa os arquivos *.class* antes de eles serem carregados. Entretanto, no MIDP a verificação foi dividida em duas fases. A primeira é realizada em tempo de construção do projeto e busca simplificar o estágio final da verificação de *bytecode* na máquina virtual CLDC, aumentando a eficiência de execução no dispositivo. A segunda é realizada pelo dispositivo móvel quando ele carrega as classes. [KNU & NOU2002]

Finalmente, as MIDlets são construídas em uma suíte de MIDlet para serem distribuídas aos dispositivos atuais.

Todas as etapas citadas acima são facilmente executadas através do J2ME *Wireless Toolkit*.

A Figura 3.4 e a Figura 3.5, do manual do *Wireless Toolkit*[SUN2002B], ilustram o processo de construção da aplicação e empacotamento, respectivamente.

3.3 O desenvolvimento do jogo “Alea Jacta Est”

O jogo “Alea Jacta Est” foi desenvolvido de acordo com a arquitetura cliente-servidor. Neste modelo, o servidor controla toda a lógica do jogo e o cliente fica responsável apenas pela renderização de imagens e outras tarefas mais simples.

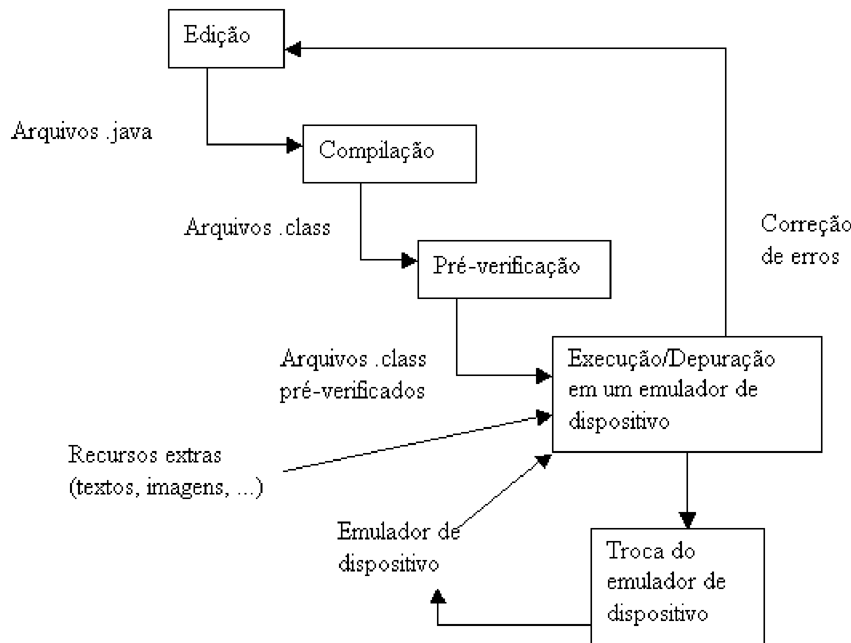

Figura 3.4: Processo de construção e teste de uma MIDlet.

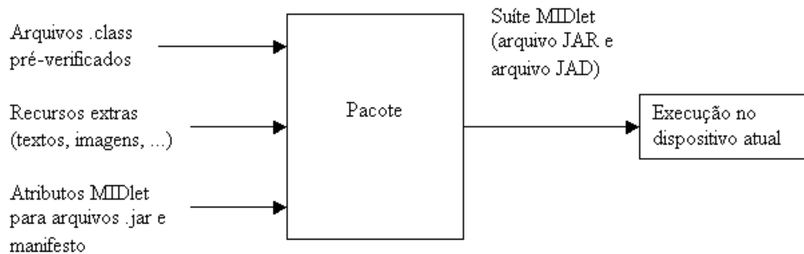

Figura 3.5: Processo de empacotamento de uma MIDlet.

Assim, no caso da construção do jogo modelo, não foi possível se preocupar apenas com a implementação do celular, que seria o cliente, mas também com a construção de um servidor para controlar toda a lógica do jogo.

O servidor, por ser o responsável por controlar todas as sessões de jogo em aberto e pela troca de dados entre os clientes conectados, não pode ser implementado em um simples aparelho celular. Há a necessidade de se executá-lo em

um computador com maior capacidade de processamento, como os computadores *desktop*, para que ele seja capaz de tratar o *overhead* de dados que possa vir a acontecer. Portanto, foi necessário utilizar outra linguagem para a implementação do servidor. A linguagem escolhida para o seu desenvolvimento, aproveitando-se da grande integração existente entre a plataforma J2ME e as outras plataformas da tecnologia Java, foi a definida pela API do J2SE.

A implementação do servidor não será apresentada neste projeto, pois o seu enfoque é na análise da plataforma J2ME. No entanto, cabe ressaltar que o J2SE supriu todas as necessidades do projeto e, em algumas vezes, foi necessário deixar de utilizar alguma funcionalidade fornecida pela arquitetura por limitações impostas pelo J2ME.

Tendo como base a experiência adquirida no desenvolvimento do "Alea Jacta Est", pode-se afirmar que, para aplicações móveis que utilizem o modelo cliente-servidor, o uso da combinação J2SE no servidor e J2ME no cliente é altamente recomendada. A integração e intercompatibilidade com outros produtos da família Java é uma característica importante de J2ME, pois permite o desenvolvimento de soluções completas baseadas em uma única tecnologia.

O cliente foi implementado de acordo com os *profiles* MIDP 1.0 e 2.0, que foram projetados especialmente para o mercado de desenvolvimento de aplicações para celulares[3]. O capítulo 2 apresenta em detalhes estes dois *profiles* e faz uma breve descrição dos outros definidos pela plataforma.

A idéia original era implementar o cliente de acordo com a especificação MIDP 1.0, até mesmo pelo fato de o MIDP 2.0 ter sido lançado durante o desenvolvimento do projeto. No entanto, a utilização da API do MIDP 2.0 tornou-se necessária devido a uma deficiência encontrada na versão inicial com relação a comunicação via *sockets*, que é a maneira como o cliente troca dados com o servidor.

"Alea Jacta Est´´ é compatível com MIDP 2.0, mas é importante deixar claro que a utilização da API fornecida por este *profile* ficou limitada apenas ao que foi necessário para suprir a deficiência encontrada na versão anterior.

No entanto, serão feitas comparações entre estas versões do *profile* MIDP durante todas as fases de implementação do jogo. Essas comparações serão úteis para mostrar as melhorias implementadas pela nova versão, principalmente as relacionadas com o novo pacote criado especialmente para facilitar o desenvolvimento de jogos baseados em *tiles*.

O conjunto de classes criadas para a implementação da parte cliente do jogo

[3]Existe uma implementação do MIDP 1.0 feita pela *Sun* para o sistema operacional Palm OS, mas o mercado de PDAs não é o objetivo deste *profile*

"Alea Jacta Est" está definido na Figura 3.6.

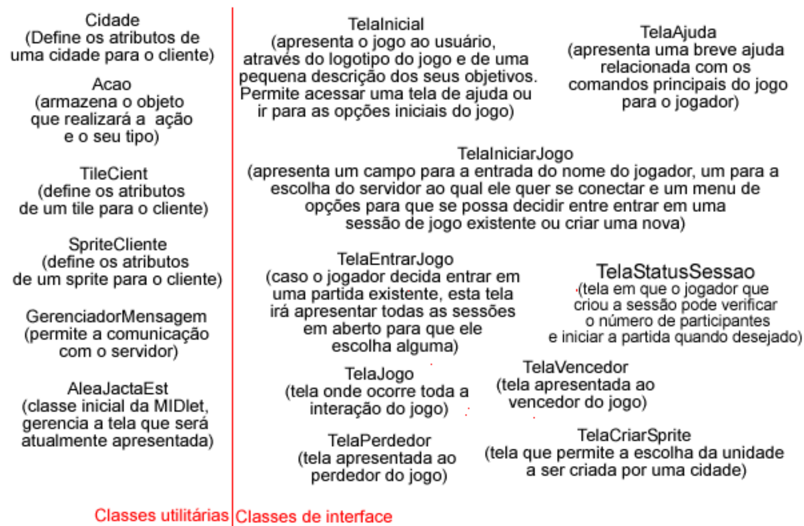

Figura 3.6: Classes que fazem parte do jogo "Alea Jacta Est".

A pilha de especificações definidas pela plataforma J2ME e que foi utilizada no desenvolvimento do "Alea Jacta Est" está ilustrada na Figura 3.7.

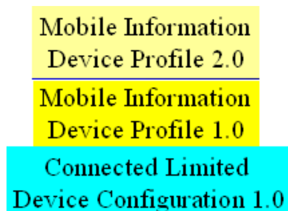

Figura 3.7: Pilha de especificações J2ME utilizada na implementacao do "Alea Jacta Est".

As próximas seções irão apresentar os detalhes de implementação (incluindo as dificuldades e facilidades encontradas), a avaliação da tecnologia e, sempre que possível, uma comparação entre as versões dos *profiles* MIDP. Tudo isto será feito de acordo com a perspectiva de interface, dinâmica de jogo e arquitetura de comunicação.

3.3.1 A interface

O fato de os dispositivos móveis, em especial os celulares, não seguirem um padrão para o tamanho de suas telas, como ocorre com os monitores tradicionais, e não possuírem grande capacidade de processamento, como os computadores *desktop*, torna a tarefa de construção de interfaces um pouco complexa.

