Compilador Reflexivo Para Adaptação de Programas a Configurações de Recursos Limitados

Sérgio Ribeiro Libório e Roberto da Silva Bigonha

Depto. de Ciência da Computação - Universidade Federal de Minas Gerais

{liborio, bigonha}@dcc.ufmg.br

Abstract. This paper presents a technique for developing a framework to provide the J2ME platform with CLDC configuration advanced reflective capabilities. It also describes a simple and efficient mechanism to automate the proposed reflection framework by means of a compiler that uses advanced reflection capabilities to carry out Java class adaptation of Java class to given execution contexts. A collateral product of this work is the implementation of serialization and objects mobility in CLDC configurations.

Resumo. Este artigo propõe uma técnica para desenvolver um arcabouço (*framework*) para disponibilizar ao ambiente J2ME/CLDC capacidades avançadas de reflexão, além de descrever um mecanismo simples e eficiente para automatizar a implementação de um compilador que utiliza capacidades avançadas de reflexão para operar adaptação em classes Java. Um subproduto deste trabalho é a possibilidade de serialização e mobilidade de objetos em ambientes com a configuração CLDC.

1. Introdução

Os recentes avanços nas áreas de hardware, telecomunicações e redes de computadores estão transformando em realidade a noção de computação móvel [Var00, Rob98], que por sua vez eleva os atuais dispositivos computacionais móveis, como laptops, assistentes pessoais digitais (PDAs), telefones celulares e pagers, a elevados níveis de popularidade. Estima-se, por exemplo, que a partir de 2005 o número de telefones celulares conectados à Internet será maior que o número de computadores pessoais [Sta01].

Além da liberdade que dispositivos dessa natureza oferecem a seus usuários, aos quais é permitido o acesso a serviços e informações em qualquer lugar e a qualquer hora, há também a maior capacidade computacional oferecida. Celulares, por exemplo, têm recebido ganhos em termos de capacidade de processamento, sendo capazes de embutir uma Máquina Virtual disponibilizada em conjunto com uma versão simplificada da linguagem Java, ambas customizadas para dispositivos móveis. Customização essa, chamada de Java 2 Micro Edition (J2ME) [Rig01], correspondendo a um subconjunto das bibliotecas de Java.

Apesar dessas iniciativas, as tecnologias de software, se comparadas às tecnologias de hardware e telecomunicações, mostram-se muito aquém do requerido atualmente pelo mercado para suportar este novo paradigma de computação [Rom00].

2. Migração Para Configurações Limitadas

A baixa disponibilidade de certos recursos, notadamente memória disponível, nos atuais dispositivos móveis levou a uma redução da Máquina Virtual Java (JVM) para a então chamada Kilobyte Virtual Machine (KVM), por meio da eliminação de funcionalidades da API (*Application Programming Interface*) de Java. Outras funcionalidades da API de Java foram também removidas não apenas por questões de indisponibilidade de memória, mas também por questões de segurança. Inclui-se aqui a reflexão, que permite a programas ter acesso livre a atributos, inclusive os privados, de classes.

Sem pode contar com reflexão computacional, importantes funcionalidades da plataforma J2SE {Nic00] não podem mais ser utilizadas em dispositivos móveis, como, por exemplo, chamada de métodos remotos (RMI) e serialização de objetos. Aplicativos que utilizem CLDC (*Connected Limited Device Configuration*), que é uma especificação voltada para dispositivos extremamente restritos em termos de memória, largura de banda e segurança [Rig01], não podem requerer recursos avançados com capacidades reflexivas. Por exemplo a implementação proposta do sistema Jamp [Val02] é inadequada para dispositivos móveis com baixos recursos que utilizam J2ME com a configuração CLDC / MIDP (*Mobile Information Device Profile*). Isto deve-se à inexistência de vários dos recursos (e.g.: reflexão, serialização, *classloader*, *dynamic proxy classes*) [Arn00] que são essenciais para a implementação de mobilidade dos *contêineres* de Jamp, que são grupos de objetos que podem ser transferidos autonomamente uma para outra máquina da rede.

O presente trabalho apresenta alternativas para os problemas da probição de uso de recursos para reflexão e serialização em aplicações que devem migrar para o ambiente móvel utilizando J2ME. O foco aqui é dado, principalmente, à reflexão e à serialização em J2ME, mais especificamente na configuração CLDC. As soluções descritas no decorrer deste trabalho mostram como resolver um conjunto de problemas oriundos da remoção de diversas funcionalidades da API de Java na especificação CLDC, imposta para possibilitar a instalação de uma Máquina Virtual Java em dispositivos móveis com grandes restrições de recursos e segurança

Reflexão computacional é o principal recurso que pode ser utilizado para auxiliar ou prover outros. Por exemplo, serialização pode ser implementada a partir de uma rotina externa que utilize as capacidades reflexivas de uma classe para examinar seus atributos, e, assim,construir uma representação serializada da mesma.

