Universidade Federal de Pernambuco

Centro de Informática

TRABALHO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TECNOLOGIA JAVA™ PARA SISTEMAS EMBARCADOS

Autor: Wellington João da Silva
Orientador: Sérgio Vanderlei Cavalcante

Recife, julho de 2001.

Resumo

Os sistemas embarcados estão cada vez mais presentes no dia a dia das pessoas, na forma de produtos fixos ou móveis de uso pessoal ou de consumo e que apresentam freqüentemente, entre outras características, alguma forma de comunicação rede e um elevado grau de sofisticação em relação às tarefas que executam (dispositivos inteligentes). A maioria das pessoas utiliza microprocessadores embutidos em algum dispositivo, como em telefones celulares e eletrodomésticos do que em computadores pessoais, por exemplo.

Esta é uma tendência que vai de encontro às idéias de Mark Weiser, em seu artigo "*The Computer for the Twenty-First Century*" ("O Computador para o Século Vinte e um") [WEI1991], no qual se previa que computadores pessoais desapareceriam do olhar das pessoas e passariam a fazer parte de todos os objetos, de forma integrada e onipresente (ubiquitous computing).

A "Ubiquitous Computing", termo cunhado pelo próprio Wiser, refere-se a ambientes com objetos operados por computador e conectados em redes sem fio. Ou seja: comunicação, mobilidade e poder de processamento integrados em vários objetos com finalidades distintas.

Escrever software para estes sistemas geralmente envolve a utilização de profissionais altamente especializados. Uma mistura de linguagens de alto e baixo nível (por exemplo, C e assembler) é freqüentemente utilizada, o que embora útil quando se deseja atingir requisitos de desempenho ou tamanho de código, e suscetível a erros, complicada, dificulta a portabilidade e manutenção, além de não ser adequada para aplicações críticas como, por exemplo, sistemas de segurança.

Uma outra característica importante dos sistemas embarcados, e em particular os de uso pessoal ou de consumo, é que muitos estão sendo projetados para apresentar funcionalidade variada, em contraste com a visão tradicional de tais sistemas (como se verá no decorrer deste trabalho, dispositivos compatíveis com Java podem ser designados para executar aplicações Java genéricas).

Java™ apresenta uma série de características que a tornam atrativa para o desenvolvimento de sistemas embarcados, tais como orientação a objetos, robustez, neutralidade de plataforma, suporte a rede e multithread, um amplo conjunto de biblioteca de classes, entre outras. Em contrapartida, questões de eficiência, não determinismo, tamanho de código e acesso ao hardware, entre outras, são colocadas como empecilhos para a utilização de Java em tais sistemas.

O objetivo deste trabalho consiste no estudo das abordagens utilizadas para tornar Java aplicável no mundo dos sistemas embarcados e algumas das principais tecnologias existentes para o desenvolvimento de sistemas embarcados utilizando Java.

O enfoque será dado nas tecnologias propostas pela Sun Microsystems™, criadora de Java. Em particular, será motivo de estudo mais detalhado a plataforma de programação *Java™ 2 Micro Edition (**J2ME**)* e a especificação de tempo real para Java (**RTSJ**) .

A J2ME foi escolhida para estudo mais detalhado por conseguir abranger, através de sua estrutura de configurações e perfis, uma ampla variedade de dispositivos embarcados.

Da mesma forma, a RTSJ foi escolhida por apresentar solução para alguns dos obstáculos mais sérios à utilização de Java em dispositivos embarcados, a saber, o gerenciamento de memória, escalonamento e sincronização, que têm forte impacto no universo de sistemas de tempo real, além de manipulação de eventos assíncronos e acesso à memória física.

O termo ***Tecnologia Java*** é utilizado para denominar desde a máquina virtual, a linguagem de programação e as bibliotecas de classe, até os ambientes de execução e aplicação, sendo a distinção realizada quando houver necessidade.

O termo ***desenvolvedor*** é utilizado para designar de forma genérica as pessoas envolvidas com a utilização da tecnologia Java para o desenvolvimento de sistemas embarcados.

Agradeço a todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho, quer através de críticas e sugestões, quer através do fornecimento de material de apoio ou indicação de fontes indispensáveis de informação.

Em particular, agradeço ao orientador, pela ajuda prestada diante das dúvidas relativas ao trabalho. Ao colega Jorge Cavalcanti, pelo auxilio dado no entendimento e desenvolvimento de aplicações utilizando **kjava**. A Pedro Henrique, pela indicação de ótimas fontes de consulta, além de dicas indispensáveis sobre a **J2ME**. A Gilmar Ferreira e Paulo Gustavo, pelo auxilio prestado na revisão deste trabalho.

Wellington João da Silva

Índice Analítico

Índice de Figuras

A maior parte das figuras presentes neste trabalho foram obtidas a partir do site da SUN Microsystems™.

1 Introdução

Sistemas embarcados estão cada vez mais presentes no dia a dia das pessoas, nos carros, eletrodomésticos, aparelhos telefônicos, nos caixas eletrônicos de bancos, etc. Tais sistemas apresentam restrições em termos de velocidade, tamanho, requisitos de memória e determinismo (capacidade de executar uma tarefa dentro de um certo período de tempo).

Java™ oferece algumas características que a tornam ideal para o desenvolvimento de sistemas embarcados, como portabilidade, reuso de código, confiabilidade, segurança, conectividade com a WEB (*World Wide Web*) e com outros sistemas embarcados. Além da melhor qualidade do código, devido às características de orientação a objetos da linguagem, projetos embarcados desenvolvidos em Java ganham em tempo e custo de desenvolvimento, em função da variedade de ferramentas de desenvolvimento e suporte existentes (§ 9.3 e § 9.3).

Por outro lado, as atuais implementações da especificação Java são muito lentas, não determinísticas e demasiadamente grandes para o desenvolvimento de sistemas embarcados, além de não proverem funcionalidades essenciais para estes sistemas, tais como controle direto do hardware (por exemplo, acesso à memória), execução a partir da ROM (memória apenas de leitura), escalabilidade e comportamento de tempo real.

Parece um contra-senso propor Java como tecnologia para o desenvolvimento de sistemas embarcados, pois enquanto um dos objetivos de Java é prover uma plataforma de desenvolvimento independente de hardware e sistema operacional subjacentes, sistemas embarcados usualmente requerem controle total dos mesmos. Contudo, o crescimento em quantidade e principalmente funcionalidade de dispositivos embarcados de uso pessoal ou de consumo fez surgir a necessidade de se desenvolver aplicações cada vez mais elaboradas e que pudessem ser facilmente portadas para uma família de dispositivos, atividade a que Java se presta muito bem.

A atual conjuntura fez surgir duas classes de desenvolvedores para sistemas embarcados: aqueles preocupados na funcionalidade da aplicação (**independência de dispositivo**) e aqueles preocupados em fazer com que a plataforma Java possa ser portada para as diversas classes de dispositivos embarcados (**amplo controle sobre o hardware**).

A Sun Microsystems™, criadora de Java, está investindo no mercado de dispositivos embarcados com algumas tecnologias que permitem aos fabricantes de dispositivos, fornecedores de serviços e programadores de aplicação desenvolver produtos embarcados e serviços várias espécies de uso de consumo.

A existência de uma máquina virtual que interpreta ***bytecodes*** é a base na qual Java se fundou para garantir o ideal "***write once, run anywhere***" (WORA), escreva uma vez, execute em qualquer lugar. No universo embarcado, todavia, não é de se esperar que qualquer dispositivo embarcado utilizando uma JVM (***J****ava* ***V****irtual* ***M****achine*) execute qualquer programa. Em tais dispositivos, é simplesmente impossível incluir todas as classes da definição completa de Java. É possível, todavia identificar classes de dispositivos para os quais um subconjunto específico de Java possa ser aplicado. Isto permitiria portabilidade dentro desta classe. Seguindo este principio, a Sun Microsystems™ tem definido várias plataformas específicas para sistemas embarcados, dentre as quais destacam-se:

- ***PersonalJava***™ - destinado a dispositivos de consumo conectados em rede, como os telefones com tela que permitem acesso à Internet (*internet screenphones*) (§ 6.1);
- ***EmbeddedJava***™ - para dispositivos embarcados de função dedicada (§ 6.2);
- ***Java Card***™- para uso em cartões inteligentes, anéis Java, e outros dispositivos com severas restrições de recursos (§ 6.3);
- ***Java 2 Micro Edition (J2ME***™ ***Platform, Micro Edition)*** - que abrange uma ampla área de dispositivos embarcados e de consumo, como telefones celulares, assistentes pessoais digitais(*PDAs)*, *pagers*, entre outros (§ 6.7);
- ***Java Phone***™ ***API –*** trata-se de uma extensão vertical da plataforma Personaljava™ consistindo de dois perfis de referência direcionados a telefones sem fio inteligentes e "internet screenphones" (§ 6.4).
- ***Java TV***™ ***API –*** adiciona interatividade aos televisores digitais (§ 6.5).

Sistemas embarcados são tipicamente sistemas de tempo real [GREHA1998]. A baixa performance de Java, por ser uma linguagem interpretada, e o gerenciamento automático de memória realizado pelo *garbage collector* (GC) padrão, devido à imprevisão quanto ao tempo que o mesmo levava para executar, impossibilitavam o uso de Java para sistemas de tempo real. Em decorrência, foram desenvolvidas várias técnicas de compilação, tais como a JIT (*just-in-time*) (§ 5.1.1) e a AOT (*ahead-of-time*) (§ 5.1.3), que otimizam o processo de interpretação de Java ou mesmo o eliminam, além de otimizações no gerenciamento automático de memória (§ 5.2).

Os maiores desafios de Java frente à programação em tempo real são o escalonamento, o gerenciamento de memória e a sincronização. A Sun Microsystems™ propôs uma **especificação de tempo real para Java**, a **RTSJ**, que tenta superar tais desafios. A RTSJ dentre outras características permite acesso à memória física, controle de escalonamento e existência de eventos assíncronos. Espera-se com esta especificação atender aos requisitos necessários para que Java seja uma plataforma efetiva para o desenvolvimento de sistemas embarcados.

O objetivo deste trabalho consiste no estudo das abordagens utilizadas para tornar Java aplicável no mundo dos sistemas embarcados e algumas das principais tecnologias

existentes para o desenvolvimento de sistemas embarcados utilizando Java. O enfoque será dado nas tecnologias propostas pela Sun Microsystems™, criadora de Java. Em particular, será motivo de estudo mais detalhado a plataforma de programação *Java™ 2 Micro Edition (**J2ME**)* e a especificação de tempo real para Java (**RTSJ**) .

1.1 Sumário dos Capítulos

- O capítulo 2 apresenta a definição e características básicas dos dispositivos embarcados.

- O capítulo 3 descreve as características básicas de Java.

- O capítulo 4 apresenta os nichos de mercado para os quais as tecnologias Java propostas pela Sun Microsystems™ são aplicáveis, e faz um paralelo entre as duas classes de desenvolvedores Java surgidas com o advento destas tecnologias.

- O capítulo 5 apresenta os vários esforços feitos no sentido de tornar Java adequada para o desenvolvimento de sistemas embarcados.

- O capítulo 6 descreve algumas das principais tecnologias Java para o desenvolvimento de sistemas embutidos, com ênfase dada a especificação de tempo real para Java (RTSJ) e a Java 2, edição Micro (J2ME™), uma plataforma Java especifica para dispositivos com restrição de recursos.

- O capítulo 7 apresenta o estudo de caso feito em cima da tecnologia J2ME.

- O capítulo 8 apresenta a conclusão do trabalho realizado, com uma análise crítica sobre o mesmo.

- O capítulo 9 corresponde ao apêndice, e apresenta uma coletânea de máquinas virtuais e ferramentas de desenvolvimento específicas para sistemas embarcados.

- No capítulo 10, as referências são descritas.

2 Sistemas Embarcados

Um sistema embarcado é um sistema de computação especializado, que faz parte de uma máquina ou sistema mais amplo. Geralmente, é um sistema microprocessado com os programas armazenados em ROM.

Alguns sistemas embarcados podem incluir um sistema operacional, mas muitos são tão especializados que tanto as tarefas a serem executadas quanto as funções especificas de um sistema operacional podem estar implementadas em um único programa.

Atualmente está cada vez mais difícil classificar um sistema como sendo ou não um sistema embarcado, em função dos recentes requisitos de funcionalidade que os mesmos vêm apresentando. Por exemplo, em [HEA1998] página 1, é colocado que um sistema embarcado é designado para um propósito especifico, não podendo ser programado pelo usuário da mesma forma que um computador pessoal e que o usuário pode fazer escolhas referentes à funcionalidade, mas não pode mudar esta funcionalidade pela adição ou substituição de software. Esta afirmação é valida quando aplicada a sistemas embarcados tradicionais onde pouca ou nenhuma alteração em termos de funcionalidade é feita no sistema depois de o mesmo ter sido concluído, mas contrasta fortemente com os novos dispositivos embarcados de uso pessoal ou de consumo cuja funcionalidade pode ser modificada dinamicamente. De acordo com a definição acima, um PALM não poderia ser considerado um dispositivo embarcado porque possibilita execução de aplicações diversas da mesma forma que um computador pessoal. De fato, tais dispositivos deveriam se chamados apenas de computadores de bolso.

Vários dispositivos utilizam sistemas embarcados: telefones celulares, assistentes pessoais digitais (PDAs), *pagers*, relógios digitais, carros, televisores, videocassetes, cartões inteligentes (*smart cards*), caixas eletrônicos, entre outros.

2.1 Requisitos de Projeto

Os sistemas embarcados, normalmente, apresentam altas restrições em termos de funcionalidade e implementação. Em particular, eles devem reagir a eventos externos em tempo real, adaptar-se a limites de tamanho e peso, gerenciar consumo de potência, satisfazer requisitos de confiabilidade e segurança e adequar-se a restrições de orçamento.

2.1.1 Resposta em Tempo Real

Um sistema é dito de Tempo Real quando ele responde a entradas e fornece saídas, rápido o suficiente para atender os requisitos do hardware ou do usuário [SCH1999]. Sistemas embarcados são essencialmente sistemas de tempo real.

2.1.2 *Resposta a eventos síncronos/assíncronos*

Sistemas embarcados necessitam responder a eventos internos ou externos. Estes eventos podem ser síncronos, caso em que uma política de escalonamento de eventos para garantir performance pode ser aplicável, ou assíncronos, caso em que uma estimativa do comportamento do mesmo deve ser feita para garantir uma performance mínima no pior caso.

2.1.3 *Tamanho e custo reduzidos*

Sistemas embarcados geralmente apresentam restrições como tamanho e peso ou custo unitário. Estas restrições são importantes para a definição da arquitetura de um sistema embarcado. O tamanho e/ou peso são restritivos quanto à escolha dos componentes físicos que podem fazer parte do sistema. Da mesma forma, as restrições de custo podem fazer com que sejam escolhidos para compor o sistema componentes de hardware/software que não são os mais modernos no mercado.

2.1.4 *Segurança e confiabilidade*

Alguns sistemas embarcados podem trazer risco aos usuários em caso de falha. É o caso, por exemplo, do controle automático de vôo em aeronaves ou sistema automático de freios em carros.

Tradicionalmente, utiliza-se alguma forma de redundância para tornar um sistema tolerante a falhas, seja ela de componentes de software ou hardware, informações ou tempo. E no caso dos sistemas embarcados, dadas as suas restrições não só em termos de custo e desempenho, mas também em relação a volume, peso e consumo de energia, a aplicação de técnicas de tolerância a falhas deve ser criteriosa.

[SAN2000] traz um estudo das várias técnicas de tolerância a falhas que podem ser adotadas para sistemas embarcados.

2.1.5 *Ambientes inóspitos*

Muitos sistemas embarcados operam em ambientes inóspitos ou mesmo inacessíveis, demandando a necessidade de proteção contra choques, vibrações, flutuações na fonte de energia, calor excessivo, fogo, água, corrosão, etc. Em tais ambientes a possibilidade de falha causada por um sinistro é elevada, o que demanda um bom mecanismo de tolerância a falhas.

2.2 Componentes Básicos

Alguns elementos de hardware/software estão sempre presentes no desenvolvimento de sistemas embarcados. O hardware básico de um dispositivo embutido deve conter um processador, memória e periféricos. Dado que sistemas embarcados são geralmente

microprocessados, uma prática comum entre desenvolvedores de sistemas embutidos é escolher um microprocessador específico e montar seu hardware em função deste. Obviamente esta escolha tem, entre outros determinantes, a disponibilidade do microprocessador no mercado e também o conjunto de bibliotecas de software existentes para o mesmo. A vantagem desta prática é que módulos de software podem ser reaproveitados entre projetos distintos.

2.2.1 PROCESSADOR

Sistemas embarcados são geralmente microprocessados. Embora a capacidade computacional dos microprocessadores tenha aumentado bastante (a capacidade de processamento dos *microchips* dobra aproximadamente a cada 18 meses), muitos sistemas embarcados continuam sendo desenvolvidos com o uso de processadores de baixa performance, de 8/16 bits. Isto se deve ao fato de haver outros fatores determinantes na escolha do processador além da performance, tais como custo, consumo de potência, ferramentas de software disponíveis, e disponibilidade do componente no mercado, entre outras.

2.2.2 MEMÓRIA

A memória é uma das partes mais importantes do projeto de um sistema embarcado e influencia diretamente na forma como o software para o sistema é projetado, escrito e desenvolvido. A memória tem duas funções básicas dentro de um sistema embarcado:

- Armazenar o software que o sistema ira rodar – feito geralmente em memória não volátil, que mantém seu conteúdo após a ausência de uma fonte de alimentação.
- Armazenar dados - tais como variáveis de programa e resultados intermediários, que são armazenados em memória volátil.

Muitos sistemas embarcados apresentam maior quantidade de memória não volátil em relação à volátil, dado o custo maior da memória volátil. Como resultado, o software para tais sistemas tem quer ser escrito de forma a minimizar os requisitos de memória volátil.

2.2.3 PERIFÉRICOS

Um sistema embarcado comunica-se com o mundo exterior através de periféricos. Dentre os principais periféricos encontrados em um sistema embarcado, destacam-se os *DISPLAYS*, saídas seriais, saídas binárias, temporizadores, etc.