Em geral, os projetistas de interface para dispositivos móveis devem se preocupar com duas questões básicas:

- **processamento**. A renderização de imagens na tela não pode exigir muito do processador do aparelho;

- **portabilidade**. Uma interface de um programa que deve executar em vários modelos de dispositivos precisa ser capaz de se adaptar automaticamente aos mais variados tamanhos de telas dos aparelhos móveis.

Segundo o MIDP Style Guide[SUN2002F], existem algumas metas que o desenvolvedor deve buscar atender ao desenvolver interfaces para os pequenos dispositivos móveis:

- **simplicidade**. Deve-se construir as interfaces de maneira mais simples possível, mas sem perder a funcionalidade;

- **previsibilidade**. O usuário deve ser capaz de saber o que vai acontecer após realizar alguma ação, mesmo sem ter lido qualquer tipo de manual;

- **separação de tarefas importantes**. As tarefas mais utilizadas e as mais importantes devem estar disponíveis de modo direto para o usuário. Esta atitude evita que ele tenha que ficar navegando entre telas para iniciar alguma tarefa comumente realizada;

- **rapidez de resposta**. A resposta deve ocorrer imediatamente após a disparada de alguma ação. A espera é uma atividade extremamente irritante para o usuário;

- **retorno constante**. O usuário deve sempre receber algum tipo de retorno para as suas ações e também ser constantemente informado sobre o estado de alguma ação que esteja realizando no momento;

O documento MIDP Style Guide[SUN2002F] apresenta uma discussão muito interessante sobre as maneiras corretas de se projetar interfaces para dispositivos

móveis utilizando o *profile* MIDP. Além disso, ele também possui uma visão geral de todos os componentes disponíveis para a construção destas interfaces. Assim, recomenda-se a sua leitura antes de se iniciar o desenvolvimento de *softwares* com esta especificação.

A API do *profile* MIDP 1.0, conforme apresentado no capítulo 2, proporciona dois modelos para a construção de interfaces: alto nível e baixo nível.

O jogo "Alea Jacta Est" foi desenvolvido sempre pensando em manter a portabilidade da aplicação. No entanto, não é possível desenvolver um jogo apenas com os componentes de alto nível disponibilizados. Há a necessidade de se controlar a tela completamente, permitindo a manipulação pixel a pixel dos elementos a serem inseridos. Assim, os elementos de alto nível, apesar de garantirem a portabilidade das aplicações, não puderam ser utilizados em todas as telas do jogo.

Os elementos de interface de alto nível foram utilizados na confecção das telas relacionadas com a apresentação do jogo, ajuda, opção de escolha de qual servidor se deseja conectar e se o participante irá criar um novo jogo ou entrar em uma sessão existente. Além destas, apenas a tela apresentada ao vencedor ou perdedor do jogo e a que possibilita a escolha da unidade a ser construída por uma cidade usaram componentes de alto nível.

A interface de baixo nível precisou ser utilizada na confecção da tela onde ocorre a interação do jogo.

"Alea Jacta Est", portanto, é uma aplicação desenvolvida de acordo com os dois modos de interface existentes.

O desenvolvimento de um jogo *multiplayer* seguirá sempre o padrão de se utilizar componentes de alto nível para as telas iniciais e de baixo nível para a tela onde ocorre a interação definida pelo jogo.

O restante desta seção irá apresentar como foi a implementação da interface do jogo, iniciando pelas de alto nível, onde serão apresentados trechos de códigos de algumas telas criadas, mostrando a facilidade de sua construção. Por fim, será feito uma apresentação do desenvolvimento da interface onde ocorre toda a interação do jogo, que foi desenvolvida com componentes de baixo nível.

Os detalhes de implementação apresentados neste capítulo dizem repeito a como os componentes foram utilizados e não como se pode usá-los. Para maiores informações sobre os métodos definidos por cada classe dos pacotes MIDP deve-se consultar a sua especificação, disponível no site ofical deste *profile*[MID2003].

Para iniciar a discussão, a Figura 3.8 apresenta o modelo navegacional das interfaces desenvolvidas, do ponto de vista das classes. Nesta figura, a única interface de baixo nível é a definida pela `TelaJogo`.

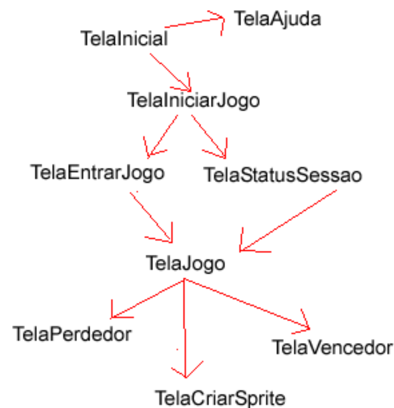

Figura 3.8: Modelo navegacional das interfaces desenvolvidas para o jogo "Alea Jacta Est". A classe `TelaInicial` é corresponde ao início do fluxo.

Para analisar os detalhes de implementação dos componentes da interface de alto nível foram escolhidas duas das classes da Figura 3.8, a `TelaInicial` e a `TelaIniciarJogo`.

A Figura 3.9 apresenta a interface definida pela classe TelaInicial e ressalta alguns detalhes de implementação.

A classe `TelaInicial`, por ser um `Form`, pode ser apresentada na tela dos celulares. O trecho de código da Figura 3.10 apresenta a definição da classe, permitindo verificar que ela herda `Form`.

O objetivo desta classe e criar uma tela de apresentação do programa. Para isso foram utilizados os seguintes elementos:

- um componente `ImageItem`, que apresenta o logotipo do jogo. O trecho de código da Figura 3.11 apresenta como ele foi implementado.

- um `StringItem` que foi utilizado como texto de apresentação e uma breve descrição do programa. O trecho de código da Figura 3.12 ilustra o seu processo de criação e utilização.

- três elementos `Command`, que permitem a realização de ações ao serem selecionados. As ações definidas para esta classe foram: sair do programa, iniciar um jogo e visualizar a ajuda. Note que, quando são inseridos mais de três `Commands` ao mesmo tempo em uma tela um menu suspenso, como o

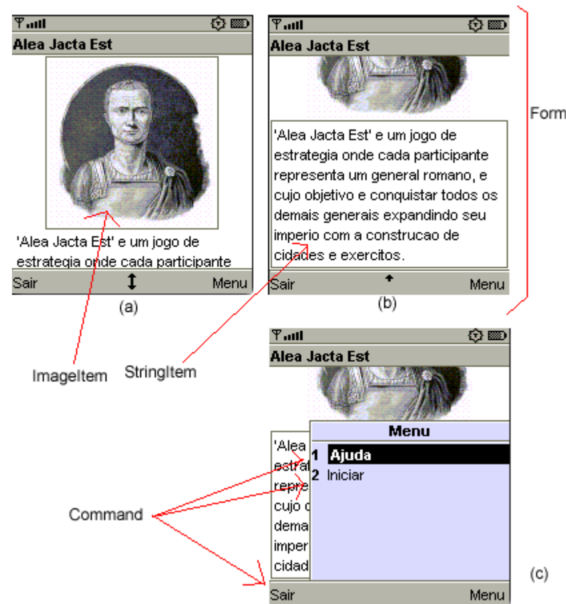

Figura 3.9: Interface definida pela classe `TelaInicial`.

```
package br.ufla.dcc.v1.aleajactaest;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TelaInicial extends Form implements CommandListener
{

    //código da classe

}
```

Figura 3.10: Trecho de código enfocando a definição da classe *TelaInicial*.

da letra **c** da Figura 3.9, é automaticamente criado. A Figura 3.13 apresenta como estes três comandos foram construídos.

É importante notar o fato de a rolagem de tela ocorrer de maneira automática, sempre que necessário. Na classe `TelaInicial`, a imagem e o texto de apresentação ocuparam um espaço maior que o *display* do celular, obrigando o surgimento da barra de rolagem.

A observação dos trechos de códigos apresentados permite verificar a facilidade na construção das interfaces de alto nível, bastando que se declare, crie e adicione, através dos métodos `append` ou `addCommand`, os componentes dese-

```
package br.ufla.dcc.v1.aleajactaest;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TelaInicial extends Form implements CommandListener
{
    private Image         imagem;
    private ImageItem     abertura;
    //...outras variáveis globais...

    public TelaInicial(AleaJactaEst princ)
    {
        //...código da função...

        // o primeiro passo para se inserir uma imagem em um Form é obter o objeto Image
        // referente ao arquivo desejado. Depois deve se criar um ImageItem a partir deste
        // objeto e adicioná-lo ao Form.
        try
        {
            //instancia o objeto que representa uma imagem a partir de um dado arquivo
            imagem = Image.createImage("/apresentacao/apresentacao.png");

            //cria o ImageItem a partir do objeto Image instanciado
            abertura = new ImageItem(null, imagem, ImageItem.LAYOUT_CENTER, null);

            //adiciona o ImageItem no Form
            append(abertura);
        }
        catch(java.io.IOException e)
        {
            //...código de tratamento de erro...
        }

        //...código da função...
    }

    //...código da classe...
}
```

Figura 3.11: Trecho de código enfocando a inserção de um `ImageItem` pela classe *TelaInicial*.

```
package br.ufla.dcc.v1.aleajactaest;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TelaInicial extends Form implements CommandListener
{
    private StringItem texto;
    //...outras variáveis globais...

    public TelaInicial(AleaJactaEst princ)

        //...código da função...

        texto = new StringItem(null, "Bem-vindo ao jogo \"Alea Jacta Est\"");
        append(texto);

        //...código da função...

        texto = new StringItem(null, "\'Alea Jacta Est\' e um jogo de estrategia onde cada "
                                   + "participante representa um general romano, e cujo "
                                   + "objetivo e conquistar todos os demais generais
                                   + "expandindo "
                                   + "seu imperio com a construcao de cidades e exercitos.");
        append(texto);

        //...código da função...
    }

    //...código da classe...
}
```

