UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Luciano Secchi

Ambiente de execução para aplicações escritas em Java no Sistema EPOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Antônio Augusto Medeiros Fröhlich

Florianópolis, fevereiro de 2004

Ambiente de execução para aplicações escritas em Java no Sistema EPOS

Luciano Secchi

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Raul Sidnei Wazlawick

Banca Examinadora

Antônio Augusto Medeiros Fröhlich

Wolfgang Schröder-Preikschat

Frank Augusto Siqueira

Rômulo Silva de Oliveira

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Exemplos

Resumo

Este trabalho mostra uma nova metodologia para executar aplicações escritas em Java em sistemas embutidos que possuam severas restrições de *hardware*. Para isto, esta metodologia faz uso da técnica de engenharia de *software* denominada *Application-Oriented System Design*.

Dada uma aplicação ou conjunto de aplicações, um ambiente de execução sob medida é construído para suprir suas necessidades. As necessidades da aplicação são obtidas de forma automática, através da análise de seu código. A análise é feita a partir do código Java compilado (*bytecode* Java). Essa abordagem possibilita que mesmo aplicações ou bibliotecas sem código fonte conhecido possam ser analisadas com esta técnica.

O ambiente de execução é construído a partir de componentes de *software* que são utilizados de acordo com regras de dependência e modelos de custo. Os componentes de *software* utilizados fazem parte do sistema EPOS, um sistema que segue os princípios da *Application-Oriented System Design*.

O código da aplicação, em *bytecode* Java, pode ser reduzido através de métodos de compactação de código e extração. Antes do passo da análise de dependências, os elementos de *software* da aplicação que são fundamentais a sua execução são selecionados e a aplicação pode ser reconstruída sem dependências desnecessárias.

O trabalho propõe a transformação da aplicação escrita em Java para código nativo do *hardware*. A transformação para código nativo diminui a necessidade de memória e melhora o desempenho das aplicações. Esta transformação pode ser feita com compiladores *Ahead-of-Time*, como é o caso do compilador GNU GCJ.

Abstract

This work presents a methodology to run Java-written applications in embedded systems with severe hardware restrictions. To do that, this methodology takes use of a software engineering technique called *Application-Oriented System Design*.

For a given application or application set, a custom run-time environment is constructed to supply its needs. The needs of the application are found automatically through code analysis. The analysis is performed in the Java compiled code (bytecode Java). This approach makes possible even applications or libraries without known source code can be analyzed with this technique.

The run-time environment is built up from software components that are used in accordance with dependence rules and cost models. The used software components are part of EPOS system, a system that follows the principles of the Application-Oriented System Design.

The application code, in Java bytecode, can be reduced through program compaction and extracting methods. Before the analysis step, the application software elements that are fundamentals to its execution are selected and the application can be rebuilt without useless dependences.

The work considers the transformation of the Java written application to hardware native code. The transformation to native code reduces the memory usage and improves the performance of the applications. This transformation can be made with *Ahead-of-Time* compilers.

Capítulo 1

Introdução

Sistemas embutidos são amplamente utilizados nos mais diversos segmentos de mercado. Aparelhos de uso cotidiano, tais como carros, aparelhos de telefone celular e fornos de microondas, freqüentemente possuem algum processador que executa uma ou mais aplicações específicas [BEU 99]. A maioria dos processadores produzidos hoje em dia é utilizada em sistemas de computação dedicada, principalmente como sistemas embutidos. Sistemas embutidos tem características bem diferentes dos computadores pessoais. Normalmente eles têm escassos recursos de *hardware* (muitos não tem MMU[1], por exemplo) e em sua maioria são constituídos por micro-controladores de 8 bits com pouca memória [TEN 00].

Em conseqüência das restrições neste tipo de plataforma, é necessário fazer sistemas que permitam obter o máximo desempenho possível do *hardware* utilizado. Para que seja obtido o desempenho máximo, tais sistemas geralmente são feitos com o uso de linguagens programação de baixo nível, tais como C e linguagem *assembly*. Por este motivo, normalmente estes sistemas são desenvolvidos exclusivamente para uma determinada plataforma de *hardware*. Caso seja necessário utilizar a mesma aplicação em uma plataforma de *hardware* diferente, o código da aplicação deve ser reescrito ou adaptado à nova plataforma.

Para auxiliar esta tarefa, muitas vezes existem bibliotecas de funções

[1] *Memory Management Unit*

que escondem particularidades de *hardware* da aplicação. Nestes casos, o porte de aplicações é simplificado pois somente as bibliotecas de funções devem ser adaptadas à nova plataforma. Entretanto, existem casos onde toda a aplicação deve ser adequada a cada nova plataforma.

Estas dificuldades na adaptação de aplicações fazem os custos de produção de *software* aumentarem pois mais tempo é gasto no desenvolvimento dos sistemas. Este tempo adicional também é prejudicial para o fabricante de sistemas embutidos, que deve lançar seus produtos o mais cedo possível para aproveitar possíveis janelas de consumo.

Java é uma plataforma que ajuda a diminuir o tempo de desenvolvimento das aplicações. Java foi desenvolvida com objetivo de ser usada em aplicações embutidas em dispositivos eletrônicos. A plataforma Java foi projetada com atenção especial para ter o mínimo de dependências de implementação possível. Desta forma, permitindo aos desenvolvedores escreverem aplicações uma única vez e estarem aptos a executá-las em qualquer lugar que forneça suporte à Java [GOS 96]. Para executar uma aplicação Java em uma determinada plataforma de *hardware* ou *software*, deve existir um ambiente de execução Java para esta plataforma. Desta forma, em vez de portar cada aplicação, somente é necessário portar o ambiente de execução. Uma vez que o ambiente de execução foi portado, pode-se executar inúmeras aplicações sem qualquer alteração em seu código.

Além de facilitar o porte de aplicações, Java também facilita seu desenvolvimento. Ao usar Java, as aplicações para sistemas embutidos podem ser desenvolvidas sem levar em conta a plataforma onde serão executadas. Desta forma, o porte de aplicações de um produto para outro é feita de forma mais fácil, uma vez que o programador não precisa conhecer muitos detalhes do *hardware* onde seu sistema será executado. A existência de um grande número de bibliotecas de classes Java é outra característica que permite o desenvolvimento rápido de aplicações.

Nestes últimos anos, Java tornou-se uma das mais populares linguagens de programação orientada a objetos [GAG 02]. Este desejo de poder utilizar a plataforma Java em sistemas embutidos é especialmente alimentado pela característica *"write once, run everywhere"* tão divulgada pelos seus criadores [BOT 97]. Java vem sendo usada

nos mais variados contextos, sendo até citada como uma linguagem de programação que poderia ser usada em processamento de alto desempenho [CAR 97, BAR 02]. Entretanto, por falta de compiladores, ambientes de execução e bibliotecas eficientes, o uso de Java neste tipo de aplicação ainda não é realidade, sua utilização é mais comum em aplicações para estações de trabalho e ambientes de rede.

Para permitir a portabilidade de código, a compilação de Java normalmente é feita para um conjunto de instruções de um computador abstrato. O código binário contendo este conjunto de instruções normalmente é chamado *bytecode* Java [LIN 99]. Desta forma, para executar uma aplicação Java normalmente é necessário um interpretador, que faz o papel do processador do computador abstrato. A interpretação faz com que aplicações Java apresentem um desempenho inferior a aplicações escritas em linguagens tradicionais e compiladas diretamente para o código nativo da plataforma de *hardware* onde são executadas. Além disso, os ambientes de execução tradicionais não são adequados a sistemas embutidos. Estes ambientes consomem mais recursos que os sistemas embutidos tradicionalmente possuem.

Para permitir a execução de Java em sistemas embutidos, a SUN[2] desenvolveu uma plataforma para ambientes de execução Java específicos chamada *Java 2 Micro Edition* (J2ME). Estes ambientes de execução tem por objetivo ser plataformas Java mínimas de forma a permitir o uso de Java em dispositivos com restrições de recursos de *hardware* [Sun 00]. Entretanto, mesmo a plataforma J2ME exige mais memória e poder de processamento que os sistema embutidos típicos possuem. Deste modo, a maioria dos sistemas embutidos não possui a possibilidade de executarem tal plataforma [BEU 00].

Além de J2ME, existem vários ambientes de execução voltados ao segmento de sistemas embutidos. Ambientes de execução para sistemas embutidos normalmente são configuráveis e muitas vezes impõem restrições às aplicações como forma de diminuir a utilização de recursos. A configuração destes ambientes de execução é feita de forma a refletirem as características do *hardware* e necessidades das aplicações. Normalmente, as necessidades das aplicações são configuradas manualmente pelo projetista do sistema. Entretanto, muitas vezes o projetista do sistema não conhece intimamente

[2]Sun Microsystems é a empresa que criou Java e detém direitos sobre esta marca.

a aplicação a ponto de determinar quais são as características do ambiente de execução que devem ser habilitados ou desabilitados. Em outros casos, o projetista do sistema não conhece com muita profundidade o projeto do ambiente de execução. Isto pode fazer com que seja realizada uma configuração do ambiente de execução que não é ideal para a aplicação. Deste modo, uma configuração feita manualmente pelo projetista do sistema pode fazer com que o ambiente de execução seja construído sem garantir todas as necessidades da aplicação. Outra possibilidade é o sistema resultante da configuração manual contendo características que não são necessárias, desperdiçando-se valiosos recursos.

Para evitar este tipo de problema, pode-se diminuir a responsabilidade do projetista em determinar as necessidades da aplicação. Um programa pode determinar parte destas necessidades através de análise do código da aplicação.

Um ambiente de execução Java para sistemas embutidos, que seja configurável e adequado às necessidades da aplicação pode ser construído seguindo-se as técnicas de *Application-Oriented System Design* (AOSD) [FRö 01]. Este ambiente pode ser feito com base nos princípios da *Application-Oriented System Design* [FRö 01]. Existe um sistema operacional chamado EPOS [FRö 99] que tem por objetivo suportar aplicações embutidas e paralelas. Este sistema operacional segue os princípios da AOSD. Isto é, um sistema sob medida é construído para dar suporte a uma determinada aplicação ou grupo de aplicações. Desta forma, a aplicação do sistema embutido tem máximo aproveitamento do *hardware* disponível. Com um suporte do EPOS e com bibliotecas Java especialmente adaptadas às características deste sistema, pode-se obter um ambiente de execução sob medida para uma determinada aplicação Java.

Para utilizar este ambiente de execução, deve-se fazer uma análise das necessidades da aplicação Java. Ferramentas com este propósito são fáceis de implementar uma vez que o arquivo `class` (ou seja, o arquivo que contém o os bytecodes Java) mantém informações suficientes, por exemplo, para permitir reconstruir o código fonte. Além disso, existem bibliotecas de classes que permitem a manipulação de classes que podem ser utilizadas para fazer estas ferramentas.

Java normalmente é associado a desempenho pobre. A fama de ter baixo desempenho se deve principalmente às primeiras máquinas virtuais Java, que eram sim-

plesmente interpretadores do *bytecode* Java, sendo executados sobre sistemas hospedeiros. Atualmente existem técnicas que permitem transformar aplicações Java para o código nativo da máquina onde a aplicação é executada. Para evitar perda de desempenho com a interpretação do código Java, pode-se fazer a utilização de compiladores *Ahead-of-Time*. Este tipo de compilador transforma programas Java em código nativo da plataforma onde é executado. Isto pode ser feito tendo como origem tanto o código fonte da aplicação como o *bytecode* Java. Com isso, o sistema perde a capacidade de utilizar o mesmo binário para plataformas diferentes. Mesmo com código compilado, o uso de Java continua sendo interessante devido ao grande número de aplicações e bibliotecas feitas em Java. O programador pode continuar pensando que seu código será executado em uma máquina virtual Java tradicional pois a transformação de código Java em código nativo será feita de forma automática e com a menor intervenção possível do programador.

1.1 Motivação e Objetivos

Um ambiente de execução Java ideal deveria ser capaz de fornecer suporte à execução de qualquer aplicação Java com mínima utilização de recursos, com desempenho máximo e sem sacrificar portabilidade e funcionalidade. Infelizmente, algumas destas características são conflitantes e deve-se ponderar sobre quais devem ser evidenciadas no momento de criar um ambiente de execução.

Tomando como ponto de referência uma aplicação ou conjunto de aplicações Java, pode-se tentar selecionar apenas a funcionalidade que estas aplicações necessitam. Desta forma, pode-se diminuir a utilização de recursos eliminando-se as funcionalidades não utilizadas. Um ambiente de execução específico poderia ser construído para esta aplicação, ou conjunto de aplicações. Entretanto, a possibilidade de executar qualquer aplicação seria sacrificada, uma vez que o ambiente de execução seria específico para determinado conjunto de aplicações. Este sacrifício não é um problema no domínio de aplicações para sistemas dedicados ou embutidos.

O objetivo deste trabalho é criar um ambiente de execução Java que seja configurado de acordo com as características necessárias a uma aplicação ou conjunto

de aplicações. Além disso, levando-se em conta as restrições de recursos existentes no mundo de sistemas embutidos, o ambiente de execução deve ser o mais enxuto possível. Isto é, o ambiente de execução deve prover todas as necessidades da aplicação, sem prover características que não sejam necessárias.

1.2 Organização do Texto

No capítulo 2 são relembrados os principais aspectos sobre Java que devem ser levados em conta durante a construção de um ambiente de execução. No capítulo 3 são abordados tópicos a respeito da utilização da linguagem Java em sistemas embutidos. A seguir, no capítulo 4, é apresentada uma metodologia de desenvolvimento de sistemas operacionais a partir das necessidades impostas pela aplicação. Esta metodologia será usada na determinação do ambiente de execução para uma dada aplicação escrita em Java. Logo após, no capítulo 5, é apresentado o ambiente de execução para aplicações escritas em Java proposto neste trabalho. Finalmente são apresentadas as conclusões obtidas e próximos passos a seguir.

Capítulo 2

Java

Java, cujo nome inicial era OAK, foi originalmente projetada para uso em aplicações embutidas em dispositivos eletrônicos [GOS 96]. O termo *Java* não é somente aplicado a uma linguagem de programação, é também usado para referenciar uma API[1] e um ambiente de execução [BAR 02]. As aplicações Java são escritas na linguagem Java e compiladas para um formato de classe binário, que é independente de plataforma de *hardware*. A API Java é constituída por um conjunto de classes predefinidas. Qualquer implementação da plataforma Java deve garantir o suporte à linguagem de programação, ao ambiente de execução e à API [LIA 99].

A linguagem de programação Java é orientada a objetos, baseada em classes, que permite programação concorrente e que tem características necessárias para uso geral. A linguagem Java, juntamente com sua biblioteca padrão e seu ambiente de execução fornecem independência de plataforma e segurança. Esta linguagem foi projetada com atenção especial para ter o mínimo de dependências de implementação possível [GOS 96]. A independência de plataforma é obtida através da compilação dos fontes de Java para um conjunto de instruções de um computador abstrato, chamado Máquina Virtual Java ou por sua sigla em inglês (JVM). Desta forma, permite aos desenvolvedores de aplicações escrever um programa uma única vez e poder executar o programa em qualquer lugar [GOS 96]. O código binário das instruções da JVM é chamado *byte-*

[1] *Application Program Interface*

code [LIN 99]. O código *bytecode* dos métodos e outras informações sobre a classe são armazenados em uma estrutura chamada arquivo `class`.

O ambiente de execução, na maioria das vezes, é uma máquina virtual [BOT 97]. Isto é, o *bytecode* Java é interpretado por um programa que está sendo executado sobre a plataforma real. Entretanto, existem casos onde o *bytecode* Java funciona como código de máquina e é interpretado diretamente pelo *hardware*. Este é o caso do *chip* chamado JAVACARD [Sun 04]. Também existem casos onde o *bytecode* Java é convertido para código nativo antes de sua execução. O sistema Toba [PRO 97] e GNU GCJ [Ope 03a] são exemplos de plataformas que permitem a transformação de aplicações Java para código nativo. Em todos estes casos, existe a necessidade de um ambiente propício que forneça suporte à execução das aplicações Java.

Neste capítulo são apresentados os conceitos básicos relacionados à linguagem Java, ao seu ambiente de execução e à sua API. Na seção 2.1, é apresentada a linguagem Java, descrevendo suas estruturas como classes, interfaces, exceções, etc. A seção 2.2 apresenta a API Java e algumas de suas implementações. Na seção 2.3 é apresentada a máquina virtual Java, ambiente de execução utilizado para execução de programas Java. Nesta mesma seção também é apresentada a maneira como programas Java são integrados com aplicações escritas em C e C++: métodos nativos.

2.1 Linguagem

Java é uma linguagem de programação orientada a objetos. Sua sintaxe é similar a C e C++, mas ela omite várias características destas linguagens. Java é uma linguagem de alto-nível e não permite acesso a muitos detalhes a respeito da representação lógica e física do ambiente de execução. Java inclui gerenciamento automático da memória, tipicamente com a utilização de coletores de lixo. Este gerenciamento existe para evitar problemas de segurança relacionados com liberação explícita de memória[2].

A seguir são apresentados conceitos da linguagem Java em relação a

[2]Tais como acesso a regiões de memória que já tenham sido liberadas com o uso de `free` em C ou `delete` em C++.

tipos, pacotes, classes e interfaces. Logo após são apresentadas características especiais da linguagem tais como o uso de *arrays*, exceções e programação concorrente.

2.1.1 Tipos

Uma das características que torna Java segura é que esta linguagem é fortemente tipada. Isto significa que cada variável e cada expressão tem um tipo que é conhecido em tempo de compilação. Os tipos limitam os valores que uma variável pode armazenar ou que uma expressão pode produzir, limitam as operações suportadas nestes valores, e determinam o significado das operações. Isto ajuda a detectar possíveis erros em tempo de compilação [GOS 96].

Os tipos da linguagem Java são divididos em duas categorias: tipos primitivos e tipos referência. Os tipos primitivos são os tipos numéricos e o tipo `boolean`. Os tipos referência são classes, interfaces e arrays. Existe também um tipo especial `null`, que não pode ser usado para declarar uma variável ou fazer *typecast*. Entretanto qualquer variável do tipo referência pode armazenar o valor `null`.