No entanto, para viabiliza o uso de reflexão em J2ME, é necessário que as classes disponibilizem informações a seu respeito (meta-dados), implementando um protocolo reconhecido por rotinas externas, por exemplo, o de serialização. O protocolo pode ser representado por um conjunto de classes abstratas que proverão características reflexivas em tempo de execução para as subclasses que as implementem.

O presente trabalho propõe um arcabouço (*framework*) que oferece as facilidades de reflexão computacional em J2ME com a configuração CLDC. Seção 2 detalha um conjunto de classes e interfaces a ser utilizado a partir de uma biblioteca em formato *jar*, cujo tamanho é da ordem de 10 KB, e que deverá estar disponível em tempo de execução no ambiente das aplicações que desejam fazer uso das capacidades reflexivas de determinadas classes.

As classes, disponibilizadas como uma API, utilizam um mapeamento baseado nas interfaces do Arcabouço que deverão ser implementadas. As interfaces permitem tornar compatível uma prévia implementação escrita em Java utilizando reflexão e a implementação necessária para que as capacidades reflexivas sejam portadas para um ambiente onde não são disponibilizadas. A seguir, será mostrado como o uso de uma biblioteca especial e a implementação de determinadas interfaces e métodos das classes que se desejam como portadoras de habilidades reflexivas possibilitam a sua utilização em um ambiente desprovido de recursos para reflexão computacional.

Não obstante seja esta solução interessante, a implementação manual de interfaces e métodos ainda representa um esforço tedioso, sujeito a erros e quase sempre repetível com pequenas variações, mas passível de se automatizado. Os benéficos desta automatização seriam mais eficiência, menos erros, redução dos custos de depuração e mais rapidez para a migração de aplicações, permitindo ao desenvolvedor se concentrar mais propriamente na solução de negócio que nos emaranhados complexos do ambiente.

A contribuição principal deste trabalho é a validação de um mecanismo para automatizar as soluções aqui descritas e detalhadas nas Seções 2 e 3. A automatização é realizada por um compilador baseado em técnicas de reflexão computacional, escrito em Java, que utiliza reflexão para analisar a estrutura de classes e assim gerar as implementações de interfaces e métodos necessárias às mesmas. A idéia básica é utilizar reflexão em um ambiente onde isso é permitido, por exemplo, J2SE, para criar em tempo de compilação toda a estrutura necessária para disponibilizar reflexão em tempo de execução, agora, em um ambiente do tipo J2ME com CLDC.

O compilador reflexivo, descrito na Seção 4, permite criar classes, que, se transportadas para o ambiente J2ME, consigam ainda usufruir as capacidades reflexivas que arcabouço proposto torna disponível às mesmas, sendo para isso suficiente a presença de uma biblioteca especial de tamanho compatível com as limitações de dispositivos móveis configurados com CLDC na plataforma J2ME.

3. O Arcabouço

3.1 Estrutura

O arcabouço compõe-se de um conjunto de interfaces e das classes que as implementam (Figura 1), provendo recursos para manipulação de informação das classes. A interface **Clazz** é implementada pela classe **Jclazz**, cujos métodos permitem acessar atributos, construtores e métodos de classes que implementem o Arcabouço. As classes **JField**, **JConstructor** e **JMethod** estendem a classe **JObject** e implementam, respectivamente, as interfaces **Field**, **Constructor** e **Method**, as quais permitem compatibilizar o presente arcabouço com a API de reflexão de J2SE.

Os métodos da **interface Reflectable**, que devem ser implementados pelas classes que se desejam portadoras de capacidades reflexivas, representam primitivas que permitem às classes **JField**, **JConstructor** e **JMethod** criar métodos mais específicos e mais práticos através de variações no seu uso. Pode-se fazer uma analogia entre tais métodos primitivos e micro-instruções de um processador que não são utilizadas diretamente, mas que permitem a construção de instruções de mais alto nível.