Figura 3.12: Trecho de código que mostra a criação de um `StringItem`.

jados. A única preocupação que deve-se possuir é com relação a ordem com que os elementos serão inseridos na tela, pois o tamanho e todas as outras características são definidas pela implementação do *profile* MIDP no celular.

```
public class TelaInicial extends Form implements CommandListener
{
    private Command ajuda, iniciar, sair; //declaração dos comandos u‹

    public TelaInicial(AleaJactaEst princ)
    {
        //instancia os comandos a serem inseridos na tela
        sair    = new Command("Sair", Command.BACK, 1);
        ajuda   = new Command("Ajuda", Command.SCREEN, 1);
        iniciar = new Command("Iniciar", Command.SCREEN, 1);

        //adiciona os comandos na janela do formulário
        addCommand(sair);
        addCommand(ajuda);
        addCommand(iniciar);

        //define o manipulador de eventos para os comandos
        setCommandListener(this);

        //...código da função...
    }

    //manipulador de eventos para os comandos inseridos no formulario
    public void commandAction(Command c, Displayable d)
    {
        if(c == sair)
        {
            //realiza ação
        }
        if(c == ajuda)
        {
            //realiza ação
        }
        if(c == iniciar)
        {
            //realiza ação
        }
    }
}
```

Figura 3.13: Trecho de código mostrando como inserir `Commands` em um `Form`.

A Figura 3.14 apresenta a interface definida pela classe `TelaIniciarJogo` e ressalta alguns detalhes de sua implementação. Agora, serão discutidos apenas os elementos que não foram apresentados na `TelaInicial`.

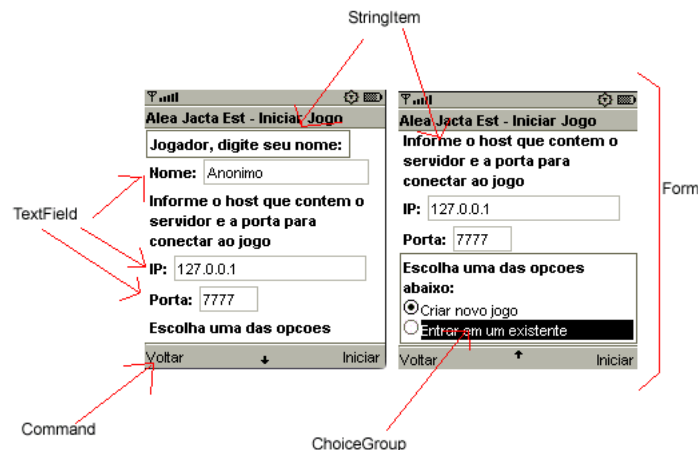

Figura 3.14: Interface definida pela classe `TelaIniciarJogo`.

A classe `TelaIniciarJogo` permite a obtenção de algumas informações do usuário, como o seu nome, o servidor e porta na qual deseja se conectar e se ele quer entrar em uma sessão de jogo existente ou criar uma nova. Esta classe também foi implementada a partir de um `Form`.

A Figura 3.15 apresenta um trecho de código da `TelaIniciarJogo` mostrando como foi inserido um `TextField` e um `ChoiceGroup` no `Form`.

```
import javax.microedition.lcdui.*;

public class TelaIniciarJogo extends Form implements CommandListener
{
    //caixas de texto utilizadas para receber entrada do usuário
    private TextField    nomeJogador, caixaIP, caixaPorta;
    //lista de elementos de seleção
    private ChoiceGroup  opcao;

    public TelaIniciarJogo(AleaJactaEst princ)
    {
        //cria um TextField e...
        nomeJogador = new TextField("Nome: ", "Anonimo", 12, TextField.ANY);
        //...o adiciona ao formulário
        append(nomeJogador);

        caixaIP = new TextField("IP: ", "127.0.0.1", 14, TextField.ANY);
        append(caixaIP);

        caixaPorta = new TextField("Porta: ", "7777", 4, TextField.ANY);
        append(caixaPorta);

        //cria uma lista de seleções do tipo exclusiva, ou seja, apenas
        // uma opção pode ser selecionada por vez
        opcao = new ChoiceGroup("Escolha uma das opcoes abaixo: ", Choice.EXCLUSIVE);

        //adiciona as opções a lista
        opcao.append("Criar novo jogo", null);
        opcao.append("Entrar em um existente", null);

        //adiciona a lista com as opçõs ao formulário para torná-la visível
        append(opcao);
    }
}
```

Figura 3.15: Trecho de código da classe `TelaIniciarJogo`.

Os elementos de construção de interfaces de alto nível disponibilizados pela versão 1.0 do MIDP atendem completamente as necessidades que possam vir a surgir quando se está desenvolvendo jogos *multiplayers*. Neste contexto, a sua utilização fica restrita a construção das telas iniciais e, em algumas vezes, de telas de *status* e de opções que possam vir a se tornar disponíveis durante o jogo.

O MIDP 2.0 adicionou novos elementos de interface de alto nível a sua API, como pode ser visto no capítulo 2. No entanto, estes novos componentes são úteis apenas na criação de aplicações *enterprise*, pois estas exigem interfaces mais complexas.

Algumas considerações que podem ser feitas com relação ao MIDP versão 1.0 no desenvolvimento de interfaces de alto nível são:

- grande facilidade na manipulação dos componentes existentes;
- alta portabilidade;
- o processamento necessário para renderizar os componentes é bastante otimizado.

A única classe do jogo "Alea Jacta Est" construída utilizando a interface de baixo nível foi a `TelaJogo`. Ela herda da classe `Canvas` e utiliza dos métodos de desenho da classe `Graphics` para pintar imagens e outras primitivas gráficas na tela.

A Figura 3.16 apresenta a interface gerada pelo código da classe `TelaJogo`.

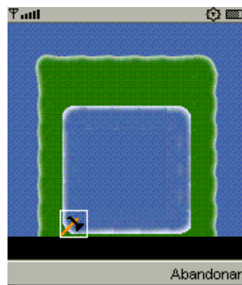

Figura 3.16: Classe `TelaJogo`, na qual toda a interação entre os jogadores ocorre.

A interface da classe `TelaJogo` foi modelada de modo a manter a portabilidade do jogo entre os mais diversos dispositivos e implementada de acordo com a idéia de *tiles*, *sprites* e janela de visualização. Assim, ao se instanciar um objeto desta classe, a primeira tarefa realizada é a verificação do número de *tiles* que podem ser inseridos no *display* do dispositivo. Esta atitude, ao invés da utilização de um número fixo, permite explorar ao máximo a tela do dispositivo do jogador e, ainda assim, garantir a portabilidade da aplicação. No entanto, quanto maior for a interface do dispositivo melhor será a jogabilidade.

Os *tiles* são utilizados para construir o mapa de fundo da tela. O "Alea Jacta Est" possui ao todo 24 *tiles* para a definição do seu mapa. Cada um deles, por sua vez, possui dimensões de 20x20 pixels. Este *tiles* são apresentados pela Figura 3.17.

Os *sprites* utilizados para a interação do jogador com os demais participantes são apresentados na Figura 3.18.

Figura 3.17: *Tiles* utilizados na construção do mapa do jogo.

Figura 3.18: *Sprites* disponibilizados para a interação com o jogo. Os números entre parênteses correspondem ao valor de ataque, defesa e tempo de construção, respectivamente.

O mapa completo do jogo constitui-se de uma matriz de altura e largura igual a 100 *tiles*. No entanto, o dispositivo MIDP mantém em memória apenas um trecho de mapa de dimensões quadradas de 20 *tiles*. Esta estrutura foi utilizada de maneira a permitir que novos mapas possam ser criados pelo servidor com os *tiles* armazenados no cliente e impedir que a matriz com os dados do mapa a serem guardados na memória fosse muito grande.

Uma alternativa seria não armazenar nenhum dado referente ao mapa em memória, fazendo com que o servidor enviasse o trecho do mapa coberto pela janela de visualização sempre que uma requisição de movimentação fosse realizada. O problema é que isto geraria um grande *overhead* de dados e atraso na repintura da tela.

A outra alternativa seria armazenar os dados completos do mapa no aparelho. Entretanto, esta abordagem faz com que seja necessário ocupar um grande espaço

da memória de armazenamento e também não permite que novos mapas sejam gerados dinamicamente pelo servidor.

A base para a rolagem de tela está na utilização da janela de visualização. Esta janela correponde a visão de usuário do mapa do jogo e possui sempre o tamanho da tela do dispositivo, limitado a dimensão do trecho de mapa. A necessidade de se rolar a tela pode ser verificada através de uma checagem se a posição (x, y) do *sprite* ou *cidade* a ser mostrado encontra-se na área determinada pela posição inicial (xjanela, yjanela) da janela e o seu tamanho.

A pintura da tela também é realizada baseada na janela de visualização. Assim, quando houver necessidade de se renderizá-la para mostrar o resultado de uma ação, o `Canvas` será preenchido de acordo com a posição inicial da janela de visualização.