2.2.4 SOFTWARE

O software de um sistema embarcado determina o que ele faz e como ele faz. Há várias formas de se desenvolver software para sistemas embarcados, dependendo da complexidade do mesmo e do tempo e dinheiro que pode ser gasto.

Um desenvolvedor de software para sistemas embarcados geralmente tem que se preocupar não só com a funcionalidade que este sistema deve apresentar, mas também com aspectos

de sincronização, escalonamento de tarefas, gerenciamento de memória, entre outros. Outra característica comum é a utilização de duas ou mais linguagens de programação, em função dos requisitos do sistema. Tais práticas aumentam o tempo de desenvolvimento do sistema, dificultam a portabilidade do código gerado além de serem suscetíveis a falhas tanto a nível de desenvolvimento quanto em tempo de execução.

3 Java™

Java é marca registrada da Sun Microsystems™ e define uma plataforma de programação incluindo uma linguagem de programação, uma máquina virtual, um formato de arquivo (arquivo class) e um conjunto de classes (a API de Java). Cada uma destas tecnologias é especificada pela Sun™.

Juntas, a máquina virtual de Java e a API de Java constituem o ambiente de execução de Java ou ainda, a plataforma Java.

3.1 A LINGUAGEM DE PROGRAMAÇÃO JAVA

Java é uma linguagem de programação de alto nível, de propósito geral, concorrente, desenvolvida pela Sun Microsystems™. Java foi originalmente chamada Oak, e projetada para dispositivos portáteis. O projeto da linguagem Java começou nos anos 90, como parte do chamado "Green Project", da Sun, cujo objetivo primordial era desenvolver software avançado para o mercado de dispositivos de consumo.Grande parte do trabalho inicial na linguagem foi realizada por James Gosling.

Em 1995 a Sun Microsystems™ mudou o nome de Oak para Java e modificou a linguagem para tomar vantagem da World Wide Web.

Dentre as características da nova linguagem, destacam-se:

- Suporte a rede – para comunicação entre dispositivos;
- Segurança – para prevenir a execução de código não autorizado;
- Confiabilidade – para evitar que os dispositivos falhem em tempo de execução;
- Tamanho reduzido – para caber em dispositivos com pouca memória;
- Independência de plataforma – para permitir a utilização em uma variedade de máquinas com diferentes CPUs e arquiteturas de sistemas operacionais;
- Multithreaded - para permitir que mais de uma atividade sejam executadas simultaneamente.

Quando o desenvolvimento da linguagem Java foi iniciado, a World Wide Web estava apenas começando no CERN. O uso inicial da linguagem Java necessitava de segurança e habilidade de executar código de *hosts* não confiáveis. Acontece que estas são virtualmente as mesmas necessidades dos usuários para a execução de programas na Web, daí a grande aceitação de Java na Internet.

Java é uma linguagem de programação orientada a objetos derivada de C++. Tem todas as vantagens de OO de C++, mas exclui algumas características como ponteiros e alocação de memória em nome da clareza, segurança e robustez.

Os arquivos com o código de fonte de Java (arquivos com extensão java) são compilados para um formato chamado ***bytecode*** (instrução da máquina virtual Java), que pode então ser executado por uma máquina virtual Java. O código compilado de Java pode funcionar na maioria dos computadores porque os interpretadores de Java e os ambientes de execução existem para a maioria de sistemas operacionais. O *bytecode* pode também ser convertido diretamente em instruções da linguagem de máquina por um compilador JIT ou AOT.

Java vem com uma coleção de bibliotecas que estende a linguagem. Há uma biblioteca de interface com o usuário chamada AWT, uma biblioteca de I/O, uma biblioteca de rede, etc.

Em Java, há dois tipos dos métodos: **Java e nativo**. Um **método Java** é escrito na linguagem Java, compilado para *bytecodes*, e armazenado em um arquivo de classe. Um **método nativo** é escrito em alguma outra língua, tal como C, em C++, ou *assembly*, e compilado para código nativo de máquina de um processador particular. Os métodos nativos são armazenados em uma biblioteca de vínculo dinâmico cuja forma exata é específica da plataforma. Métodos Java são independentes de plataforma; os métodos nativos não são. Quando um programa Java chama um método nativo, a JVM carrega a biblioteca de vínculo dinâmico que contém o método nativo e o invoca.

Java pode ser utilizado para se criar vários tipos de aplicativos, desde aplicações "standalone" até aplicações designadas para serem controladas pelo software que as executa, tais como APPLETSs que são carregados pela WEB e executados dentro de um *browser*, SERVLETS, que são designados para serem executados dentro de um servidor WEB, MIDLETs, designados para serem executados dentro de dispositivos móveis, XLETS, que são aplicações para receptores de TV digital ou *set top boxes* (dispositivo que estende a funcionalidade de um receptor de TV digital) que suportam a plataforma Java TV™, entre outros .

Abaixo são apresentados alguns aspectos positivos e negativos de Java, levando-se em conta os sistemas embarcados.

3.1.1 BENEFÍCIOS DE JAVA

- Todos os tipos primitivos de Java têm um tamanho fixo;
- Checagem automática de limites evita o programa de ler ou escrever fora dos limites de um *array*;
- Testes de condição devem retornar um valor booleano;
- Suporte para manipulação de *strings*;

3.1.2 FRAQUEZAS DE JAVA

- Sem acesso a registradores ou memória;
- Tamanho do código;
- Ineficiência;
- Não determinismo;
- Gerenciamento automático de memória (*garbage collection*).

3.2 A API DE JAVA

A API (Interface de programa aplicativo) de Java consiste em um conjunto de bibliotecas de tempo de execução que fornecem ao desenvolvedor de software uma forma padrão de acessar os recursos do sistema.

A especificação da API de Java, bem como a máquina virtual, devem ser implementados para cada dada plataforma, o que garante independência de plataforma para os programas que rodam sobre os mesmos. Para acessar recursos nativos do sistema, a API Java invoca método nativos.

Esta API contribui não apenas com a independência de plataforma, mas também com a segurança, pois os métodos da mesma verificam se possuem permissão para efetuar qualquer ação potencialmente prejudicial (como por exemplo, apagar arquivos).

A Sun Microsystems™ tem definido alguns subconjuntos da especificação Java, tais como Enterprise Java™, PersonalJava™, EmbeddedJava™, Java Card™, J2ME™, entre outras. Estas especificações objetivam definir os diversos elementos funcionais e bibliotecas de classes que mais se adequam a uma dada categoria de aplicações.

3.3 O FORMATO DE ARQUIVO CLASS

O formato de arquivo class é um formato binário, independente de hardware ou sistema operacional, e que representa o código compilado a ser executado pela máquina virtual Java. Tipicamente é armazenado em um arquivo conhecido como formato de arquivo class, embora não necessariamente precise ser armazenado desta forma.

O formato de arquivo class define precisamente a representação de uma classe ou interface, e contém as instruções da máquina virtual Java (bytecodes) e uma tabela de símbolos, bem como outras informações adicionais.

3.4 A MÁQUINA VIRTUAL JAVA

A máquina virtual Java (**JVM**) define um computador abstrato. Sua especificação define a funcionalidade que toda máquina virtual deve ter, mas dá liberdade quase total aos projetistas de cada implementação. Por exemplo, cada JVM pode usar qualquer técnica para

executar *bytecodes*. De fato, cada JVM pode ser implementada em hardware como em software, ou ambos. Esta flexibilidade na especificação da JVM objetivou permitir a implementação da mesma em uma larga variedade de computadores e sistemas operacionais [LIND1999].

Portabilidade e segurança são características fundamentais da plataforma Java obtidas pela existência de uma máquina virtual para a qual os programas Java são feitos.

3.4.1 CARGA E EXECUÇÃO DE CLASSES

A atividade principal de uma JVM é carregar classes e executar os bytecodes que elas contêm. O diagrama de blocos simplificado, com os subsistemas principais de uma JVM, é apresentado abaixo.

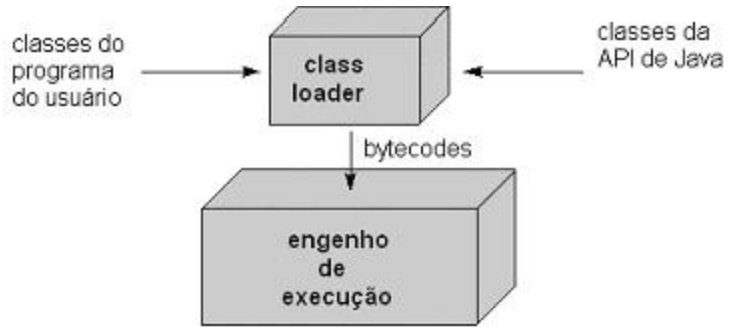

Figura 1- Diagrama de blocos de uma JVM

3.4.1.1 Class Loader

A JVM contém um "*class loader*", ou carregador de classes, que carrega as classes oriundas do programa e da API de Java. Apenas as classes da API que são atualmente necessárias pelo programa em execução são carregadas pela JVM.

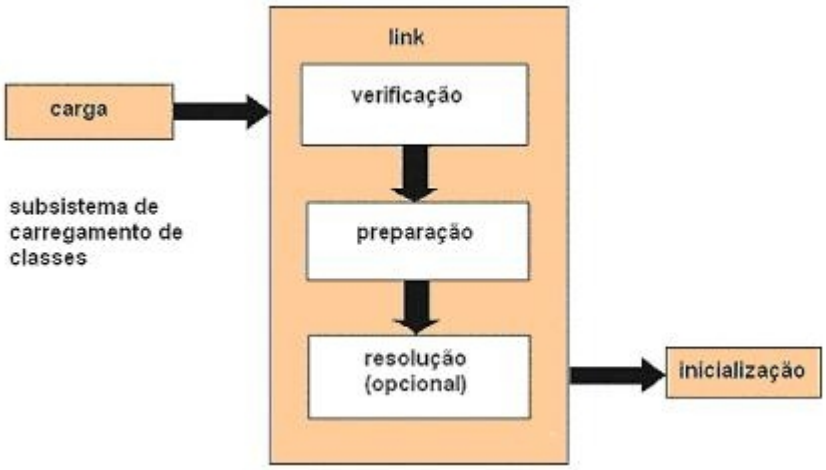

Figura 2 - Subsistema de um carregador de classes

As atividades do "*class loader*" são compostas das fases de **carga** e "***linkagem***", a qual divide-se em **verificação**, **preparação** e opcionalmente **resolução**. Isto é exemplificado no diagrama em blocos apresentado na figura acima.

Durante a **carga** de uma classe, o "*class loader*" é responsável por encontrar uma representação binária para a mesma. Caso não seja possível, um erro será gerado. A fase de **verificação** checa se a estrutura binária da classe carregada é bem formada. A **preparação** envolve a criação dos campos estáticos de classes ou interfaces e iniciação dos mesmos com seus valores padrões. Na fase de **resolução**, referências simbólicas são validadas e, tipicamente, substituídas com referências diretas.

3.4.1.2 Engenho de execução ("*execution engine*")

Os *bytecodes* são executados dentro de um engenho de execução, que pode variar em diferentes implementações. A forma mais simples de engenho de execução consiste em se interpretar os *bytecodes* um por vez. Outra opção é a compilação *just-in-time*, onde os bytecodes são convertidos para código nativo na primeira vez que o método é invocado; o código nativo será então posto numa *cache* e reutilizado na próxima vez que o método for chamado. Uma outra opção é a compilação adaptativa que converte para código nativo apenas as áreas de código que são mais utilizadas no programa. O quarto tipo de engenho de execução, conhecido como *ahead-of-time*, produz, a partir dos bytecodes, um programa executável para a plataforma alvo. Por fim, pode adotar como alternativa a implementação da JVM inteiramente em hardware.

A diversidade de tecnologias que podem ser adotadas para o engenho de execução de uma JVM faz com que o termo **"linguagem interpretada"**, que muitas vezes é utilizado em referência a linguagem de programação Java, perca um pouco seu significado.

4 Java e Sistemas Embarcados

A quantidade e diversidade de dispositivos pessoais e de consumo fizeram com que a Sun Microsystems™ entrasse no mercado de sistemas embarcados e desenvolvesse uma série de tecnologias que facilitassem o desenvolvimento de sistemas embarcados e serviços para uma ampla classe de dispositivos de consumo.

Java apresenta várias características que a tornam atrativa para o desenvolvimento de sistemas embarcados, tais como portabilidade, segurança, simplicidade, conectividade, desenvolvimento rápido de aplicações, entre outras. Por outro lado, características como ausência de ponteiros, não determinismo, ineficiência, latência do GC *e* impossibilidade de acesso direto ao hardware, são apresentadas como grandes empecilhos para a aplicação de Java no universo embarcado.

O modelo de caixa de areia de Java (*Java sandbox*) e a ausência de ponteiros a tornam mais segura e robusta, porém dificultam o controle do hardware. Coleta de lixo automática facilita a programação, mas impede que operações possam ser executadas de forma determinística. Além disto, Java tipicamente é lenta tem um grande requisito de memória.

Sistemas embarcados apresentam restrições tais como memória, velocidade e determinismo, que devem ser atendidas para que Java possa ser aplicada aos mesmos. Por conseqüência, esforços têm sido feitos no sentido de torná-la mais enxuta, rápida e previsível.

A diversidade de sistemas embarcados existentes torna inviável a existência de uma única plataforma Java que atenda aos seus variados requisitos. Para garantir o ideal "escreva uma vez, execute em qualquer lugar", a Sun Microsystems™ tem definido várias especificações para serem usadas no universo dos dispositivos embarcados, que ou são mais enxutas que a versão padrão de Java, como a PersonalJava™ e a EmbeddedJava™ , ou são especificas para uma categoria de dispositivos, como a Java Card™ , a Java TV™ e a JavaPhone™.

Em 1999, a Sun Microsystems™ lançou uma versão da sua plataforma para sistemas embarcados, e que abrange várias classes de dispositivos embarcados. Esta plataforma é a Java 2 Micro Edition™ (J2ME, desenvolvida para atender o melhor possível as necessidades especificas de cada tipo de dispositivo embutido. Também em 1999, o RTJEG (*Real-time for Java Experts Group*), de acordo com o JCP (*Java Community Process)*, começou a desenvolver a especificação de tempo real para Java (RTSJ) [BOL2000].

Até a data de entrega deste Trabalho, não havia sido liberada uma implementação de referência. Estas duas tecnologias (RTSJ e J2ME) são as peças principais no sentido de adaptar Java para o desenvolvimento de aplicações embarcadas.

4.1 Segmentos de Mercado

Convém salientar que as tecnologias adotadas pela Sun Microsytems™ não são aplicáveis a para qualquer tipo de sistema embarcado. Em particular, elas se encaixam bem em dispositivos embarcados de uso pessoal ou de consumo, tais como telefones celulares, assistentes pessoais digitais, *pagers*, etc. Especificamente, a Sun tem proposto várias tecnologias que atendem às necessidades dos seguintes segmentos ou perfis de mercado:

- **Wireless** – abrange os dispositivos sem fio, tais como telefones celulares, pagers, PDAs, leitores de cartões inteligentes (*smart card*), terminais de ponto de venda (POS), entre outros. Tecnologias relacionadas:
  - Móveis
    - J2ME (CLDC, perfil MIDP, perfil PDA)
    - Java Card™
  - Estacionários
    - J2ME(CDC, perfil Personal, perfil Foundation)

- **Home Gateway** – Espera-se que as casas modernas, num futuro próximo, possuam um *home gateway* que servirá a duas finalidades preliminares: como um *hub* para conectar e controlar os dispositivos inteligentes na casa e como um *gateway* das comunicações entre a casa e o mundo exterior. Como exemplos de tecnologia desenvolvida para *home gateway* pode-se citar:
  - Java Embedded Server™
  - Jini™

- **Automotivo** – Objetiva a integração de serviços de entretenimento, acesso a Internet, e sistemas de navegação nos automóveis. Hoje em dia, muitos automóveis já utilizam sistemas de localização por satélites. Tecnologias relacionadas:
  - Java Card™
  - Java Car™
  - Java TV API™

- **TV Digital Interativa** – TV Digital Interativa adiciona interatividade ao modelo padrão de televisão digital acoplada a um "*set top box*". Com este novo modelo, além de poder utilizar serviços convencionais como o "*pay-per-view"* , vai ser possível, por exemplo, solicitar mais informações sobre um filme em exibição, ou escolher um final alternativo para o mesmo, customizar a programação de uma emissora em função do perfil do espectador, responder a pesquisas através do próprio aparelho de TV ou efetuar compras *online*. Tecnologias relacionadas:
  - Java TV API™
  - Java Embedded Server™

Para complementar a informação contida nesta seção, recomenda-se a leitura de [SEG0001], especificamente a parte referente a tendências de mercado.

4.2 Transparência de hardware versus Controle do hardware

Um desenvolvedor de software para sistemas embarcados tradicional necessita ter amplo controle do hardware sobre o qual as aplicações irão executar. Isto é verdade porque, no geral, tal desenvolvedor não está focado exclusivamente na implementação da funcionalidade do sistema, mas em outros aspectos como escalonamento de atividades, controle de interrupções, compartilhamento de dados, tamanho de código, eficiência, etc., os quais exigem domínio do hardware.

As tecnologias Java para sistemas embarcados, por serem desenvolvidas para categorias ou classes de dispositivos específicas, já abstraem o hardware subjacente, o que significa dizer que o desenvolvedor do software não precisa ter conhecimento de hardware.

Com Java, o desenvolvedor de software pode preocupar-se exclusivamente com as tarefas que a aplicação irá executar. Isto tem aspectos positivos e negativos. Como aspecto positivo cita-se, além da óbvia portabilidade do código, dado que não se faz suposição sobre o hardware sobre o qual a aplicação Java irá executar, o ganho em tempo e custo de desenvolvimento, pois o desenvolvedor, além de não necessitar ser um especialista no hardware para o qual se está desenvolvendo a aplicação, pode dedicar-se exclusivamente a implementação da funcionalidade que a aplicação deve apresentar. Em contrapartida, utilizar a tecnologia Java em outras categorias de dispositivos que não as definidas pela Sun Microsystems™, torna-se bastante difícil, exigindo um especialista não só em hardware, mas também na plataforma Java que se está tentando portar.