Um objeto em Java é uma instância dinamicamente criada de uma classe ou array[3]. Os valores possíveis para uma variável do tipo referência são referências a objetos. Todos os objetos, incluindo arrays, suportam os métodos da classe `java.lang.Object` [GOS 96].

O tipo de um objeto pode ser verificado em tempo de execução. Para verificar o tipo de objeto pode-se utilizar o operador `instanceof`. Isto permite, por exemplo, determinar o tipo de um objeto armazenado em uma determinada posição de um array de `java.lang.Object`.

A seguir são apresentados os pacotes, estrutura hierárquica utilizada para restringir conflitos de nomes de tipos. Logo após são descritas as classes e interfaces em Java.

[3]Diferentemente de C e C++, onde arrays são conjuntos de estruturas organizadas linearmente na memória, em Java cada array é uma instância de uma classe criada dinamicamente. A seção 2.1.2 descreve arrays em Java.

Pacotes

Os programas em Java são organizados como conjuntos de pacotes. Um pacote pode conter classes, interfaces e sub-pacotes. A estrutura de nomes de pacotes é hierárquica. Cada pacote tem seu próprio conjunto de nomes para tipos, o que ajuda a prevenir conflitos. Um tipo é acessível fora do pacote que o declara somente se ele é declarado como público.

O conjunto de pacotes disponíveis ao programa Java é definido pelo ambiente de execução no qual está sendo executado. Entretanto, este sistema deve sempre incluir pelo menos os três pacotes padrões: `java.lang`, `java.util` e `java.io` [GOS 96].

Classes

Declarações de classe definem novos tipos referência e descrevem como eles são implementados. O nome de uma classe tem como escopo todas as declarações de tipo no pacote no qual a classe é declarada. Se uma classe é declarada como pública, ela pode ser referenciada em outros pacotes.

Uma classe pode ser abstrata e pode ser declarada abstrata caso ela seja incompletamente implementada. Este tipo de classe não pode ser instanciada, mas pode ser estendida por subclasses.

A classe `java.lang.Object` é especial e não possui uma superclasse. Todas as demais classes são subclasses de alguma classe. Entretanto cada classe só pode ter uma única superclasse. Se uma classe não declara sua superclasse, é considerada uma especialização de `java.lang.Object`. Todas as classes podem implementar interfaces.

O corpo de uma classe declara membros (atributos e métodos), inicializadores estáticos e construtores. Inicializadores estáticos são blocos de código executável que podem ser usados para ajudar a inicializar uma classe quando esta é carregada. Declarações de atributos, métodos e construtores podem incluir os modificadores de acesso `public`, `protected` ou `private`. Estes modificadores permitem restringir o acesso aos membros da classe. Além disso, os atributos e campos de uma superclasse podem ser

sobrecarregados.

Interfaces

Diferentemente de C++, a linguagem Java possui tipos específicos para representar o conceito de interface de uma classe. Uma declaração de interface cria um novo tipo referência cujos membros são constantes e métodos abstratos. Este tipo não tem implementação, mas outras classes podem fornecer implementações para seus métodos abstratos.

```
class Voador {
        public:
                virtual void voe(Local para) = 0;
};

class Passaro : public Animal, public Voador {
        public:
                void voe(Local para) { ... };
};

class Aviao : public Veiculo, public Voador {
        public:
                void voe(Local para) { ... };
};
```

Exemplo 2.1: Utilização do conceito de interfaces em C++.

No caso da linguagem C++, uma determinada classe declara tanto uma interface quanto uma superclasse da mesma maneira. De fato, a maneira utilizada para definir uma interface é igual à maneira utilizada para definir uma classe abstrata. O Exemplo 2.1 mostra a implementação em C++ da interface `Voador` e de duas classes que a implementam. Neste exemplo, caso o código da interface não estivesse presente, poderia-se interpretar incorretamente a classe `Passaro` como sendo um exemplo de `Voador`

que tem como característica ser `Animal`.

```
public interface Voador {
        public void voe(Local para);
}

public class Passaro extends Animal implements Voador {
        public void voe(Local para) { ... };
}

public class Aviao extends Veiculo implements Voador {
        public void voe(Local para) { ... };
}
```

Exemplo 2.2: Utilização de interfaces em Java.

A maneira que Java implementa interfaces possibilita diferenciar claramente no código de uma determinada classe a sua superclasse das interfaces por ela implementadas. No Exemplo 2.2 é fácil identificar que `Passaro` é um exemplo de `Animal` que tem como característica ser `Voador`. Esta possibilidade de identificar claramente classes de interfaces também está presente na estrutura binária da classe, o arquivo `class`. Esta característica permite que sejam feitas otimizações no código de uma aplicação. Como um exemplo, pode-se de eliminar do código de uma aplicação interfaces implementadas por uma única classe.

Ao contrário das classes, uma interface pode ser declarada como sendo uma extensão direta de uma ou mais interfaces, fazendo assim com que ela especifique implicitamente todos os métodos abstratos e constantes das interfaces que estende.

Uma classe pode ser declarada de forma a implementar uma o mais interfaces. Qualquer instância de uma classe implementa todos os métodos especificados pelas interfaces que a classe declara implementar. A herança múltipla nas interfaces permite aos objetos suportarem múltiplos comportamentos comuns sem compartilharem qualquer implementação [GOS 96].

Uma variável cujo tipo declarado é uma interface pode ter como valor uma referência a qualquer objeto que seja instância de uma classe que implementa esta interface. Entretanto, para que uma classe implemente uma interface, *não* é suficiente que a classe implemente todos os métodos declarados na interface. Para isto, é necessário que a classe, ou uma superclasse sua, declare implementar esta interface através da palavra reservada `implements`.

2.1.2 Características Especiais

Para que um ambiente de execução Java seja construído, algumas características da linguagem Java devem ser levadas em conta. Um ambiente de execução deve tratar estas características de forma especial. Destas características pode-se destacar a implementação de Arrays, Exceções e Programação Concorrente.

A seguir são apresentadas características especiais da linguagem Java. Inicialmente é apresenta a forma como os arrays são tratados. Logo após, é apresentado o mecanismo de exceções da linguagem. Finalmente são mostrados aspectos da linguagem relativos à programação concorrente: *threads* e *locks*.

Arrays

Diferentemente de C e C++, onde arrays são conjuntos de objetos organizados linearmente na memória, em Java os *arrays* são objetos que são criados dinamicamente. Sendo assim, eles podem ser atribuídos a variáveis do tipo `java.lang.Object` e todos os métodos da classe `java.lang.Object` podem ser invocados em um array. Além de métodos e atributos da classe `java.lang.Object`, os arrays ainda possuem o atributo público chamado `length`, que contém o número de elementos do array.

Todos os componentes de um array tem um mesmo tipo. Entretanto, caso o tipo dos componentes de um array seja `java.lang.Object`, ele pode conter qualquer tipo de objeto, incluindo outros arrays. O tipo do componente de um array pode ser um tipo array. Desta forma, os componentes de um array podem conter referências a subarrays.

Threads e Locks

Todas as implementações de ambiente de execução Java devem suportar múltipas *threads*[4] executadas simultaneamente. Estas *threads* independentemente executam código que age sobre valores e objetos residentes em uma memória compartilhada. *Threads* podem ser suportadas com vários processadores, compartilhando um único processador ou compartilhando vários processadores. Em Java, a utilização de *threads* é feita através das classes `java.lang.Thread` e `java.lang.ThreadGroup` [GOS 96].

Para garantir o comportamento determinístico de programas que usem várias *threads*, a linguagem de programação Java provê mecanismos de sincronização. Para prover esta sincronização, são usados monitores [HOA 74]. O comportamento dos monitores é implementado através de *locks*. Em Java, existe um *lock* associado a cada objeto instanciado.

Para permitir concorrência, métodos ou trechos de código devem ser declarados como exclusivos através da palavra reservada `synchronized`. A declaração `synchronized` causa duas ações:

- Depois de calcular a referência a um objeto e antes de executar seu corpo o *lock* associado ao objeto é trancado;
- Depois da execução do corpo ter sido completada, tanto normalmente como de forma abrupta por ocorrência de uma exceção, o *lock* associado ao objeto é liberado.

Entretanto, as ações executadas em um bloco `synchronized` são relevantes apenas em operações com várias *threads*. No caso de haver um único fluxo de execução, estas ações continuam sendo executadas, mas não têm qualquer efeito sobre o funcionamento do programa.

Todos os objetos instanciados, além de um *lock*, possuem uma fila de espera. Esta fila pode ser acessada através dos métodos `wait()`, `notify()` e `notifyAll()` da classe `java.lang.Object`. Estes métodos permitem uma transferência de controle de execução de uma *thread* para outra. Desta forma uma *thread* pode suspender-se

[4] Fluxos de execução

usando `wait()` até o momento em que outra *thread* a acorda usando `notify()` ou `notifyAll()` [GOS 96].

A existência de *locks* em todos os objetos instanciados no sistema e o uso de métodos ou blocos sincronizados em aplicações com apenas um único fluxo de execução fazem com que recursos dos sistema, como memória e processamento, sejam gastos sem necessidade. O desperdício de recursos com sincronização é freqüentemente alvo de pesquisas. Bogda e Hölzle [BOG 99] conseguiram aumentar o desempenho de aplicações em até 36% ao eliminar sincronizações desnecessárias de aplicações Java.

Exceções

Quando um programa Java viola a semântica da linguagem Java, o ambiente de execução sinaliza este erro ao programa como uma exceção. Um exemplo disso ocorre quando existe uma tentativa de acessar um índice fora do escopo de um array.

Programas em Java também podem lançar exceções explicitamente. Isto pode ser feito com o uso da declaração `throw`. Este mecanismo fornece uma alternativa à prática de reportar erros retornando valores como `-1` onde valores negativos não são esperados.

Cada exceção é representada por um objeto da classe `Throwable` ou uma de suas subclasses. Este objeto pode ser usado para carregar informação do ponto no qual a exceção ocorreu para o tratador que a captura.

O mecanismo de exceções da linguagem Java é integrado com o modelo de sincronismo. Desta forma, *locks* são liberados quando existem exceções na execução de trechos de código ou de métodos declarados como `synchronized`.

2.2 API

Um dos motivos de Java ser popular está ligado ao uso de bibliotecas de classe com API padronizada. Pacotes como `java.lang` e `java.io` são usados para permitir o uso de recursos do sistema sem ter conhecimento de como estes recursos são

implementados na plataforma onde a aplicação é executada. Além de garantir compatibilidade e transparência aos programas Java, a API padronizada também fornece uma série de componentes e algoritmos de uso geral. Dentre os componentes de uso geral, pode-se destacar pacotes gráficos como `java.awt` e `javax.swing` e o pacote com utilitários para manipulação e armazenamento de dados (`java.util`). A API padrão de Java é especificada pela SUN e pela comunidade envolvida com a linguagem.

De acordo com a especificação da API Java, três pacotes devem existir em todas as bibliotecas que a implementem: `java.lang`, `java.util` e `java.io`.

Existem várias implementações de bibliotecas com esta API padronizada. Entre estas implementações, pode-se destacar a biblioteca de classes desenvolvida pela SUN e o projeto da *Free Software Fundation* que almeja construir uma biblioteca Java de código aberto, o GNU Classpath.

2.2.1 Bibliotecas de Classes

Existem várias implementações de bibliotecas de classes que seguem a API Java. Algumas das classes definidas nesta API são dependentes do ambiente de execução [GOS 96]. Esta dependência faz com que a maioria dos ambiente de execução Java implementados também tenham uma biblioteca de classes próprias.

A seguir são apresentadas duas bibliotecas de classes que implementam a API Java: a biblioteca desenvolvida pela SUN e a biblioteca GNU Classpath.

J2SE

Para cada versão do *Java 2 Standard Edition* (J2SE), existe uma versão das bibliotecas de classes. O desenvolvimento destas bibliotecas de classes, assim como a evolução da API Java, é feito com ajuda da comunidade envolvida com Java. Esta biblioteca, que é fornecida pela SUN, contém a implementação referência de todas as classes definidas na API. A biblioteca do J2SE que implementa a API Java é a mais conhecida e mais utilizada. Entretanto, por restrições em sua licença de uso, não pode ser utilizada por qualquer ambiente de execução Java.

GNU Classpath

GNU Classpath é um projeto da *Free Software Foundation* que almeja desenvolver um conjunto de bibliotecas essenciais à linguagem Java. As bibliotecas desenvolvidas pelo projeto são utilizadas em substituição às bibliotecas de classes da SUN. O motivo que levou à necessidade de desenvolver uma nova biblioteca de classes está associado à licença de uso da biblioteca de classes da SUN. Esta licença é muito restritiva segundo o ponto de vista dos projetos de Software Livre. A licença de uso adotada pela biblioteca GNU Classpath permite sua livre utilização e distribuição, desde que seu código fonte continue sendo aberto. Por este motivo, a biblioteca GNU Classpath pode ser usada em projetos de ambientes de execução com código aberto tais como Kaffe [Ope 03c], ORP [CIE 02] e SableVM [GAG 00].

O estágio de desenvolvimento atual da biblioteca permite a execução de grande parte das aplicações que utilizam bibliotecas Java compatíveis com as especificações 1.1 e 1.2 da API Java. Atualmente existe um esforço para unificar as implementações da GNU Classpath com a biblioteca Java usada pelo GNU GCJ, mas esta unificação não está completa [Ope 03b].

Como algumas classes têm dependência da máquina virtual, a biblioteca GNU Classpath possui "ganchos" para permitir que as máquinas virtuais posam ser implementadas sem alterar o código original da GNU Classpath. Estes "ganchos" são constituídos por classes abstratas e métodos nativos.

2.3 Máquina Virtual

A Máquina Virtual Java é um computador abstrato. Como um computador real, ela tem um conjunto de instruções e áreas de memória no momento de sua execução. Entretanto, a JVM não pressupõe uma determinada arquitetura de *hardware* ou sistema operacional onde será executada [LIN 99].

A máquina virtual Java é responsável pela interpretação de código Java compilado (ou *bytecode*). Ela pode ser construída em cima de um sistema pré-existente,

como uma aplicação para um determinado sistema, ou como todo um sistema novo, resolvendo questões inerentes ao sistema como *threads*, E/S, etc.

A maioria das máquinas virtuais existentes é construída como uma aplicação a ser executada em um determinado sistema. Existem, porém, casos onde a máquina virtual pode ser o próprio sistema. Este é o caso dos sistemas JX [GOL 02] e JAVACARD [Sun 04] que são executados diretamente sobre o *hardware*.

Para executar um programa escrito em Java é necessário um ambiente adequado. Este ambiente normalmente é uma máquina virtual. Esta máquina virtual tem a capacidade de interpretar o *bytecode* Java e de executar o código associado a estas instruções no processador nativo.

Programas escritos em Java são geralmente compilados para um formato binário chamado *bytecode*. Cada classe é representada por um único arquivo que contém os dados relativos a esta classe, assim como as instruções em *bytecode* de seus métodos, construtores e inicializador de classe. Estes arquivos são carregados dinamicamente em um interpretador e executados. Este interpretador é a parte de uma ambiente de execução responsável por emular o comportamento de um processador que execute *bytecodes* Java.

O ambiente de execução Java deve fornecer as características necessárias às aplicações desenvolvidas em Java. Normalmente estas características incluem:

- Um carregador de classes (*ClassLoader*), responsável por carregar, verificar a integridade e inicializar as classes usadas pela aplicação;
- Um coletor de lixo (*Garbage Collector*), responsável pela procura e liberação de memória que não está mais sendo utilizada pela aplicação;
- Um ambiente para a execução das aplicações, que pode possuir:
  - Um interpretador, responsável por ler as instruções em *bytecode* e executar um código associado a cada instrução;
  - Um conjunto composto por interpretador e compilador *Just-In-Time*, responsável por traduzir o *bytecode* em código nativo no momento de sua execução.

Fazendo assim com que o programa seja executado nativamente, mas só fazendo esta transformação no momento em que o trecho de código é necessário;

  - Um compilador que transforme *bytecode* em código nativo antes da execução da aplicação e um conjunto bibliotecas. Estas bibliotecas devem dar suporte às necessidades da aplicação transformada para código nativo.

- Áreas de memória para armazenar vários tipos de dados:

  - Uma pilha de execução para cada *thread*;
  - Uma área para *Heap* por máquina virtual. Esta área é usada para armazenar todas as instâncias de classes e arrays e é compartilhada entre todas as *threads*;
  - Uma área para manter o código dos métodos. Existe uma única área de métodos por máquina virtual que é compartilhada por todas as *threads*;
  - Uma área para manter constantes a serem usadas em tempo de execução. Existe uma área de constantes para cada classe ou objeto instanciado;
  - Uma pilha de execução para código nativo para cada *thread*;
  - Uma área de porções de memória usadas como memória temporária. Estas áreas são criadas a cada invocação de método e são destruídas assim que o método termine;

- Uma unidade aritmética de ponto flutuante.

A seguir são apresentadas as características de ambientes de execução Java que devem ser analisadas ao se desenvolver um novo ambientes de execução. Inicialmente é apresentado um mecanismo utilizado para melhorar o desempenho das aplicações Java, os compiladores *Just-In-Time*. Logo a seguir são apresentadas as classes da API Java que são dependentes da implementação do ambiente de execução. Finalmente são apresentados métodos nativos. Este é o mecanismo utilizado para integração de aplicações Java com aplicações nativas ao sistema hospedeiro.

2.3.1 Compiladores *Just-In-Time*

Originalmente, os ambientes de suporte a execução de programas escritos em Java eram constituídos totalmente por interpretadores de *bytecode* [GAG 02]. Para executar uma aplicação escrita em Java era necessário carregar o interpretador na memória, carregar o código da aplicação e interpretar instrução a instrução todo o código da aplicação. Desta forma, as aplicações Java tinham um desempenho muito inferior se comparado às aplicações executadas pelo *hardware*.