Note-se, também, que, embora as classes devam implementar tais métodos, o controle sobre a utilização dos mesmos deve pertencer única e exclusivamente às classes do Arcabouço. Isto é, as classes que implementam a interface Reflectable podem acessar tais primitivas. No entanto, não faz sentido para as mesmas uma utilização descontextualizada, pois apenas o Arcabouço possui as informações necessárias para executar tais métodos de forma coerente e que produza o resultado desejado. Dessa forma, a interface Reflectable é a única dentre as aqui apresentadas que não possui uma implementação pronta, devendo, assim, ser realizada para cada classe que utilize os recursos do Arcabouço. A interface Reflectable representa o elo entre as classes que desejam adquirir capacidades reflexivas e o Arcabouço.

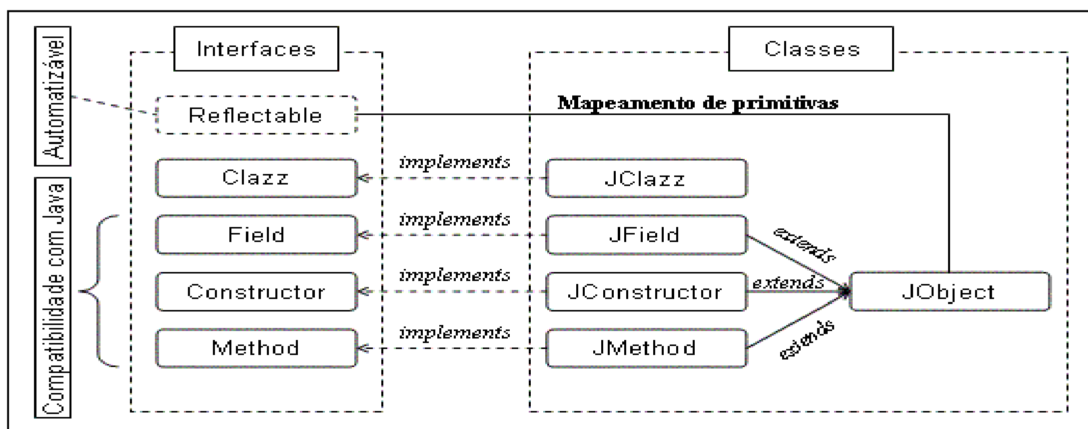

Figura 1. Framework para reflexão em J2ME

3.2 Interfaces do Arcabouço

A interface **Clazz** (Programa 1) define o acesso a atributos, construtores e métodos das classes reflexivas.

```
public class Clazz
   public  Field getField(String name);
   public  Field  [ ]getFields( );
   public  Constructor getConstructor(Class parameterTypes[]);
   public  Constructor[ ] getConstructors() ;
   public  Method getMethod(Class parameterTypes[]);
   public  Method[ ] getMethods() ;
}
```

Programa 1: Interface Clazz

A interface **Reflectable** (Programa 2) possui métodos para recuperação e alteração de objetos (get e set), e também para criação de instâncias de classes (**newInstance**) e invocação de seus métodos (invoke). Os métodos para recuperação e alteração de objetos são utilizados pela classe que implementa a interface **Field** (Programa 3). O

método para criação de instâncias de classes é útil para a classe que implementa a interface **Constructor** (Programa 4) e o método para invocação de métodos é para a classe que implementa a interface **Method** (Programa 5).

```
public interface Reflectable {
    public Object get(int position);
    public void set(int pos, Object value);
    public Object newInstance(int pos, Object[ ] initargs);
    public Object invoke(int pos, Object[ ] args);
    public Clazz getClazz ( );
    public int getFieldsLength( );
    public int getConstructorsLength( );
   public int getMethodsLength( );
}
```

Programa 2: Interface Reflectable

```
public interface Field {
    public Class getType( );
    public Boolean equals(Object obj);
    public Object get(Object obj);
    public void set(Object obj, Object value);
    public boolean getBoolen(Object obj);
    public byte getByte(Object obj);
    public byte getChar(Object obj);
    public byte getInt(Object obj);
    public byte getLong(Object obj);
    public byte getShort(Object obj);
    public void setBoolen(Object obj);
    public void setByte(Object obj);
    public void setChar(Object obj);
   public void setInt(Object obj);
   public void setLong(Object obj);
   public void setShort(Object obj);
}
```

Programa 3: Interface Field

3.3 Utilização do Arcabouço

Na utilização do arcabouço, deve-se observar os seguintes pontos importantes: (i) é necessário implementar a interface **Reflectable** para cada classe que se deseje reflexiva;