5 Adaptando Java para Sistemas Embarcados

Durante execução, há dois aspectos de implementação da JVM que têm maior impacto na aplicabilidade da tecnologia Java para sistemas embarcados: **os métodos usados para executar os bytecodes** e o **gerenciamento dos recursos do sistema** – principalmente memória.

Três importantes atributos que têm um efeito significativo na aplicação final serão determinados pela forma como serão implementados a execução do bytecode e o gerenciamento de memória. O primeiro atributo, que é mais afetado pelos métodos utilizados para execução do bytecode, é **performance.** O segundo é o **requisito de memória**, e o terceiro, **determinismo.**

Para aumentar a performance, várias alternativas à tradicional forma de execução do bytecode foram apresentadas. As principais são apresentadas em (§ 5.1).

O gerenciamento automático de memória facilita a vida do programador porque evita erros comuns de programação, mas causa problemas de determinismo e eficiência, quando aplicado ao universo de sistemas embarcados. Em (§ 5.2) são apresentadas algumas tecnologias utilizadas para melhorar o desempenho do GC.

Por fim, em (§ 5.3) é apresentada a especificação de tempo real para Java, que visa superar as limitações que a impedem de ser eficientemente utilizada em sistemas de tempo real, como o são, essencialmente, a maioria dos sistemas embarcados. Uma característica desta especificação é que ela vai além do seu propósito básico, no sentido de que fornece ao desenvolvedor de sistema embutido algumas características fundamentais e com as quais ele está acostumado a lidar, como o acesso à memória física do sistema e manipulação de eventos assíncronos.

5.1 Técnicas de compilação

Uma máquina virtual Java (JVM) tradicional converte o código fonte presente nos arquivos java em uma seqüência de *bytecodes*, que são armazenados em um arquivo class, os quais poderão ser depois **interpretados** em qualquer máquina para a qual haja uma JVM. As técnicas de compilação citadas a seguir apresentam alternativas para este modelo interpretado de compilação [CT0001].

5.1.1 Compilador just-in-time (jit compiler)

Um compilador *JIT* é uma parte opcional de um interpretador Java que compila os bytecodes para código nativo de máquina, imediatamente antes de executá-los.

Compiladores JIT são just-in-time no sentido de que eles compilam os métodos das classes em código nativo imediatamente antes destes serem chamados. Se um método for chamado mais de uma vez, o JIT irá re-executar o código nativo.

JIT aumenta os requisitos de tamanho e memória de uma JVM tradicional. O compilador aumenta os requisitos estáticos da JVM. Além disto, JIT utiliza memória para compilar os métodos, os quais geralmente requerem mais RAM que o *bytecode* original.

Em sistemas com uma quantidade de recursos razoável (por exemplo, computadores pessoais), estes requisitos adicionais de tamanho e memória são mais que justificados pelo resultante aumento de velocidade; em sistemas embarcados, todavia, constituem-se num grande problema.

JIT nem sempre torna o código mais rápido que o código interpretado. Se o código da aplicação apresenta vários laços de execução, então a técnica de re-utilizar o código previamente compilado será vantajosa; todavia, caso o programa execute em uma única etapa, sem laços ou repetições de trechos de código, o custo da operação do JIT pode ser maior que simplesmente interpretar o código. Se a aplicação criar novos objetos constantemente, a operação do JIT irá interferir com a operação do GC.

Outro aspecto importante para sistemas embarcados é que o tempo de *start-up* da aplicação também pode ser maior com a utilização do JIT.

Uma descrição um pouco mais detalhada de JIT e da sua arquitetura pode se encontrada em [JIT0001].

5.1.2 Compilação adaptativa dinâmica

Compilação adaptativa dinâmica é similar a um JIT no sentido que ele compila bytecode em tempo de execução. Ele difere do JIT no que compila, como e quando decide compilar.

Durante a execução interpretada dos bytecodes, a JVM monitora a aplicação e determina onde estão os "gargalos" de execução. Ela então chama o compilador dinâmico adaptativo para compilar os segmentos de bytecode que são executados com mais freqüência. As instruções nativas resultantes são armazenadas na memória para acesso rápido. A JVM continua neste processo até que todos os *buffers* de código tenham sido utilizados.

Por ser adaptativo, caso ele encontre um novo segmento de código que está executando com mais freqüência que algum código previamente compilado e armazenado, ele compilará este novo bytecode e o armazenará em um buffer contendo código menos freqüentemente utilizado. O usuário pode configurar a quantidade de memória do sistema utilizada para a compilação, bem como o tamanho e o numero de *buffers* de código utilizados para armazenar as instruções nativas compiladas. Também pode ser dado às aplicações controle explicito sobre o *thread* do compilador dinâmico adaptativo durante

execução para maior determinismo, fornecendo um comportamento mais apropriado para sistemas embarcados [WORK1999].

5.1.3 COMPILADOR ESTÁTICO ("AHEAD-OF-TIME" COMPILER)

Um compilador estático é, em linhas gerais, um compilador cruzado que gera, a partir do código fonte (ou *bytecode*), um programa executável para a plataforma alvo.

Um compilador estático compila classes fora do dispositivo alvo. Estas classes são armazenadas em um formato que pode ser ligado a outras classes (*linkable file format*). Quando a JVM roda no dispositivo alvo, quaisquer classes pré-compiladas são carregadas. Devido à compilação não se dar no dispositivo alvo, mais tempo e memória podem ser gastos em otimização para gerar o código nativo. Diferentemente da compilação JIT um dispositivo usando compilação estática não requer espaço ou memória adicional para executar.

O programa Java resultante de compilação estática pode executar praticamente à mesma velocidade que programas escritos em outras linguagens como, por exemplo, C++. Além disto, a utilização de compilação estática elimina a necessidade de uma JVM no dispositivo alvo, o que diminui em muito os requisitos de memória do mesmo.

Compilação estática é ideal para dispositivos com restrição de recursos, especialmente nas situações onde várias classes Java são conhecidamente "*ahead of time*".

5.2 Garbage collection

A especificação da máquina virtual da Sun™ define os requerimentos funcionais da JVM, mas é aberta quanto à sua forma de implementação. Em decorrência, existem várias técnicas de gerenciamento de memória que podem ser utilizadas em implementações da JVM, algumas sendo mais aplicáveis a sistemas embarcados que outras. Para uma introdução ao tema, recomenda-se a leitura de [GC0001] e [JONES1996].

5.2.1 MODOS DE EXECUÇÃO

Os modos básicos de execução do GC são: em **batch**, **incremental**, **concorrente**.

Um GC que roda **em batch** não pode ser interrompido, ou seja, o programa principal é suspenso enquanto este é executado, o que pode tornar a sua aplicação inapropriada para a maioria dos dispositivos embarcados por causa do seu não determinismo.

Um GC **incremental** intercala as suas atividades com as do programa em execução. Esta intercalação geralmente é ordenada, ou seja, o coletor e o programa do usuário não acessam ou modificam dados de forma simultânea. Este tipo de implementação pode levar a um

maior grau de determinismo do que um GC operando em batch, contudo, se não for preemptível, um GC incremental pode bloquear a aplicação.

Em um GC **concorrente**, o coletor e programa do usuário podem modificar ou acessar dados de forma simultânea. A intercalação das atividades é desordenada e preempção pode ocorrer a qualquer hora.

Um GC também pode ser de **tempo real**, ou seja, o custo associado com qualquer operação do mesmo é garantido estar restrito a um pequeno limite de tempo.

5.2.2 TÉCNICAS BÁSICAS DE GARBAGE COLLECTION

Há duas abordagens básicas para separar lixo de objetos vivos (que possuam referências para si): **contagem de referência** e **"tracing"**.

O termo "***tracing*"** é usado para designar as técnicas de **cópia** e ***marcação ("mark")***. Objetos situados em posições de memória que são direta ou indiretamente apontados por um conjunto inicial de ponteiros chamado conjunto raiz (ex: variáveis globais), são considerados vivos e, portanto não passíveis de serem coletados pelo GC. Algoritmos de "**tracing**" procuram identificar tais objetos. Há duas formas básicas de algoritmos de "**tracing": "cópia"** e "***mark and sweep***".

5.2.2.1 Contagem de Referências

Contagem de referência envolve a contagem das referências que cada objeto tem para si, contagem esta que é utilizada para determinar se o objeto pode ou não ser reciclado.
Em um sistema de contagem de referência, cada objeto possui um contador de referências (ponteiros) associado ao mesmo, que é incrementado toda vez que uma referência ao objeto é feita, e decrementado quando uma referência ao objeto existente é eliminada. A memória ocupada pelo objeto pode ser reciclada quando o seu contador for igual a zero, que indicaria que o programa em execução não pode mais atingi-lo.

A vantagem da contagem de referência é a natureza incremental da maior parte de suas operações (a atualização dos ponteiros pode ser intercalada com a operação da aplicação). Ela pode ser feita totalmente incremental e em tempo-real.

Há alguns problemas com a contagem de referência em GCs: Primeiro, os contadores utilizados ocupam espaço, o que constitui-se em problema no caso de sistema com restrição de memória como é o caso dos sistemas embarcados em geral. Segundo, a contagem de referência falha ao tentar reciclar estruturas cíclicas, pois caso os ponteiros em um grupo de objetos formem um ciclo direcionado, os contadores de referências dos mesmos nunca chegarão a zero mesmo que não haja caminho entre o conjunto raiz e os objetos. O terceiro

problema da contagem de referência é a eficiência, pois seu custo é geralmente proporcional às atividades realizadas pela aplicação em execução.

5.2.2.2 Mark and sweep

M*ark and sweep* tem duas fases: marcar os objetos acessíveis a partir do conjunto raiz e examinar todo o ***heap*** reclamando objetos não marcados.

Há alguns problemas com GCs que utilizam o ***"mark-and-sweep"*** tradicional. Primeiro, é difícil lidar com objetos de diferentes tamanhos sem causar fragmentação de memória. Como os objetos não são movidos, à medida que a memória vai sendo ocupada por objetos e estes são reciclados, a memória torna-se bastante fragmentada, dificultando a alocação de espaço contínuo para objetos de maior tamanho. Um GC que não manipula a fragmentação da memória depende da expansão do *heap* para acomodar a alocação de espaço para um objeto que seja demasiado grande para todos os segmentos disponíveis de memória. Uma vez que a memória fragmentou bastante e o heap alcançou o tamanho máximo, erros do tipo "*out of memory*" ocorrerão. Nos sistemas embarcados onde a memória é limitada, ignorar a fragmentação é inaceitável.

Segundo, o custo da coleta é proporcional ao tamanho do *heap*, pois todo objeto vivo deve ser marcado, e todo lixo deve ser encontrado no *heap* coletado. O terceiro problema envolve localidade de referência. Dado que objetos nunca são movidos, objetos vivos permanecem na memória intercalados com espaço livre.

5.2.2.3 Cópia

A técnica de **cópia** move todos os objetos para uma área, e o restante do *heap* é considerado como disponível, pois contém apenas lixo.

O algoritmo de **cópia** usa um **heap** dividido em metades chamadas semi-espaços. Um semi-espaço, chamado o espaço atual, é usado para a alocação de memória. Quando a coleta de lixo ocorre, objetos ativos do espaço corrente são rastreados copiados para o segundo espaço. Ao final deste processo, o espaço atua conterá apenas lixo, estando os objetos ativos no outro espaço.

Este algoritmo tem as vantagens de assegurar a alocação rápida de novos objetos, reduzir a fragmentação, compactar os dados de uma aplicação, e ser mais eficiente (porque o custo da coleta de lixo é proporcional somente ao tamanho dos dados alcançáveis a partir do conjunto raiz, não sendo relacionado ao tamanho da área onde os objetos estão situados). Por outro lado, ele divide a memória disponível à aplicação o que se constitui em um enorme empecilho quanto a sua aplicação em sistemas embarcados que usualmente apresentam restrições de memória.

5.2.3 Coleta de lixo Precisa versus Conservativa

Durante o processo de rastreio, os coletores de lixo podem usar a estratégia **precisa** ou a **conservativa** para identificar referências para os objetos. O GC conservativo não pode determinar com certeza quais variáveis e campos são ponteiros e quais são tipos primitivos (ex: inteiros), então ele pode tratar valores incertos como ponteiros. Esta abordagem pode causar escapes de memória ("***memory leak***"), caso algo que não seja ponteiro contenha um valor que aponta para um objeto inativo na memória, pois tanto o objeto quanto os objetos que ele referencia não serão coletados. Como os escapes de memória são dependentes dos dados, aplicações que executavam normalmente podem repentinamente vir a ter problemas de falta de memória.

Em ambientes com alta restrição de memória, um GC que atua de forma **precisa** é a alternativa adequada. Tais GCs são capazes de determinar inequivocamente se um dado valor é ou não um ponteiro, de forma que ele sempre coleta todos os objetos não utilizados.

Também os sistemas críticos necessitam de uma forma precisa de coleta de lixo, pois não é admissível que os mesmos deixem de funcionar de uma hora para outra, em função dos dados que ele manipule, o que pode trazer riscos não só ao sistema quanto às pessoas que eventualmente interajam com ele.

5.2.4 Não utilização do garbage collector

Uma estratégia que pode ser utilizada em sistemas embarcados, principalmente naqueles que necessitam ter comportamento deteminístico, de tempo real, ou em sistemas críticos, consiste em não se utilizar nenhuma técnica de reciclagem de memória, e sim alocar uma área de memória fixa ou estática durante *start-up*, onde todos os objetos necessários serão previamente criados. Esta abordagem é usual em aplicações cujos requisitos de memória apresentados em tempo de execução são previamente conhecidos.

Alocação prévia da memória necessária garante comportamento deteminístico porque a aplicação não necessita ficar esperando que a memória seja reciclada. A utilização do GC não impede que aplicações cujos requisitos dinâmicos de memória em tempo de execução venham a apresentar erros do tipo ***Out Of Memory,*** o que é inaceitável em aplicações criticas como, por exemplo, no controle de instrumentação cirúrgica.

A Especificação de Tempo Real para Java (RTSJ) (§ 5.3) corrobora esta estratégia, através da definição de áreas de memória nas quais o GC é impossibilitado de reciclar objetos.

5.2.5 Reuso de objetos

Uma técnica de escrita de código que pode ser utilizada para minimizar os efeitos negativos da ação do GC consiste em se reutilizar os objetos alocados. Ao invés de se criar uma nova instância de uma classe toda vez que um objeto for necessário, pode-se reaproveitar um

objeto já alocado e modificar seus atributos para refletir o estado desejado do novo objeto. Esta técnica é particularmente útil em *arrays* ou objetos que contêm *arrays*.

5.3 Especificação de tempo real para Java (RTSJ)

Uma grande parte dos sistemas embarcados apresenta requerimentos de performance de **tempo real**. Tal requerimento não implica necessariamente em execução imediata, e sim execução de forma previsível, ou seja, saber com antecedência quando certo trecho de código irá começar e quando irá terminar.

Na especificação Java tradicional, o **escalonamento** e a **sincronização** não são suficientemente precisos para programação em tempo real. Em Java regular, o **gerenciamento automático** de memória tipicamente causa atrasos não determinísticos nas tarefas.

Com o objetivo de desenvolver uma plataforma que fornecesse ao programador controle sobre o comportamento temporal do software em execução, o RTJEG (*Real Time Java Expert Group*) começou a desenvolver, em março de 1999, uma especificação de tempo real para Java, a RTSJ (*The Real Time Specification for Java*), cuja primeira liberação ocorreu em maio de 2000. Até a data de entrega deste trabalho não havia sido liberada uma implementação de referência da RTSJ.

Convém salientar que o RTJEG não é o único grupo trabalhando no desenvolvimento de tecnologias Java relacionadas com tempo real. O ***J Consortium*** tem sua própria especificação de tempo real, que difere em alguns pontos da defendida pelo RTJEG, embora ambos os trabalhos sejam baseados em um *workshop* realizado em 1998-1999 e patrocinado pelo **NIST (*National Institute of Technologiy*).** Maiores informações sobre o *J Consortium* podem ser obtidas a partir de [JCON1999].

Optou-se por apresentar neste trabalho a especificação definida pelo RTJEG simplesmente por ser este o grupo de trabalho apoiado pela Sun Microsystems™ e neste trabalho está sendo dada ênfase às soluções propostas pela mesma. Contudo, o estudo comparativo do trabalho destes dois grupos é colocado como uma proposta de trabalho futuro.

A especificação de Tempo Real para Java estende a plataforma Java™ para suportar a prática atual de programação e a programação avançada de aplicações para sistemas de tempo real.

O RTJEG identificou três características principais na especificação da máquina virtual e da linguagem para o não determinismo de Java, o **escalonamento**, o **gerenciamento de memória** e a **sincronização**. Além destas características, foram identificadas outras quatro, que visam atender as necessidades dos desenvolvedores de sistemas de tempo real:

manipulação assíncrona de eventos, **transferência assíncrona de controle**, **término assíncrono de *threads***, e **acesso à memória física**.

Estas sete áreas modificadas, segundos os autores da RTSJ, fornecem uma plataforma de desenvolvimento de software de tempo real que atende aos requisitos de uma ampla faixa de aplicações [BOL2000].