A Figura 3.19 ilustra o processo de pintura determinado pela classe `TelaJogo`. Nesta figura, é interesssante notar as constantes definidas, que são referentes as dimensões do mapa, e o método `paint`, que permite o desenho em um `Canvas`. A ordem estabelecida para a pintura dos elementos na tela é a apresentada pelas funções inseridas no corpo do método `paint`. O Objeto `Graphics`, para os desenhos de primitivas gráficas, é disponibilizado como parâmetro pelo gerenciador da MIDlet quando uma renderização da tela é solicitada.

A tela do jogo sempre é repintada por completo. Assim, para evitar atraso nesta renderização, uma vez que os *tiles* são imagens que necessitam ser carregadas de um arquivo local, modelou-se um sistema que acessa os arquivos de imagens apenas quando se muda de trecho de mapa.

O sistema de otimização de pintura da tela consiste em armazenar os índices dos *tiles* do trecho de mapa em um vetor e fazer uma busca por aqueles que são utilizados neste trecho, armazenando os *tiles* necessários para a pintura da tela em uma lista de objetos `TileClient` (que armazenam o índice e uma referência a sua imagem de arquivo correspondente). Assim, quando se for repintar a tela, basta procurar nesta lista pelo *tile* correspondente ao índice desejado e recuperar a referência a sua imagem. Deste modo, utiliza-se as referências em memória ao invés de se acessar o sistema de arquivos. Este processo é realizado sempre que uma notificação informando o novo trecho de mapa for recebida.

O tratamento de colisão entre os *sprites* é realizado pelo servidor. Quando o jogador do turno faz uma solicitação de movimentação de *sprite*, o servidor verifica se a nova posição coincide com algum *sprite* ou cidade dos outros jogadores da mesma partida e se o *tile* de fundo também permite o movimento. Caso alguma colisão ocorra, o próprio servidor toma as decisões necessárias para tratá-la. A

```
public class TelaJogo extends Canvas implements CommandListener
{
    public static final int LARGURA_MAPA   = 100;
    public static final int ALTURA_MAPA    = 100;
    public static final int LARGURA_TRECHO = 20;
    public static final int ALTURA_TRECHO  = 20;
    public static final int DIMENSAO_TILE  = 20;

    protected void paint(Graphics g)
    {
        //...
        int         posX, posY;

        if(acaoCorrente < totalAcoes)
        {
            g.setColor(0, 0, 0);
            g.fillRect(0, 0, getWidth(), getHeight());


            //...

            pintarFundo(g, posX, posY);
            pintarCidadesInimigas(g, posX, posY);
            pintarSpritesInimigos(g, posX, posY);
            pintarCidades(g, posX, posY);
            pintarSprites(g, posX, posY);

            //...

        }
    }

    //...

}
```

Figura 3.19: *Tiles* utilizados na construção do mapa do jogo.

classe `Tile` implementada no servidor possui um atributo que indica se o *tile* pode ser ultrapassado ou não pelo cliente.

As primitivas de desenho do MIDP 1.0 foram suficientes para os propósitos do jogo desenvolvido e, acredita-se, que sejam também adequadas para a construção de outros jogos *multiplayers* baseados em *tiles*. No entanto, a API não disponibiliza nenhuma facilidade para a manipulação e criação de *tiles*, *sprites* e da janela de visualização, sendo necessário construir todas as classes para abstrair os conceitos destas primitivas. A Figura 3.20 ilustra a relação entre as classes construídas. Entretanto, o projeto de jogos que não seguem o modelo 2D baseados em *tile* não é tão simples de ser elaborado nesta plataforma, apesar de ser possível.

Uma das novidades da especificação MIDP 2.0 foi a criação de um novo pacote exclusivo para a manipulação de jogos. Este pacote simplifica o desenvolvimento de jogos que seguem o estilo do "Alea Jacta Est". Para ser mais claro, ele disponibiliza toda a API necessária para a manipulação de *sprites*, *tiles* e janelas de visualização. A seção 2.4.5 do capítulo 2, apresenta estas classes de maneira

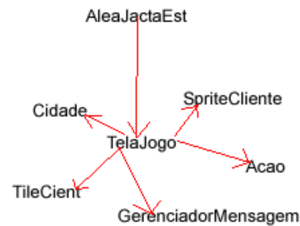

Figura 3.20: Estrutura de classes criadas para utilizar a idéia de *sprite*, *tile* e janela de visualização.

completa.

Os tópicos a seguir apresentam como algumas classes do pacote de jogos do MIDP 2.0 poderiam facilitar e melhorar a implementação do "Alea Jacta Est".

- A classe **Sprite** implementa métodos de verificação de colisão com outro *sprite* ou *tile* na tela, mas como esta verificação é feita pelo servidor não chega a ser uma vantagem para jogos *multiplayers*. A criação de *sprites* animados é possível através da definição de um conjunto de quadros de mesmo tamanho e uma seqüência de exibição. A utilização de tais *sprites* seria interessante para aplicar animação as unidades militares disponibilizadas.

- A classe **TiledLayer** já abstrai toda a estrutura de matriz de *tiles* criada para a manipulação da tela.

- A classe **LayerManager** possibilita a manipulação de várias `TiledLayer` de modo a gerenciar o modo como a tela é repintada. Abstrai também o conceito de janela de visualização, permitindo que a classe que gerencia o jogo se preocupe apenas com a manipulação correta dos índices (x, y) que indicam a posição atual da janela no sistema de coordenadas do `LayerManager`.

A utilização da API de jogos do MIDP 2.0 é altamente recomendada pelo fato de já fornecer as implementações otimizadas de quase todos os elementos necessários a criação de jogos 2D baseados em *tiles*.

Caso o "Alea Jacta Est" tivesse sido elaborado a partir da API de jogos do MIDP 2.0 o seu desenvolvimento se daria de um modo muito mais rápido e um ganho de performance seria obtido.

A classe `TelaJogo` é a responsável por gerenciar os processos de renderização da interface do jogo.

Algumas outras considerações que podem ser realizadas com relação ao projeto de interfaces em MIDP 1.0 para jogos *multiplayers* são:

- dificuldade de se manipular gráficos 3D, pois não existe suporte a valores de pontos flutuante e primitivas gráficas de desenhos de triângulos. Com isso, não é possível realizar os desenhos de formas geométricas em terceira dimensão. Esta característica também permanece na versão 2.0 do *profile*. O suporte a ponto flutuante é uma característica da configuração. A configuração CLDC versão 1.1 deve prover este tipo de suporte.

- não apresenta suporte a inclusão de sons nas aplicações, o que pode prejudicar a interação do jogo. No entanto, MIDP 2.0 já apresenta um pacote dedicado a este propósito.

- o suporte a transparência nas figuras não é obrigatório na versão 1.0, apenas na 2.0. Isto faz com que seja necessário criar um arquivo para cada combinação de *sprite* e fundo possível de ser ultrapassado para que se obtenha o efeito de superposição. Essa abordagem consome mais espaço no aparelho.

3.3.2 A dinâmica do jogo

No cliente, a classe responsável por coordenar a renderização da tela, armazenar as informações necessárias do jogador, manipular estes dados e formatar as mensagens que serão definidas a partir das interações do jogador com o dispositivo é a `TelaJogo`.

Para ilustrar o fluxo de execução implementado, o processo de criação de uma cidade pelo jogador que se encontra em seu turno será detalhado. Em MIDP 1.0 existem três *threads* que em conjunto determinam o fluxo de controle do jogo: a de manipulação de eventos de teclado, a de pintura e a de controle. A Figura 3.21 ilustra o fluxo de jogo definido para a *thread* de manipulação de eventos de teclado no “Alea Jacta Est”.

Voltando ao exemplo proposto, o jogador que deseja criar uma cidade pressiona o botão 0 do aparelho celular. Ao pressionar uma tecla o manipulador de eventos de teclado é ativado e inicia o processamento desta tecla. No caso do “Alea Jacta Est”, ele primeiro verifica se a ação que está sendo executada é de um *sprite* ou de uma cidade, pois para cada um destes elementos as teclas são mapeadas de modo diferente. Como a ação é de um *sprite* ele vai iniciar o processamento

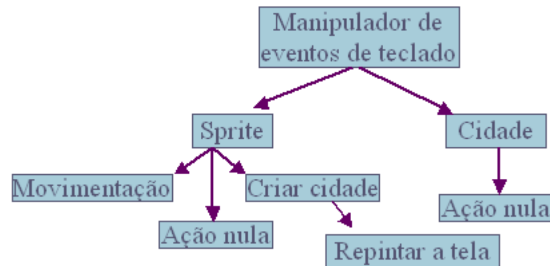

Figura 3.21: Fluxo de execução criado para a *thread* de manipulação de eventos de teclado no “Alea Jacta Est”.

que foi definido para a tecla 0 deste elemento de interação. Neste caso, é iniciado o processo de criação de uma cidade, envolvendo troca de mensagens com o servidor. A troca de mensagens com o servidor para a situação onde não há problema com relação a criação da cidade no local desejado é ilustrada na Figura 3.22.

Wendel --> CID_ACT;0; --> Servidor
Wendel <-- SPR_DES;0; <-- Servidor
Wendel <-- CID_NOT;0; <-- Servidor

Figura 3.22: Mensagens trocadas entre o jogador Wendel e o servidor durante o processo de criação de uma cidade com sucesso.