A principal forma de melhorar o desempenho de programas Java nos dias de hoje é a utilização de compiladores *Just-In-Time* (JIT) nas máquinas virtuais. A técnica consiste em traduzir o código Java, *bytecode*, para código nativo em tempo de execução. Para isto, compiladores específicos para esse fim ficam permanentemente aguardando a requisição para a transformação de um novo método. Os métodos traduzidos são normalmente armazenados em memória para serem utilizados posteriormente sem a necessidade de nova tradução. Isto faz com que a necessidade de memória aumente, uma vez que tanto o compilador quanto os métodos traduzidos ocupam memória extra durante o processamento do programa. Em contrapartida, o desempenho das aplicações é melhorado uma vez que o código que já foi traduzido é executado diretamente ao invés de ser interpretado como nas máquinas virtuais sem compiladores *Just-In-Time*.

2.3.2 Suporte à Biblioteca de Classes

A máquina virtual Java deve fornecer suporte suficiente para a implementação das bibliotecas de classes da plataforma. Algumas das classes destas bibliotecas não podem ser implementadas sem a cooperação da máquina virtual Java [GOS 96]. As classes que se encaixam neste perfil contém métodos nativos cujas implementações devem estar disponíveis em tempo de execução.

Classes que necessitam de suporte especial da máquina virtual Java incluem aquelas que tem:

- Reflexão, como as classes do pacote `java.lang.reflect` e a classe `Class`;

- Carga ou criação de classe ou interface, como no caso da classe `ClassLoader`;
- Ligação e inicialização de uma classe ou interface, também como a classe `ClassLoader`;
- Segurança, como as classes do pacote `java.security` e outras classes como `SecurityManager`.
- Multithreading, como a classe `Thread`.
- Referências fracas, como as classes do pacote `java.lang.ref`.

2.3.3 Métodos Nativos

Programas escritos em Java tradicionalmente são executados por máquinas virtuais que são responsáveis pela interpretação e execução do *bytecode* contido nas classes da aplicação. Em muitos casos, o ambiente de execução Java usa métodos nativos para fornecer suporte a várias operações necessárias à aplicação. Nas implementações de ambientes de execução Java, grande parte das funções de E/S e funções dependentes do sistema tem como base métodos nativos.

Além disso, métodos nativos são a maneira que Java utiliza para que sejam feitas integrações com códigos desenvolvidos por outras linguagens de programação. Nestes casos é necessário uma integração entre a aplicação, escrito em *bytecode*, e algum código desenvolvido especificamente para o sistema hospedeiro onde a aplicação está sendo executada.

A utilização de métodos nativos também pode ser realizada para se obter um ganho de desempenho, entretanto esta não é sua principal função. Na verdade, o uso excessivo de métodos nativos pode diminuir o desempenho de programas Java, uma vez que a chamada a esses métodos é bastante custosa [Ope 03b]. Se o objetivo for ganho de desempenho, deve-se optar por outras alternativas como compiladores *Just-In-Time* ou otimização do código através de outras ferramentas.

Em Java, os métodos nativos só determinam a assinatura do método. Sua implementação deve ser feita através de uma linguagem de programação tradicional que esteja disponível no sistema hospedeiro onde a aplicação Java será executada. O

suporte a métodos nativos normalmente possibilita que o código destes métodos possa ser implementado em C e C++. O Exemplo 2.3 mostra a declaração de uma classe que possui métodos nativos. A palavra reservada `native` é usada para definir se um método é nativo.

```
public class ClassWithNaviteMethod {
        public native int showString(String arg);
        public native String useMethod();
        public static native ClassWithNaviteMethod newObject();

        public static void main(String args[]) {
                ClassWithNaviteMethod obj;
                obj = ClassWithNaviteMethod.newObject();
                obj.showString(obj.useMethod());
        }
}
```

Exemplo 2.3: Classe em Java com métodos nativos.

O ambiente de execução Java necessita ter acesso ao código dos métodos nativos para que ele possa ser executado. Este acesso normalmente é feito através da utilização de um mecanismo de bibliotecas dinâmicas. Desta forma, o código nativo da implementação dos métodos nativos deve estar contido em uma biblioteca que possa ser carregada. Esta biblioteca deve ser carregada pela aplicação Java em tempo de execução, através do método *System.LoadLibrary(String)*. Outra possibilidade para permitir o acesso ao código dos métodos nativos é ligá-los diretamente ao código do ambiente de execução. Neste caso, o uso do método *LoadLibrary* não é necessário pois o código passa a estar disponível desde a inicialização do ambiente de execução.

A seguir serão apresentados três tipos de métodos nativos. Inicialmente é apresentado o JNI, tipo de método nativo mais utilizado. Em seguida é apresentado o método KNI, uma versão do JNI adaptada para a máquina virtual KVM. Finalmente é apresentado o método CNI, desenvolvido pela Cygnus Solutions para ser utilizado com

os compiladores do pacote GNU GCC.

JNI – Java Native Interface

Java Native Interface (JNI), definida pela SUN em [LIA 99], é o método mais conhecido pois foi desenvolvido e é divulgado pelos criadores da linguagem Java. O JNI é constituído por um conjunto de funções em C e C++ e determinadas regras de utilização.

O Exemplo 2.4 mostra a implementação dos métodos nativos da classe apresentada no Exemplo 2.3 usando JNI. Note que, são necessárias várias chamadas a funções da API JNI para que um método nativo possa criar um objeto ou fazer uma chamada a um dos métodos da classe a qual pertence. Para criar um objeto no método `newObject`, por exemplo, é necessário primeiro obter a identificação do construtor *default* da classe. Somente depois o objeto é criado a partir das identificações da classe, recebida como parâmetro do método, e do identificador do construtor. Note ainda que o código faz referência ao arquivo `ClassWithNativeMethod.h`. Este arquivo é criado automaticamente a partir do código binário da classe. No caso do JNI, o arquivo é criado com o auxílio da ferramenta `javah`, distribuída juntamente com o pacote de desenvolvimento Java da SUN.

KNI – K Native Interface

O uso de JNI em ambientes de execução voltados a sistema embutidos é inviável. Essa nova implementação de interface para métodos nativos, K Native Interface, é necessária pois a implementação habitual, JNI, requer muitos recursos para ser utilizada [Sun 02b]. De acordo com [Sun 00], a implementação de JNI aumentaria muito o tamanho da máquina virtual KVM. Por este motivo, a SUN desenvolveu uma nova forma de fazer integração entre aplicações Java e aplicações nativas voltada a dispositivos com restrições de memória.

```
#include "ClassWithNaviteMethod.h"
#include <iostream>

JNIEXPORT jint JNICALL Java_ClassWithNaviteMethod_showString
  (JNIEnv *env, jobject obj, jstring str) {
        const char *tmp = env->GetStringUTFChars(str,NULL);
        std::cout << tmp << std::endl;
        env->ReleaseStringUTFChars(str,tmp);
        return 1;
}

JNIEXPORT jstring JNICALL Java_ClassWithNaviteMethod_useMethod
  (JNIEnv *env, jobject obj) {
        jclass classobj = env->GetObjectClass(obj);
        jmethodID mId = env->GetMethodID(classobj,
                        "toString","()Ljava/lang/String;");
        return (jstring) env->CallObjectMethod(obj,mId);
}

JNIEXPORT jobject JNICALL Java_ClassWithNaviteMethod_newObject
  (JNIEnv *env, jclass classobj) {
        jmethodID mId = env->GetMethodID(classobj,
                        "<init>","()V");
        return env->NewObject(classobj,mId);
}
```

Exemplo 2.4: Implementação dos métodos nativos da classe apresentada no Exemplo 2.3 usando JNI.

CNI – Cygnus Native Interface

Além do JNI e KNI, desenvolvidos pela SUN existe também o *Cygnus Native Interface* – CNI [Cyg 99], desenvolvido pela Cygnus Solutions para o compilador Java do GNU Compiler Collection[5].

O CNI é uma técnica mais natural de escrever métodos nativos Java usando C++. Entretanto, é uma alternativa menos portável que a padrão JNI. É fortemente tipado e não permite utilização de estruturas exclusivas de C/C++. Para evitar esta limitação existe uma classe, `gnu.gcj.RawData`, que pode ser usada como qualquer tipo de dados. Em outras palavras, variáveis declaradas com o tipo `RawData` podem conter qualquer tipo de dados e não são verificadas pelo compilador de nenhuma forma.

O grupo de desenvolvedores do GCJ prefere CNI segundo a alegação que ele é mais eficiente, fácil de escrever e fácil de depurar. Eles usam CNI pois o acham a melhor solução, especialmente para uma implementação de Java que é baseada na idéia que Java é apenas outra linguagem de programação que pode ser implementada usando técnicas padrões de compilação. Posto isso, e tendo em vista a idéia que linguagens implementadas usando GCC devem ser compatíveis[6], chega-se à conclusão que a convenção de chamadas de métodos e passagem de parâmetros de Java deve ser tão similar quanto possível ao usado por outras linguagens, especialmente C++, uma vez que pode-se pensar Java como um subconjunto desta linguagem. CNI é apenas um conjunto de funções e convenções que ajudam na idéia que C++ e Java têm a *mesma* convenção de chamada de funções e passagem de parâmetros e *layout* de objetos; neste caso, eles são compatíveis binariamente [Ope 03a].

Esta compatibilidade binária elimina a necessidade do uso de adaptadores entre classes escritas em Java e C++, facilitando assim a tarefa de integração entre programas escritos nestas linguagens.

O Exemplo 2.5 mostra a implementação dos métodos nativos da classe apresentada no Exemplo 2.3 usando CNI. Note que no método `newObject` a criação do novo objeto é feita exatamente da mesma forma que em um programa inteiramente em

[5]GNU GCJ

[6]Onde esta compatibilidade faz sentido.

C++. Isto é possível por causa da compatibilidade binária fornecida pelo GNU GCJ.

```
#include "ClassWithNaviteMethod.h"
#include <gcj/cni.h>
#include <iostream>

jint ClassWithNaviteMethod::showString(jstring str) {
        jsize len = JvGetStringUTFLength(str);
        char tmp[len+1];
        JvGetStringUTFRegion(str,0,len,tmp);
        tmp[len] = 0;
        std::cout << tmp << std::endl;
        return 1;
}

jstring ClassWithNaviteMethod::useMethod() {
        return this->toString();
}

ClassWithNaviteMethod *ClassWithNaviteMethod::newObject() {
        return new ClassWithNaviteMethod();
}
```

Exemplo 2.5: Implementação dos métodos nativos da classe apresentada no Exemplo 2.3 usando CNI.

Ainda no Exemplo 2.5, o arquivo `ClassWithNativeMethod.h` é criado automaticamente a partir do código binário da classe. No caso do CNI, o arquivo é criado com o auxílio da ferramenta `gijh`, distribuída juntamente com o GNU GCJ.

2.4 Comentários Sobre Java

O termo Java se aplica a três diferentes conceitos: uma linguagem de programação, uma API na forma de uma biblioteca de classes e um ambiente de execução. Para que uma plataforma forneça suporte a Java, deve haver um compilador que gere o código a ser executado, um ambiente de execução responsável por garantir os recursos necessários à execução das aplicações e uma biblioteca de classes que implemente a API Java, responsável pelo integração da aplicação com o ambiente de execução.

A linguagem Java possui características similares às demais linguagens de programação orientadas a objetos. A linguagem é fortemente tipada e usa tipos para determinar características da aplicação. A utilização do método `start()` da classe `java.lang.Thread` ou uma de suas subclasses, por exemplo, indica a existência de mais de um fluxo de execução no sistema.

A biblioteca de classes Java possui dependências do ambiente de execução. Desta forma, muitas vezes a biblioteca de classes deve ser reimplementada para ser usada em um ambiente de execução diferente. Para solucionar este problema, o projeto GNU Classpath está desenvolvendo uma biblioteca de classes Java com o objetivo de ser livremente disponível e de ser facilmente adaptada a novos ambiente de execução.

O ambiente de execução é responsável por permitir a execução de aplicações com o melhor desempenho e menor consumo de recursos possíveis. Para melhorar o desempenho das aplicações são usados compiladores *Just-In-Time*. Entretanto, a presença destes compiladores demanda maior consumo de recursos.

O acesso à características dependentes da plataforma e o suporte à integração com sistemas legados é feito através de métodos nativos. As implementações de interfaces entre Java e métodos nativos são intimamente ligadas ao ambiente de execução. Das interfaces de métodos nativos existentes, a JNI é a mais amplamente usada e conhecida enquanto a CNI, implementada no GNU GCJ, é a que melhor adapta a C++ a Java.

Capítulo 3

Java em Sistemas Embutidos

Dispositivos como aparelhos de telefone celular, câmeras digitais e aparelhos de GPS[1] estão cada vez mais presentes na vida cotidiana. Entretanto, poucas pessoas lembram que existe um processador embutido em cada um desses aparelhos. Segundo Wolf [WOL 01], "*sistema embutido* é qualquer dispositivo que inclui um processador programável mas não é um computador de uso geral".

Como os computadores de uso geral, os sistemas embutidos também possuem aplicações adequadas às suas funções. Um telefone celular, por exemplo, normalmente possui uma "agenda de telefones" e outras aplicações como jogos, editor de mensagens, etc. Entretanto estas aplicações não podem ser tão complexas quanto as usadas em computadores de uso geral. Esta diferença existe pois, normalmente, sistemas embutidos têm severas restrições em poder de processamento e quantidade de memória. Tais restrições são motivadas por vários fatores, entre os quais pode-se destacar: custo, consumo de energia de baterias e medidas físicas como tamanho e peso [WOL 01].

Tendo em vista estas restrições e para que seja obtido o máximo desempenho do *hardware*, os *softwares* contidos em um sistema embutido são, normalmente, escritos em C e linguagem *assembly*. Também tendo em vista o desempenho dos aparelhos, estes *softwares* são compilados para código nativo, específico ao tipo de processador usado.

[1] Sigla em inglês para Sistema de Posicionamento Global (*Global Positioning System*).

Java é uma alternativa ao uso das linguagens C e C++. Existem várias bibliotecas e *frameworks* desenvolvidos para esta linguagem. A utilização destes componentes de *software* permite economia de tempo e dinheiro em um projeto de sistema embutido. Java também é extremamente portável. Esta portabilidade é devido ao fato de que detalhes a respeito da plataforma são escondidas pelo ambiente de execução. Desta forma, para portar uma aplicação Java para uma nova plataforma, basta a existência de um ambiente de execução adequado.

Entretanto, existem temores de que o desempenho final do sistema não seja satisfatório com o uso de Java. Este temor advém da má impressão causada pelo desempenho das primeiras máquinas virtuais, onde o programa escrito em Java era totalmente interpretado e as otimizações existentes eram escassas [GAG 02].

Atualmente, existem métodos de resolver este problema. As máquinas virtuais são providas de sofisticados mecanismos para aumentar o desempenho das aplicações. Dentre estes mecanismos deve-se destacar o uso de compiladores *Just-In-Time*, que transformam o código *bytecode* para nativo em tempo de execução. Entretanto, o uso de compiladores *Just-In-Time* não é recomendado a sistemas embutidos, vez que estes compiladores devem permanecer ativos na memória e devem manter áreas para armazenar o código dos métodos transformados. Outro método para aumentar o desempenho de aplicações Java é a transformação do *bytecode* para código nativo antes da aplicação ser executada. Esta abordagem é mais adequada a sistemas embutidos, uma vez que elimina-se a necessidade de um processo para transformação do código *bytecode* em tempo de execução. Os compiladores usados para este fim são chamados de compiladores *Ahead-of-Time*.

As seções seguintes apresentam alternativas para execução de aplicações Java em sistemas embutidos. Inicialmente é apresentada uma metodologia para isolamento da aplicação Java. Esta metodologia permite isolar componentes da aplicação e bibliotecas das quais a aplicação depende, diminuindo o tamanho final das aplicações. A seguir são apresentados métodos para transformar código Java para código nativo da plataforma onde a aplicação é executada. Finalmente são apresentados ambientes de execução definidos para permitir a execução de aplicações Java em sistemas embutidos. Ini-

cialmente é apresentada a solução *Java 2 Micro Edition*, proposta pela Sun Microsystems. Logo após é apresentada a solução proposta para o sistema PURE.

3.1 Isolamento do Código Necessário à Aplicação

Sistemas construídos inteiramente a partir do zero geralmente têm todo seu código realmente utilizado pela aplicação durante a sua execução. Entretanto, atualmente, poucas são as aplicações que tem seu código feito inteiramente do zero. Normalmente, os programadores optam pelo uso de bibliotecas de classes que implementam parte das necessidades da aplicação. Desta forma, reduzindo o custo de produção de *software*.

Bibliotecas de classes estão amplamente disponíveis e cobrem boa parte das necessidades das aplicações. Caso as bibliotecas utilizadas não estejam disponíveis na plataforma alvo, o que é comum para sistemas embutidos, elas podem ser instaladas juntamente com a aplicação. Entretanto, o uso de bibliotecas de classes acarreta na inclusão de código que na realidade não é usado pela aplicação durante sua execução. Várias destas bibliotecas são implementadas como *frameworks* orientados a objetos ou contém um conjunto bem maior de funcionalidades que a aplicação necessita. *Frameworks* orientados a objetos [JOH 97], por serem voltados a um conjunto de aplicações, têm classes métodos e atributos que são desnecessários a uma aplicação específica. Isto faz com que ocorra um aumento desnecessário no tamanho da aplicação, desperdiçando-se valiosos recursos.

Como os recursos disponíveis nos sistemas embutidos são escassos, é inaceitável o desperdício com código que não é utilizado pela aplicação. Usando técnicas de extração de código das aplicações Java baseadas em bibliotecas de classes [TIP 03], pode-se reduzir o uso destes recursos. Estas técnicas isolam apenas o código utilizado pela aplicação, eliminando classes, métodos e atributos desnecessários. Desta forma, estas técnicas podem ser usadas para reduzir o tamanho de código das aplicações.