Class Hierarchy

- Class java.lang.Object
  - class javax.realtime.**AsyncEvent**
    - class javax.realtime.**Timer**
      - class javax.realtime.**OneShotTimer**
      - class javax.realtime.**PeriodicTimer**
  - class javax.realtime.**AsyncEventHandler** (implements javax.realtime.Schedulable)
    - class javax.realtime.**BoundAsyncEventHandler**
  - class javax.realtime.**Clock**
  - class javax.realtime.**GarbageCollector**
  - class javax.realtime.**HiResTime**
    - class javax.realtime.**AbsoluteTime**
    - class javax.realtime.**RelativeTime**
  - class javax.realtime.**MemoryArea**
    - class javax.realtime.**ImmortalMemory**
    - class javax.realtime.**ImmortalPhysicalMemory**
    - class javax.realtime.**ScopedMemory**
      - class javax.realtime.**CTMemory**
      - class javax.realtime.**ScopedPhysicalMemory**
      - class javax.realtime.**VTMemory**
  - class javax.realtime.**MemoryParameters**
  - class javax.realtime.**MonitorControl**
    - class javax.realtime.**PriorityCeilingEmulation**
    - class javax.realtime.**PriorityInheritance**
  - class javax.realtime.**PhysicalMemoryFactory**
  - class javax.realtime.**PosixSignalHandler**
  - class javax.realtime.**ProcessingParameters**
  - class javax.realtime.**RawMemoryAccess**
  - class javax.realtime.**RealtimeParameters**
  - class javax.realtime.**RealtimeSecurity**
  - class javax.realtime.**RealtimeSystem**
  - class javax.realtime.**ReleaseParameters**
    - class javax.realtime.**AperiodicParameters**
      - class javax.realtime.**SporadicParameters**
    - class javax.realtime.**PeriodicParameters**
  - class javax.realtime.**Scheduler**

- class javax.realtime.**SchedulingParameters**
- class java.lang.Thread (implements java.lang.Runnable)
  - class javax.realtime.**RealtimeThread**
    - class javax.realtime.**NoHeapRealtimeThread**
- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Error
    - class javax.realtime.**MemoryAccessError**
    - class javax.realtime.**ThrowBoundaryError**
  - class java.lang.Exception
    - class java.lang.InterruptedException
      - class javax.realtime.**AsynchronouslyInterruptedException**
        - class javax.realtime.**Timed**
    - class javax.realtime.**MemoryScopeException**
    - class javax.realtime.**NoSuchElementException**
    - class javax.realtime.**OffsetOutOfBoundsException**
    - class javax.realtime.**ResourceLimitException**
      - class javax.realtime.**AdmissionControlException**
    - class java.lang.RuntimeException
      - class javax.realtime.**ThrowBoundaryException**
    - class javax.realtime.**SizeOutOfBoundsException**
- class javax.realtime.**WaitFreeDequeue**
- class javax.realtime.**WaitFreeReadQueue**
- class javax.realtime.**WaitFreeWriteQueue**

Interface Hierarchy

- interface javax.realtime.**Interruptible**
- interface java.lang.Runnable
  - interface javax.realtime.**Schedulable**

Hierarquia de classes da RTSJ

5.3.1 ESCALONAMENTO

Um dos objetivos da programação em tempo real é garantir a previsibilidade da execução de uma seqüência de instruções, ou seja, garantir pela análise do programa, se uma seqüência de instruções irá ou não terminar dentro de um certo limite de tempo. Em um programa de tempo real, não só a corretude é importante, mas também o tempo que o mesmo irá levar para realizar uma seqüência de operações. Por exemplo, em um sistema automático de freio para automóveis e essencial que ele realize a sua função básica: garantir uma frenagem segura para o motorista. Contudo se ele vai fazer isto após alguns milésimos de segundos ou após alguns segundos faz uma enorme diferença da qual dependerá a própria integridade física de quem estiver dentro do veículo.

A RTSJ dá aos implementadores flexibilidade para instalar algoritmos de escalonamento arbitrários e algoritmos de análise de viabilidade em uma implementação da especificação.

Ela fornece três classes, ***Scheduler***, ***SchedulingParameters*** e ***ReleaseParameters***, e subclasses destas que encapsulam requisitos de tempo.

O algoritmo de escalonamento padrão é preemptivo, de prioridade fixa, com pelo menos 28 níveis únicos de prioridade.

5.3.2 Gerenciamento de Memória

Uma das grandes críticas dos desenvolvedores de sistemas de tempo real quanto à adoção de Java para o desenvolvimento de sistemas de tempo real refere-se ao não determinismo introduzido pela atuação do *GC* no *heap.* A RTSJ supera esta dificuldade através da definição de varias extensões ao modelo de memória, que suportam gerenciamento de memória de uma maneira que não interfere com a habilidade do código de tempo real em fornecer comportamento deteminístico. Este objetivo é conseguido pela possibilidade de se alocar objetos fora do *heap* de memória, local onde os objetos são tradicionalmente alocados em uma implementação de Java padrão.

5.3.2.1 Criação de Threads

A RTSJ define duas classes básicas para a criação de threads: ***RealTimeThread*** (RT) e **NoHeap*RealTimeThread*** (NHRT), que é uma subclasse de RT.

RTs podem acessar objetos no *heap* de memória, estando portanto sujeitos a atrasos decorrentes da atuação do GC. Para evitar tais atrasos, pode-se utilizar NHRTs, que não podem acessar objetos no heap, o que implica que eles podem estar em execução concorrentemente com o GC.

NHRTs são aplicáveis para código com muito pouca tolerância a atrasos. RTs possuem uma tolerância maior a tais atrasos. Para código que não apresenta restrições de tempo, pode-se utilizar threads regulares de Java.

5.3.2.2 Áreas de Memória

Uma área de memória representa uma porção da memória que pode ser utilizada para a alocação de objetos. Algumas áreas de memória existem foram do *heap* e impõem restrições sobre o que o sistema ou o GC pode fazer com objetos alocados dentro das mesmas. Embora o GC seja capaz de analisar estas áreas de memória através de referências de objetos presentes no *heap*, objetos contidos em algumas das áreas de memória especificadas não são passíveis de reciclagem.

A RTSJ usa a classe abstrata ***MemoryArea***, para representar áreas de memória. Esta classe tem três subclasses: **memória física (*Physical Memory*)**, **memória imortal (*immortal memory*) e memória com escopo (*scoped memory*)**.

Memória com escopo constitui-se em um mecanismo para lidar com objetos que tem um tempo de vida limitado. Memória física permite que objetos sejam criados dentro de regiões especificas da memória física e que têm características particulares, como por exemplo, memória com acesso mais rápido ou memória mapeada em algum dispositivo de IO (*I/O mapped memory*). Memória imortal representa uma área de memória contendo objetos que uma vez alocados, existem ate o fim da aplicação.

5.3.2.2.1Memória Imortal

Objetos alocados em uma memória imortal existem até o final da aplicação, não sendo passíveis de *garbage collection*, embora alguns algoritmos de reciclagem de memória possam varrer a memória imortal para o seu correto funcionamento. Um objeto imortal só pode conter referências para outros objetos imortais ou para objetos no *heap*. Memória imortal é um recurso de memória compartilhado por todos os threads de uma aplicação.

5.3.2.2.2Memória Física

A RTSJ define duas classes para que os programadores possam acessar a memória física diretamente a partir de código Java.

RawMemoryAccess define métodos que possibilitam a construção de um objeto que representa uma faixa de endereços físicos e então acessar a memória física com granularidade de byte, word, long, float, ou seqüência de bytes. O conteúdo da memória física pode então ser acessado por um conjunto de métodos get() e set().

Um objeto *RawMemoryAccess* não pode conter objetos Java ou referências para objetos Java, pois isto pode ser inseguro(porque poderia ser utilizado para vencer a verificação de tipos de Java) e propenso a erros (porque é sensível às escolhas de implementação específicas feitas pelo compilador Java).

A segunda classe, ***PhysicalMemory,*** permite a construção de objetos que representam uma faixa de endereços de memória física onde o sistema pode localizar objetos Java. Para construir um novo objeto Java em um objeto *PhysicalMemory* específico, pode-se utilizar os métodos enter() ou new Array().

5.3.2.2.3Memória com escopo

Memória com escopo é utilizada para limitar o tempo de vida de qualquer objeto alocado dentro da mesma. Quando o sistema entra em um escopo sintático, todo uso do operador "*new*" faz o sistema alocar memória da área de memória com escopo ativa. Cada área de memória com escopo normalmente mantém um contador do numero de referências externas para si. Quando um escopo termina ou o sistema o deixa, o sistema normalmente decrementa o contador de referências da memória para zero, destrói quaisquer objetos alocados dentro da mesma e chama seus finalizadores.

Escopos podem ser aninhados. Quando se entra em um escopo aninhado, todas as alocações seqüentes são feitas em cima da memória associada com o novo escopo.

Memória com escopo é implementada na classe abstrata ***ScopedMemory.*** Duas subclasses concretas são disponíveis para instanciação: **LTMemory** (Linear time) e **VTMemory** (variable time), onde o termo *time* (tempo) refere-se ao custo para se alocar um novo objeto.

5.3.3 SINCRONIZAÇÃO

RTSJ utiliza o termo "prioridade" de uma forma suave que na literatura de tempo real convencional. Por exemplo, quando se afirma que um *thread* é o que possui maior prioridade, simplesmente se está afirmando tratar-se do *thread* mais elegível, aquele que deveria ser escolhido pelo escalonador dentre todos os *threads* prontos para execução.

5.3.3.1 Filas de Espera (*wait queues*)

O sistema deve enfileirar todos os *threads* esperando para ter acesso a algum recurso em ordem de elegibilidade. Se for possível a existência de *threads* com a mesma prioridade, estes devem ser enfileirados utilizando-se a política FIFO (*first in, first out)*.

5.3.3.2 Evitando a inversão de prioridade

A implementação da primitiva ***synchronized*** deve possuir um comportamento padrão que previna a inversão de prioridade entre *threads* que compartilham um recurso. O protocolo de herança de prioridade deve ser implementado por padrão. Este é um protocolo bem conhecido na literatura de tempo real, e funciona da seguinte forma:
Se *thread* t1 tenta adquirir um *lock* que é mantido por um thread t2 de mais baixa prioridade, a prioridade de t2 é feita igual a de t1 enquanto t2 mantiver o *lock* (e recursivamente se t2 esta esperando para adquirir um *lock* mantido por um *thread* de mais baixa prioridade ainda).

Uma segunda política, "*priority ceiling emulation protocol*" (ou *highest locker protocol*) também é especificada para sistemas que o suportam.

5.3.4 MANIPULAÇÃO ASSÍNCRONA DE EVENTO

As duas classes principais do mecanismo de eventos assíncrono proposto pela RTSJ são a ***AsynEvent*** e a ***AsynEventHandler***.

Um objeto ***AsynEvent*** representa algo que pode ocorrer, como, por exemplo, uma interrupção de hardware, um evento computado ou um sinal POSIX.Um ***AsynEventHandler*** é um objeto escalonável , similar a um *thread.* Quando ocorre um evento, o sistema invoca os métodos run() dos *handlers* associados.

Diferente de outros objetos do tipo "*Runnable*", um ***AsynEventHandler*** tem associados parâmetros de memória(***MemoryParameters***), escalonamento(***SchedulingParameters***), e liberação(***ReleaseParameters***) que controlam a execução do *handler* uma vez que este foi acionado. Quando o evento é acionado, o sistema executa os *handlers* de forma assíncrona, escalonando-os de acordo com os objetos associados ***ReleaseParameters e SchedulingParameters***. O resultado é que o *handler* aparenta ter sido atribuído ao seu próprio *thread.*

Uma forma especializada de ***AsynEvent*** é a classe ***Timer***, que representa um evento cuja ocorrência é controlada pelo tempo. Há duas formas de Timers: ***OneShotTimer*** e ***PeriodicTimer***. Instâncias de ***OneShotTimer*** são acionadas uma vez, no tempo especificado. Se o tempo corrente for maior que o especificado, o *handler* é acionado imediatamente. Instâncias de ***PeriodicTimer*** são acionadas uma vez, no tempo especificado, e então periodicamente de acordo com um intervalo previamente especificado.

A RTSJ representa relógios usando a classe Clock. As implementações da RTSJ podem oferecer mais de um relógio. Um objeto Clock especial, Clock.getRealTimeClock(), representa o relógio de tempo real e deve estar presente em todas as implementações.

5.3.5 TRANSFERÊNCIA ASSÍNCRONA DE CONTROLE

Transferência de controle assíncrona é uma das características presentes em sistemas de tempo real. Em decorrência, a RTSJ inclui um mecanismo que estende a manipulação de exceções tradicional de Java para permitir que o ponto corrente de execução de um programa possa ser transferido de forma eficiente e imediata para outra localização.

Convém salientar que a RTSJ restringe a transferência de controle assíncrona para métodos escritos com o entendimento que seu foco de controle pode mudar de forma assíncrona. Isto é feito especificando que tais métodos levantam a exceção *AsyncronoulyInterruptedException* (AIE), na sua cláusula *throws*. Quando tal método estiver executando no topo da pilha de execução de threads e o sistema chamar java.lang.Thread.interrupt() naquele *thread*, o método irá imediatamente atuar como se o sistema tivesse levantado uma AIE. Se o sistema chama uma interrupção em um thread que não está executando tal método, o sistema irá colocar a AIE em um estado pendente para o thread e irá levantá-la na próxima vez que o controle passar por tal método, ou chamando-o ou retornando do mesmo. O sistema também coloca a AIE em estado pendente enquanto o controle está em blocos sincronizados, entra, ou retorna dos mesmos.

5.3.6 TÉRMINO ASSÍNCRONO DE THREADS

As primeiras versões da linguagem Java forneciam mecanismos para realizar esta tarefa, como os métodos stop() e destroy() na classe Thread. Entretanto, dado que stop() poderia

deixar objetos compartilhados em um estado inconsistente, stop() foi depreciada (“deprecated”). O uso de destroy() pode levar a deadlock(caso o thread seja destruído enquanto mantém um lock) e, embora não tenha ainda sido depreciado, seu uso é desencorajado. O objetivo da RTSJ foi atender aos requerimentos de término assíncrono de threads sem introduzir os perigos dos métodos stop() e destroy().

Para implementar término assíncrono de threads, o sistema pode utilizar AIE diretamente ou em conjunto com eventos assíncronos.

6 Tecnologias Java para Sistemas Embarcados

Neste capítulo são descritas as principais tecnologias Java disponíveis para utilização em sistemas embarcados. Elas são basicamente compostas de ambientes de aplicação e extensões da plataforma Java. Antes de descrever as tecnologias, alguns conceitos básicos devem ser explicados.

- **API –** as tecnologias listadas como APIs correspondem a extensões da plataforma forma, geralmente desenvolvidas através de um processo aberto pela Sun MicroSystems™ e líderes de mercado naquela tecnologia.

- **Ambiente de execução Java –** corresponde ao ambiente de software necessário a execução de código Java em um sistema embarcado. Geralmente é composto de:
  - **Maquina virtual Java –**traduzir os *bytecodes* em código nativo de máquina.
  - **Biblioteca de classes –** em formato de *bytecode*.
  - **Métodos nativos –** corresponde a funções que são escritas em alguma outra linguagem, pré-compiladas e juntadas com a JVM. Geralmente implementam funções que são dependentes de plataforma.
  - **Sistema operacional multitarefa –** que fornece a implementação dos mecanismos de sincronização e threads.
  - **Garbage collector –** para a reciclagem da memória utilizada.

- **Ambientes de aplicação Java –** Para garantir o ideal WORA (escreva uma vez, rode em qualquer lugar), a Sun definiu varias plataformas de execução padrão que atendem aos requisitos de certas classes ou categorias de aplicativos. A estas plataformas deu-se o nome de ambientes de aplicação.

Em seqüência segue a descrição das principais tecnologias Java pesquisadas.

6.1 PersonalJava™

O ambiente da aplicação **PersonalJava™** é uma plataforma Java para construir aplicações em rede para eletrodomésticos, equipamentos de escritório e de uso móvel, tais como celulares, PDAs, pagers, CATV e outros sistemas com uma capacidade razoavelmente alta de processamento e memória. Ela é composta da máquina virtual de Java e de uma versão otimizada da biblioteca de classes do Java Padrão.

Este ambiente é designado para dispositivos conectados via WEB que estão freqüentemente executando applets da rede. Para garantir esta funcionalidade de forma genérica, é necessária a implementação completa do seu conjunto fixo de bibliotecas de classes.

Para atingir os requerimentos de tais dispositivos, a plataforma **PersonalJava™** reimplementa o conjunto de completo de APIs definido pelo ambiente de aplicação Java, a fim de caber em dispositivos menores e com maior limitação de memória. A plataforma **PersonalJava** também oferece um kit de ferramentas gráfico, o Truffle™ toolkit, criado especialmente para dispositivos portáteis que requerem conectividade com a WEB.

A tecnologia **PersonalJava™** Será embutida na arquitetura J2ME(Java 2 PlataForm, Micro Edition) como a soma de uma configuração (CDC) e de um perfil , o perfil *Personal*.

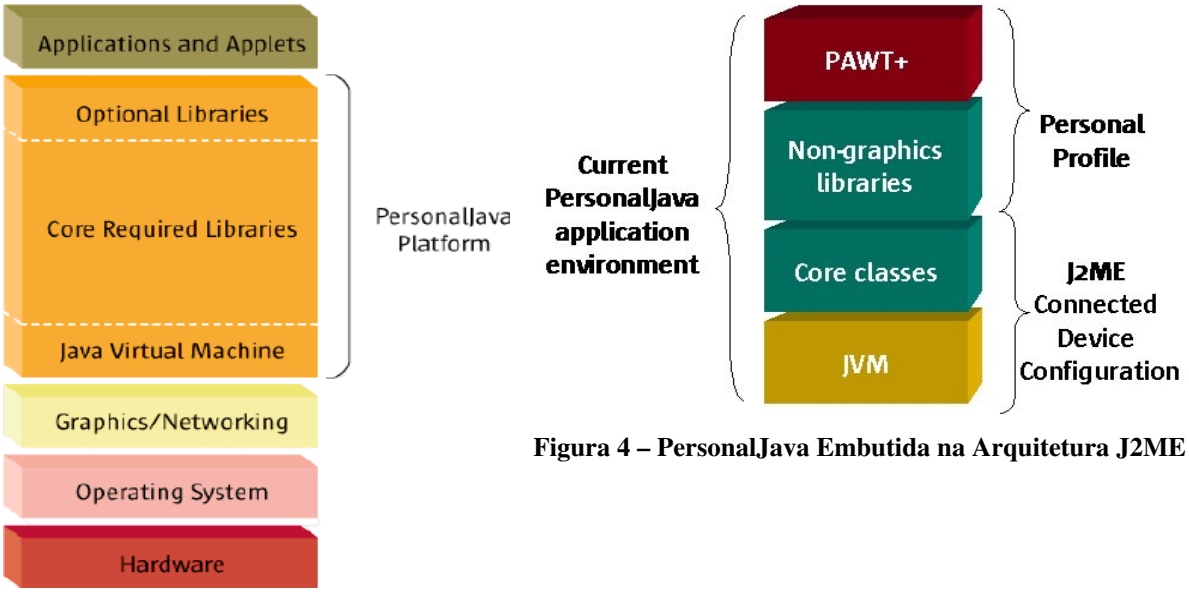

Figura 4 – PersonalJava Embutida na Arquitetura J2ME

Figura 3 - Arquitetura PersonalJava

6.1.1 Ferramentas de desenvolvimento Para PersonalJava™

As informações contidas nesta seção são referentes às ferramentas de desenvolvimento que auxiliam no desenvolvimento de aplicações e dispositivos que aderem às especificações PersonalJava™ e EmbeddedJava™, inclusive.