Após trocadas as mensagens e atualizado o estado do jogo no celular é solicitada uma repintura da tela. Neste ponto, uma falha da especificação MIDP 1.0 pode ser identificada. Não há como garantir que o método `paint()` seja executado no momento em que é chamado, pois por estar em uma *thread* separada quem determina isso é a MIDlet. A única coisa que se pode fazer é solicitar que uma renderização seja feita quando houver a oportunidade. A classe `GameCanvas` do MIDP 2.0 corrige este problema permitindo que se force a repintagem da tela e que se obtenha o objeto de desenho `Graphics` de qualquer ponto do jogo.

O controle de execução baseado em *threads* separadas dificulta a criação de um jogo e a manutenção do seu estado. Além disso, permite a ocorrência de situações em que o jogador executa duas ações sem ter sua tela repintada ou, até mesmo, renderizada de acordo com uma mistura dos dados de cada uma dessas ações.

No exemplo adotado, supondo que a renderização da tela ocorresse simultaneamente com a sua solicitação, a cidade seria desenhada no lugar do *sprite* constru-

tor que iniciou o seu processo de construção.

Uma das dificuldades que este tipo de controle provocou no desenvolvimento do "Alea Jacta Est" foi com relação a atualização da tela do jogador que não se encontra em seu turno. Este jogador permanece em um *loop* onde ele fica escutando por mensagens do servidor e processando-as. Algumas destas mensagens estão relacionadas com a destruição de seu próprio *sprite* por um inimigo ou a movimentação deste. Assim, o ideal seria que o jogador tivesse a sua tela atualizada de maneira que ele pudesse acompanhar o desenrolar da ação do jogo. No entanto, apesar de ser solicitada uma repintura de tela sempre que mensagens deste tipo sejam recebidas ela não ocorre até que o jogador inicie o seu turno. Esta característica prejudicou muito a interatividade do "Alea Jacta Est".

Para cada ação realizada por um jogador em seu turno, uma mensagem é enviada para o servidor. O número de mensagens de retorno, entretanto, é variável.

Pode-se fazer mais duas considerações sobre a dinâmica de execução em MIDP 1.0 para jogos *multiplayers*:

- não é permitido verificar o estado atual das teclas do aparelho. A única maneira de se tratar as ações do usuário é através do manipulador de eventos padrão, que executa em uma *thread* separada. Entretanto, MIDP 2.0 permite esse acesso;

- o acesso as teclas do dispositivo é realizada de maneira simples e portável pela API.

A Figura 3.23 apresenta de uma maneira clara como é o *loop* de controle implementado por um jogo nas duas versões do *profile* MIDP.

A Figura 3.24 apresenta o trecho de código de um *loop* de controle implementado de acordo com o *profile* MIDP 1.0 e a Figura 3.25 o implementado pela versão 2.0 deste *profile*.

A classe `GameCanvas` permite que toda a funcionalidade do jogo ocorra dentro de um simples *loop*. Neste *loop* geralmente são realizadas as seguintes tarefas:

1. obtenção do objeto `Graphics` para desenho na tela;

2. atualização do estado do jogo;

3. obtenção do estado das teclas do jogo;

4. processamento da entrada do cliente;

Figura 3.23: *Loops* de controle de execução definidos pelos *profiles* MIDP 1.0 e 2.0.

```
public void Game
    extends Canvas
    implements Runnable {
  public void run() {
    while (true) {
      // Atualiza o estado do jogo.
      repaint();
      // Espera por um tempo.
    }
  }

  public void paint(Graphics g) {
    // Código de pintura.
  }

  protected void keyPressed(int keyCode) {
    // Tratamento de eventos de teclado
  }
}
```

Figura 3.24: Trecho de código que mostra a estrutura tradicional utilizada por uma classe em MIDP 1.0 para um *loop* de jogo.

5. código de pintura da tela;

6. forçar a pintura dos elemento modificados pelos passos anteriores;

7. esperar por algum tempo.

```
public void Game
    extends GameCanvas
    implements Runnable {
  public void run() {
    while (true) {
      // Atualiza o estado do jogo
      int keyState = getKeyStates();
      // Responde ao estado das teclas pressionadas.
      Graphics g = getGraphics();
      // Código de repintura da tela.
      flushGraphics(); // Força a repintagem de tela
      // Espera por um instante de tempo.
    }
  }
}
```

Figura 3.25: Trecho de código que mostra a estrutura tradicional utilizada por uma classe em MIDP 2.0 para um *loop* de jogo.

3.3.3 A arquitetura de comunicação

"Alea Jacta Est", como já foi mencionado anteriormente, foi implementado utilizando o modelo de comunicação cliente-servidor. Este tipo de arquitetura é freqüentemente encontrada nos jogos *multiplayers* de computadores *desktop*.

A seção 2.7 apresenta outro tipo de arquitetura muito conhecida, o modelo *peer-to-peer*. Nesta abordagem, ao invés de se centralizar todas as decisões do jogo em um único computador que executa um aplicativo servidor, os clientes são os responsáveis por controlar a lógica do jogo. Assim, exige-se uma maior capacidade de processamento e o armazenamento local de quase todos os dados relativos ao jogo.

Alto grau de processamento e grande capacidade de armazenamento não são características comuns em aparelhos celulares. Portanto, a arquitetura cliente-servidor é a mais indicada para se trabalhar com este tipo de aparelho. Ela permite realizar o processamento todo do lado do servidor e deixar o cliente livre para a execução de tarefas mais simples, como a renderização da tela e a captura das ações dos usuários. O servidor é o único elemento dentro da arquitetura que mantém todos os dados do jogo constantemente atualizados.

Na arquitetura *cliente-servidor*, a implementação do servidor e os modelos de mensagens trocados entre ele e o cliente devem ser muito otimizadas. Assim, evita-se o *overhead* de dados e a incapacidade de processamento de todas as requisições das conexões atuais.

Neste ponto da implementação foi verificada uma grande deficiência na API da versão 1.0 do *profile* MIDP, a falta de suporte a abertura de conexões via *sockets*. O único protocolo exigido pela especificação MIDP é o HTTP. Este protocolo,

por ser do tipo *request-response*, não pode ser utilizado pelo jogo e muito menos por qualquer outro jogo *multiplayer*, uma vez que há situações onde é necessário enviar dados para o cliente sem que este os tenha requisitado.

Outra característica do protocolo HTTP que dificulta a sua utilização em aplicações *multiplayers* está no fato de ele ser baseado em sessões e cada uma ser independente da outra. Por isso, feito um *request* e recebido o *response* a sessão de comunicação será encerrada, não havendo meio de se manter o estado da comunicação durante toda a partida sem a utilização de algum tipo de controle extra. A sessão também é encerrada caso o *response* demore para ocorrer após um tempo pré-definido.

No "Alea Jacta Est", quando não se possui a vez, o jogador é colocado em uma situação onde permanece escutando por mensagens do servidor e processando-as. Este tipo de comunicação é impossível de ser realizada sobre um protocolo que não mantém informações sobre os clientes conectados durante toda a partida e também não permite o envio de mensagens quando estas não são solicitadas.

No entanto, a versão 2.0 do *profile* MIDP apresenta a estrutura de classes necessárias a abertura de conexões via *sockets*, mas ainda não obriga a sua implementação. O MIDP 2.0 exige apenas que o aparelho tenha suporte ao HTTP e ao HTTPS. Entretanto, enquanto na versão 1.0 não é definido nem ao mesmo as classes que um fabricante deveria escrever caso quisesse permitir o uso do protocolo de *sockets*, a versão 2.0 deixa claro que caso ele venha a ser oferecido deve-se implementar as classes presentes na documentação da especificação.

No MIDP 1.0 o suporte a *sockets* é realizado apenas através de APIs próprias do fabricante, obrigando uma reescrita do código de acordo com o aparelho em que se deseje executar a aplicação. Entretanto, no MIDP 2.0 todos os dispositivos que permitem a utilização deste protocolo implementam a mesma API.

No MIDP 2.0 utiliza-se da classe `SocketConnection` do pacote `javax.microedition.io` para fazer alguma requisição de conexão a um servidor via *sockets*.

O fato de não haver suporte obrigatório a um protocolo de comunicação tão comum quanto *sockets* em MIDP 1.0 representa uma grande falha desta plataforma na hora de se desenvolver jogos *multiplayers*. No caso de aplicações *enterprise* o protocolo HTTP é suficiente. Já o MIDP 2.0, apesar de definir uma API para *sockets*, ainda não obriga a sua implementação. Assim, o problema de suporte a este tipo de conexão foi apenas parcialmente resolvido na nova versão da especificação.

O jogo "Alea Jacta Est" só não é completamente portável devido ao fato de utilizar *sockets* para se conectar ao servidor.

A Figura 3.26 apresenta um trecho de código com a abertura de uma conexão via *sockets* com o servidor escolhido pelo jogador.

```
// Armazena a conexao a ser estabelecida com o servidor
private SocketConnection conexao = null;
// Armazena o fluxo de entrada
private InputStream      fluxoEntrada;
// Armazena o fluxo de saida
private OutputStream     fluxoSaida;

// Inicia a conexao com o servidor especificado, obtendo
// os fluxos de I/O
private void iniciarConexao(String url)
{
    // Tentando iniciar a conexao
    try
    {
        conexao = (SocketConnection) Connector.open(url);

        fluxoEntrada = conexao.openInputStream();

        fluxoSaida = conexao.openOutputStream();
    }
    catch(IOException e)
    {
        // Tratamento de erros
    }
}
```

Figura 3.26: Trecho do código do “Alea Jacta Est” que mostra a abertura de conexões por *sockets*.