Normalmente as bibliotecas de classes são distribuídas em forma binária, em *bytecode*, que é uma representação de alto nível, e contém informações sobre a hierarquia de classes, tipos e métodos utilizados no corpo de cada método, etc. O iso-

lamento do código utilizado é possível pois a estrutura que armazena as classes em Java possui referências textuais[2] a todas as classes, métodos e atributos utilizados na classe em questão. Outra característica que permite uma análise no código é a ausência de aritmética de ponteiros e *typecasting* inseguro [TIP 03].

A Figura 3.1 mostra como funciona a utilização destas técnicas sobre uma hierarquia de classes da qual uma aplicação só necessita de alguns de seus membros. Da mesma forma que classes inteiras podem não ser utilizadas por uma determinada aplicação, também existem casos de classes das quais apenas uma parte de seus métodos e atributos são utilizados. Para estes casos, pode-se também isolar somente os métodos e atributos que são realmente necessários.

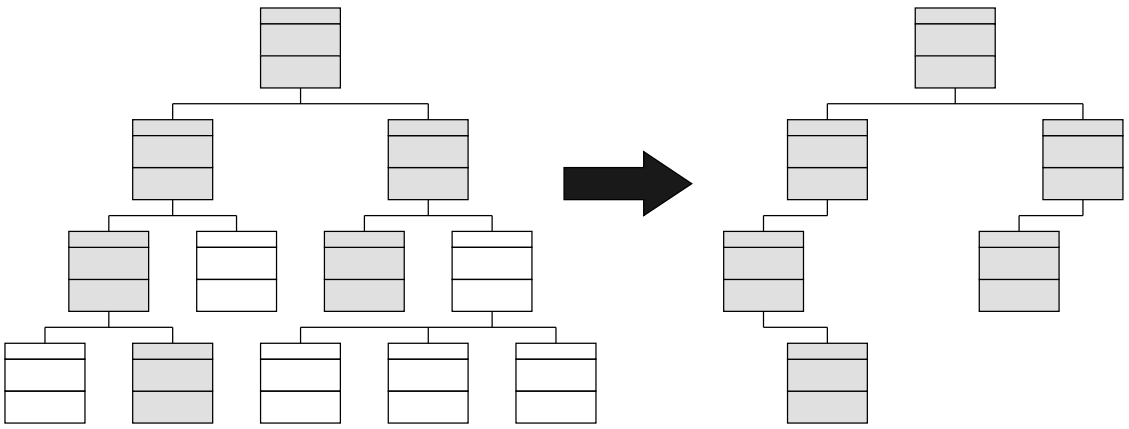

Figura 3.1: Frameworks e bibliotecas de classes geralmente apresentam mais recursos que a aplicação necessita. O isolamento do código que a aplicação necessita reduz o tamanho total do sistema.

Além de selecionar o código realmente utilizado pela aplicação, pode-se fazer otimizações nas estruturas de classes. A ferramenta de extração de aplicações JAX [TIP 99], por exemplo, seleciona e otimiza o código das aplicações. A utilização do JAX resulta em uma redução média de 48,7% nos tamanhos das aplicações. Entre outras otimizações, esta ferramenta elimina atributos redundantes, elimina chamadas virtuais quando possível e reorganiza a estrutura de classes de forma a restar o menor número de

[2] São armazenadas as assinaturas das classes, métodos e atributos utilizados.

classes. Neste processo, campos dos arquivos `class` que são utilizados para depuração, tais como tabela de nomes de variáveis locais e tabela de correspondência das linhas no arquivo fonte, também são eliminados. Estas otimizações, além de diminuir o tamanho do código também diminuem o tempo da início da aplicação. Isto é possível uma vez que existem menos classes para serem carregadas na inicialização do ambiente de execução [TIP 99].

Existe uma restrição para o isolamento do código da aplicação. Classes carregadas dinamicamente, usando por exemplo o método `forName` da classe `java.lang.Class`, não podem ser identificadas automaticamente. Esta restrição existe pois este tipo de método usa uma *string* para a identificação das classes a serem carregadas. São usados os nomes de classes no formato legível para humanos. Desta forma, os nomes de classes podem ser facilmente lidos e alterados em arquivos de configuração, por exemplo. A utilização de literais ou variáveis contendo os nomes das classes impede a previsão de qual tipo será utilizado após a classe ter sido carregada.

Sempre que existirem chamadas a métodos que carreguem classes dinamicamente, o nome das classes carregadas deve ser informado à ferramenta de isolamento da aplicação. No caso de sistemas embutidos, na maioria das vezes, o nome das classes carregadas dinamicamente é conhecido no momento da criação do sistema. Isto permite desenvolver ferramentas que possam ser alimentadas com estas informações antes de iniciarem o isolamento do código da aplicação.

3.2 Transformando Java em código nativo

Para melhorar o desempenho, aplicações em Java podem ser transformadas para código nativo do *hardware* no qual serão executadas. Para isso, a metodologia mais utilizada é transformar o código Java em tempo de execução com o uso de compiladores *Just-In-Time*. Este tipo de compilador permanece ativo durante a execução de um programa à espera de métodos a converter para código nativo. Cada vez que um método é invocado, a máquina virtual procura a versão nativa deste método. Caso o método ainda não tenha sido transformado, a máquina virtual pede ao compilador *Just-In-Time* que o

traduza para código nativo. Uma vez traduzidos, os métodos convertidos são armazenados à espera de nova invocação.

Entretanto, a utilização de compiladores *Just-In-Time* não é a mais adequada a sistemas embutidos. O motivo desta utilização não ser adequada é o consumo adicional de memória e processamento, uma vez que o compilador fica ativo durante toda a execução da máquina virtual. Além disso, os métodos traduzidos também são armazenados na memória, aumentando a necessidade de memória para manter um mesmo método.

Outra metodologia existente consiste em transformar o código antes da execução da aplicação. Pode-se transformar o código Java para código nativo e permitir a execução em conjunto com *bytecodes* Java. São exemplos desta técnica os ambientes de execução Harissa [MUL 97] e KVM [Sun 00]. A máquina virtual KVM, por exemplo, possui uma ferramenta que converte programas escritos em Java para a linguagem C. Este código é então compilado e ligado à máquina virtual. As classes transformadas em código nativo ficam então disponíveis à máquina virtual desde sua inicialização. Desta forma, classes compiladas para *bytecode* podem ser carregadas e executadas na máquina virtual e interagir com classes transformadas em código nativo.

Além de ligar o código nativo à máquina virtual, pode-se eliminar a necessidade desta. Para isso, é necessária a utilização de bibliotecas que forneçam serviços tradicionalmente prestados pela máquina virtual. Neste caso, fica impossibilitada a execução de código em *bytecode*. Este é o caso do sistema Toba [PRO 97], que permite a execução de aplicações Java sem a necessidade de um interpretador de *bytecodes* ou compilador JIT. Este sistema é constituído por um compilador capaz de transformar *bytecodes* Java para C, um coletor de lixo, um pacote que implementa *threads* e suporte à API Java. Em [PRO 97] o termo *"Way-Ahead-of-Time"* (WAT) é usado para descrever o sistema Toba, como uma forma para diferenciar do termo *"Just-In-Time"* (JIT).

Além do uso de compiladores WAT, que traduzem o *bytecode* Java para código C, pode-se transformar código Java diretamente para código nativo com o uso de compiladores *Ahead-of-Time*. Esta técnica é mais simples e permite uma série de vantagens. A principal vantagem é a possibilidade de compilar aplicações Java de tercei-

ros sem que haja a necessidade de acesso aos seus códigos fonte. O compilador GNU GCJ [Ope 03a] é um exemplo de compilador *Ahead-of-Time* que pode transformar programas em Java para código nativo e possui uma biblioteca com suporte a aplicações Java.

3.2.1 *Just-In-Time versus Ahead-of-Time*

Sem a utilização de compilador JIT temos um menor uso de memória, mas tempos um desempenho pior, pois todo o processamento é interpretado. Com a utilização de um compilador JIT ativo na memória, transformando todos os métodos utilizados temos uma melhora no desempenho de métodos freqüentemente usados. Entretanto, também temos um maior gasto de memória para o compilador e para os métodos compilados. Além disso, o desempenho de métodos pouco usados é prejudicado com a transformação de seu código.

Com a utilização de um compilador JIT ativo na memória, transformando somente métodos freqüentemente utilizados temos uma melhora desempenho de métodos freqüentemente usados. Métodos pouco utilizados não tem seu desempenho degradado pois não são transformados para código nativo. Entretanto, também temos um maior gasto de memória para o compilador e para os métodos compilados. Nesta caso também existe um gasto com processamento de heurísticas para determinar os métodos freqüentes.

Com a utilização de um compilador *Ahead-of-Time* durante a carga da aplicação o processo tem desempenho melhorado pois todo processamento é feito nativamente. Entretanto, é observado que o tempo para o início da aplicação aumenta e existe perda de desempenho ao receber classes via rede.

Com a utilização de um compilador *Ahead-of-Time* durante a “criação” do sistema todo processamento é feito nativamente. Entretanto, existe a perda de portabilidade e problemas com código carregado da rede.

3.2.2 GNU GCJ

O GCJ faz parte do pacote *GNU Compiler Collection* (GCC) e foi inicialmente uma iniciativa da RedHat (Cygnus Solutions) de desenvolver um compilador que transformasse Java em código nativo. Tanto fontes de aplicações Java quando programas compilados para *bytecode* (arquivos `class`) podem ser convertidos para código nativo com o uso deste compilador. A transformação de Java para código nativo permite um melhor desempenho das aplicações.

Além do compilador, o GCJ também dispõe de um ambiente de execução Java. Este ambiente de execução pode ser usado tanto para execução de aplicações Java transformadas para código nativo quanto para interpretação de aplicações Java em *bytecode*.

Aplicações compiladas com GCJ são ligadas ao ambiente de execução de sua biblioteca, `libgcj`. Esta biblioteca contém as classes básicas, um coletor de lixo e um interpretador de *bytecode*. A biblioteca `libgcj` possui métodos que permitem carregar e interpretar dinamicamente arquivos de classes em *bytecode*, resultando em aplicações mistas com código compilado e interpretado. As classes que compõem a API Java desta biblioteca foram desenvolvidas especificamente para uso com o GCJ. Entretanto, atualmente existe um esforço para unificar esta biblioteca à biblioteca GNU Classpath [Ope 03b], processo que foi iniciado em março de 2000.

Da forma como o ambiente de execução do GCJ foi implementado, existe a necessidade de implementação da interface POSIX no sistema onde a aplicação gerada é executada. A dependência da interface POSIX existe pois *threads*, E/S, exceções e outras características são implementadas através de invocações de funções definidas nesta interface. Desta forma, qualquer tentativa de utilizar a biblioteca `libgcj` implica em utilizar também uma biblioteca de funções POSIX.

Os principais pontos de dependência de POSIX são:

- **Threads**: A camada de *threads* da biblioteca *libgcj* foi projetada para expor precisamente as partes do sistema de *threads* que não são necessários a um ambiente de execução Java.

- **E/S**: As classes do pacote `java.io` necessitam de alguma conexão com o sistema de arquivos do sistema operacional na qual o programa Java está sendo executado. Existem basicamente duas partes dependentes do sistema. Primeiro a classe `File-Descriptor` é usada para representar um arquivo aberto. Ela tem métodos nativos para abrir, fechar, ler, escrever, etc. Depois a classe `File` pode ser usada para manipular um sistema de arquivos através de métodos como `mkdir`, `rename`, etc.

- **Sinais**: Quando um sinal é enviado a um programa Java, o ambiente de execução Java deve lançar uma exceção que deve ser capturada pela aplicação. Exemplos de sinais que geram exceções são `SIGSEGV`[3] que causa uma `NullPointerException` e `SIGFPE`[4] que origina uma `ArithmeticException`.

O tamanho do arquivo executável gerado pelo GNU GCJ, considerando todas as dependências fica muito grande. A Tabela 3.1 mostra a comparação dos tamanhos de algumas aplicações compiladas com GNU GCJ. As aplicações utilizadas nesta comparação foram: *Hello World!*, uma aplicação que simplesmente apresenta uma frase na saída padrão; *Webserv 0.2.0*, um pequeno servidor Web e *JOrbis 0.0.12*, um decodificador de arquivos de som no formato Ogg Vorbis. Como pode ser visto nesta tabela, a aplicação *Hello World!*, compilada de forma estática e sem símbolos, tem seu tamanho por volta de 2,3 MB. Estes tamanhos são explicados pelo fato dos executáveis serem dependentes da biblioteca `libcgj` e, por conseqüência, de uma implementação da interface POSIX.

Dados os valores dos tamanhos de executáveis fica evidente que portar diretamente o compilador e a biblioteca para uma plataforma de sistema embutido é inviável. Para fazer o uso desta tecnologia em sistemas embutidos existe a necessidade de adaptar o ambiente de execução da *libgcj* de forma a torná-lo mais modular. Desta forma, pode-se selecionar os componentes que devem ser utilizados no sistema final.

[3] Segment Violation
[4] Arithmetic Exception

Aplicação	HelloWorld	Webserv	Jorbis
Código original em *bytecode* Java	426	14.422	106.585
Código nativo, ligação dinâmica sem símbolos	4.984	40.500	323.460
Código nativo, ligação dinâmica com símbolos	9.053	55.259	448.299
Código nativo, ligação estática sem símbolos	2.310.660	2.340.900	2.622.884
Código nativo, ligação estática com símbolos	18.966.809	19.003.788	19.392.674

Tabela 3.1: Tamanhos expressos em *bytes* de executáveis gerados pelo GNU GCJ. Os programas apresentados nesta tabela foram compilados para um computador Pentium com Linux. Na linha que apresenta o tamanho do *bytecode* Java, não está computado o tamanho do ambiente de execução Java. Para comparação, o tamanho do ambiente J2ME CLDC, incluindo a KVM versão 1.0.4 para Linux e biblioteca de classes, é 821.603 *bytes*.

3.3 Java 2 Micro Edition

A SUN tem uma alternativa para execução de programas Java em sistemas com poucos recursos denominada Java 2 Micro Edition (J2ME). Existem dois esquemas propostos pela SUN: *Connected Device Configuration* (CDC) e *Connected, Limited Device Configuration* (CLDC).

O CDC se destina a dispositivos que estejam conectados à rede e que tenham algumas restrições de recursos. Estas restrições, entretanto, não podem ser muito severas, uma vez que o ambiente CDC exige, no mínimo 2MB de memória RAM para ser executado. Este ambiente é destinado a dispositivos com memória RAM entre 2MB e 16MB. Acima deste limite a SUN aconselha a usar o ambiente normal de execução Java (J2SE ou J2EE).

O CLDC se destina a dispositivos com severas restrições de recursos. Entretanto, segundo a SUN [Sun 02a], existe a necessidade de 160 KB a 512 KB de RAM para que ele seja executado. Este ambiente dispõe de uma máquina virtual diferenciada, chamada KVM. A KVM é uma máquina virtual projetada para ser compacta e portável, mantendo quando possível todos as principais características da linguagem Java. Ela é

escrita em C/C++ e seu código ocupa até 80KB de memória estática do sistema. Entretanto, a KVM não dispõe de muitos recursos que a máquina virtual tradicional da SUN possui. Entre outras características, a KVM[5] não implementa suporte aos tipos `float` e `double`, não permite o uso de reflexão ou referências fracas, não invoca o método `finalize()` quando um objeto é destruído e não implementa a maioria das subclasses de `java.lang.Error` [Sun 00]. Estas restrições fazem com que a maioria das aplicações desenvolvidas em Java devam ser adaptadas ou reescritas para serem executadas em um ambiente CLDC.

A Figura 3.2 mostra como é o relacionamento entre as classes que podem ser utilizadas nas configurações CDC, CLDC e *Java 2 Standard Edition*. A compatibilidade das bibliotecas do J2ME com as bibliotecas tradicionais (J2SE) está restrita às classes do pacote `java`[6]. Além disso, classes fora do contexto do J2SE não podem usar o *namespace* `java.*` [Sun 00].

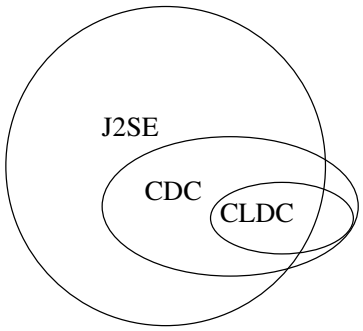

Figura 3.2: Relacionamento entre as configurações do J2ME (CLDC e CDC) e Java 2 Standard Edition(J2SE). Figura retirada da documentação do J2ME [Sun 00].

O J2ME CLDC tem como alvo dispositivos com pelo menos 160Kb de memória total[7] disponível para a plataforma Java [Sun 02a].

O pacote J2ME possui uma ferramenta que converte aplicações em Java para código C portável. Desta forma, este código pode ser compilado e ligado à máquina virtual. O código que é ligado a máquina virtual passa a estar disponível deste a inicia-

[5] Versão 1.0.4.

[6] Pacotes como `java.lang`,`java.io`,`java.util` entre outros.

[7] Incluindo memória principal (RAM) e memória para armazenamento de código (ROM ou Flash).

lização do ambiente de execução. Entretanto, para que uma classe possa passar por este processo é necessário que todas as classes por ela referenciadas também sejam ligadas à máquina virtual. A vantagem deste método é que o interpretador de código *bytecode* continua estando disponível. Classes recebidas da rede, por exemplo, podem ser carregadas e executadas em adição ao código ligado à máquina virtual.

3.4 JPURE – Estratégia para execução de Java do sistema PURE

JPURE é uma metodologia desenvolvida para permitir que aplicações escritas em Java sejam executadas em redes de micro-controladores. Para construir o ambiente de execução, o JPURE utiliza a combinação do sistema PURE [BEU 99] com ferramentas *software* livre como GNU GCJ.

A estratégia adotada no JPURE consiste em traduzir código Java para código nativo. Esta transformação é feita com o compilador GNU GCJ. O código nativo da aplicação é ligado à biblioteca `libgcj` e roda sobre o sistema PURE, como mostra a Figura 3.3.