Na execução tradicional de Java, todo o código é verificado antes de ser executado para garantir que o código a ser executado não viole nenhuma regra de segurança de Java. Este processo consome tempo e memória, o que é um problema em sistemas embarcados. Em decorrência foram desenvolvidas algumas ferramentas que pré-compilam as classes Java e possibilitam a sua execução diretamente da ROM.

O *JavaCodeCompact™* recebe classes Java como entrada e as converte para arquivos fonte da linguagem C que são independentes de plataforma. Estes arquivos podem então ser compilados para um formato de ROM.

O *JavaDataCompact*™ é similar ao *JavaCodeCompact*, mas possibilita a inclusão de dados como imagens ou sons dentro da ROM do dispositivo alvo.
JavaFilter™ é uma ferramenta utilizada na Embeddedava, apenas, e que constrói uma lista de campos e métodos utilizados pela plataforma Java para executar a aplicação.

Para produzir a imagem final em ROM todos os arquivos são ligados ("*linked*") juntos utilizando-se ferramentas de desenvolvimento disponíveis do sistema operacional de tempo real sobre o qual PersonalJava™ (ou EmbeddedJava™) esta executando.

Uma descrição completa destas ferramentas pode ser encontrada em [TOO0001];

6.1.2 Otimizações da PersonalJava™

Um dos objetivos de projeto primários do ambiente de aplicação PersonalJava™ foi minimizar os requisitos estáticos de memória (principalmente memória ROM) e o uso de memória em tempo de execução (RAM). Para tanto, foram feitas algumas otimizações na implementação da máquina virtual Java e do JavaCodeCompact, as quais são resumidamente listadas abaixo:

- Requisito de memória
  - Estática
    - Compartilhamento do corpo de strings
    - Eliminação de duplicação, movendo estruturas de dados para segmentos apenas de leitura
    - Compartilhamento/fusão de elementos de tabelas
  - Dinâmica
    - Remoção de informações de debug
    - Remoção de inicializadores estáticos largos
    - Redução de estruturas de dados
- Uso de pilha nativa
  - Remoção de recursão e introdução de verificações dinâmicas
  - Medida do consumo de pilha da aplicação atual
- Pilha Java
- Uso do *heap*
- Tamanho do código C

Em [PER1998] encontra-se uma descrição mais detalhada das otimizações citadas acima.

6.2 EmbeddedJava™

O ambiente de aplicação EmbeddedJava™ é para dispositivos com funcionalidade dedicada e grande limitação de memória.

Este ambiente é direcionado para dispositivos embutidos de função dedicada que podem ser conectados em rede ou dispositivos autônomos. Estes dispositivos de função dedicada irão realizar um conjunto definido de atividades de software que são bem conhecidas durante a concepção do produto.

Apenas as classes que são necessárias para suportar este conjunto específico de tarefas de software são incluídas no dispositivo. Por causa disto, a especificação **EmbeddedJava™** pode ser implementada de forma diferente de um dispositivo embarcado para outro, já que a implementação completa da plataforma não é esperada.

A vantagem disto é que a implementação EmbeddedJava™ freqüentemente tem um menor requisito de memória do que o ambiente de aplicação PersonalJava™. Como contrapartida, perde-se a portabilidade do código.

Algumas informações adicionais sobre a EmbeddedJava tais como sua arquitetura e processo de desenvolvimento podem ser encontradas [BECK0001] , onde é feita uma comparação deste ambiente e da PERC-VM (§ 9.3.6).

6.3 Java Card™

Java Card™ é designado especialmente para cartões inteligentes (smart cards), que são dispositivos com fortes restrições de processamento e memória.

Um cartão inteligente é idêntico em tamanho a um cartão de credito comum. Este cartão armazena e processa informações através dos circuitos eletrônicos embutidos em silicone no substrato de plástico do cartão. Há dois tipos de cartões inteligentes: de memória e inteligente. Um cartão de memória armazena dados localmente, mas não contém uma CPU para realizar operações sobre os dados. Um cartão inteligente inclui um microprocessador e pode realizar operações sobre os dados armazenados localmente.

A API para a tecnologia Java Card define as convenções de chamada pelas quais um applet acessa o ambiente de tempo real do cartão Java (Java Card Runtime Environment) e serviços nativos. A API Java Card permite que aplicativos escritos para uma plataforma Java Card rodem em qualquer outra plataforma que suporte Java Card.

A API Java Card é compatível com padrões internacionais tais como, ISO7816, e padrões específicos de indústrias, tais como, Europay/Master Card/Visa (EMV).

A tecnologia Java Card preserva muito dos benefícios da linguagem de programação Java – produtividade, segurança, robustez, ferramentas e produtividade – enquanto possibilitando o uso em cartões inteligentes. A máquina virtual, a definição da linguagem, e os pacotes centrais foram feitos mais compactos e sucintos para trazer a tecnologia Java para o ambiente restrito dos cartões inteligentes.

Em [HOW2001] são descritos os passos necessários para se desenvolver uma aplicação utilizando a tecnologia JavaCard™. Informações específicas sobre a plataforma podem ser encontradas em [CARD2001]

6.4 JavaPhone™ API

A API JavaPhone™ é uma extensão vertical da plataforma Personaljava™ consistindo de dois perfis de referência direcionados a telefones sem fio inteligentes e "internet screenphones".

Esta API é designada para fornecer acesso as seguintes funcionalidades:

- Controle direto do telefone
- Mensagens de datagrama
- Informações de calendário e agenda eletrônica
- Acesso ao perfil do usuário
- Monitoramento de potencia

Em [PHONE2001] pode-se informações mais detalhadas sobre esta tecnologia.

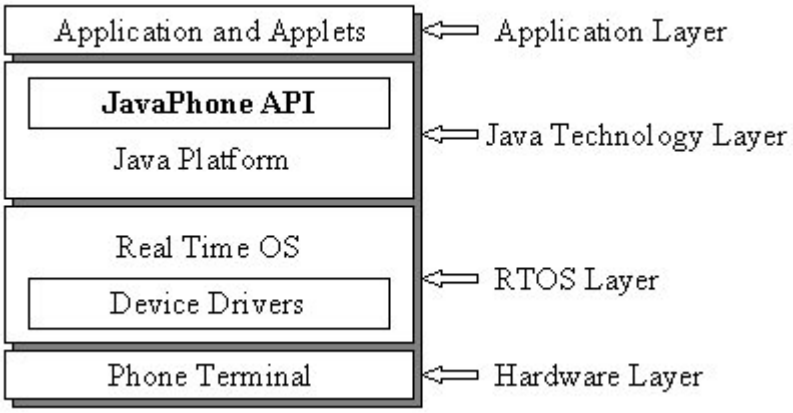

Figura 5 - pilha de Software tipica em um aparelho telefônico usando JavaPhone

6.5 Java TV™ API

A API Java TV é uma extensão da plataforma Java é designada para permitir que aplicações Java possam acessar as funcionalidades do servidor de televisão no qual eles são executados. A plataforma Java, junto com a API Java TV™ fornecem uma plataforma ideal para o desenvolvimento e distribuição de serviços interativos.

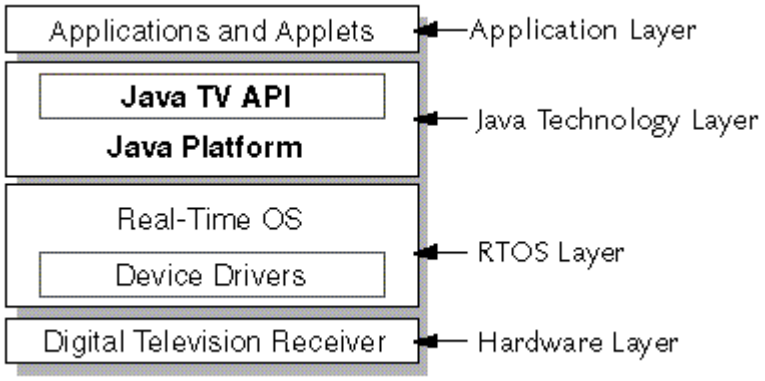

Figura 6 - JavaTV API

Através da API Java TV, que irá prover acesso ao conteúdo de programação de televisão, seleção de conteúdo (guia de programa), e controle sobre a aparência da tela da televisão. As aplicações podem rodar em uma **JVM** designada para "set top boxes", aparelhos de digitais ou dispositivos de tempo real. Os detalhes do hardware subjacente são abstraídos, deixando os desenvolvedores livres para se concentrar no desenvolvimento do conteúdo interativo.

A partir de [TV2001], pode-se obter informações mais detalhadas sobre esta tecnologia.

6.6 Java Embedded Server™

Java EmbeddedServer™ (JES) é um servidor de aplicações de com baixo requisito de recursos, e designado especialmente para suportar aplicações embutidas sob demanda. Ele é composto de dois componentes principais, o ***ServiceSpace***, que roda dentro de uma plataforma Java de execução (EmbeddedJava or PersonalJava) em um dispositivo que suporte alguma forma de acesso em rede. O ***JavaServer engine,*** gerencia a carga e execução das aplicações correspondentes aos serviço os. Os serviços incluem servidores http, agentes SNTP, serviços de escalonamento e gerenciamento de Servlets.

JES pode ser instalado em modems ou set top boxes, por exemplo, para transformá-los em gateways residenciais. Um gateway residencial é uma caixa na residência na qual os aparelhos domésticos se conectam e que se conecta com a rede externa (Internet). Isto pode dar aos moradores acesso a serviços tais como telefone sob demanda, segurança residencial, controle de energia, entre outros.

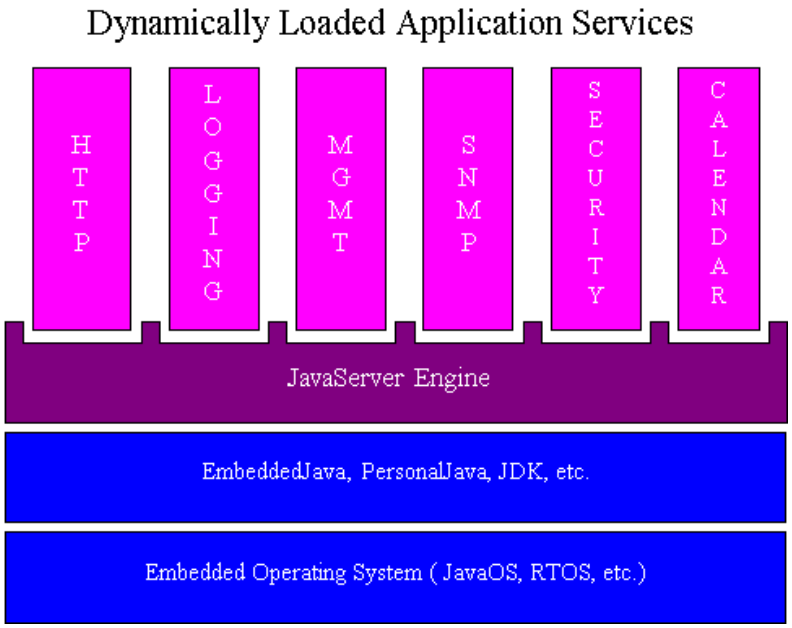

Figura 7 - Java Embedded Server

Para obter informações mais detalhadas sobre a tecnologia EmbeddedServer™ recomenda-se a leitura de [JES2001]

6.7 J2ME (Java 2 Micro Edition)

Java 2 Platform Micro Edition™ (J2ME) é a edição da plataforma Java que atende as necessidades de espaço de eletrodomésticos e dispositivos embarcados de uso pessoal e de consumo. Ela abrange uma ampla área de dispositivos, tais como telefones celulares, *pagers,* set top boxes, PDAs, Palms, entre outros.

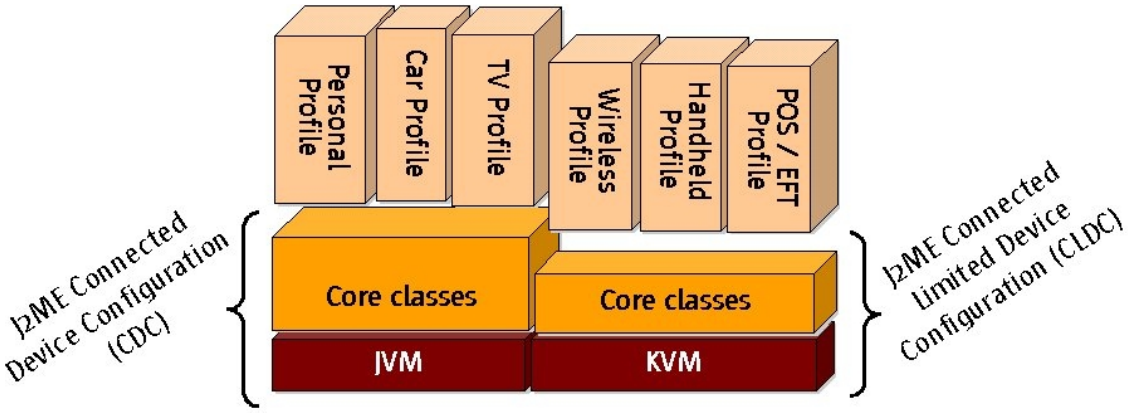

Figura 8 - arquitetura J2ME

A arquitetura J2ME é composta de **configurações** e **perfis**.

6.7.1 CONFIGURAÇÃO

Dispositivos embarcados variam em forma, funcionalidade e características básicas. Para melhor atender a estes diferentes requisitos, a plataforma J2ME suporta uma configuração mínima da máquina virtual, e fornece APIs que atendem as necessidades de cada espécie de dispositivo. Ao nível de implementação, uma configuração J2ME determina um conjunto horizontal de APIs para uma família de produtos que têm requisitos similares. Uma configuração define:

- As características suportadas pela linguagem de programação Java
- As características da suportadas pela máquina virtual
- Bibliotecas de classes e APIs suportadas

Existe uma variedade enorme de dispositivos embarcados e de consumo. Tais dispositivos podem ser aproximadamente divididos em duas categorias: dispositivos que são tipicamente fixos e dispositivos móveis, que tendem a ter mais restrições de processamento, alimentação (fornecimento de energia) e memória que os primeiros, por não poderem contar com uma conexão direta com fios ou alguma forma de rede. Dadas estas variações em termos de requisitos que apresentam os dispositivos embarcados, a Sun Microsystems™ definiu duas configurações, a *connected device configuration* (**CDC**), para dispositivos geralmente fixos e com uma quantidade razoável de processamento ou memória, e a *connected device limited configuration* **(CLDC)**, para dispositivos móveis que apresentam grande restrição de recursos.

6.7.1.1 J2ME CDC (connected device configuration)

A configuração CDC é baseada na especificação da máquina tradicional clássica, que define um ambiente de execução com toda a funcionalidade encontrada nos ambientes de execução Java para computadores pessoais. Esta configuração é designada para dispositivos fixos um pouco maiores, com pelo menos alguns megabytes de memória disponível e alguma forma de acesso à Internet ou a outros dispositivos como set top boxes, telefones que dão acesso a WEB, e sistemas de navegação. Tipicamente, estes dispositivos funcionam em microprocessadores de 32 bits e têm mais que 2.0MB da memória total para o armazenamento da máquina virtual e bibliotecas de classes.

Os dispositivos que se enquadram nesta configuração apresentam como características:

- Conectividade com alguma espécie de rede;
- Requerem um mínimo de 512Kb de memória ROM e 256Kb de memória RAM;
- Suporte para uma implementação completa da especificação da máquina virtual Java, segunda edição;
- Interface com usuário com variado grau de sofisticação.

O CDC contém a máquina virtual CVM, que é uma máquina virtual completa e projetada para os dispositivos que necessitam de toda a funcionalidade presente na edição 2 de Java, apresentando porem bem menos requisitos de memória.

6.7.1.2 J2ME CLDC (connected limited device configuration)

A configuração CLDC consiste de uma máquina virtual, a KVM, e um conjunto de bibliotecas de classes para serem utilizados dentro de um perfil definido pela industria, tal como o MIDP (mobile *information device profile*) e o PDA (*personal digital assistant*).

A CLDC foi projetada pela Sun para ser uma configuração padrão, portável, com requisitos mínimos, para ser utilizada em dispositivos móveis, pequenos e com grande restrição de recursos, tais como *pagers*, telefones celulares, assistentes pessoais digitais e terminais de ponto de venda.

Os dispositivos que se enquadram nesta configuração apresentam como características:

- Processadores de 16 ou 32 bits;
- Requerem de 160Kb a 512Kb de memória total disponível para a plataforma Java;
- Baixo consumo de potência, freqüentemente são dispositivos operados por bateria ;
- Conectividade com alguma espécie de rede, freqüentemente com uma conexão intermitente, sem fio e com largura de banda limitada (9600bps ou menos).

A especificação da CLDC não impõe nenhum requisito de hardware especifico, a não ser o requisito de memória de 160Kb-512Kb. Mais especificamente:

- 128Kb de memória não volátil para a máquina virtual e bibliotecas CLDC;
- Pelo menos 32Kb de memória volátil para o ambiente de execução e objetos alocados.

Como requisito de software, a especificação da CLDC assume que há um Sistema Operacional ou *kernel* disponível para gerenciar o hardware subjacente. Este sistema operacional deve fornecer pelo menos uma entidade escalonável para rodar a máquina virtual Java.