Na arquitetura cliente-servidor, além de se preocupar com o protocolo de comunicação, é necessário definir o formato de mensagens a serem trocadas.

As primitivas envolvidas na comunicação do jogo modelo construído são classificadas em três categorias: primitivas de ação (formatadas pelo cliente), primitivas de notificação (formatadas pelo servidor) e primitivas de gerência de sessão de jogo (abertura e manutenção das partidas). Um subconjunto delas está relacionado na listagem abaixo:

- `NOV_ACT`: utilizada para solicitar ao servidor a criação de uma nova sessão de jogo.
- `SES_NOT`: primitiva enviada como resposta a uma solicitação de criação de sessão de jogo ou em resposta a solicitação de participar de uma sessão de jogo pré-existente.
- `COM_ACT`: primitiva enviada pelo jogador que criou a sessão solicitando o início da mesma.
- `COM_NOT`: primitiva enviada pelo servidor para o jogador que realizou uma solicitação `COM_ACT` indicando o início do jogo.

- `MOV_ACT`: primitiva utilizada para indicar uma ação envolvendo o movimento de alguma unidade militar.

- `CID_ACT`: primitiva utilizada para indicar uma requisição para construir uma cidade a partir de uma unidade *construtora*.

- `SPR_ACT`: primitiva utilizada para indicar uma requisição para construir unidades militares a partir de uma cidade pré-existente.

- `MAP_ACT`: primitiva utilizada para codificar uma requisição para o servidor enviar um trecho do mapa do jogo.

Para cada ação de um cliente enviada ao servidor ele recebe uma notificação correspondente. Um exemplo desta comunicação pode ser verificado através da troca que ocorre quando uma mensagem de MOV_ACT é enviada pelo cliente ao servidor. O formato da primitiva MOV_ACT é o seguinte:
MOV_ACT;<id_sprite>;<posX>;<posY>;
Uma possível "conversa" com o servidor na qual o cliente quer movimentar o *sprite* de identificador igual a 5 para a posição (10, 15) ocorreria da seguinte maneira:

1. Cliente envia para o servidor: MOV_ACT;5;10;15;

2. Servidor recebe a mensagem, processa, verifica que a movimentção é possível e retorna para o cliente: MOV_NOT;5;10;15;

O formato das demais mensagens trocadas entre o jogador e o servidor não será apresentado neste projeto. A discussão será realizada com relação as maneiras utilizadas para se manipular tais mensagens.

As duas versões do *profile* MIDP não apresentam suporte a serialização de objetos. Por isso, não existe a possibilidade de se transmitir e receber objetos através dos fluxos de comunicação com o servidor. Para se transmitir dados é necessário convertê-los em *bytes*. Esta abordagem exige que se utilize um sistema de cabeçalho fixo em conjunto com as mensagens a serem transmitidas.

A estrutura de cabeçalhos foi implementada da seguinte forma:

- Primeiro definiu-se um tamanho padrão para o cabeçalho, tanto do lado do servidor quanto do cliente. Este valor foi fixado em três *bytes*;

- Ao se transmitir uma mensagem, deve-se calcular o seu tamanho, montar o cabeçalho de acordo com este tamanho, acrescentar a mensagem em *bytes* ao cabeçalho e, finalmente, enviar os dados.

- Ao receber uma mensagem, deve-se, inicialmente, ler os três primeiros *bytes* do fluxo de entrada, verificar o valor obtido (que corresponde ao tamanho da mensagem), criar um novo vetor de *bytes* para a mensagem com o tamanho especificado no cabeçalho e só então ler o fluxo de entrada para obter o restante dos dados.

A Figura 3.27 ilustra o processo de definição de um cabeçalho para uma primitiva de comunicação.

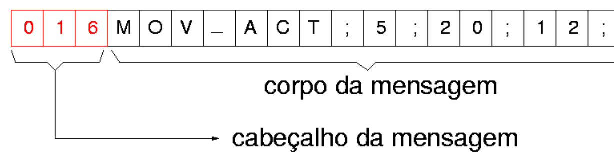

Figura 3.27: Processo de construção de um cabeçalho para uma primitiva de comunicação.

A utilização de um sistema de cabeçalhos é útil pelo fato de permitir que se leia de um fluxo o número de *bytes* corretos de uma mensagem. Com isso, impede-se que se leia caracteres a menos, prejudicando a interpretação da mensagem, ou a mais, obtendo um trecho da próxima mensagem enviada ou fazendo o fluxo esperar por um caracter que ainda não foi enviado pelo servidor.

Em jogos *multiplayers* os dados devem ser transmitidos de maneira rápida entre o cliente e o servidor, para evitar atrasos de renderização nas imagens. Por isso, como a transmissão de *bytes* é mais rápida, ela deve ser preferencialmente utilizada. Assim, o fato de a transmissão de objetos não ser possível em MIDP não representa um ponto negativo da plataforma J2ME.

No *"Alea Jact Est´´*, as mensagens são geradas em formato de `String` e convertidas em *bytes* antes de serem enviadas. A API do MIDP 1.0 foi bastante completa nas tarefas de conversão de uma `String` em um vetor de *bytes* a ser enviado e na de um vetor recebido em uma `String` a ser processada. Não há nenhuma crítica negativa a ser feita com relação a esta parte da implementação.

No jogo "Alea Jacta Est", um objeto da classe `GerenciadorMensagem` é o responsável por enviar e receber dados. Este objeto, ao ser instanciado, obtém o

número da porta e o endereço do servidor no qual o cliente deseja se conectar. Por ser uma *thread*, ele deve ser iniciado logo em seguida.

A utilização da classe `GerenciadorMensagem` como uma *thread* foi necessária pelo fato de a leitura e escrita em *sockets* travar a execução da MIDlet caso não ocorra em um contexto de execução separado.

Ao se iniciar a execução da *thread* é aberta uma conexão com o servidor, obtido os fluxos de entrada e saída e, em seguida, passa-se para o estado de processamento das mensagens.

O trecho de código ilustrado na Figura 3.28 apresenta o *loop* que permite o envio ou o recebimento de mensagens.

```
private void processar()
{
    emExecucao = true;

    while(emExecucao)
    {
        if(enviar)
        {
            enviarMensagem(stringEntrada);

            enviar = false;
        }

        if(receber)
        {
            stringSaida = receberMensagem();

            receber = false;
        }

    }
}
```

Figura 3.28: Trecho de código mostrando a maneira como o cliente envia ou recebe mensagens.

Na classe `GerenciadorMensagem` foram definidas rotinas que lêem e escrevem nos fluxos de acordo com o sistema de cabeçalhos definido.

A seqüência de troca de mensagens entre o servidor e a MIDlet ocorre da seguinte maneira: ao receber uma mensagem de notificação dizendo que o turno é do jogador em questão, inicia-se uma contagem do número de ações que poderão ser realizadas pelo jogador (que varia de acordo com o número de *sprites* e de cidades que não estão em produção). Passada esta etapa, o jogador pode executar ações de movimento, construção de *sprites*, construção de cidades entre outros.

Cada ação gera uma mensagem que é enviada ao servidor para que ele a processe e retorne as notificações necessárias. Enquanto isso, os demais participantes encontram-se em um *loop* que os obriga a ler o fluxo de entrada com o servidor até que o jogador atual termine seu turno e o servidor decida qual jogador possuirá a

vez.

A Figura 3.29 apresenta o trecho de código onde o jogador, que não está na sua vez, fica parado lendo as mensagens enviadas pelo servidor.

```
private void escutarNotificacao()
{
    String  mensagem = "";

    if(emJogo)
    {
        while(!meuTurno)
        {
            // Recebendo notificacoes oriundas do servidor
            mensagem = principal.gerenciadorMensagem().ler();

            processarMensagem(mensagem);
        }

        prepararAcoes();

        renderizarTela();
    }
}
```


Figura 3.29: Trecho de código mostrando o *loop* onde o jogador permanece até que inicie o seu turno.

Capítulo 4

Resultados e discussões

Este capítulo objetiva expor em maiores detalhes e de um modo mais claro as caracteríticas positivas e negativas encontradas na plataforma J2ME para o desenvolvimento de jogos *multiplayers* para celulares.

Para cumprir o objetivo proposto, o capítulo foi dividido em duas seções. A seção 4.1 apresenta as características identificadas no projeto que justificam a escolha da plataforma J2ME para o desenvolvimento de jogos *multiplayers* e a seção 4.2 aquelas que poderiam ser responsáveis pela não escolha da tecnologia no desenvolvimento de aplicações com estas características. Estas análises serão realizadas tendo como base o *profile* MIDP 1.0, pelo fato de este perfil ser o mais adequado para o desenvolvimento de aplicações para celulares em J2ME e a versão 1.0 ter sido utilizada durante a maior parte da implementação do jogo modelo. No entanto, sempre que for necessário, será realizada uma referência a versão 2.0 do *profile*.