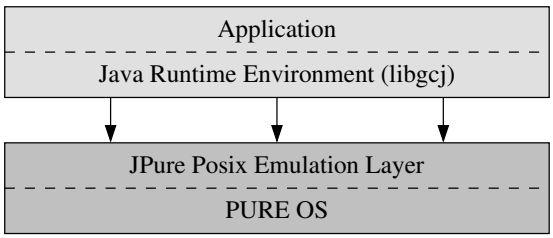

Figura 3.3: Esquema de execução de aplicações Java no JPURE.

Para ser possível a execução de aplicações transformadas, foi necessário desenvolver uma nova família para PURE. Esta família foi construída especialmente para fornecer a interface POSIX à biblioteca `libgcj`. Esta necessidade é devida ao fato de que a biblioteca a `libgcj` utiliza funções POSIX, como visto na seção 3.2 deste capítulo. Além disso, também foi necessário modificar a `libgcj` para torná-la mais

modular [BEU 00].

Em um teste documentado em [BEU 00], foi feito um sistema para execução de uma aplicação simples[8] em um equipamento com um processador Intel Pentium. Neste teste, o sistema completo, incluindo aplicação, `libgcj` e JPURE, teve seu tamanho em torno de 311 KB. Deste total, a utilização de memória da biblioteca `libgcj` somada à aplicação correspondeu a 94% do consumo de memória do sistema.

3.5 Comentários Sobre Java em Sistemas Embutidos

A produção de sistemas embutidos é regida pelo mercado consumidor. O tempo gasto para finalizar o desenvolvimento de um sistema embutido é essencial. Desenvolver aplicações do zero implica em maior custo e tempo para a liberação do produto para o mercado consumidor. A vantagem de usar Java em sistemas embutidos está ligada à sua portabilidade, à grande quantidade de bibliotecas de classes existentes e à sua grande popularidade em programadores.

As aplicações podem ser otimizadas de forma a reduzirem a necessidade de memória. Otimizações feitas no código das aplicações não demandam modificações no ambiente de execução. Desta forma, o código gerado continua apto a ser usado em qualquer ambiente de execução Java. Entretanto, essas otimizações não são suficientes para garantir a execução de aplicações Java em sistemas embutidos.

Sistemas embutidos, entretanto, têm uma série de características que limitam a disposição de recursos de *hardware* às suas aplicações. Estas limitações proíbem o uso dos mesmos ambiente de execução Java utilizados em sistemas de uso geral. Desta forma é necessário adaptar ou construir ambientes de execução especiais de forma a permitir seu uso em sistemas embutidos. Estes ambientes de execução especiais normalmente impõem limites aos programadores, que passam a não dispor de todas as vantagens da linguagem Java.

Uma maneira alternativa ao porte de máquinas virtuais às plataformas alvo é a transformação da aplicação diretamente para código nativo com o uso de compi-

[8] Neste teste foi utilizada a aplicação “Hello World!”.

ladores *Ahead-of-Time*. Uma vantagem desta abordagem é o aumento do desempenho das aplicações, uma vez que não é necessário o uso de interpretadores de código. Entretanto, além da transformação do código Java para código nativo, também é necessário fornecer um suporte de execução à aplicação compilada. Este suporte é composto por classes da API Java e pelo suporte destas classes que é dependente de sistema operacional.

A abordagem utilizada pelo GNU GCJ, de utilizar uma camada POSIX para fornecer suporte às aplicações Java permite que este ambiente de execução seja facilmente portado para novas plataformas. Entretanto, as dependências de bibliotecas do executável gerado pelo GNU GCJ fazem com que seu tamanho seja muito superior ao recomendado para sistemas embutidos. Estas dependências devem ser minimizadas para permitir sua utilização em sistemas embutidos.

Capítulo 4

Application-Oriented System Design

Neste capítulo é apresentada a metodologia de projeto de sistemas denominada *Application-Oriented System Design* [FRö 01]. Esta metodologia é utilizada neste trabalho por permitir o desenvolvimento de um sistema de suporte sob medida para uma aplicação ou conjunto de aplicações.

Application-Oriented System Design (AOSD) é uma metodologia para projeto de sistemas de suporte à execução fortemente compromissados com as aplicações. A idéia fundamental é fornecer todas as necessidades da aplicação alvo, sem contudo gerar desperdício de recursos com a adição de serviços que não sejam necessários à aplicação. O sistema operacional é construído a partir de um conjunto de componentes de *software* que são selecionados com base nas necessidades das aplicações que nele serão executadas. Para isso, o programador deve explicitar a necessidade de tais recursos na aplicação. As necessidades da aplicação são expressas na forma de invocações de métodos a objetos de sistema. Para determinar as necessidades de uma aplicação, programas de análise de requisitos pressupõem a necessidade de determinados recursos a partir de inferências (tais como determinar o tipo de escalonador a partir dos métodos de escalonamento explicitados no código) e regras.

"Um sistema operacional orientado a aplicação somente é definido considerando a aplicação correspondente, para a qual ele implementa o suporte em tempo de execução necessário que é entregue como foi pedido" [FRö 01]. Sendo assim, não

pode existir uma instância de um sistema operacional orientado a aplicação sem que haja conhecimento prévio de todas as aplicações que nele serão executadas.

Como esta metodologia só pode ser utilizada em sistemas onde se tem conhecimento da aplicação que será executada no momento da construção do sistema, esta metodologia não pode ser aplicada a todos os sistemas operacionais. Sistemas de uso geral tais como sistemas operacionais para estações de trabalho não têm, e nem podem ter, informações sobre todas as suas possíveis aplicações. Por outro lado, existem tipos de sistemas que normalmente têm informações sobre as aplicações que neles serão executadas. Exemplos deste tipo de sistema são bastante comuns em sistemas de computação dedicada tais como sistemas embutidos e computação de alto desempenho.

Gerar um sistema sob medida é o caso ideal para aplicações dedicadas e que exigem desempenho máximo. Desenvolver um sistema do zero pode ser uma boa idéia quanto a implementação de sistemas sob medida. Entretanto, esta abordagem não permite o reuso de código. Esta abordagem também demanda maior tempo para o desenvolvimento do sistema.

Uma solução para este problema pode recair sobre o uso de repositórios de componentes de *software* que podem ser utilizados para gerar um sistema sob medida para uma determinada aplicação. Com a utilização da AOSD, pode-se fornecer um suporte *run-time* adequado a cada aplicação. Este suporte *run-time* é feito com base em um repositório de componentes reutilizáveis e configuráveis. O repositório de componentes reutilizáveis é usado para evitar gastos de tempo desnecessários com desenvolvimento de sistemas. Desta forma, basta selecionar os componentes necessários e juntá-los para fazer um novo sistema sob medida.

A metodologia prevê que a seleção de componentes do sistema seja fácil, por permitir configurar boa parte das características automaticamente. A utilização de ferramentas para determinar automaticamente as necessidades das aplicações faz com que não exista a necessidade de que os programadores de aplicações conheçam a fundo detalhes de desenvolvimento dos componentes do sistema operacional.

A utilização de componentes reutilizáveis ajuda a evitar a necessidade de construir um novo sistema a cada nova aplicação para garantir um sistema sob me-

dida. Entretanto, esta abordagem tem como requisito a granularidade dos componentes. Sistemas configuráveis com muitas opções de configuração são difíceis de operar. Um grande número de opções faz com que o programador possa configurar incorretamente o sistema. Isto acontece pois o programador pode não conhecer detalhadamente cada uma das opções e pode optar por uma opção menos eficiente ou até mesmo inadequada. Por outro lado, sistemas configuráveis com poucas opções de configuração geralmente resultam em sistemas que apresentam mais funcionalidades que as necessárias à aplicação. Desta forma, são desperdiçados recursos podendo causar degradação no desempenho do sistema.

Na próxima seção são apresentados os principais conceitos relacionados à AOSD. A seguir são apresentados como são formados os *frameworks* de componentes de acordo com a metodologia AOSD. Logo após o sistema EPOS é apresentado. Este é um exemplo de sistema que segue os princípios da AOSD.

4.1 Sistemas Orientados à Aplicação

A construção de sistemas operacionais orientados a aplicação é iniciada com a decomposição do domínio das aplicações. Segundo Fröhlich [FRö 01], o processo de decomposição do domínio orientado à aplicação é um método de análise de domínio que envolve muitos paradigmas. Ele possibilita a construção de sistemas operacionais orientados à aplicação através decomposição do domínio correspondente em famílias de abstrações independentes do cenário de execução e de seus aspectos de cenário. Os relacionamentos entre as famílias de abstrações são utilizados para construir *frameworks* de componentes. Este processo é ilustrado pela Figura 4.1.

Ao se analisar o domínio das aplicações, são identificadas abstrações que o compõem. Estas abstrações são agrupadas em famílias de acordo com suas similaridades. Dentro das famílias as abstrações são separadas em diferentes membros de acordo com suas diferenças. As abstrações são separadas dos aspectos de cenário de execução. São exemplos de cenários de execução a utilização de métodos remotos, a necessidade de sincronização, etc. Isto faz com que o número de componentes do repositório seja menor,

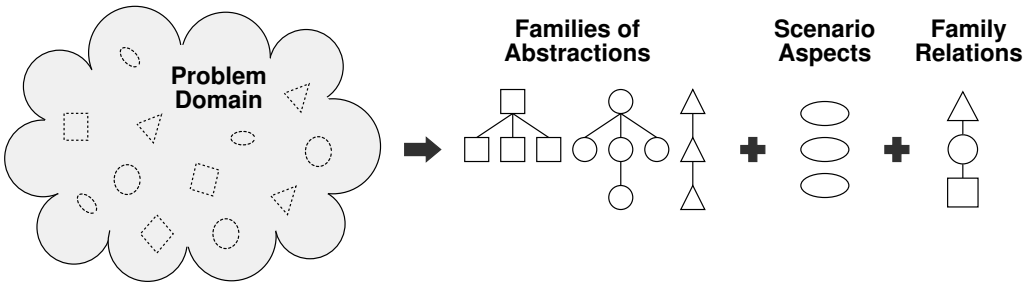

Figura 4.1: Decomposição de domínio orientado à aplicação. Figura retirada de [FRö 01].

além de permitir que os componentes sejam configurados de acordo com os cenários de execução.

Para que seja possível detectar automaticamente quais são as abstrações a serem utilizadas no sistema final, foi desenvolvido o conceito de interfaces infladas. Interfaces infladas reúnem em uma só entidade a interface de toda uma família de abstrações. Desta forma, o programador da aplicação pode utilizar a interface para explicitar as necessidades da aplicação, sem ter conhecimento de qual componente será utilizado. Estas necessidades são resolvidas através de uma ferramenta que investiga qual das abstrações disponíveis realiza melhor as necessidades descritas no código da aplicação.

A seguir são detalhados os conceitos de Famílias de Abstrações Independentes de Cenário, Aspectos de Cenário e Interfaces Infladas segundo a AOSD.

4.1.1 Famílias de Abstrações Independentes de Cenário

Após a decomposição do domínio, os componentes são agrupados em famílias de abstrações de maneira similar à metodologia *Family-Based Design* [PAR 76]. As abstrações são agrupadas de acordo com suas similaridades em famílias de abstrações específicas. Além disso, as abstrações são separadas de informações que são necessárias somente em determinados aspectos do cenário de execução. Os aspectos de cenários de execução, tais como localidade, são extraídos e armazenados independentemente como aspectos de cenário. Isto é feito para diminuir o número de membros de cada família e facilitar o gerenciamento e desenvolvimento do repositório de componentes. Esta se-

paração também permite desenvolver componentes configuráveis. Para determinar um componente específico deve-se selecionar tanto a abstração adequada quanto os aspectos do cenário de execução onde o componente será usado.

As famílias de abstrações são os componentes básicos do repositório de componentes. Dentro de uma família, as abstrações podem ser relacionadas por herança, ou simplesmente por terem funcionalidade semelhante, sem entretanto compartilharem qualquer aspecto de implementação. A utilização de famílias de abstrações facilita na seleção automática de componentes para o sistema final. Desta forma, o programador não necessita selecionar componentes de uma vasta lista de abstrações. As abstrações são utilizadas indiretamente nas aplicações através do uso de interfaces infladas. Para isto, o programador simplesmente faz invocações de métodos ou utilizada tipos disponíveis para uma determinada família de abstrações. A seleção da abstração adequada é feita de forma automática a partir de regras que levam em conta os métodos da família que são utilizados e modelos de custo.

4.1.2 Aspectos de Cenários

Aspectos de cenários de execução representam determinadas características das abstrações que são sensíveis à variação de cenários de execução. Os aspectos de cenário de execução permitem um melhor gerenciamento do repositório de componentes. Com a utilização de aspectos, o número de membros das famílias diminui bastante. Em um repositório com dois aspectos de cenário de execução, por exemplo, as abstrações podem ser utilizadas com ambos os aspectos ligados, com apenas um deles ligados ou com ambos os aspectos desligados. Neste exemplo, existe um número de combinações possíveis quatro vezes maior que o número de abstrações disponíveis. O número de combinações possíveis cresce exponencialmente em relação ao número de aspectos de cenários de execução.

Além de reduzir o número de membros de uma família, os aspectos de cenário também podem ser usados para representar as necessidades da aplicação que não estão explícitas no código. Como exemplo, um computador com múltiplos processadores

não é identificado através da análise do código da aplicação. Para isso são necessárias informações adicionais, utilizadas como uma especificação da plataforma onde o sistema será executado. A seleção de aspectos de cenário pode ser feita de forma manual, através de uma aplicação gráfica. Neste caso, o programador deve simplesmente descrever o ambiente de execução, quais são suas características, tais como tipo e número de processadores, dispositivos existentes a serem utilizados, etc.

A utilização dos aspectos de cenário de execução é feita através da aplicação destes aspectos às abstrações por intermédio de adaptadores de cenário. Os adaptadores de cenário, como o nome propõe, são implementados de maneira similar ao padrão de projeto *Adapter* [GAM 95]. Ou seja, a aplicação invoca um método de uma abstração através de um adaptador de cenário. O adaptador faz seu serviço e repassa a requisição ao componente que está realizando o serviço da abstração. Os adaptadores de cenário podem ter suas funções *ligadas* ou *desligadas* de acordo com os aspectos de cenário de execução.

4.1.3 Interfaces Infladas

Interfaces Infladas são interfaces que agregam todos os métodos e atributos que estão disponíveis nos membros de uma determinada família de abstrações. Isto permite que uma família inteira de abstrações possa ser vista de uma única maneira. Além disso, o uso de interfaces infladas faz com que o programador possa escrever programas tendo como base um nível mais alto de abstração. Desta forma, o programador da aplicação não precisa saber qual abstração será utilizada na construção do sistema de suporte a sua aplicação.

Diferentemente do conceito de interfaces apresentado no Capítulo 2, os membros de uma família normalmente *não* realizam todos os métodos declarados na interface. Sendo assim, de acordo com os métodos da interface utilizados na aplicação pode-se determinar as abstrações que melhor se adaptam às necessidades da aplicação. Por este motivo, interfaces infladas são utilizadas como ferramentas para facilitar a identificação dos membros das famílias de abstrações que são adequados a uma determinada

aplicação. A abstração a ser utilizada pode ser determinada automaticamente a partir de métodos e tipos usados pela aplicação. Desta forma, a seleção de componentes usados no sistema pode ser feita com o auxílio de regras de inferência e modelos de custo.

Existem quatro categorias de famílias de abstrações de acordo com sua representação por interfaces infladas:

- Uniforme: são usadas em famílias de abstrações onde todos os membros tem a mesma interface. Desta forma, o conceito de Interfaces Infladas Uniformes é muito semelhante ao conceito de interfaces em Java;

- Incremental: são usadas em famílias de abstrações onde cada nova abstração da família implementa todas as funcionalidades anteriormente existentes e agrega alguma funcionalidade nova. Neste tipo de família, os membros mais complexos implementam todos os métodos das interfaces;

- Combinada: são usadas em famílias de abstrações onde os membros não possuem intersecção de funcionalidades, mas os membros podem ser mesclados para produzir novos membros com funcionalidades combinadas;

- Desassociada: são usadas em famílias de abstrações que não se encaixam nas categorias anteriores.

Quando referenciados por uma aplicação, as três primeiras categorias de famílias de abstrações podem sempre ter membros associados. Essa associação pode ser feita por um membro existente ou por um novo membro produzido pela mescla de membros existentes, no caso de uma família combinada. Entretanto, no caso de famílias desassociadas, a associação só é feita caso o subconjunto de métodos da interface inflada utilizada pela aplicação coincidir com um único membro da família.

4.2 Framework de Componentes

Frameworks [JOH 97] são estruturas ativas que permitem o reuso da arquitetura de um sistema. Eles são constituídos por um conjunto de componentes e re-

gras de como os componentes são arranjados e como se inter-relacionam. *Frameworks* são definidos através de um processo incremental, utilizando experiências anteriores na construção de sistemas de um certo tipo. Esta experiência pode ser capturada em uma especificação de arquitetura que pode ser reusada em novo sistemas. O processo de evolução de um *framework* é iniciado com uma estrutura sem muitas implementações de cada uma das abstrações. Com o passar do tempo novas implementações e funcionalidades vão sendo incorporadas até o ponto em que todo o domínio visado esteja representado no *framework*.

Em AOSD, a arquitetura do sistema começa a ser desenvolvida durante a decomposição do domínio. Isto é feito através do identificação de relacionamentos entre as famílias de abstrações. O relacionamento entre componentes do *framework* também é alimentado com características dos cenários de execução. Propriedades de sistema podem ser mapeadas com o uso de aspectos do cenário de execução. Para cada família de abstrações o *framework* de componentes apresenta uma interface inflada, usada como uma forma de permitir uma visão única da família de abstrações, um membro da família de abstrações e um adaptador de cenário, que é utilizado para permitir que os aspectos do cenário de execução sejam aplicados sobre as abstrações. Este relacionamento é ilustrado pela Figura 4.2.