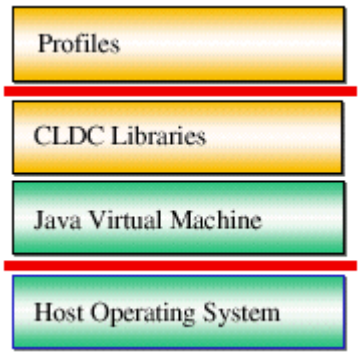

Figura 9 - Arquitetura da CLDC

A CLDC abrange as seguintes áreas:

- Características da máquina virtual e da linguagem;
- Entrada/saída;
- Acesso a rede;
- Segurança;
- Internacionalização;
- Bibliotecas de classes e APIs suportadas (java.util.*, java.lang.*) .

As seguintes áreas não são cobertas pela CLDC, devendo ser abrangidas por perfis montados no topo da CLDC:

- Interface com o usuário;
- Manipulação de eventos;
- Gerenciamento do ciclo de vida da aplicação (instalação, remoção, etc.)

6.7.1.2.1Características eliminadas da J2SE(Java 2 Standard Edition)

Algumas características foram eliminadas da JVM que suporta CLDC (KVM) porque as bibliotecas de classes incluídas na CLDC são mais limitadas que as presentes na edição padrão de Java ou sua presença poderia causar problemas de segurança na ausência de um modelo de segurança Java completo. São elas:

- Suporte a operações de ponto flutuante;
- Finalização;
- Java native interface(JNI);
- *Class loaders* definidos pelo usuário;

- Reflexão;
- Grupos de threads e daemon threads;
- Referências fracas.

6.7.1.2.2Processo de Verificação

A verificação de classes é responsável por detectar e rejeitar arquivos de classe inválidos. Por ser um processo caro e demorado de ser realizado, principalmente em pequenos dispositivos que apresentam restrição de recursos, este processo foi divido em duas etapas: pré-verificação e verificação no dispositivo. A pré-verificação geralmente é efetuada fora do dispositivo, ou seja, na máquina onde as classes Java foram compiladas ou no servidor de onde as aplicações Java foram baixadas. O dispositivo é responsável por algumas poucas verificações sobre as classes pré-verificadas para garantir que elas foram verificadas e continuam válidas.

6.7.1.2.3APIs suportadas

A API da CLDC pode ser dividida em duas categorias:

- Aquelas que são subconjuntos da edição padrão de Java (J2SE)
  - java.lang – classes fundamentais da linguagem de programação Java
  - java.io – entrada/saída através de *streams* de dados
  - java.util – variadas classes utilitárias
- Aquelas que são específicas da CLDC
  - javax.microedition.io – classes para conecções genéricas

Salienta-se que as classes presentes nos pacotes da CLDC herdados da J2SE apresentam um subconjunto reduzido das classes presentes na mesma. Para uma descrição completa das classes presentes em cada pacote da CLDC recomenda-se a leitura da especificação da API da CLDC.

6.7.1.2.4KVM (máquina virtual K)

A **KVM** (máquina virtual K) é uma máquina virtual compacta e portável, designada para dispositivos com pouca memória, recursos limitados e alguma forma de conexão em rede, tais como aparelhos celulares, *pagers* e PDAs . Estes dispositivos geralmente possuem microprocessadores de 16/32bits e requisito mínimo de memória de 128Kb.

Segundo a especificação da J2ME, a KVM é designada para ser:

- Pequena e com baixo requisito de memória;
- Modular e customizável;
- Enxuta e portável;
- Tão completa e rápida quanto possível.

A máquina virtual K é derivada de um sistema de pesquisa chamado **Spotless** dos laboratórios da Sun™. O objetivo do projeto **Spotless** foi implementar o sistema Java para o *Palm Connected Organizer*.

6.7.1.2.5KVM Porting

A maioria do código de fonte de KVM é comum a todas as implementações. A relativamente pequena quantidade de código dependente de máquina ou específico de uma plataforma está isolada em um número reduzido de arquivos. Ao se portar a KVM para novas plataformas é necessária a criação de novas versões dos mesmos.

Funções que implementam as seguintes operações devem ser re-implementadas para permitir a correta interação entre a KVM e o sistema operacional subjacente:

- Iniciações.
- Finalizações (clean-up).
- Alocação de memória no heap/liberação de memória.
- Manipulação de eventos.
- Tempo atual

Informações sobre como adaptar a KVM para novas plataformas podem ser obtidas no manual ***KVM Porting,*** que faz parte da documentação presente na implementação de referencia da CLDC, e que pode ser obtida a partir de [CLDC2001].

6.7.2 PERFIL

Perfis são direcionados para um tipo específico de dispositivo. Note-se que a definição de um perfil pode especificar uma das configurações. Também, um perfil pode conter outro perfil em sua definição.

6.7.2.1 Perfil MIDP (mobile information device profile)

O Perfil MIDP fornece uma plataforma padrão para dispositivos móveis, com restrição de recursos, pequenos, sem fio, como é o caso dos aparelhos telefônicos celulares, como exemplo. Tais dispositivos apresentam como características:

- 512Kb de memória total disponível para o ambiente de execução de bibliotecas de classes;
- Potência limitada. Tipicamente são operados a bateria;
- Conexão com algum tipo de rede sem fio com largura de banda possivelmente limitada;
- Interface com o usuário com variado grau de sofisticação.

MIDP é o primeiro perfil disponível para a plataforma J2ME. A combinação do CLDC e do MIDP fornece um ambiente completo de desenvolvimento para a criação de aplicações em celulares e *pagers,* por exemplo.

Aplicações que rodam em dispositivos que suportam MIDP são denominados MIDlets. A semelhança dos Applets, MIDlets são controlados pelo software que os executa. Por exemplo, no caso de um aparelho celular que suportasse MIDP/CLDC, os MIDlets seriam controlados pelo software que implementasse o perfil/configuração.

6.7.2.2 MIDP APIs

Por ser designado para operar sobre um perfil (CLDC) o perfil MID agrega os pacotes citados na descrição da CLDC, a saber:

- java.lang – classes fundamentais da linguagem de programação Java
- java.io – entrada/saída através de *streams* de dados
- java.util – variadas classes utilitárias
- javax.microedition.io – classes para conecções genéricas

Em adição, foram acrescentados outros pacotes específicos do perfil MIDP, são eles:

- javax.microedition.lcdui – classes utilizadas para a implementação de interfaces com o usuário para aplicações MIDlet;
- javax.microedition.rms – MIDP fornece um mecanismo para que os MIDlets armazenem e recuperem dados de forma persistente;
- javax.microedition.midlet – define aplicações MIDP (MIDlets) e as interações entre estas aplicações e o ambiente no qual a aplicação executa.

As classes de interface com o usuário, existentes no MIDP, estão localizadas no pacote **javax.microedition.lcdui.** Elas são constituídas de APIs de baixo nível e de alto nível.

A API de baixo nível fornece pouca abstração e é designada para aplicações que necessitam ter controle preciso sobre o posicionamento e atuação dos componentes gráficos, além de necessitar manipular eventos de baixo nível. Esta API é implementada pelas classes **Canvas** e **Graphics**. É importante salientar que a utilização da API de baixo nível não garante a portabilidade da aplicação porque elas acessam detalhes que são específicos de um dispositivo em particular.

A API de alto nível é designada para aplicações que rodam em dispositivos nos quais a portabilidade é importante. Para conseguir portabilidade, a API de alto nível fornece pouco controle sobre como as coisas devem aparecer. Por exemplo, aplicações utilizando API de alto nível não podem acessar entradas de dispositivos, como teclas específicas.

O pacote de **javax.microedition.rms** contém as classes necessárias para implementar uma base de dados de armazenamento persistente no dispositivo. Essa base de dados é limitada em suas potencialidades para armazenar e recuperar a informação devido às limitações do tamanho do dispositivo.

O pacote de **javax.microedition.midlet** contém a classe **MIDlet**. Esta classe executa o ciclo de vida do MIDlet e fornece o método ***getAppProperty(key)*** para recuperar

informação a partir das propriedades da aplicação definidas em um arquivo **jad,** que contém a descrição de algumas das características da mesma.

Para uma descrição completa das classes presentes em cada pacote do MIDP recomenda-se a leitura da especificação da API do MIDP que pode ser obtida a partir de [MIDP2001].

6.7.2.3 Mobile Information Device Profile (MIDP) for Palm OS

O perfil MIDP para Palm OS é um ambiente de execução de aplicações J2ME™ baseado nas especificações da CLDC e do MIDP. Este perfil foi liberado no início de Junho de 2001.

MIDP para Palm OS é designado para dispositivos portáteis rodando o Palm OS, versão 3.5 e inclui as seguintes funcionalidades:

- Uma versão binária da CLDC e do MIDP para a plataforma Palm OS;
- Um utilitário para converter MIDlets em arquivos PRC (**P**ilot **r**esource **d**atabase);
- Alguns exemplos de aplicações MIDlets;
- Documentação.

Dentre as características deste produto, destacam-se:
Características da especificação de MIDP 1.0:

- API gráfica de baixo nível (**Canvas**);
- API gráfica de alto nível (**LCDUI**);
- Acesso a banco de dados(**RMS**);
- Suporte a rede(**HTTP**).

Características específicas do Palm OS:

- Preferências;
- Suporte a HotSync;

7 Estudo de Caso

O estudo de casos consistiu no desenvolvimento de duas aplicações Java utilizando a tecnologia J2ME.

A primeira aplicação desenvolvida corresponde a um programa que utiliza a plataforma J2ME, ou mais especificamente, a configuração CLDC e o perfil MIDP, já vistos anteriormente. Trata-se de uma aplicação simples, que pode ser utilizada em *pagers* ou telefones celulares. O objetivo básico para o desenvolvimento desta aplicação é demonstrar a utilização da tecnologia J2ME e mostrar em que aspectos ela difere da programação para sistemas embarcados tradicional.

Esta aplicação desenvolvida constitui-se em um programa que acessa a caixa postal do usuário e exibe sua lista de e-mails. O usuário pode então ler seus e-mails e removê-los, ou mesmo enviar uma mensagem a partir do dispositivo no qual a aplicação está sendo executada.

A segunda aplicação corresponde a uma agenda telefônica e foi desenvolvida especificamente para PALM, usando a API kjava. Trata-se de um conjunto de bibliotecas de interface com o usuário que não fazem parte da especificação da CLDC, pois, como já foi salientado, a CLDC não define componentes de interface com o usuário.

7.1 A aplicação MIDP

A aplicação desenvolvida é denominada **MIDPine**, é constitui-se em um programa de e-mails que roda dentro do dispositivo que suporta a plataforma J2ME (CLDC/MIDP) e apresenta as seguintes funcionalidades:

- Acesso à caixa postal do usuário
- Leitura de mensagens/e-mails;
- Apagar mensagens/e-mails;
- Envio de mensagens a partir do dispositivo.

Esta aplicação explora os aspectos principais da plataforma, tais como acesso a APIs de alto e baixo nível e acesso a rede.

7.1.1 Estrutura da aplicação

Para que a aplicação funcione corretamente, são necessários três módulos independentes. Cada módulo constitui-se em uma aplicação distinta e responsável por uma atividade em particular.

Na figura abaixo, é apresentada a estrutura da aplicação, com a identificação dos módulos que a constituem e a interdependência entre os mesmos. Detalhes sobre cada módulo são apresentados na seqüência.

Figura 10 - Estrutura da Aplicação

7.1.2 MÓDULO MAILMIDLET

O primeiro módulo é o **MailMidlet,** que é o módulo responsável pela interface com o usuário e processamento dos comandos que o mesmo executa. Este módulo foi desenvolvido utilizando-se o perfil MIDP da J2ME que é designado para dispositivos móveis com algum acesso a rede.

A figura a seguir mostra um modelo simplificado de funcionamento do módulo MailMidlet. Neste diagrama, as setas representam transições e correspondem a ações do usuário ao clicar em botões específicos da interface gráfica. Por sua vez, os balões representam estados e correspondem a telas da interface com o usuário. Por exemplo, o estado **Listar e-mails** representa uma tela da interface que possui três botões (**ok, voltar** e **del**) e onde são listados os e-mails. Ao se clicar no botão **ok**, o usuário é levado para uma outra tela na qual ele poderá ler o e-mail selecionado. **Voltar** leva o indivíduo à tela anterior. Para finalizar, clicar em **del** remove o e-mail selecionado.

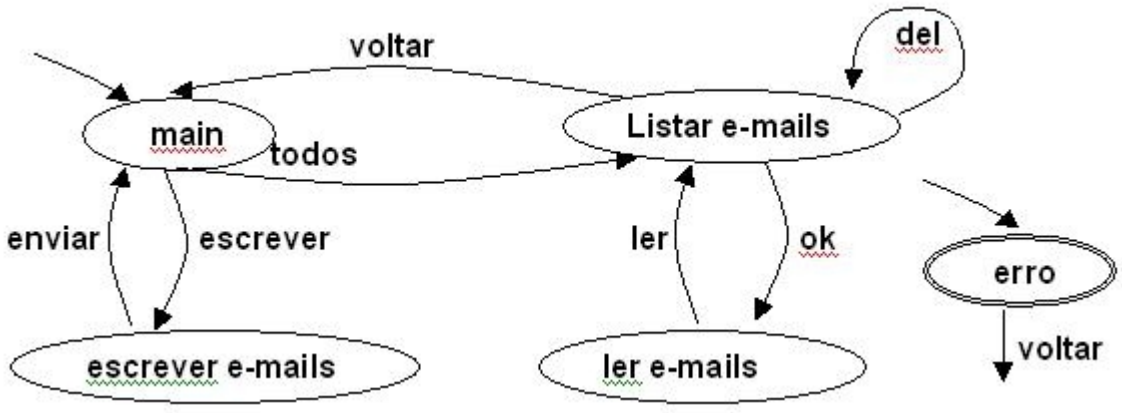

Figura 11 - diagrama de estados da aplicação

O módulo **MailMidlet** é o único que executa dentro do dispositivo que suporta a J2ME. Para se comunicar com o mundo exterior, é preciso alguma forma de acesso a rede. A configuração CLDC define algumas classes para entrada/saída incluindo acesso à rede. A estas classes, o perfil MIDP adiciona a interface ***HttpConnection***, para acesso ao protocolo HTTP, que define constantes e métodos necessários para uma conexão HTTP. A utilização do protocolo HTTP garante também que a aplicação seja portável entre todos os dispositivos moveis compatíveis com J2ME.

7.1.3 MÓDULO MAILSERVLET

Para acessar a caixa postal do usuário, o módulo **MailMidlet** solicita serviços a um segundo módulo, o **MailServlet**, através de uma interface de comunicação HTTP. Devido aos módulos estarem distribuídos, a API reduzida do MIDP/CLDC, e a utilização do protocolo HTTP para comunicação entre estes módulos, torna-se necessária à codificação dos dados a serem transmitidos/recebidos em um formato que possa ser transmitido via WEB e posteriormente restaurado do lado receptor. Por exemplo, quando o dado a ser trocado trata-se de um objeto, é necessária a codificação do mesmo em um formato textual para que possa ser transmitido pela WEB.

O **MailServlet** é responsável por receber as requisições e dados do módulo MailMidlet, convertê-los para um formato de dados conhecido e processá-las. O MailServlet é implementado como um Servlet, que é uma aplicação Java que roda no lado servidor (por exemplo em um *WEB server*) é que processa requisições HTTP. O MailServlet é uma espécie de *proxy* que recebe as requisições do módulo MailMidlet, interpreta-as e solicita o serviço correspondente ao terceiro módulo o **MailServer**.

7.1.4 MÓDULO MAILSERVER

O **MailServer** é o modulo que de fato acessa a caixa postal do usuário. Ele utiliza a API JavaMail™ para acessar a caixa postal do usuário e efetuar operações como:

- Ler a caixa postal do usuário;
- Remover e-mails;
- Enviar e-mails.

O MailServer comunica-se diretamente apenas com o módulo MailServlet, de forma que nenhuma conversão de dados entre os mesmos é necessária.

7.1.5 DESENVOLVIMENTO

Nesta secção será descrito apenas o processo de desenvolvimento do módulo **MailMidlet**, que corresponde a interface com o usuário, por ser o único modulo que utiliza a tecnologia J2ME. Os outros dois módulos são implementados com tecnologias bem interessantes no universo Java (Servlets e JavaMail API™) , mas, por não se tratarem de tecnologias específicas para o desenvolvimento de sistemas embarcados, não são apresentadas com mais detalhes.

O módulo **MailMidlet** constitui-se em uma aplicação MIDlet. MIDlets são pequenas aplicações escritas em Java e que rodam em telefones móveis e pagers que suportam o perfil MIDP. Para desenvolvê-lo, a etapa inicial consistiu em obter e instalar todo o software necessário, a saber:

- JDK 1.2 ou superior
- Connected Limited Device Configuration (CLDC)
- Mobile Information Device Profile (MIDP)

Instalados os softwares acima e configurado o ambiente de trabalho (por exemplo, paths e variáveis de ambiente), a próxima etapa consistiu na escrita da aplicação MIDlet cujo comportamento desejado está exemplificado no diagrama de estados apresentado em (§ 7.1.2).

Depois de criada, a aplicação MIDlet deve ser compilada. O processo de compilação adiciona uma terceira etapa ao processo tradicional de compilação em Java: a pré-verificação. A pré-verificação pré-processa o programa para ser usado pela máquina virtual K (KVM), sem, contudo mudar o nome da classe. Se esta etapa for esquecida, ocorrerá um erro quando se tentar executar a aplicação.
Depois de concluída a compilação e pré-verificação, a próxima etapa consiste em se executar a aplicação, chamando o programa **midp,** que vem junto com o software supracitado.

7.1.6 TESTES

Os teste da aplicação MIDPine foram realizados com a utilização do programa **midp**, cuja interface é similar a de um telefone celular. Na figura abaixo, é possível visualizar a interface deste programa, que é similar à de um telefone celular. Também são mostradas as telas principais que compõem a aplicação MIDPine.