4.1 Vantagens na utilização do *profile* MIDP 1.0 para a construção de jogos *multiplayers*

- **Integração com outros produtos da plataforma Java**. Esta característica permite o desenvolvimento de soluções completas baseadas na tecnologia Java de ponta a ponta;

- **linguagem de programação com sintaxe simples**. A linguagem Java, que é a utilizada por J2ME, possui uma sintaxe de fácil compreensão, responsável por facilitar o desenvolvimento e a depuração dos códigos;

- **linguagem orientada a objetos**. A linguagem Java é puramente orientada a objetos. Esta característica é muito importante no desenvolvimento de grandes projetos, como um jogo *multiplayer*;

- **manipulação de todos os botões padrões do teclado de celular de um modo portável**. Esta característica garante que a função atribuída a um botão será a mesma em todos os aparelhos celulares;

- **manipulação de eventos gerados por todos os botões do celular**. Em MIDP, pode-se responder a eventos dos *soft buttons*, dos botões de navegação e de todo o teclado padrão do celular;

- **primitivas de desenho bem completas**. Os métodos da classe `Graphics` para o desenho em um `Canvas` são bem completos para o caso de primitivas 2D;

- **API bem estruturada**. As classes são bem organizadas em pacotes dentro da especificação, facilitando o entendimento dos seus objetivos;

- **disponibilidade de ferramentas de desenvolvimento**. A *Sun Microsystems* disponibiliza todas as ferramentas necessárias para o desenvolvimento em J2ME de modo gratuito;

- **documentação completa**. A *Sun Microsystems* também disponibiliza em seu site oficial uma série de artigos e tutoriais que cobrem todos os tópicos desta arquitetura;

- *garbage collector*. O coletor de lixo automático permite o desenvolvimento de códigos mais confiáveis, permitindo que não se preocupe com alocação de memória;

- **pacotes otimizados para a maioria das estruturas de dados**. A API do MIDP inclui classes otimizadas para a maioria das estruturas de dados comumente utilizadas, como a `Vector`;

- **suporte a** ***multithreading***. A utilização de *threads* é útil em jogos de tempo real, onde uma *thread* de execução pode ser definida para escutar o fluxo de entrada de dados do servidor enquanto outra controla o fluxo normal do programa.

- **facilidade na manipulação de** *bytes*. A API do MIDP 1.0 é bastante completa com relação ao fornecimento de métodos para a manipulação de *bytes*. Esta característica facilita a manipulação das mensagens a serem trocadas entre o cliente e o servidor.

4.2 Desvantagens na utilização do *profile* MIDP 1.0 para a construção de jogos *multiplayers*

- **falta de suporte a ponto flutuante e primitivas de desenho de triângulos**. Esta característica dificulta o uso de gráficos 3D no celular;

- **repintagem da tela ocorre em** *thread* **separada**. Uma vez solicitada uma requisição de repintura da tela, não há como saber quando este processo será realizado. Isto pode fazer com que sejam geradas imagens estranhas no *display* do usuário, já que pode-se repintar a tela durante o processo de mudança do estado atual do jogo, por exemplo. No MIDP 2.0 pode-se forçar a repintura no fluxo normal de execução do programa, fazendo com que a execução do código não continue até que esta seja completada;

- **manipulação dos eventos de pressionamento de teclas ocorre em** *threads* **separadas**. Esta característica pode causar o mesmo tipo de problema do tópico anterior, ou seja, pode-se acionar um botão durante a repintagem da tela, fazendo com que o resultado da ação não seja mostrado corretamente no aparelho. No MIDP 2.0 este problema foi solucionado pela adição de um método que permite a verificação do estado atual das teclas do jogo durante a *thread* principal do programa;

- **falta de suporte a transparência nas imagens**. Esta característica impede que se obtenha o efeito de superposição de *tiles*. No MIDP 2.0 este suporte já foi implementado;

- **suporte único a imagens do tipo** *png*. Este tipo de imagem gera arquivos muito grandes, ocupando grande quantidade de espaço no dispositivo;

- **falta de suporte ao protocolo de comunicação via** *sockets*. Esta é a principal deficiência do MIDP, uma vez que este protocolo é essencial em alguns tipos de aplicações;

- **falta de suporte a sons**. Em alguns casos, a interface com o jogador pode ficar prejudicada pela falta de sons que poderiam indicar a completude de uma ação. No MIDP 2.0 foi incluída uma API específica para a utilização de elementos de multimídia;

- **linguagem interpretada**. O fato de J2ME ser uma plataforma interpretada, motivo pelo qual sua portabilidade é garantida, faz com que aplicativos escritos nesta linguagem não executem tão rapido quanto como aqueles escritos em linguagens que geram códigos nativos do aparelho.

- **inexistência de qualquer tipo de abstração para o desenvolvimento de jogos baseados em** *tiles*. A API do MIDP 1.0 não fornece nenhuma facilidade com relação ao desenvolvimento deste tipo de jogo. Entretanto, MIDP 2.0 definiu um pacote específico para este tipo de aplicação.

Capítulo 5

Conclusões

Este capítulo apresenta as conclusões finais do projeto com relação ao uso da tecnologia J2ME no desenvolvimento de aplicações *multiplayers* para celulares e algumas sugestões para trabalhos futuros. Para tal, o capítulo foi dividido em duas seções: a seção 5.1 apresenta as conclusões finais e a seção 5.2 as propostas para trabalhos futuros.

5.1 Conclusões

A utilização do *profile* MIDP 1.0, apesar dos problemas encontrados, pode ser considerada adequada para o desenvolvimento de jogos *multiplayers* para celulares.

Entretanto, apesar de ainda não ser implementado por nenhum dispositivo real, a utilização do *profile* MIDP versão 2.0 torna o o desenvolvimento de jogos no estilo 2D, baseados na idéia de *tiles*, muito simples e também deixa as aplicações mais eficientes. Estas melhoras ocorrem pois a API desta versão do MIDP possui classes altamente otimizadas que abstraem a maioria dos conceitos necessários ao desenvolvimento de jogos deste estilo.

O único problema real verificado nas duas versões do *profile* MIDP está relacionado com o fato de não ser obrigatória a implementação de outros protocolos de comunicação que não o HTTP. Este problema deve ser bem dimensionado ao se desenvolver qualquer tipo de aplicação em J2ME para celulares, pois não pode ser resolvido através de nenhuma técnica fornecida pela linguagem. Esta questão é totalmente dependente da arquitetura do aparelho alvo. A falta de suporte a conexões via *sockets* é uma de suas principais deficiências.

Conclui-se, então, que a utilização da plataforma J2ME ou, mais especificamente, do *profile* MIDP no desenvolvimento de jogos *multiplayers* baseados em *tiles* para celulares é satisfatória quando se estiver utilizando a versão 1.0 deste perfil. No entanto, ao se utilizar a versão 2.0, esta qualificação sofrerá uma melhora considerável, devido as grandes facilidades implementadas. A integração com as outras plataformas da tecnologia Java e a facilidade de construção de códigos nesta linguagem são características que acrescentam pontos positivos a plataforma J2ME.

A construção de jogos *multiplayers* que não usam *tiles* para construir a sua interface torna-se mais complexa nesta plataforma, pois exige a manipulação constante das primitivas gráficas e um controle maior sobre os métodos de desenho na tela. A API de jogos do MIDP 2.0 não apresenta muita utilidade nesta situação. No entanto, ainda é viável utilizar J2ME para este tipo de aplicação.

A Tabela 5.1 apresenta as considerações mais importantes obtidas com relação a utilização da plataforma J2ME no desenvolvimento de aplicações *multiplayers* para celulares.

5.2 Sugestões para trabalhos futuros

Através dos resultados obtidos por este projeto pode-se ainda desenvolver outros trabalhos. Algumas propostas de continuidade são:

- melhorar a especificação do jogo modelo, permitindo também uma melhora no quesito diversão proporcionada ao jogador;
- implementar a versão cliente do jogo modelo de uma maneira mais eficiente, utilizando ainda a versão 1.0 do *profile* MIDP;
- implementar a versão cliente do jogo modelo utilizando-se das melhorias oferecidas pela API do MIDP 2.0;
- definir métricas que permitam uma comparação mais detalhada das duas versões do *profile* MIDP no desenvolvimento de jogos *multiplayers*;
- realizar testes em aparelhos e emuladores de dispositivos reais. Estes testes permitiriam verificar a performance da aplicação em uma rede celular real.
- definir uma proposta de metodologia para o desenvolvimento de jogos *multiplayers* em celulares usando o *profile* MIDP. Esta metodologia forneceria

Tabela 5.1: Resumo das conclusões obtidas com relação a utilização da plataforma J2ME no desenvolvimento de jogos *multiplayers*.

Características	**Vantagens**	**Desvantagens**
Potencial para comercialização	Grande número de dispositivos com suporte a MIDP 1.0.	MIDP 2.0 ainda não disponível no mercado.
Manipulação de bytes	Fácil manipulação.	–
Fluxo de jogo	Controle centralizado no MIDP 2.0.	Threads separadas no MIDP 1.0.
Interface	Fácil manipulação.	Não suporta gráficos 3D.
Imagens	Suporte a transparência no MIDP 2.0.	Suporte único a imagens do tipo png.
Arquitetura de comunicação	Fácil integração com outras plataformas Java.	Suporta apenas o protocolo HTTP.
Linguagem Java	Simples, OO e portável.	Interpretada.
API	Bem estruturada e otimizada. API específica para jogos no MIDP 2.0.	–

técnicas para tratar melhor os problemas que possam ocorrer na implementação deste tipo de aplicação e maximizar os benefícios oferecidos pela plataforma.

Referências Bibliográficas

[POD2002] *Revista Poder.* URL: http://www.poderonline.com.br. Acessado em novembro de 2002.

[PAW2002] Pawlan, Monica. *Introduction to Wireless Technologies.* URL: http://wireless.java.sun.com/getstart/articles/intro/. 09 de outubro de 2002.