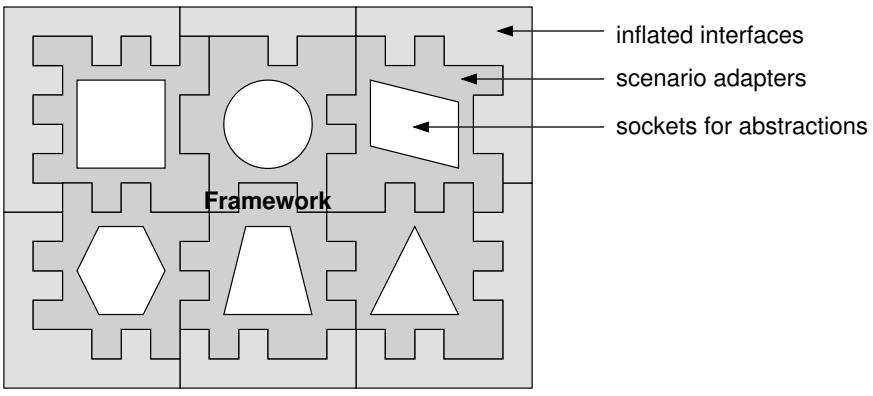

Figura 4.2: Framework de componentes orientado à aplicação. Figura retirada de [FRö 01].

Nem todas as composições de componentes são permitidas. A maneira como os adaptadores de cenário são agrupados no *framework* determina a arquitetura dos

sistemas resultantes.

4.3 Sistema EPOS

EPOS [FRö 99, FRö 01] é um sistema operacional que foi projetado seguindo as diretrizes de projeto de sistema orientado a aplicação. O domínio visado pelo sistema EPOS é a computação dedicada de alto desempenho. Este domínio inclui principalmente aplicações embutidas e paralelas. A sigla EPOS significa *Embedded Parallel Operating System*. O sistema EPOS fornece às aplicações um ambiente que disponibiliza um eficiente gerenciamento de recursos. A aplicação de AOSD, utilizada no EPOS, não produz um único sistema operacional, mas uma coleção de componentes que podem ser especializados e combinados de forma a definir uma grande variedade de sistemas operacionais.

O sistema de suporte do EPOS pode ser embutido em uma aplicação ou ser usado como um *micro-kernel*. O sistema pode ser embutido na aplicação no caso de aplicações com uma única tarefa. Desta forma, todo seu código é acessível pela aplicação e está disponível dentro do seu espaço de endereçamento. No caso de ambientes com múltiplas tarefas, o EPOS pode ser usado como um *micro-kernel*. Desta maneira, suas funções, serviços e dispositivos são protegidos em um espaço de endereçamento de sistema operacional.

O sistema EPOS, para ser um sistema operacional orientado a aplicação, segue as seguintes diretrizes:

- **Funcionalidade:** O sistema fornece a cada aplicação um suporte à execução de alto desempenho.
- **Customização:** O sistema é altamente configurável, permitido a criação de sistemas sob medida para as aplicações. Esta configuração deve ser feita, na medida do possível, de forma automatizada.
- **Eficiência:** O sistema fornece somente os componentes que são realmente necessários. Estes componentes são construídos de forma a não causarem *overhead*,

garantindo assim o melhor desempenho possível.

- **Invisibilidade:** O sistema provê as interfaces de acordo como são pedidas na aplicação. Esta invisibilidade é possível através do fornecimento de algumas APIs do Unix, incluindo `libc`, `libm` e `libstdc++`. Entretanto, o EPOS não compartilha nenhum aspecto de desenvolvimento com o UNIX.

Sendo assim, este sistema é ideal para desenvolver um ambiente de execução de programas escritos em Java, uma vez que o sistema pode ser configurado de forma a só fornecer os recursos necessários à aplicação e ao ambiente de execução Java.

4.4 Comentários sobre *Application-Oriented System Design*

Application-Oriented System Design é um método de projeto de sistema que possibilita a construção de sistemas operacionais orientados à aplicação. A utilização deste método é iniciada com a decomposição do domínio de forma a serem identificadas famílias de abstrações e aspectos de cenário de execução. Informações sobre os relacionamentos entre as famílias de abstrações são usadas para desenvolver um *framework* de componentes. Este *framework* é composto por regras na forma de relacionamentos entre adaptadores de cenário, usados para possibilitar a utilização dos vários aspectos de cenário sobre membros das famílias de abstrações. Este processo é ilustrado pela Figura 4.3.

Application-Oriented System Design pode ser usada para resolver os problemas de sistemas de computação dedicada, sendo que aplicações de sistemas embutidos são beneficiadas com este método. O método permite que sejam feitos sistemas sob medida para uma determinada aplicação ou conjunto de aplicações. Estes sistemas são projetados para garantir apenas as necessidades da aplicação, não desperdiçando valiosos recursos de *hardware* que não seriam utilizados pela aplicação.

O princípio da invisibilidade, almejado pela *Application-Oriented System Design*, determina que desenvolver aplicações tendo como alvo o sistema EPOS seja

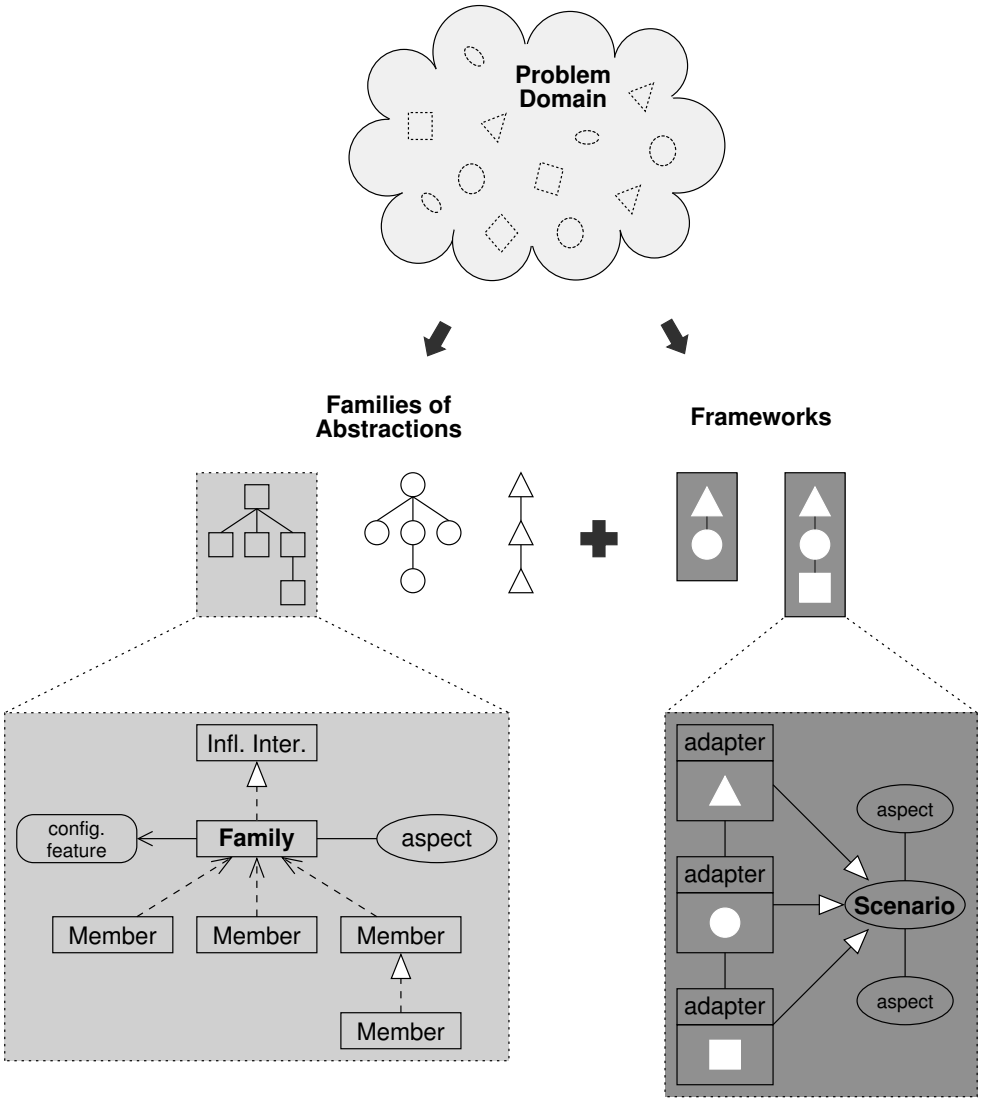

Figura 4.3: Projeto de sistema orientado à aplicação. Figura retirada de [FRö 01].

igual a desenvolver um sistema para outras plataformas. Isto é possível pois o EPOS é desenvolvido como um *framework* de componentes de *software* e pode disponibilizar estes componentes de acordo com interfaces padronizadas tais como POSIX, etc. Deve ser feita uma implementação de interfaces para o EPOS que possibilite a execução de programas escritos em Java. Desta maneira, aplicações Java podem fazer uso do sistema EPOS através de invocações de métodos de "objetos de sistema". As classes destes objetos devem ser implementadas de forma que recaiam sobre classes, métodos e atributos dos pacotes `java.lang`, `java.io`, etc. Desta forma, uma aplicação desenvolvida em Java pode ser executada em um sistema da família EPOS sem que tenha que sofrer alterações.

Capítulo 5

Ambiente de Execução Java Orientado à Aplicação

Como visto no Capítulo 2, o ambiente de execução tradicionalmente utilizado em Java é composto por uma máquina virtual. Esta máquina virtual é executada sobre um sistema operacional, conforme ilustra a Figura 5.1. Desta forma, a portabilidade das aplicações escritas em Java[1] depende unicamente do porte do ambiente de execução para uma nova plataforma de *software* ou *hardware*.

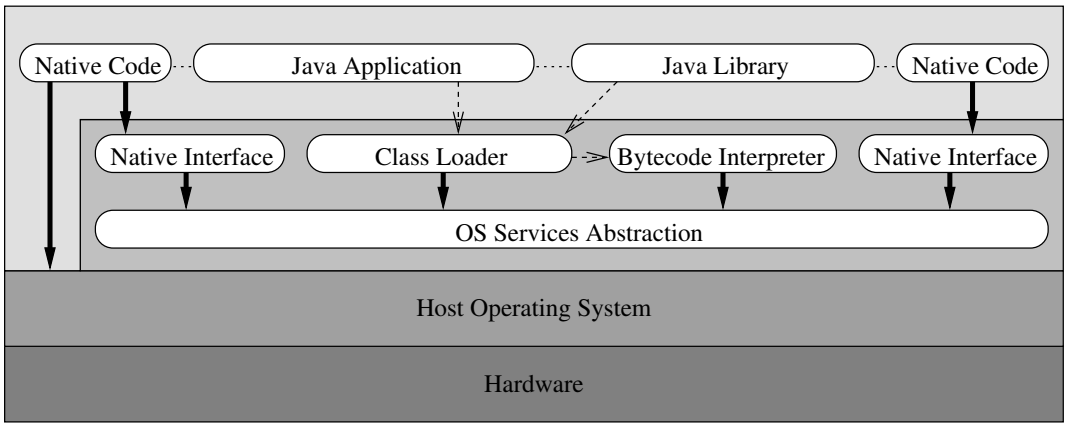

Figura 5.1: Esquema tradicional do ambiente de execução do Java.

Entretanto, essa abordagem não é muito interessante quando o novo

[1]Considerando apenas aplicações inteiramente escritas em Java, sem uso de métodos nativos.

ambiente em questão é um sistema com severas restrições de *hardware*, como visto no Capítulo 3. Portar um ambiente de execução Java com todas as suas características e bibliotecas para um sistema embutido não é uma boa solução pois muitos recursos seriam utilizados sem necessidade. Sendo assim, as principais tentativas de trazer Java para o mundo de sistemas embutidos são constituídas por ambientes de execução configuráveis. O projeto destes ambientes de execução permite a seleção de recursos que serão utilizados na plataforma alvo. Desta forma, o projeto destes ambientes precisa ser bem conhecido pelo desenvolvedor da aplicação a ser executada no novo sistema. Isso é necessário para garantir que a aplicação possa ser executada sem perda de funcionalidades e mesmo assim possibilitar uma utilização mínima de recursos.

A maioria das tentativas de fornecer ambientes de execução para sistemas embutidos também impõe algumas restrições na utilização das bibliotecas Java. Estas restrições variam em cada uma das implementações de ambiente de execução, podendo limitar o uso de classes Java ou recursos como métodos nativos. As restrições na utilização das bibliotecas Java podem variar desde impedir o uso de classes de pacotes gráficos[2] até impedir o uso de tipos primitivos como `float`. Estas restrições dificultam o uso de aplicações pré-existentes e aplicações baseadas em bibliotecas de classes de terceiros em sistemas embutidos.

Uma alternativa às tentativas tradicionais de restringir o uso de determinadas funcionalidades é construir um ambiente de execução com apenas as funcionalidades que são necessárias para a execução da aplicação em questão. Desta forma, pode-se reduzir a utilização de recursos pelo ambiente de execução sem impor limites à aplicação. Usando-se técnicas de *Application-Oriented System Design* pode-se definir um sistema sob medida para a aplicação que se deseja executar em um sistema embutido.

Pode-se utilizar um sistema operacional orientado à aplicação, conforme as técnicas de AOSD, para servir de suporte ao ambiente de execução Java. Desta forma, pode-se portar uma máquina virtual para este sistema operacional orientado à aplicação. Neste caso, as necessidades do ambiente de execução seriam usadas para definir o sistema.

[2] Tais como classes dos pacotes `java.awt`, `javax.swing`, etc.

No lugar de portar uma máquina virtual mínima para estes sistemas, pode-se optar por uma transformação de código Java para código nativo da plataforma alvo. Utilizando-se compiladores *Ahead-of-Time*, pode-se pensar Java da mesma forma como qualquer outra linguagem de programação. Desta forma, a utilização de Java é muito similar às linguagens tradicionais como C, C++, Pascal, Fortran, entre outras [BOT 03]. Utilizando a transformação de código Java (*bytecode*) para código nativo, pode-se eliminar a necessidade de uma máquina virtual para interpretar as instruções de um programa Java. Entretanto, a necessidade de um ambiente propício que dê suporte à execução deste código continua existindo.

Para fornecer este ambiente propício, pode-se agregar funcionalidades típicas do ambiente de execução Java ao sistema operacional. Desta forma, pode-se fazer com que o sistema operacional e o ambiente de execução sejam a mesma entidade, como ilustra a Figura 5.2. Neste caso, não existe a necessidade de um interpretador de *bytecodes*, uma vez que a aplicação será executada diretamente pela CPU. Também não são necessárias entidades do ambiente de execução responsáveis pela carga e validação de classes Java.

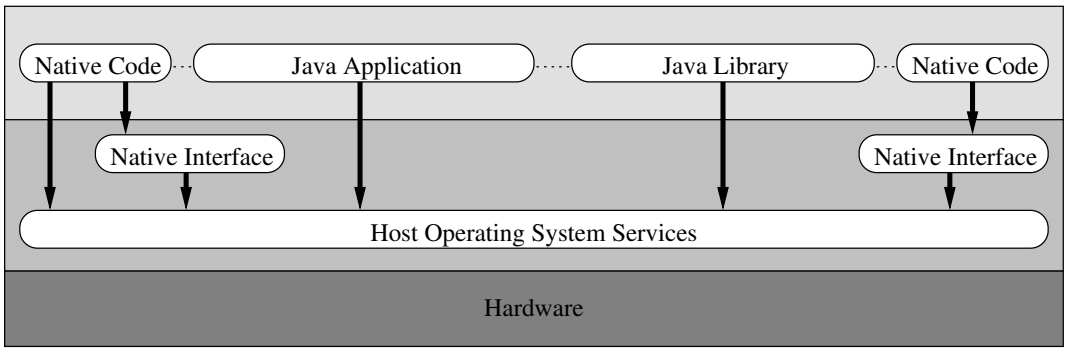

Figura 5.2: Esquema do ambiente de execução Java proposto.

Neste capítulo é apresentada uma proposta de um ambiente de execução para aplicações escritas em Java. O ambiente proposto é baseado no sistema EPOS, na análise de requisitos da aplicação, na determinação dos requisitos mínimos para o ambiente imposto pela aplicação e na transformação de código *bytecode* para código nativo usando um compilador *Ahead-of-Time*, como é o caso do GNU GCJ. A seguir é apre-

sentada a descrição de um método para construção do ambiente de execução com base nos requisitos da aplicação e logo após é mostrado um exemplo de utilização da técnica proposta.

5.1 Descrição do Método

O método apresentado a seguir baseia-se na extração de código de aplicações baseadas em bibliotecas de classes Java [TIP 03], na compactação do código obtido [De 03], na análise de dependências da aplicação e na configuração de um sistema sob medida para a aplicação de acordo com as técnicas de *Application-Oriented System Design* [FRö 01]. Finalmente, o sistema é construído ao se ligar componentes de *software* do sistema EPOS com o código nativo da aplicação. O código nativo da aplicação é obtido a partir da transformação do código Java com uso de um compilador *Ahead-of-Time*. Desta forma, pode-se dividir o método nas seguintes etapas:

1. Preparação do código da aplicação: nesta etapa o código é compilado e a aplicação, juntamente com as bibliotecas de classes das quais depende, são otimizadas de forma a eliminar código desnecessário;

2. Extração das necessidades da aplicação: neste etapa, o resultado obtido com a etapa anterior é submetido à análise de necessidades, que determina quais tipos e métodos de sistema estão sendo utilizados;

3. Seleção de componentes do sistema operacional: a seleção de componentes leva em conta as necessidades da aplicação e informações a respeito da plataforma onde o sistema será executado;

4. Construção do sistema: o código isolado na primeira etapa é transformado em código nativo com o uso de um compilador *Ahead-of-Time*; este código é então ligado com o código dos componentes selecionados na etapa anterior.

Estas etapas são apresentadas mais detalhadamente nas próximas seções.

5.1.1 Preparação do Código da Aplicação

Para que seja possível manipular as informações a respeito das aplicações Java, é necessário que o código da aplicação esteja representado em um formato de fácil manipulação e que mantenha a semântica da aplicação. As informações contidas nos arquivos `class`, incluindo o código *bytecode* dos métodos, atributos diversos sobre a classe e seus membros e informações para depuração podem ser utilizadas para este fim. Por este motivo, o *bytecode* Java é utilizado como linguagem intermediária no processo de gerar um ambiente de execução sob medida.