Figura 12 - Aplicação MailMidlet no programa midp

Tela de leitura

Tela de escrita

Listagem de mensagens

7.1.7 Dificuldades Encontradas

Alguns problemas foram verificados durante o processo de desenvolvimento da aplicação MIDlet.

O primeiro deles diz respeito à instalação e a configuração do software necessário para se desenvolver as aplicações. O processo adotado no projeto consistiu em baixar a partir do site da Sun, os pacotes CLDC e MIDP e instalá-los, seguindo os passos descritos em artigos acessíveis a partir do site da Sun. Estes pacotes traziam o código fonte a ser compilado. O problema ocorrido foi que a informação descritiva da estrutura de diretórios dos pacotes CLDC e MIDP, depois de descompactados, não correspondia à obtida ao se seguir os passos indicados no artigo. Em decorrência não se conseguia compilá-los. A solução inicial adotada foi obter a partir de terceiros os pacotes já previamente compilados e instalados. Com o tempo descobriu-se como solucionar o problema: tratava-se se de uma pasta chamada classes que era criada após se executar os arquivos ***make*** que vinham com os pacotes e devia ser movida para o local apropriado.

O outro problema consistiu em se configurar o ambiente de execução e definir a partir de onde as classes deveriam ser executadas.

O processo de execução é relativamente complicado, já que envolve a compilação, pré-verificação e execução da aplicação. Um erro comum ocorrido foi o de se tentar executar

uma classe da aplicação que ainda não havia sido pré-verificada. A solução encontrada para evitar erros deste tipo foi concentrar as operações de compilação, pré-verificação e execução em um arquivo batch.

Desenvolver aplicações gráficas que utilizem a API de baixo nível (por exemplo, Canvas), é trabalhoso caso não se utilize alguma IDE.

Uma alternativa para superar os problemas de instalação e compilação da aplicação é a utilização do J2ME Wireless toolkit™, que é bem mais simples de se instalar e utilizar.

7.1.8 Análise da tecnologia

O desenvolvimento da aplicação transcorreu de forma relativamente simples. As maiores dificuldades encontradas foram relativas à instalação do software necessário para o desenvolvimento da aplicação. Um programador familiarizado com Java padrão dificilmente encontrará dificuldades em desenvolver aplicações com CLDC/MIDP porque:

- Trata-se de uma solução de software com a qual o mesmo está familiarizado;
- O hardware é completamente abstraído.

Por sua vez, esta mesma transparência de hardware é um enorme empecilho caso se queira customizar o hardware do dispositivo onde se está executando a aplicação, ou executar as aplicações em um novo hardware.

Como visto em (§ 6.7.1.2.5) , a máquina virtual KVM foi escrita inteiramente em C e opera sobre um Sistema Operacional que fornece transparência de hardware a mesma, entre outras coisas. Portar a KVM para uma nova plataforma não e um processo trivial, pois envolve no mínimo a utilização de duas linguagens de programação (linguagem C e Java) além do conhecimento do funcionamento dos módulos de software que compõem a KVM e sua lógica de funcionamento e das características do S.O. sobre o qual a máquina irá executar.

7.2 A aplicação CLDC para PALM

Como parte do estudo de caso, foi desenvolvida uma aplicação que utiliza a configuração CLDC. A versão A KVM utilizada foi uma versão específica para Palm PDAs. Como já especificado no decorrer deste trabalho, a CLDC não define nenhum componente gráfico, contudo foram utilizadas as classes presentes no pacote com.sun.kjava que constituem a GUI kjava e fornecem vários elementos gráficos (Button, RadioButton, TextField) os quais são genericamente chamados “widgets”.

7.2.1 Estrutura da aplicação

A aplicação é bem simples e constitui-se em uma agenda telefônica onde são armazenados o nome e o telefone dos indivíduos. O objetivo foi desenvolver uma aplicação para PALM

que utilizasse componentes gráficos e tivesse alguma forma de armazenamento persistente de dados.

A funcionalidade básica da aplicação consiste nas operações de inserção e busca de elementos na base de dados. Os elementos são compostos de nome e telefone. A busca dos mesmos é feita pelo nome.

O armazenamento dos elementos foi inicialmente temporário (dados permaneciam armazenados enquanto a aplicação executava). Posteriormente o armazenamento persistente foi introduzido, com a utilização da classe Database, que faz parte da biblioteca kjava.

7.2.2 DESENVOLVIMENTO

O desenvolvimento da aplicação foi relativamente simples, e baseado em exemplos de programas para PALM usando kjava são amplamente disponíveis na Internet. Por tratar-se de uma solução essencialmente de software, não é necessário nenhum conhecimento especifico e qualquer indivíduo com um conhecimento mínimo da linguagem Java e das bibliotecas de classes disponíveis para desenvolver tais aplicações não enfrenta maiores dificuldades.

7.2.3 TESTES

Os teste foram realizados utilizando-se o programa kvm, que vem com a CLDC da Sun, e com o POSE (Palm OS Emulator).

Uma característica interessante observada é que ao utilizar o programa kvm, observou-se que a funcionalidade de banco de dados que se queria testar não estava implementada no mesmo. A solução foi testar a aplicação no POSE, onde se obteve um resultado satisfatório. Abaixo é mostrado o resultado do teste da aplicação AgendaPalm tanto usando o programa kvm quanto através do Palm OS™ Emulator (POSE).

Figura 13 - Aplicação AgendaPalm vista no POSE

Figura 14 – A mesma aplicação vista no programa KVM

7.2.4 Dificuldades Encontradas

A instalação do software necessário ao desenvolvimento da aplicação e instalação e configuração do mesmo constituiu-se na maior dificuldade encontrada. Em particular, A instalação do POSE (PALM OS Emulator) demandou espaço físico em área de trabalho inicialmente não disponível. As imagens em ROM do sistema operacional do PALM foram cedidas por terceiros, não havendo a necessidade de se obter uma a partir do "download" de um Palm PDA real.

A maior dificuldade de programação refere-se sem dúvida à dificuldade em se posicionar os componentes gráficos (*widgets*) na tela do PALM sem o auxílio de ferramentas de programação com algum suporte gráfico. Embora a aplicação desenvolvida fosse deveras simples, a não utilização de tais ferramentas praticamente inibiu qualquer anseio em se desenvolver uma aplicação com maior apelo gráfico, até porque o objetivo central do desenvolvimento da aplicação não era este.

Em (§ 9.2) são apresentadas algumas ferramentas que podem auxiliar no desenvolvimento de aplicações Java para sistemas embarcados e em particular aquelas aplicações que possuem uma interface gráfica com o usuário.

7.2.5 ANÁLISE DA TECNOLOGIA

Como analise da tecnologia kjava, valem as mesmas observações feitas em (§ 7.1.8), com a ressalva que a utilização da API kjava deve ser evitada primeiro porque não faz parte da especificação de nenhum perfil ou configuração, e segundo porque já existe um perfil baseado no MIDP específico para PALM e cuja implementação de referência foi recentemente disponibilizada (§ 6.7.2.3).

8 Conclusão e trabalhos futuros

Este trabalho procurou mostrar as características básicas de Java e dos Sistemas Embarcados, e os esforços feitos no sentido de adequar Java para tais sistemas. Além disto, procurou-se descrever algumas das principais tecnologias Java existentes para o desenvolvimento de sistemas embarcados, com ênfase na J2ME e na RTSJ. Um estudo de casos foi realizado para mostrar a utilização destas tecnologias (especificamente, da J2ME) e o processo de desenvolvimento de aplicações utilizando tais tecnologias.

Como resultado, obteve-se um conhecimento mais profundo da estrutura e funcionamento de Java e dos seus principais elementos constituintes, como a máquina virtual, a linguagem de programação, o ambiente de execução, APIs e ambientes de aplicação. Este conhecimento foi essencial para o perfeito entendimento das técnicas de otimização utilizadas para fazer com que Java satisfizesse os requisitos e restrições dos sistemas embarcados, tais como memória e processamento.

Alguns dos principais entraves quanto à aplicação de Java em sistemas embarcados consistiam no tamanho de código elevado, ineficiência, não-determinismo e ainda ausência de suporte a operações em tempo real. Soluções para estes problemas foram apresentadas no decorrer do projeto: o tamanho de código pôde ser reduzido com a utilização de ferramentas como a JavaCodeCompact™ ou definição de plataformas com requisitos mínimos de memória para serem utilizadas em sistemas embarcados. A ineficiência de Java pôde ser atacada com a utilização de algumas técnicas de compilação, umas das quais são mais aplicáveis a sistemas embarcados que outras. Como alternativa de solução para os problemas de determinismo de Java que são, algumas das principais alternativas para o algoritmo de coleta de lixo padrão foram apresentadas. Por fim, para dar suporte a operações de tempo real, que é um requisito presente na maior parte dos sistemas embarcados, a especificação de Java para tempo real foi proposta. Esta especificação procurou fortalecer a semântica do modelo de programação concorrente de Java.

Ela objetivou também atender a alguns dos principais requisitos dos sistemas embarcados, como acesso ao hardware (através da memória física e técnicas como Entrada/Saída mapeada em memória), escalonamento e manipulação de eventos assíncronos.

Há, todavia limitações que não foram tratadas pelas tecnologias Java estudadas e que são relevantes para o desenvolvimento de sistemas embarcados.

Em primeiro lugar, Java não dá suporte a todo e qualquer tipo de sistema embarcado. Em particular, ela é aplicável apenas as classes ou categorias de dispositivos que pertençam a alguns dos segmentos de mercado descritos em (§ 4.1) e para os quais a Sun Microsystems™ tem proposto tecnologias Java. Algumas tecnologias Java são estruturadas

de forma a facilitar a transferência das mesmas para uma plataforma ou dispositivo ("*port*"), todavia, esta é uma operação complexa que exige um alto grau de especialização de quem a executa, pois este deve dominar no mínimo duas linguagens de programação (Java, certamente e pelo menos a linguagem C, posto que a maior parte do código fonte das tecnologias Java disponível está escrita nesta linguagem), conhecer a estrutura da tecnologia Java que se está tentando portar, além do domínio do Sistema Operacional sobre o qual a máquina virtual executa.

O exposto acima mostra que é difícil, portanto, tentar customizar o hardware de um dispositivo para atender a um requisito em particular, por mais simples que seja a alteração, como, por exemplo, mudar um display utilizado em um dispositivo, de um display LCD de uma linha outro de duas linhas, pois não há amplo suporte a manipulação do hardware (A RTSJ permite apenas que dispositivos sejam mapeados em memória, portanto o acesso a hardware continua bastante limitado).

Manipulação direta do hardware é um requisito importantíssimo quando do desenvolvimento de sistemas embarcados porque o hardware que os compõe pode variar muito de um sistema para outro, em função dos requisitos de projeto. Geralmente, a espinha dorsal é mantida (ou seja, a família de microprocessadores utilizada), contudo os periféricos que realizam entrada/saída variam bastante mesmo entre sistemas que desempenhem funções similares.

Em cima do modelo de desenvolvimento de tecnologias para sistemas embarcados adotado em Java, no qual as tecnologias são desenvolvidas para um perfil definido pelo mercado e que geralmente atendem a uma categoria ou classe de dispositivos, propõe-se outro no qual componentes de software (ou beans, como chamados em Java), sejam desenvolvidos para abstrair componentes específicos do hardware que compõe um dispositivo. Por exemplo, haveria beans para teclados, displays, processadores, memória ou qualquer outro componente de software que pudesse ser utilizado. A junção destes beans formaria um sistema embarcado completo.

A vantagem teórica deste modelo é que seria mais fácil desenvolver e distribuir os beans, os quais poderiam ser utilizados para modelar os mais diversos tipos de arquitetura de hardware. Desta forma, este modelo poderia ser utilizado por uma variedade maior de desenvolvedores. Obviamente, aspectos de desempenho e tamanho deveriam ser reavaliados.

Muitos aspectos relativos ao tema central deste trabalho não foram aqui abordados, em função da abrangência do tema do trabalho. Estes são colocados como indicações para trabalhos futuros a serem feitos sobre o tema e são descritos nas próximas linhas.

O estudo apresentado neste trabalho pode ser colocado como essencialmente o levantamento e análise de tecnologias de software existentes. Contudo, várias tecnologias de hardware auxiliam na utilização de Java em ambientes embutidos.

Vários fabricantes, de forma independente da Sun Microsystems™, têm desenvolvido tecnologias baseadas em Java (seguindo as especificações definidas pela Sun ou baseadas nas mesmas) que por vezes são mais eficientes que as alternativas da própria Sun.

9 Apêndice

9.1 TERMINOLOGIA

Bytecode – instrução da máquina virtual Java.
Clean-room – que não depende da Sun ou outro fabricante
Determinismo – capacidade de executar uma tarefa dentro de um certo período de tempo.
Escalabilidade – habilidade de uma aplicação de computador ou produto de continuar sua execução normalmente caso ele (ou seu contexto) seja modificado em tamanho ou volume para atender às necessidades do usuário.
Garbage Collector - O garbage collector (GC) é um programa que executa em segundo plano (em background) e libera memória previamente alocada e que não está mais acessível às aplicações Java.
Gateway – em redes, hardware/software que liga dois tipos diferentes de redes.
Handler – Manipulador.
HUB – ponto comum de conexão para dispositivos em uma rede.
Heap – em programação, o termo heap refere-se a uma área de memória reservada para os dados que são criados em tempo de execução.
Jad file – arquivo de manifesto utilizado em aplicações MIDlet e que contém informações sobre o mesmo.
JCP – Java Community Process – é a maneira que a plataforma Java evolui. É uma organização aberta de desenvolvedores Java de todo o mundo, cujo objetivo é desenvolver e revisar especificações da tecnologia Java e implementações de referência.
Kernel – corresponde à parte central de um sistema operacional de computador, a qual fornece os serviços básicos para as demais partes do mesmo.
Máquina Virtual Java – A máquina virtual Java (JVM) define um computador abstrato. Sua especificação define a funcionalidade que toda máquina virtual deve ter, mas dá liberdade quase total aos projetistas de cada implementação.
PDA – Personal Digital Assistant – dispositivo portátil que combina características de telefone, fax, agenda, computação e rede.
Preempção – processo realizado pelo Sistema Operacional que consiste em interromper uma tarefa em execução e passar o controle a uma outra tarefa.
Sandbox – o termo *sandbox*, ou caixa de areia, refere-se a um ambiente restrito, no qual aplicações Java tais como applets executam e têm suas atividades verificadas para prevenir que atividades potencialmente perigosas, tais como abertura de conexões em rede, possam ser realizadas.
Sistema Embarcado – sistema de computação especializado, que faz parte de uma máquina ou sistema mais amplo.
Site – Site equivale a um "local" na Web que contém uma pagina inicial (*home page*), além de arquivos e documentos adicionais.

Tempo Real – Um sistema é de Tempo Real quando ele responde a entradas e fornece saídas, rápido o suficiente para atender os requisitos do hardware ou do usuário [SCH1999].

9.2 FERRAMENTAS DE DESENVOLVIMENTO PARA JAVA

Abaixo são apresentadas algumas ferramentas de desenvolvimento para Java. As ferramentas forma escolhidas em função do suporte que dão ao desenvolvimento de sistemas embarcados. Não se levou em conta nenhum aspecto de desempenho tampouco se pretendeu expor todas as ferramentas disponíveis, que são muitas. As informações fornecidas foram coletadas dos respectivos sites.

9.2.1 FORTE™ FOR JAVA

Forte™ for Java é um ambiente de desenvolvimento integrado (IDE) para desenvolvedores que utilizam a tecnologia Java™.
URL: http://www.sun.com/forte/ffj/

9.2.2 CODEWARRIOR FOR JAVA™

CodeWarrior for Java™ é um ambiente integrado de desenvolvimento para programar aplicações 100% Java, desde o *desktop* aos dispositivos sem fio utilizando os padrões J2SE™, PersonalJava™ e J2ME™.
URL: http://www.metrowerks.com/desktop/java/

9.2.3 WHITEBOARD™ SDK

O WHITEboard™ SDK fornece um ambiente completo de desenvolvimento para a criação e teste de aplicações Java sem fio para dispositivos móveis compatíveis com J2ME™.
O WHITEboard SDK é o primeiro kit de desenvolvimento para estender a funcionalidade Bluetooth para o desenvolvimento de aplicações Java sem fio.
URL: http://www.zucotto.com/whiteboard/

9.2.4 JBUILDER™ HANDHELD EXPRESS™

JBuilder Handheld Express fornece suporte para o desenvolvimento de soluções utilizando a plataforma J2ME™.
URL: http://www.borland.com/jbuilder/hhe/

9.2.5 VisualAge Micro Edition

A VisualAge Micro Edition foi considerada a melhor ferramenta de desenvolvimento para sistemas embarcados de 2001, em pesquisa realizada pela revista JavaPro e pela Sun Microsystems™.
URL: http://www.embedded.oti.com/

9.3 COLETÂNEA DE MÁQUINAS VIRTUAIS JAVA PARA SISTEMAS EMBARCADOS

Abaixo são apresentadas algumas máquinas virtuais Java existentes para o universo dos sistemas embarcados, com a descrição de suas características básicas. A coletânea de JVMs aqui apresentada é fruto do resultado de pesquisa até a data de entrega deste trabalho, não objetivando cobrir todas as implementações da JVM existentes.

A descrição das características das JVMs apresentadas foram obtidas a partir dos *sites* dos respectivos desenvolvedores, cujo url é apresentado antes da descrição de cada JVM.

9.3.1 JALAPEÑO

URL: http://www.research.ibm.com/jalapeno/about.html

Jalapeño é uma máquina virtual (**VM**) desenvolvida pela IBM™. Suas principais características são:

- A máquina virtual é toda implementada em Java,
- A JVM utiliza dois compiladores e nenhum interpretador,
- Utiliza compilação adaptativa.

9.3.2 CHARIS PICO VIRTUAL MACHINETM(PVM) FOR JAVA

URL: http://www.charis.com/

A Charis pVM (pico Virtual Machine) é uma máquina virtual totalmente compatível com a especificação da máquina virtual Java, da Sun™.