[INF2002A] InfoExame, Plantão Info. *32 milhões de celulares no país, diz Anatel.* URL: http://www.infoexame.com.br. 21 de novembro de 2002.

[INF2002B] InfoExame, Plantão Info. *Há mais linhas celulares do que fixas na AL.* URL: http://www.infoexame.com.br. 08 de janeiro de 2002.

[INF2001] InfoExame, Plantão Info. *Mais de 90% dos jovens finlandeses têm celular.* URL: http://www.infoexame.com.br. 17 de julho de 2001.

[IBI2002] *IBiznet 2002, Estatísticas.* URL: http://www.ibiznet.com.br. Acessado em 26 de novembro de 2002.

[FOX2002] Fox, David. *Will Mobile Games Sweep the Nation?* URL: http://www.onjava.com. 10 de dezembro de 2002

[NOK2002] *Site Oficial da Nokia.* URL: http://www.nokia.com.br. Acessado em 26 de novembro de 2002.

[DAT2002] *Data Monitor Market Analysis.* URL: http://datamonitor.com/. Acessado em 28 de novembro de 2002.

[TEL2002] *Telecom Web.* URL: http://www.telecomweb.com.br. Acessado em 26 de novembro de 2002.

[SUN2003A] Sun Microsystems. *Introduction to Wireless Java® Technology.* URL: http://wireless.java.sun.com/getstart/. 04 de fevereiro de 2003.

[SUN2003B] Sun Microsystems. *User's Guide Wireless Toolkit Version 2.o Beta 2 - Beta Draft. Java® 2 Platform, Micro Edition.* Janeiro de 2003.

[SUN2002A] Sun Microsystems. *A Brief History of the Green Project*. URL: http://java.sun.com/people/jag/green/. 28 de novembro de 2002.

[SUN2002B] Sun Microsystems. *User's Guide Wireless Toolkit Version 1.0.4.* Java® 2 Platform, Micro Edition. Junho de 2002.

[SUN2002C] Sun Microsystems. *Version 2.0 of Mobile Information Device Profile Specification.* URL: http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html. 2002.

[SUN2002D] Sun Microsystems. *Wireless Toolkit.* URL: http://java.sun.com/j2me/toolkit. Acessado em novembro de 2002.

[SUN2002E] Sun Microsystems *The Java® Tutorial* URL: http://java.sun.com. 2002.

[SUN2002F] Sun Microsystems *MIDP Style Guide, Mobile Information Device Profile (MIDP) 1.0a.* URL: http://java.sun.com Agosto de 2002.

[JON2002] Pawlan, Jonathan. *World of Wireless Communications.* URL: http://wireless.java.sun.com/getstart/articles/hardware/. 09 de outubro de 2002.

[BYO2002] Byous, Jon. *Java® Technology: an early history.* URL: http://java.sun.com/features/1998/05/birthday.html. 07 de junho de 2002.

[GOS & MCG1996] Gosling, James & McGilton, Henry. *The Java Language Environment. A White Paper*. URL: http://java.sun.com/docs/white/langenv/index.html. Maio de 1996.

[GOS, JOY, STE & BRA2000] Gosling, James & Joy, Bill & Steele, Guy & Bracha, Gilad. *The Java Language Specification.* Second Edition. 2000.

[CAM & WAL2002] Campione, Mary & Walrath, Kathy. *About the Java Technology*. URL: http://developer.java.sun.com/developer/onlineTraining/new2java/overview.html. Janeiro de 2002.

[LIN & YEL1999] Lindholm, Tim & Yellin, Frank. *The Java® Virtual Machine Specification.* Second Edition. 1999.

[ORT & GIG2003] Ortiz, C. Enrique & Giguère, Eric. *Java Comunnity Process.* URL: http://jcp.org. Acessado em 22 de março de 2003.

[ORT2002A] Ortiz, C. Enrique. *A Survey of J2ME Today.* URL: http://wireless.java.sun.com/getstart/articles/survey/. Novembro de 2002.

[ORT2002B] Ortiz, C. Enrique. *Introduction to OTA Application Provisioning.* URL: http://wireless.java.sun.com/midp/articles/ota/. November 2002.

[ORT & GIG2001] Ortiz, C. Enrique & Giguère, Eric. *Mobile Information Device Profile for Java 2 MicroEdition.* Wiley Computer Publishing. 2001.

[JSE2002] *Java 2 Standard Edition.* URL: http://java.sun.com/j2se/. Acessado em 02 de novembro de 2002.

[JEE2002] *Java 2 Enterprise Edition.* URL: http://java.sun.com/j2ee/. Acessado em 02 de novembro de 2002.

[JME2002] *Java 2 Micro Edition.* URL: http://java.sun.com/j2me/. Acessado em 02 de novembro de 2002.

[MUC2002] Muchow, John W. *Core J2ME Technology & MIDP.* Sun Press. 2002.

[WHI & HEM2002] White, james P. & Hemphill, David A. *Java 2 Micro Edition, Java in Small Things.* Manning Publications. 2002.

[GIG2002] Giguere, Eric. *J2ME[tm] Optional Packages.* URL: http://wireless.java.sun.com/midp/articles/optional/. Dezembro de 2002.

[CLC2003] *Connected Limited Device Configuration - CLDC.* URL: http://java.sun.com/products/cldc. Abril de 2003.

[CDC2003] *Connected Device Configuration - CDC*. URL: http://java.sun.com/products/cdc. Abril de 2003.

[MID2003] *Mobile Information Device Profile - MIDP*. URL: http://java.sun.com/products/midp. Abril de 2003.

[MAH2003] Mahmoud, Qusay H. *Future Java Technology for the Wireless Services Industry*. URL: http://wireless.java.sun.com/midp/articles/j2mefuture/. Fevereiro de 2003.

[KNU2003] Knudsen, Jonathan. *Creating 2D Action Games with the Game API*. URL: http://wireless.java.sun.com/midp/articles/game/. Março de 2003.

[KNU2002] Knudsen, Jonathan. *What's New in MIDP 2.0*. URL: http://wireless.java.sun.com/midp/articles/midp20/. Novembro de 2002.

[KNU & NOU2002] Knudsen, Jonathan & Nourie, Dana. *Wireless Development Tutorial Part I: Getting Started with MIDlet Development*. URL: http://wireless.java.sun.com/midp/articles/wtoolkit. 12 de fevereiro de 2002.

[ONE2003] *Sun One Studio - Sun One for Developers*. URL: http://sunonedev.sun.com/. Acessado em março de 2003.

[JBU2003] *JBuilder - The leading Java® development solution*. URL: http://www.borland.com/jbuilder/index.html. Acessado em março de 2003.

[JCR2003] *JCreator LE por Wendel de Witte*. URL: http://www.jcreator.com. Acessado em março de 2003.

[JED2003] *Jedit - Open Source programmer's text editor*. URL: http://www.jedit.org. Acessado em março de 2003.

[POS2003] *Palm OS Emulator*. URL: http://www.palmos.com/dev/tools/emulator. Acessado em março de 2003.

[DAY2001] Day, Bill. *Developing Wireless Applications using the Java® 2 Platform, Micro Edition. 2001*. URL: http://www.billday.com. 2001.

[MIC2003] Microsoft *Microsoft Age of Empires* URL: http://www.microsoft.com/catalog/display.asp?subid=22&site=10977&x=45&y=11 2003.

[MIP2003] Microprose *Civilization* URL: http://www.infogrames.com.br/jogos.asp?jogo_id=17 2003.

Resumo estendido

Assis, Wendel Malta. AVALIAÇÃO DA TECNOLOGIA J2ME NO CONTEXTO DE DESENVOLVIMENTO DE JOGOS MULTIPLAYERS PARA CELULARES. 121p. 2003. Este trabalho apresenta uma avaliação da utilização da plataforma *Java 2 Platform Micro Edition* (J2ME) da *Sun Microsystems*, que vem se destacando no mercado de construção de aplicações para dispositivos móveis, no desenvolvimento de jogos *multiplayers* para celulares.

Atualmente, a telefonia celular encontra-se em uma fase de grande expansão, momento em que estão surgindo novas tecnologias de transmissão e aparelhos cada vez mais modernos e com maiores funcionalidades. Estes novos aparelhos, por apresentarem um maior poder computacional, permitem o desenvolvimento de aplicações mais complexas, como os jogos.

Dentre o mercado de jogos para celulares e outros dispositivos móveis, os jogos *multiplayers* vem se destacando por permitirem disputas entre vários jogadores ao mesmo tempo e não apenas contra o computador, garantindo um maior nível de diversão durante as partidas. Assim, apresentam um maior grau de interação com os usuários e exigem um controle de concorrência avançado, representando um grande desafio de implementação.

O objetivo deste trabalho consiste em apresentar as dificuldades e facilidades encontradas ao se utilizar a plataforma J2ME no desenvolvimento de um jogo modelo *multiplayer* para celulares, chamado *"Alea Jacta Est´´*, e, em seguida, avaliá-la de um modo geral com relação ao desenvolvimento deste tipo de aplicação.

Para se desenvolver aplicações para celulares utilizando J2ME deve-se escolher entre uma das versões da especificação *Mobile Information Device Profile* (MIDP) definida pela plataforma. A conclusão final do projeto foi que a utilização do MIDP 1.0 cumpre satisfatoriamente com as necessidades de um desenvolvedor de jogos *multiplayers* para celulares. No entanto, a versão 2.0 do MIDP, ainda não implementada em aparelhos reais, proporciona melhores ferramentas para o desenvolvimento de tais aplicações.