Deste modo, a preparação da aplicação para ser submetida a este processo começa pela sua compilação. A compilação da aplicação é opcional se seu código binário estiver disponível. A compilação é feita com a utilização de qualquer compilador Java e não necessita otimizações ou cuidados especiais. Isto é possível pois tais otimizações podem ser feitas posteriormente sobre o código *bytecode*. Outro motivo para não serem feitas otimizações neste passo é que parte do código fonte da aplicação, principalmente de bibliotecas de classes de terceiros, pode não estar disponível.

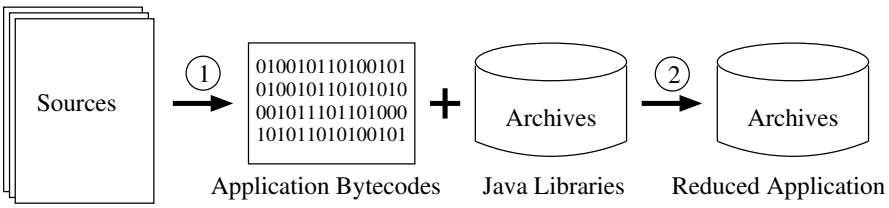

Figura 5.3: Compilação e isolamento da aplicação

A Figura 5.3 ilustra a preparação da aplicação para ser utilizada com o EPOS-J. Inicialmente, no ponto 1, o código fonte da aplicação é compilado para *bytecode*. A seguir, no ponto 2, o código Java (*bytecode*) da aplicação e as bibliotecas Java de que depende são submetidos ao processo de isolamento da aplicação. Como visto na seção 3.1, o código das classes é analisado estaticamente a procura de classes, métodos e atributos que realmente são necessários à aplicação. Reduzir o tamanho do código da aplicação é a idéia central deste passo.

Este processo reduz o tamanho do código da aplicação através da elimi-

nação do código desnecessário da aplicação e das bibliotecas da qual a aplicação depende. Além do isolamento do código da aplicação que é realmente utilizado, neste passo também podem ser feitas otimizações. Pode-se fazer uma compactação deste código usando-se técnicas de compactação de código tais como a eliminação de uma classe que só é estendida por uma única subclasse. O código isolado é armazenado em um arquivo `jar`[3] para ser utilizado nos passos posteriores.

5.1.2 Extração das Necessidades da Aplicação

A análise da aplicação em busca de suas dependências é feita sobre o resultado obtido pelo isolamento e compactação do código da aplicação. É importante destacar que este processo seja feito somente depois do código da aplicação ter sido isolado. Caso a análise para extração das necessidades da aplicação seja feita antes do isolamento de código, pode-se ter um ambiente de execução com mais recursos que a aplicação realmente necessita. O analisador deve apontar os recursos de sistema usados e também classes e métodos "de sistema", tais como as classes do pacote `java.lang`.

O método para extração de dependências da aplicação proposto neste trabalho é baseado no registro de classes, métodos e atributos utilizados pela aplicação. O mapeamento destes elementos é feito com um algoritmo que analisa as classes iniciais das aplicações e, de maneira recursiva, todos os elementos referenciados por ela. As classes iniciais das aplicações geralmente são aquelas que implementam o método cuja assinatura é "`public static void main(String[]);`". Entretanto é responsabilidade do programador determinar por quais classes, e métodos, a análise deve ser iniciada.

Esta metodologia permite a determinação de todas as classes da aplicação e bibliotecas utilizadas de maneira simples. Porém existem certos pontos a serem levados em consideração. O principal ponto é que muitas classes das bibliotecas são dependentes da máquina virtual utilizada [GOS 96].

Em um experimento realizado com as máquinas virtuais da SUN[4] e

[3] Arquivo que normalmente é utilizado para armazenar código Java.

[4] Java HotSpot(TM) Client VM (build 1.4.1_01-b01, mixed mode)

Kaffe[5], inicialmente foi determinado o conjunto total de classes necessárias para execução na máquina virtual da SUN sem a dependência das classes padrão[6]. Utilizando a máquina virtual da SUN, foi possível executar a aplicação. Utilizado o mesmo conjunto de classes na máquina virtual do *Kaffe* isso não foi possível. O problema detectado é devido o fato que certas classes dependem da implementação de métodos nativos e estes são, muitas vezes, exclusivos à máquina virtual. Assim sendo, os métodos nativos necessários à máquina virtual da SUN são diferentes daqueles necessários à máquina virtual do *Kaffe*.

Sendo assim, algumas classes não devem ser analisadas pelo algoritmo recursivo proposto, devendo considerar uma implementação padrão para classes dos pacotes `java.lang` e `java.io`, por exemplo. Desta forma é necessário construir uma biblioteca mínima necessária a partir dos métodos restritos[7]. Essa construção usa a saída da análise da aplicação e as interfaces infladas do EPOS para determinar a biblioteca Java mínima necessária para tal aplicação. Esta biblioteca mínima é construída a partir do repositório de classes de sistema do EPOS-J, que é construída a partir do GNU Classpath [Ope 03b]. Estas classes são implementadas de tal forma que sua interdependência é mínima e muitas de suas funções são realizadas por abstrações do EPOS.

A seguir, é apresentado o algoritmo utilizado para fazer a extração de necessidades da aplicação.

- 1º passo: Determinar um método a ser analisado. O início da análise é feito no(s) métodos da(s) classes de identificam a aplicação. Em Java, o método "`public static void main(String[] args)`" de qualquer classe pode ser usado como método inicial de uma aplicação. Fazendo-se a análise a partir destes métodos pode-se identificar as necessidades de determinada aplicação.

- 2º passo: Analisar o corpo do método à procura de referências a classes, métodos e atributos, armazenando-se suas assinaturas.

[5]Kaffe Virtual Machine (Engine: Just-in-time v3, Version: 1.0.7, Java Version: 1.1)

[6]Classes contidas no arquivo `rt.jar`

[7]java.lang.*, java.io.*, etc.

- 3º passo: Analisar cada assinatura armazenada recursivamente. A existência de referência a atributos de classes identifica a necessidade de analisar o construtor de cada uma destas classes.

Após a realização da análise sobre a aplicação, vários resultados podem ser conhecidos. Os resultados da análise necessários são:

- Classes da *biblioteca de sistema* necessárias. Este ítem pode conter referências às classes, métodos e campos utilizados. De forma a determinar quais classes do repositório serão utilizadas na *biblioteca de sistema* e quais recursos do ambiente de execução estas classes utilizarão. Vale lembrar que estas classes do repositório são, em sua maioria, implementadas diretamente como referência a métodos de abstrações do EPOS. São utilizações de Interfaces infladas;

- Dependências estruturais do ambiente de execução como coletor de lixo, gerenciamento de memória, locks (caso o ambiente final seja multi-threaded), etc.

Estes dados são usados nas próximas etapas da construção do sistema.

Carregamento Dinâmico de Classes

Java possui a facilidade de criar objetos a partir de referências textuais às classes. Isto é feito através do método `forName()` da classe `java.lang.Class`. A utilização deste recurso não pode ser detectada pela ferramenta de análise de dependências, uma vez que não existem referências diretas as estas classes no código. Em aplicações que utilizam este recurso devem ter analisadas, além das classes com métodos `main`, classes que são carregadas dinamicamente.

Aplicações com Métodos Nativos

Caso a aplicação tenha classes com métodos nativos, a extração de dependências também deve ser executada sobre o código dos métodos nativos com as ferramentas usuais do sistema EPOS. Deve haver uma integração entre os resultados da análise

de dependências da aplicação Java e da análise de dependências do código dos métodos nativos.

Entretanto existem problemas com a abordagem de analisar o código dos métodos nativos. Nem sempre o código fonte dos métodos nativos está disponível. A utilização da forma binária de bibliotecas que implementem os métodos nativos para determinada plataforma deve ser desconsiderada pois é muito provável que ele não seja compatível com a plataforma alvo. Mesmo de posse de código fonte dos métodos nativos, existe a possibilidade de eles não serem compatíveis com o EPOS por usarem bibliotecas não disponíveis neste sistema. Caso os métodos nativos estejam implementados com CNI e o isolamento da aplicação determinar que um método nativo é desnecessário, existe a possibilidade do código resultante não funcionar. Isto acontece pois em CNI, a estrutura da classe é toda declarada em C++. Caso exista uma divergência entre a declaração feita em C++ e a declaração em Java, o sistema não funciona.

Para resolver este problema existem duas estratégias possíveis: impedir o uso de métodos nativos na aplicação ou reescrever o código dos métodos nativos e fazer uma análise de dependências sobre este código de acordo com as regras do EPOS. Impedir o uso de nativos na aplicação é a estratégia utilizada em alguns ambientes de execução para sistemas embutidos. Esta restrição é usada para garantir a compatibilidade com demais sistemas Java e para garantir uma melhor segurança com propriedades vitais do sistema. Entretanto, esta restrição impede a utilização deste método em aplicações que possuam métodos nativos e desta forma algumas aplicações pré-existentes não poderiam ser utilizadas em conjunto com o EPOS-J. Por outro lado, reescrever ou adaptar o código dos métodos nativos para cada nova plataforma é a estratégia adotada pelos projetistas da linguagem Java e vem sendo usada com sucesso nos mais variados ambiente de execução. Por este motivo, esta é a estratégia adotada neste projeto.

A utilização de código de métodos nativos legados também é possível caso esteja disponível o código fonte dos métodos nativos usando JNI. Nesta caso basta adaptar o código de forma a garantir que todos os recursos necessários estejam disponíveis no sistema EPOS. Ao contrário de CNI, que implementa métodos nativos como métodos de uma classe em C++, em JNI a implementação dos métodos nativos é feita através

de funções independentes. Ao se fazer a extração da aplicação, métodos nativos não utilizados podem ser descartados e portanto não são incluídos na versão final do sistema.

A utilização de CNI e JNI em um mesmo sistema é possível. Ao ligar um código JNI a uma aplicação Java, o compilador GNU GCJ gera *stubs* que são usados para permitir a integração de métodos nativos escritos em JNI com o *layout* binário dos objetos utilizados.

5.1.3 Seleção de Componentes

De acordo com as necessidades detectadas na análise de dependências, são selecionadas classes da biblioteca do EPOS-J. Estas classes, por sua vez, são baseadas em componentes do EPOS. Como as classes apresentam uma implementação completa, isto é com todos os métodos previstos na API Java, deve-se selecionar os métodos que realmente foram invocados na aplicação a fim de determinar quais membros das famílias de abstrações do EPOS são mais adequados à aplicação.

Esta seleção de métodos é feita sobre o código bytecode das classes do EPOS-J. A implementação típica dos métodos de uma classe do EPOS-J é feita com o uso de métodos nativos. Cada método nativo é implementado em um arquivo separado. Isto permite fazer a seleção de métodos utilizados sobre o bytecode de classes do EPOS-J. Uma vez que as classes são customizadas à aplicação, os métodos nativos que devem ser incluídos são identificados e o código de cada um dos arquivos contendo a implementação destes métodos é adicionada à lista de código a ser compilado.

Sobre o código dos métodos nativos é feita a análise de dependências tradicional do EPOS. De acordo com os resultados desta análise, os componentes do EPOS adequados são selecionados.

5.1.4 Construção do Ambiente de Execução

O ambiente de execução final é construído a partir das necessidades da aplicação. As necessidades são mapeadas em métodos e atributos de das classes de sistema. A biblioteca de classes do sistema é construída com o uso das Interfaces Infladas

das abstrações e mediadores do sistema EPOS.

A Figura 5.4 ilustra este processo de criação do ambiente de execução, tendo como ponto de partida o resultado da aplicação isolada. Após o código ter sido preparado, a aplicação é analisada e as necessidades impostas ao ambiente de execução são determinadas. São selecionados componentes do EPOS de acordo com as necessidade da aplicação e propriedades da plataforma onde a aplicação será executada. Todo o código é compilado e ligado formando o sistema final.

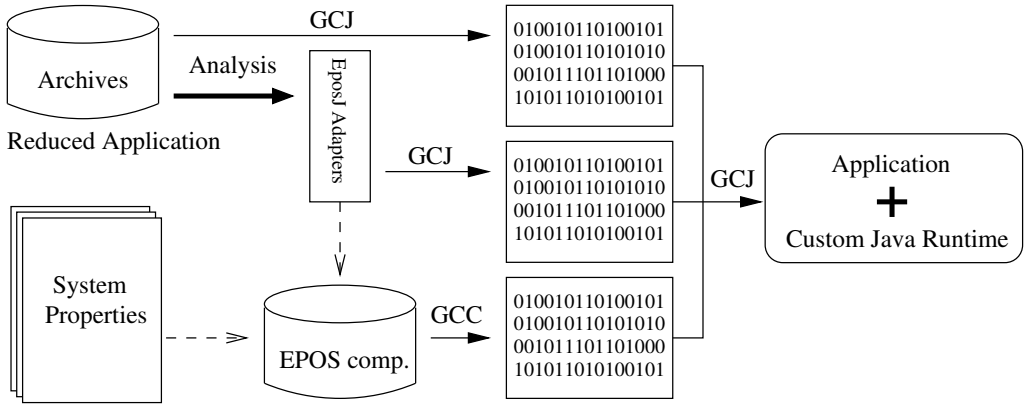

Figura 5.4: Processo de construção do sistema.

As bibliotecas de classes do EPOS-J são feitas com base nos componentes de *software* do EPOS e na biblioteca GNU Classpath, além de porções selecionadas da *libgcj*. As abstrações do EPOS são exportadas como interfaces infladas em Java. Estas interfaces são compiladas de forma a estarem disponíveis como arquivos `class`. Os métodos que devem ser implementados de cada classe da biblioteca são obtidos através da análise de métodos que foram invocados. O código fonte de cada método é colocado em um arquivo fonte distinto, de forma a possibilitar a retirada de métodos da classe original. Esta retirada de métodos é feita a partir dos resultados da análise de dependências da aplicação. Um programa é utilizado para compilar apenas os métodos identificados como necessários. Todo o código, incluindo aplicação, ambiente de execução e classes do EPOS-J são ligadas para gerar a aplicação.

5.2 Implementação

Seguindo o método apresentado anteriormente, a implementação do EPOS-J é feita da seguinte forma: As ferramentas de extração e análise das necessidades da aplicação são construídas com o uso da biblioteca BCEL [DAH 01]. Uso de GNU Classpath para implementação da biblioteca de classes Java do EPOS-J. A biblioteca do EPOS-J deve ser mínima, entretanto deve conter todas as classes que são necessárias e que dependam do ambiente de execução. Uso de GCJ para transformar código bytecode em código nativo.

5.2.1 Preparação do Código da Aplicação

Tanto a compilação quanto o isolamento do código da aplicação podem ser feitos através do uso de qualquer ferramenta para este fim que esteja disponível. Entretanto teste trabalho foi utilizada uma ferramenta para o isolamento do código implementada durante os testes com protótipos. A ferramenta de isolamento implementada é bastante simples e não muitos recursos de otimização de código. Ferramentas comerciais tais como WebSphere SmartLinker [IBM 04], devem ser usadas para garantir um melhor aproveitamento desta metodologia.

A ferramenta implementada neste projeto é a mesma ferramenta utilizada para fazer a análise de dependências da aplicação. Além de sua tarefa principal, ela também isola o código das classes utilizadas pela aplicação e elimina informações para depuração dos arquivos `class`. As classes isoladas são então armazenadas em um arquivo `jar` para serem utilizadas na construção do sistema.

5.2.2 Extração das Necessidades da Aplicação

O protótipo da ferramenta de análise de dependências e isolamento da aplicação usa a biblioteca BCEL [DAH 01] para manipular informações contidas em arquivos de classes de aplicações escritas em Java.

BCEL, *Byte Code Engineering Library*[8], é uma biblioteca de classes escrita em Java que possibilita a criação e manipulação de arquivos de classe e que contém código relativo à classe em forma compilada, isto é, em *bytecode* [DAH 01]. A biblioteca BCEL pode ser utilizada para analisar, criar ou transformar dinamicamente código binário Java (*bytecode*). Entre as muitas aplicações possíveis pode-se destacar otimizadores de *bytecode*, compiladores e implementações dos conceitos de *Programação Orientada a Aspectos* e de meta programação [DAH 99]. Esta biblioteca permite, ler, criar, remover e redefinir métodos de uma determinada classe. Permite também alterar suas estruturas internas e remover informações específicas para depuração. Este tipo de informação não é adequado em um sistema embutido em produção.

Analisador de Dependências

Para verificar como poderia ser feito um analisador de dependências para aplicações escritas em Java, foi desenvolvido um protótipo. Este protótipo foi construído com o auxílio da biblioteca BCEL. O protótipo foi construído de forma a analisar recursivamente todas as classes utilizadas pela aplicação.

O protótipo implementado executa duas atividades:

- Determina as classes de sistema utilizadas pela aplicação através das referências contidas nas classes analisadas;
- Constrói uma nova biblioteca com as classes necessárias para a execução da aplicação analisada.

O protótipo desenvolvido deve ser executado em um ambiente com as mesmas bibliotecas de classes que estão disponíveis à aplicação durante sua execução. O analisador deve conhecer e não verificar as interdependências da biblioteca de sistema utilizada. Ele deve apenas apontar as dependências da aplicação em relação à biblioteca de sistema. Isto somente é necessário pois o protótipo desenvolvido executa simultaneamente a análise de dependências e o isolamento da aplicação.

[8] Inicialmente o nome desta biblioteca era JavaClass, nome usado em [DAH 99].

A primeira versão do protótipo implementado fazia uma análise muito superficial, fazendo a análise apenas das classes utilizadas, não determinando quais métodos eram utilizados. Nesta versão, a análise era feita inclusive sobre as classes que são dependentes de implementação da máquina virtual. Desta forma, o resultado obtido é o conjunto de classes utilizadas para execução da aplicação na máquina virtual onde a análise foi realizada.