Ela é otimizada para dispositivos embarcados que são baseados em microcontroladores de 8/16/32 bits tais como telefones celulares, *pagers*, dispositivos industriais de controle, cartões inteligentes, etc. A pVM é desenvolvida independentemente e pode rodar com um sistema operacional de tempo real ou sem nenhum sistema operacional subjacente.

A implementação completa da pVM usa 32K de ROM (ou menos no caso de JavaCard™ VM) , tanto para o código da VM quanto para as bibliotecas de classes suportadas. Ela também suporta carga automática de classes, *garbage collection*, acesso a funções C via JNI, programação concorrente através do uso das bibliotecas JavaActor™ e picoActor™.

9.3.3 CHAIVM

URL: http://chai.hp.com/

A ChaiVM é uma máquina virtual desenvolvida na Hewlett-Packard Embedded Software Operation (EMSO), sendo compatível com a especificação da máquina virtual Java , e fornecendo suporte para JNI.

Os requisitos mínimos de memória da ChaiVM é de 228Kb (ROM). Os requisitos de memória ROM das bibliotecas de classes principais java.lang, java.útil e java.io são de 89Kb, 37Kb e 63Kb, respectivamente.

O algoritmo de *garbage collection* utilizado na ChaiVM é o tri-color mark-and-sweep. O *GC* executa no seu próprio thread, concorrentemente com os threads dos usuários. A freqüência, duração, e prioridade do GC pode ser controlada pelo desenvolvedor, fazendo a ChaiVM dar suporte a aplicações de tempo real.

9.3.4 KADAVM

URL: http://www.kadasystems.com/kada_vm.html

A VM KadaVM é uma implementação "clean-room", completamente funcional, da máquina virtual Java. Ela vem agregada com as APIs Kada para fornecer um ambiente de distribuição Java para dispositivos portáteis com restrições de recursos..

Existem duas configurações da Kada VM, a Kada Compact VM , para aplicações em rede e com interface gráfica restrita, e a Kada Standard VM, para aplicações de banco de dados.

A KadaVM tem baixo requisito de memória , 155Kb (Compact) ou 330Kb (Standard) .
KadaVM é a primeira implementação independente, completamente funcional, compacta, da especificação PersonalJava para a plataforma PALM.

A KadaVM utiliza compilação JIT adaptativa - o compilador adaptativo JIT monitora a execução e só compila código executado freqüentemente. Isto pode aumentar o desempenho de 3 a 14 vezes, quando comparado a execução normal do bytecode. JIT adaptativa não compila todos os métodos, só os executados mais freqüentemente. Adaptativo também significa que o compilador usa espaço do heap para compilação.

Dentre as outras características da KadaVM , cita-se o suporte a Java Native Interface(JNI), operações em ponto flutuante, multithreading, class reflection, serialização de objetos, e capacidade de *trace*.

9.3.5 KAFFE

URL: http://www.kaffe.org/

Kaffe é um ambiente de execução Java completamente funcional, aberto e “clean room”. Ele é Open Source e vem com código fonte completo, e é distribuído sob a GNU Public License (GPL). Seus três componentes principais são a máquina virtual, a biblioteca de classes compatíveis com Java 1.1 e um conjunto de bibliotecas nativas implementadas em várias plataformas alvo.

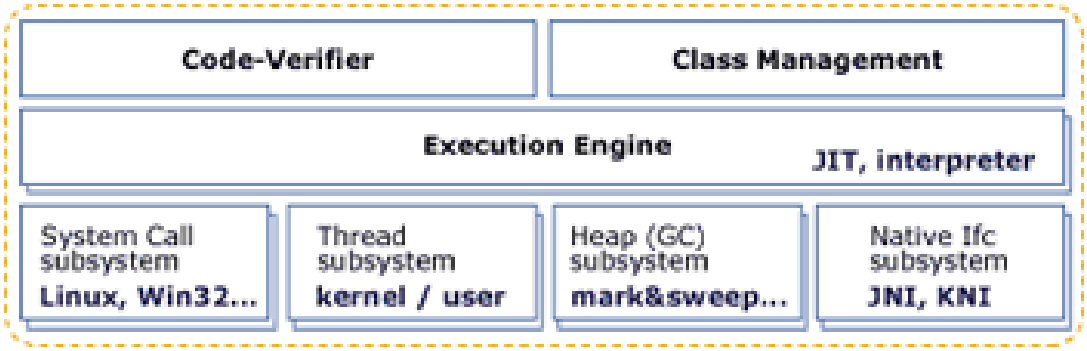

Figura 15 - Modular Java 1.1 Compliant Virtual Machine

A máquina virtual Kaffe é modular e utiliza subsistemas para threading, gerenciamento de memória, interface de métodos e chamadas de sistemas nativos.

O algoritmo utilizado pelo GC do Kaffe é o *mark-and-sweep*, contudo, Kaffe permite que se substitua o garbage collector com outro mais apropriado para a aplicação, tal como reference counting, generational collector ou copy collector para alocação mais rápida.

Há duas opções no Kaffe para se implementar métodos nativos: JNI – para escrever bibliotecas nativas que são portáteis entre diferentes VMs (para mesma plataforma de HW/ SW) e KNI (Kaffe native interface), que é mais rápida que JNI porem menos portável.

O engenho de execução da Kaffe vem em três diferentes sabores: interpretado, JIT e AOT. O interpretador é menor e mais portável, o JIT requer uma camada de macros para ser escrito, contendo as instruções em assembler. Isto permite que código seja traduzido para código nativo sob demanda. O AOT permite que o código seja diretamente compilado para código nativo objetivando máxima performance.

Dentre as plataformas suportadas pela Kaffe VM: Embedded Linux, VxWorks, LynxOS, SMX, ThreadX, Linux (todas as distribuições), DOS, Windows NT 4.0, Windows 98, Windows 2000, Windows CE, Solaris, SunOS, BSDI, FreeBSD, NetBSD, OSF/1, Unixware, HP-UX, NextStep, OpenStep. Processadores suportados incluem ix86, Sparc, m68k, StrongARM, MIPS, Alpha, PowerPC and PARisc.

9.3.6 PERC VM

URL: http://www.newmonics.com/perc/info.shtml

A Perc JVM é uma máquina virtual desenvolvida pela NewMonics Inc. para sistemas embarcados. Ela é compatível com a JDK versão 1.1 e suporta *garbage collection* em tempo real, incremental, exata e defragmentante, além de tarefas de tempo real.

Ela inclui todos os componentes padrões: o interpretador, o *class loader*, o verificador de bytecode e bibliotecas de classes.

A Perc VM requer 128Kb, enquanto PERC com bibliotecas típicas apresenta requisitos de ROM da ordem de 256 Kb -900 Kb e RAM da ordem de 64 Kb -128Kb.

Com a máquina virtual PERC, é possível substituir, modificar, e melhorar aplicações existentes sem ter que reiniciar o computador, através da carga dinâmica de classes.

A Perc VM pode ser ajustado para privilegiar o desempenho em tempo de execução, o tempo de *start-up* ou o requisito de memória. Ela está disponível para uma variedade de sistemas operacionais (Linux, Wind River's VxWorks, ISI's pSOSystem, Windows® NT,Phar Lap's ETS, VenturCom's RTX) e processadores (x86, PPC, MIPS, ARM, SPARC, e 68k).

9.3.7 SIMPLERTJ

URL: http://www.rtjcom.com/

SimpleRTJ é uma implementação da máquina virtual Java que apresenta requisitos mínimos de memória(17Kb-19Kb na maioria dos microcontroladores) e direcionada para dispositivos embarcados e sistemas operacionais de tempo real. Ela foi projetada para executar em sistemas de 8/16 bits e com pouquíssima memória.

Para minimizar o tempo de iniciação (start-up) das aplicações Java e aumentar a velocidade de execução do bytecode, a simpleRTJ utiliza aplicações Java pré-ligadas. Os arquivos de classe são ligados no computador servidor usando-se um compilador cruzado, e os arquivos gerados são introduzidos no dispositivo alvo para execução direta pela SimpleRTJ.

SimpleRTJ foi portada para uma série de processadores (MC68302, MC68376/332, 68HC11, 68HC16, 8051, 8051XA, ARM , H8S/2241, STi5512, DSP56300) .

Este é um produto comercial, contudo, seu uso está livre para avaliação, ou para aplicações não comerciais.

9.3.8 WABA

URL: http://www.wabasoft.com/

Waba define uma linguagem, uma máquina virtual, um formato de arquivo class e um conjunto de bibliotecas de classes que são um subconjunto das respectivas especificações Java. Em decorrência, programas Waba podem rodar em uma máquina virtual Java, enquanto apenas aplicações Java que utilizam o subconjunto de funcionalidade Waba podem executar em sua máquina virtual.

Waba é uma plataforma de programação *Open Source* que permite o desenvolvimento de programas para PalmPilots ou WindowsCE, caso os mesmos possuam uma máquina virtual Waba, ou para qualquer dispositivo com suporte para Java.

9.3.9 WIND RIVER'S VXWORKS

URL: http://www.wrs.com/internet/html/java.html

Wind River´s Personal JWorks™ é uma implementação do ambiente de aplicação PersonalJava™, da Sun sobre o sistema operacional de tempo real VxWorks, da WindRiver® Personal Jworks possui certificado de total compatibilidade com a especificação PersonalJava™.

9.3.10 TINYVM

URL: http://tinyvm.sourceforge.net/

TinyVM é uma máquina virtual open source e firmware de substituição para o microcontrolador MCX, da Lego MindStorms™. O requisito de memória da TiniVM é aproximadamente 10Kb no RCX.

TinyVM suporta um conjunto reduzido da especificação Java. Dentre as suas características destacam-se:

- Acesso a botões RCX
- Recursão
- Não necessidade de se instalar um compilador cruzado
- O firmware da tinyVM permite que ele mesmo seja substituído.

Dentre as limitações mais importantes e características ausentes na TiniVM, cita-se:

- Ausência de *garbage collection*
- Sem suporte a operações de ponto flutuante
- Constantes do tipo string são ignoradas

- Sem declarações **switch**

Uma alternativa mais completa (embora menos compacta) é a **leJOS**, da qual pode-se obter maiores informações a partir do url **http://sourceforge.net/projects/lejos** .

9.3.11 JBED™ RTOS

URL: http://www.jbed.com/

O Jbed™ RTOS é uma combinação de uma máquina virtual Java e de um Sistema Operacional de Tempo Real. Que esta disponível para uma variedade de plataformas embarcadas, como os processadores ARM, PowerPC, 68xxx e Coldfire. Jbed™ RTOS é designado para executar diretamente sobre o hardware, sem a necessidade de nenhum software externo adicional.

Há outras versões da Jbed, como a Jbed light, que é uma versão resumida da Jbed RTOS, e a Jbed Micro Edition CLDC, que é uma máquina virtual para telefones celulares, PDAs e aparelhos para internet. Ela é compatível com a especificação da CLDC definição sob o JCP (*Java community process),* mas é, segundo os fabricantes, cerca de 50 vezes mais rápida que qualquer outra JVM porque sempre compila as classes Java, ao invés de interpretá-las. Ao contrario do Jbed RTOS e do Jbed™ light, o Jbed Micro Edition CLDC™ também pode executar sobre um sistema operacional como o Palm OS. Além de poder executar diretamente sobre o hardware.Uma implementação do MIDP também está disponível (Jbed™ profile for MID).

9.3.12 KVM

URL: http://java.sun.com/products/kvm/

Detalhes sobre a **kvm (máquina virtual K)** podem ser encontrados em (§ 6.7.1.2.4).

9.3.13 SKELMIR

URL: http://www.skelmir.com/

Skelmir coloca-se como uma companhia de software especializada em soluções para sistemas embarcados utilizando tecnologia Java.

Skelmir introduz o CEE-J (The **C**ompatible **E**xecution **E**nvironment for **J**ava) que é uma implementação *clean-room* de Java da Sun™ e que apresenta, entre outras características:

- Suporte a dois tipos de garbage collection: mark-and-sweep e coleta generacional (baseada no tempo de vida dos objetos).

- Requisitos de memória
    - Biblioteca de classes - 360kb sem AWT. 760kb com AWT
    - Código binário da VM - 300kb sem AWT. 700kb com AWT

  O tamanho do código binário varia em função do processador alvo e compilador.
- Suporte a JNI
- Suporte a arquivos class comprimidos

9.3.14 JAMAICAVM

URL: http://www.aicas.com/

10Referências

[HEA1998] HEATH, S. Embedded Systems Design. Reading: Butterworth-Heinemann, 1998. 350p.

[SCH1999] SCHULTZ, T.W. C and the 8051-Building efficient applications. Vol. 2. Prentice Hall PTR, 1999. 458p. p22-25.

[JONES1996] JONES, R., LINS, R. Garbage Collection, Algorithms for Automatic Dynamic Memory Management. Wiley, 1996

[GREHA1998] GREHAN, R.; MOOTE, R.; CYLIAX, I. Real-Time Programming – A guide to 32-bit embedded development. Reading: Addison-Wesley Longman, 1998. 694p.

[LIND1999] LINDHOLM, T.; YELLIN, F. The Java Virtual Machine Specification (2nd Edition). ISBN 0-201-43294-3. Reading: Addison-Wesley Longman, 1999. 473p

[SAN2000] SANTOS, A. C. O. Tolerância a Falhas para Sistemas Embarcados. 2000.

[LEAO2000] LEÃO, H.B.S. Java para sistemas embarcados.

[BOL2000] BOLLELLA G., GOSLING J., TURNBULL M., DIBBLE P.BROSGOL B., FURR S., HARDIN D. The Real Time Specification for Java™. Reading: Addison-Wesley Longman, 2000. 204p. ISBN: 0-201-70323-8. Disponível: site *RTJ.org*.
URL: http://www.rtj.org/

[JUN1996] JUNIOR, P. J. K. Embedded System Design Issues (the Rest of the Story) Disponível: *School of Computer Science, Carnegie Mellon site*. Consultado em 05 mar. 2001. URL: http://www.cs.cmu.edu/~koopman/iccd96/iccd96.html

[PER1998] PersonalJavaTM 1.1 Application Environment Memory Usage Technical Note. Disponível: site *Sun*. URL: http://java.sun.com/products/personaljava/MemoryUsage.html

[WEI1991] WEISER, M. The Computer for the Twenty-First Century.
URL: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[SEG0001] Consumer and Embedded Technologies. Disponível: site *Sun Microsystems*™.
URL: http://java.sun.com/products/consumer-embedded/

[TOO0001] PERSONALJAVA AND EMBEDDEDJAVA DEVELOPMENT TOOLS. Disponível: site *Sun Microsystems*™.

URL: http://java.sun.com/products/personaljava/pjava_and_ejava_tools.html

[HOW2001] Chen, Z. How to write a Java Card applet: A developer's guide. Disponível: site *Javaworld*.
URL: http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard_p.html

[CARD2001] JAVA CARD™ TECHNOLOGY. Disponível: site *Sun Microsystems*™.
URL: http://java.sun.com/products/javacard/

[JES2001] THOMAS A., SEYBOLD, P. Java Embedded Server™ - White papers. Disponível: site *Sun Microsystems*™.
URL: http://www.sun.com/software/embeddedserver/whitepapers/whitepaper1.html#java

[PHONE2001] JavaPhoneTM API – White Paper Disponível: site *Sun Microsystems*™.
URL: http://java.sun.com/products/javaphone/overview/

[TV2001] JavaTM Technology in Digital TV Disponível: site *Sun Microsystems*™.
URL: http://java.sun.com/products/javatv/

[MIDP2001] Mobile Information Device Profile (MIDP). Disponível: site Sun Microsystems™ (11 abr. 2001).
URL: http://java.sun.com/products/midp/

[CLDC2001] CLDC and the K Virtual Machine (KVM). Disponível: site Sun Microsystems™ (11 abr. 2001).
URL: http://java.sun.com/products/cldc/

[CT0001] Compilation Technologies. Disponível: site Skelmir.
URL: http://www.skelmir.com/compilers.html Consultado em 12 abr. 2001

[JIT0001] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java virtual machine for server applications: The Jvm on OS/390. Disponível: site IBM.
URL: http://www.research.ibm.com/journal/sj/391/dillenberger.html

[GC0001] No Silver Bullet - Garbage Collection for Java in Embedded Systems
Petit-Bianco, A. Disponivel: Cygnus Sourceware site
URL: http://gcc.gnu.org/java/papers/nosb.html

[BECK0001] Análise Comparativa dos Ambientes EmbeddedJava e Real-Time Java
Becker, L. B., Geyer C. F. Consultado em 01 abr. 2001.
URL: http://www.delet.ufrgs.br/~cpereira/temporeal_pos/www/JavaRTvsEmb.html

[JCON1999] J Consortium. Disponível site J Consortium.
URL: http://www.j-consortium.com/

[WORK1999] WORKMAN, R. C. The Evolution of Java™ Technology for Embedded Devices. Consultado em 03 jul. 2001.
URL: http://www.circuitcellar.com/eiw/99eiw/RWorkman/workman.htm

[HEIS2001] HEISS J. J. Embedded Systems Go Real Time With Java™ Technology. Disponível: site *Sun Microsystems* (27 abr. 2001).
URL: http://java.sun.com/features/2001/04/embed.html?frontpage-headlinesfeatures

[PERR2001] PERRIER, V. Can Java Fly - Adapting Java to Embedded Development. Disponível: site *Wind River Systems*. Consultado em 12 abr. 2001
URL: http://www.wrs.com/internet/html/can_java_fly.html#

[PAW2001] PAWLAN, M.; Introduction to Consumer and Embedded Technologies. Disponível: site *Sun Microsystems* (28 ago. 2000). URL:
http://developer.java.sun.com/developer/technicalArticles/ConsumerProducts/intro/

[DAY1999]DAY, B. Program Java Devices – An Overview. Disponível: site *JavaWorld* (01 mai. 2001) URL: http://www.javaworld.com/javaworld/jw-07-1999/jw-07-device.html

[PETI1998] PETIT-BIANCO A. Java Garbage Collection for Real-Time Systems
Disponível: site *Dr. Dobb´s* (out 1988).
URL: http://www.ddj.com/articles/1998/9810/9810a/9810a.htm

_____

Wellington João da Silva
Aluno

_____

Sérgio Vanderlei Cavalcante
Orientador