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello World!");
        }
}
```

Exemplo 5.1: Código analisado – “Hello World!”.

A versão inicial do protótipo, ao analisar a aplicação mostrada no Exemplo 5.1, na máquina virtual incluída no pacote J2SE[9], obteve a surpreendente número de 1279 classes referenciadas. É claro, nem todas estas classes são realmente utilizadas na execução da aplicação. Este número é grande devido ao escopo, a classe inteira, utilizado na análise.

Posteriormente, com o incremento do algoritmo de análise, melhores resultados foram obtidos na análise da mesma aplicação. A Tabela 5.1 mostra os resultados obtidos na análise da aplicação “Hello World!” com cada uma das versões do protótipo implementadas. Como esperado, a versão mais recente do protótipo teve melhor resultado, identificando apenas 12 classes de sistema.

Exemplo de Utilização

O Exemplo 5.1 mostra o código de uma aplicação muito simples que apenas escreve uma frase no dispositivo padrão para saída de texto. Esta aplicação simples, compilada para código nativo e com todo o suporte da `libgcj` ligado estaticamente, ocupa 18MB de disco. O motivo deste tamanho é que o suporte *runtime* do GCJ inclui

[9] Java HotSpot(TM) Client VM (build 1.4.1_01-b01, mixed mode)

Versão do protótipo	classes	métodos	atributos
0.1	1279	–	–
0.2	157	491	309
0.3	12	22	14

Tabela 5.1: Resultados obtidos com análise da aplicação “Hello World!”.

um interpretador de *bytecodes*, coletor de lixo, e toda a biblioteca de classes Java mesmo que algumas destas características não sejam necessárias à aplicação.

Ao se executar o isolamento do código da aplicação pode-se notar que classes que o número de classes realmente utilizadas é reduzido, entretanto ainda existem mais classes que um programa feito em C/C++ utilizaria.

No Exemplo 5.2 é apresentada a listagem das classes, métodos e atributos obtidas com o uso do analisador em uma aplicação simples[10]. De acordo com o analisador, foram encontradas 12 classes necessárias a esta aplicação. Destas classes, somente 22 métodos e 14 atributos realmente são utilizados.

5.2.3 Bibliotecas Java do EPOS

Para que programas Java possam ser executados no EPOS é definida uma biblioteca de classes mínima. Esta biblioteca é composta pelas classes básicas, dos pacotes `java.lang` e `java.io` e por classes que sejam dependentes do ambiente de execução tais como classes para segurança, reflexão, etc.

A biblioteca Java do EPOS é construída com base na GNU Classpath. Parte dos pacotes `java.lang` e `java.io` é reimplementada, de forma que as classes sejam definidas com base em componentes do EPOS. Dependendo das necessidades da aplicação, classes como `java.lang.Object` poderão ter retirados seus atributos e métodos desnecessários. Para uma aplicação com apenas uma *thread*, por exemplo, pode-se remover os métodos `wait()`, `notify()` e `notifyAll()` da classe

[10] Aplicação apresentada no Exemplo 5.1.

```
HelloWorld
        public static void main(String[] arg0)
java.io.BufferedWriter
        private char[] cb
        private void ensureOpen()
                throws java.io.IOException
        void flushBuffer()
                throws java.io.IOException
        private java.lang.String lineSeparator
        protected java.lang.Object lock
        public void newLine()
                throws java.io.IOException
        private int nextChar
        private java.io.Writer out
java.io.IOException
        public void <init>(String arg0)
java.io.OutputStream
        public void flush()
                throws java.io.IOException
java.io.OutputStreamWriter
        void flushBuffer()
                throws java.io.IOException
        private final sun.nio.cs.StreamEncoder se
java.io.PrintStream
        private boolean autoFlush
        private java.io.OutputStreamWriter charOut
        private void ensureOpen()
                throws java.io.IOException
        private void newLine()
        protected java.io.OutputStream out
        public void print(String arg0)
        public void println(String arg0)
        private java.io.BufferedWriter textOut
        private boolean trouble
        private void write(String arg0)
java.io.Writer
        protected java.lang.Object lock
        public void write(String arg0)
                throws java.io.IOException
        public void write(String arg0, int arg1, int arg2)
                throws java.io.IOException
        public abstract void write(char[], int, int)
                throws java.io.IOException
        private char[] writeBuffer
java.lang.Exception
        public void <init>(String arg0)
java.lang.String
        public void getChars(int arg0, int arg1, char[] arg2, int arg3)
        public int indexOf(int arg0)
        public int length()
java.lang.System
        public static final java.io.PrintStream out
java.lang.Thread
        public static native Thread currentThread()
        public void interrupt()
sun.nio.cs.StreamEncoder
        public void flushBuffer()
                throws java.io.IOException
```

Exemplo 5.2: Classes, métodos e atributos utilizados no “Hello World!”.

`java.lang.Object`.

Como exemplo pode-se usar a classe `java.lang.Thread`. Em Java, a classe `java.lang.Thread` implementa a interface `Runnable`. Além disso, qualquer *thread* em Java pressupõe a execução do método `run()` de um objeto que implemente a interface `Runnable`.

```
public class Thread implements Runnable {
   public Thread(Runnable target) {
      this.target = target;
   }
   public native void run();
   public native void start();
   public native void suspend();
   public native void resume();
   private Runnable target;
   private gnu.gcj.RawData id;
}
```

Exemplo 5.3: Implementação da classe `java.lang.Thread`. Foram omitidos alguns métodos para permitir brevidade.

Para que seja feito um mapeamento entre a classe `java.lang.Thread` e a abstração `Thread` do EPOS pode-se utilizar métodos nativos CNI. Desta forma, pode-se implementar a classe `java.lang.Thread` da seguinte maneira: Inicialmente, define-se uma classe em Java com a maioria de seus métodos nativos. O Exemplo 5.3 mostra a declaração da classe `java.lang.Thread`.

A seguir, implementa-se os métodos nativos com o uso de CNI, como mostrado no exemplo 5.4. Neste exemplo, a abstração de `Thread` do EPOS método para ser executado em um novo fluxo de execução. Note que a implementação é feita, basicamente, através de invocações de métodos das interface inflada da família de abstrações *Thread*.

A implementação dos métodos nativos pode também empregar técni-

```
#include "Thread.h"
#include <gcj/cni.h>
#include <thread.h>
#include <framework.h>

void Thread::run () {
   target->run();
}

int glue(::java::lang::Runnable *obj) {
   obj->run();
}

void Thread::start () {
   id = (::gnu::gcj::RawData*)
     new System::Thread<::java::lang::Runnable *>(glue,target);
}

void Thread::suspend () {
   ((System::Thread<::java::lang::Runnable *>)id)->suspend();
}

void Thread::resume () {
   ((System::Thread<::java::lang::Runnable *>)id)->resume();
}
```

Exemplo 5.4: Implementação dos métodos nativos da classe definidos no exemplo 5.3 usando-se CNI. A classe `System::Thread` referenciada no método `start()` pertence à API do EPOS.

cas mais sofisticadas de engenharia de *software*. A programação orientada a aspectos [KIC 97], por exemplo, pode ser usada definir somente os métodos nativos que são realmente utilizados pela aplicação.

5.2.4 Montagem do Sistema

Usando as técnicas de engenharia de *software* utilizadas no EPOS, deve-se projetar um ambiente de execução Java a partir das reais necessidades da aplicação. Estas necessidades são obtidas a partir da análise da aplicação e bibliotecas utilizadas. O ambiente de execução também pode ter algumas necessidades intrínsecas como gerência de memória que não podem ser desprezadas.

De posse dos dados obtidos com a análise da aplicação, pode-se partir para determinação do ambiente de execução necessário para executar a aplicação em questão. Deve-se montar um ambiente de execução mínimo para execução desta aplicação.

Finalmente o sistema completo, composto pela aplicação, bibliotecas, ambiente de execução e sistema operacional, pode ser carregado no sistema embutido e executado.

5.3 Resultados Obtidos

Capítulo 6

Conclusão

Para executar programas Java em um sistema embutido com grandes restrições de hardware é necessário um ambiente de execução específico para este fim. Este ambiente deve fornecer todos os serviços necessários à aplicação, ser compacto e ter um bom desempenho.

Projetos como J2ME e JPURE propõem ambientes de execuções para sistemas embutidos altamente configuráveis. Entretanto esta configuração deve ser feita pelo projetista do sistema e está sujeita a erros. Em um cenário onde as aplicações de um sistema embutido são bem conhecidas existe uma alternativa a configuração manual. A utilização de *Application-Oriented System Design* no desenvolvimento de ambientes de execução Java permite fornecer um sistema sob medida para cada aplicação. A necessidades da aplicação são extraídas e utilizadas, juntamente com as informações de configuração da plataforma alvo, para a seleção dos componentes que serão utilizados no ambiente de execução. O sistema EPOS pode ser usado como sistema de suporte ao ambiente de execução.

A transformação das aplicações Java para código nativo aumenta o desempenho da aplicação, além de dispensar a necessidade de um interpretador no ambiente de execução. Esta transformação pode ser feita com compiladores *Ahead-of-Time*, como é o caso do GNU GCJ . Entretanto, não se deve ligar o código gerado à biblioteca `libgcj`, uma vez que esta implementação depende da existência de uma interface POSIX para o

ambiente de execução. A necessidade de uma interface POSIX aumenta sensivelmente o tamanho final do sistema. Por este motivo, uma biblioteca Java modular deve ser construída para substituir a biblioteca `libgcj`. Esta biblioteca pode ser baseada na GNU Classpath, sendo necessário apenas adaptá-la para a utilização no EPOS. Ao se transformar o código Java para código nativo perde-se a possibilidade de fazer carga de novas classes via rede. Entretanto, caso esta característica seja necessária, pode-se incluir um interpretador de *bytecodes* no ambiente de execução. Esta abordagem acarreta no aumento de recursos utilizados pelo sistema e na diminuição do desempenho da aplicação.

A análise das necessidades das aplicações escritas em Java pode ser feita sobre seu código compilado, *bytecode*. A estrutura binária do arquivo de classes permite um acesso fácil às informações sobre classes e métodos utilizados. Além disso, uma vez que foram identificadas as classes e métodos que sistema utilizados, a biblioteca de classes Java do EPOS pode ser modificada de forma a somente apresentar as classes e os métodos necessários. Para fazer isto, deve-se escrever o código nativo de cada um dos métodos das classes Java do sistema em um arquivo diferente. Isto possibilita a utilização de CNI para implementar os métodos nativos das classes de sistema.

A utilização destas técnicas em conjunto permite desenvolver um ambiente de execução Java sob medida para uma determinada aplicação ou conjunto de aplicações. Como as necessidades da aplicação são extraídas automaticamente, a possibilidade de erros na configuração do sistema é diminuída. Isso possibilita a eliminação de recursos do sistema que não seriam utilizados pela aplicação. Desta forma, pode-se construir um ambiente de execução ideal para uma determinada aplicação.

Referências Bibliográficas

[BAR 02] BARCELLOS, M. P. Programação paralela e distribuída em java. In: ERAD 2002 – ESCOLA REGIONAL DE ALTO DESEMPENHO, 2002. **Proceedings...** São Leopoldo, RS, Brasil: SBC, 2002. p.181–192.

[BEU 99] BEUCHE, D. et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In: PROCEEDINGS OF THE 2ND IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 1999. **Proceedings...** St Malo, France: [s.n.], 1999.

[BEU 00] BEUCHE, D. et al. JPURE – a purified java execution environment for controller networks. In: DIPES, 2000. **Proceedings...** Schloß Eringerfeld, Germany: Kluwer, 2000. v.189 of **IFIP Conference Proceedings**, p.65–74.

[BOG 99] BOGDA, J.; HöLZLE, U. Removing unnecessary synchronization in java. In: PROCEEDINGS OF THE 14TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 1999. **Proceedings...** Denver, Colorado, United States: ACM Press, 1999. p.35–46.

[BOT 97] BOTHNER, P. **Compiling Java for Embedded Systems**. Outubro, 1997. Disponível em <http://gcc.gnu.org/java/papers/compjava.pdf>.

[BOT 03] BOTHNER, P. Compiling Java with GCJ. **Linux Journal**, [S.l.], v.2003, n.105, p.4, Janeiro, 2003.

[CAR 97] CARPENTER, B. et al. Experiments with HP Java. **Concurrency: Practice and Experience**, [S.l.], v.9, n.6, p.633–648, Junho, 1997.

[CIE 02] CIERNIAK, M.; LEWIS, B. T.; STICHNOTH, J. M. Open runtime platform: flexibility with performance using interfaces. In: PROCEEDINGS OF THE 2002 JOINT ACM-ISCOPE CONFERENCE ON JAVA GRANDE, 2002. **Proceedings...** Seattle, Washington, USA: ACM Press, 2002. p.156–164.

[Cyg 99] Cygnus Solutions. **The Cygnus Native Interface for C++/Java Integration**. Fevereiro, 1999. Disponível em <http://www.gnu.org/software/gcc/java/papers/cni/t1.html>.

[DAH 99] DAHM, M. Byte code engineering. In: JAVA-INFORMATIONS-TAGE, 1999. **Proceedings...** Düsseldorf: [s.n.], 1999. p.267–277.

[DAH 01] DAHM, M. Byte Code Engineering with the BCEL API. Freie Universit at Berlin, Abril, 2001. Relatório TécnicoB-17-98.

[De 03] De Bus, B. et al. Post-pass compaction techniques. **Communications of the ACM**, [S.l.], v.46, n.8, p.41–46, Agosto, 2003.

[FRö 99] FRöHLICH, A. A. M.; SCHRöDER-PREIKSCHAT, W. High performance application-oriented operating systems – the EPOS aproach. In: PROCEEDINGS OF THE 11TH SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 1999. **Proceedings...** Natal, Brazil: [s.n.], 1999. p.3–9.

[FRö 01] FRöHLICH, A. A. M. **Application-Oriented Operating Systems**. Sankt Augustin, Germany: GMD - Forschungszentrum Informationstechnik GmbH, Agosto, 2001. Tese de Doutorado.

[GAG 00] GAGNON, E. M.; HENDREN, L. J. SableVM: A research framework for the efficient execution of java bytecode. McGill University, Novembro, 2000. 27–40 p. Relatório Técnico2000-3.

[GAG 02] GAGNON, E. **A Portable Research Framework for the Execution of JAVA Bytecode**. School of Computer Science - McGill University, Montreal, Dezembro, 2002. Tese de Doutorado.

[GAM 95] GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[GOL 02] GOLM, M. et al. The jx operating system. In: 2002 USENIX ANNUAL TECHNICAL CONFERENCE, 2002. **Proceedings...** Monterey, California, USA: USENIX, 2002. p.45–58.

[GOS 96] GOSLING, J.; JOY, B.; STEELE, G. **The JAVA Language Specification**. Addison-Wesley, Agosto, 1996.

[HOA 74] HOARE, C. A. R. Monitors: An Operating System Structuring Concept. **Communications of the ACM**, [S.l.], v.17, n.10, p.549–557, Outubro, 1974.

[IBM 04] IBM. **WebSphere SmartLinker**. Disponível em <http://www-306.ibm.com/software/wireless/wsdd/>. Acesso em: Janeiro.

[JOH 97] JOHNSON, R. E. Frameworks = (components + patterns). **Communications of the ACM**, [S.l.], v.40, n.10, p.39–42, Outubro, 1997.

[KIC 97] KICZALES, G. et al. Aspect-Oriented Programming. In: PROCEEDINGS OF THE EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING'97, 1997. **Proceedings...** Jyväskylä, Finland: Springer, 1997. v.1241 of **Lecture Notes in Computer Science**, p.220–242.

[LIA 99] LIANG, S. **The Java Native Interface**. Addison-Wesley, Junho, 1999.

[LIN 99] LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. 2. ed. Addison-Wesley, 1999.

[MUL 97] MULLER, G. et al. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In: COOTS, 1997. **Proceedings...** Portland, Oregon, USA.: USENIX, 1997. p.1–20.

[Ope 03a] Open Source. **GCJ Home Page**. Disponível em <http://gcc.gnu.org/java>. Acesso em: Novembro.

[Ope 03b] Open Source. **GNU Classpath Home Page**. Disponível em <http://www.gnu.org/software/classpath/classpath.html>. Acesso em: Dezembro.

[Ope 03c] Open Source. **Kaffe Home Page**. Disponível em <http://www.kaffe.org>. Acesso em: Dezembro.

[PAR 76] PARNAS, D. L. On the Design and Development of Program Families. **IEEE Transactions on Software Engineering**, [S.l.], v.SE-2, n.1, p.1–9, Março, 1976.

[PRO 97] PROEBSTING, T. A. et al. Toba: Java for applications – a way ahead of time (wat) compiler. In: COOTS, 1997. **Proceedings...** Portland, Oregon, USA.: USENIX, 1997. p.41–54.

[Sun 00] Sun Microsystems. Java(tm) 2 platform micro edition (J2ME(tm)) technology for creating mobile devices. Sun Microsystems, Maio, 2000. Relatório técnico.

[Sun 02a] Sun Microsystems. **J2ME CLDC Reference Implementation – Version 1.0.4**, Outubro, 2002.

[Sun 02b] Sun Microsystems. **K Native Interface (KNI)**, Outubro, 2002.

[Sun 04] Sun Microsystems. **Java Card Platform**. Disponível em <http://java.sun.com/products/javacard/>. Acesso em: Janeiro.

[TEN 00] TENNENHOUSE, D. Proactive computing. **Communications of the ACM**, [S.l.], v.43, n.5, p.43–50, 2000.

[TIP 99] TIP, F. et al. Practical experience with an application extractor for java. In: PROCEEDINGS OF THE 14TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED

PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 1999. **Proceedings...** Denver, Colorado, United States: ACM Press, 1999. p.292–305.

[TIP 03] TIP, F.; SWEENEY, P. F.; LAFFRA, C. Extracting library-based java applications. **Communications of the ACM**, [S.l.], v.46, n.8, p.35–40, Agosto, 2003.

[WOL 01] WOLF, W. H. **Computers as components: priciples of embedded computing system design**. CA, USA: Morgan Kaufmann Publishers, 2001.