(ii) o uso das interfaces e métodos é bastante semelhante ao conhecido para a API reflexiva padrão de Java; (iii) é permitida a reutilização de implementações já escritas para a plataforma J2SE. Contudo devem ser feitas as ressalvas de que onde for utilizada a classe **Class** para obtenção de **Field**, **Constructor** e **Method**, deve-se substituí-la por **Clazz**, e onde for utilizado o método **Getclass**, deve-se substituí-lo por **getClazz**.

```
public interface Constructor {
  public Class[ ] getParameterTypes( );
  public Object newInstance(Object[ ] initargs);
}
```

Programa 4: Interface Constructor

```
public interface Method {
  public Class[ ] getParameterTypes( );
  public Class getReturnType( );
  public Object invoke(Object obj, Object [ ] args);
}
```

Programa 5: Interface Method

4. Serialização em J2ME

4.1 Visão Geral

A interface **ByteStream** é implementada pela classe **JByteStream** (Figura 2), representando a classe responsável pela tarefa de "empacotamento", i.e., conversão de determinados objetos e tipos primitivos em bytes. A interface **Marshalizable** deve ser implementada pelas classes que utilizam serialização, podendo tal tarefa ser automatizada, como será visto na Seção 4.

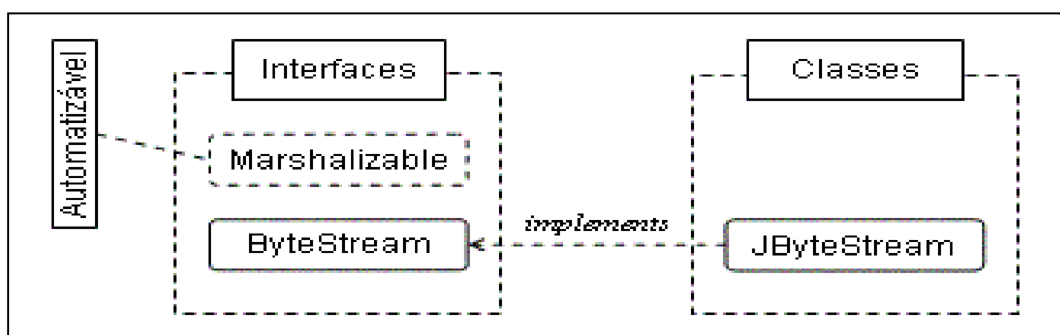

Figura 2: Interfaces e classes para serialização em J2ME

Para tornar serialização disponível no ambiente J2ME, deve-se implementar as interfaces ByteStream e Marshalizable do pacote Midleware (Programa 6).

```
package middleware;
public interface ByteStream {
  public void write(Object o);
  public Object read(Object o);
  public void setBytes(byte
bytes[]);
```

```
package middleware;
public interface Marshalizable {
  public void marshal(ByteStream
bs);
  public void
unmarshal(ByteStream bs);
```

Programa 6: Interfaces de classes de serialização

4.2 Serialização Como um Cliente da Reflexão

Serialização pode ser implementada a partir de uma rotina externa que utiliza as capacidades reflexivas de uma classe para examinar seus atributos e, assim, formar sua representação serializada. Essa solução retira a complexidade de implementação da serialização das classes e a centraliza em uma rotina independente, facilitando manutenção e depuração. Entretanto, para que isso seja possível em J2ME com CLDC é necessário o uso do arcabouço reflexivo proposto na Seção 2. Para ilustrar o problema, considere-se a necessidade de serialização de uma lista de objetos:

```
serializarLista(listaDeObjetos) {
   para cada objeto da listaDeObjetos faça  serializarObjeto(objeto);
}
```

Programa 7: Serializador de Lista de Objetos

Para simplificar o algoritmo, cada objeto da lista é, então, serializado com uma chamada ao procedimento serializarObjeto, definido da seguinte forma:

```
serializarObjeto(objeto) {
  para cada atributo do objeto faça
      se o atributo for empacotável  então empacotar(atributo);
      senão serializarObjeto(atributo);
}
```

Programa 8: Serializador de Objetos

O método **serializarObjeto** obtém todos os atributos de um dado objeto, verifica e empacota cada um deles. Se um atributo for *empacotável,* i.e., for do tipo primitivo ou puder ser transformado em um conjunto de bytes, utiliza-se o procedimento **empacotar** diretamente. Caso contrário, o atributo, que é um objeto, deve ser serializado via uma chamada recursiva ao método **serializarObjeto**. O método **empacotar** simplesmente realiza a conversão de tipos primitivos em conjuntos de bytes, podendo, por exemplo, armazenar o conjunto total serializado em uma estrutura de dados para posterior utilização ou enviar o mesmo para outra máquina através da rede.

Pode ser interessante alterar o algoritmo acima para percorrer o grafo de objetos até determinada profundidade ou para construir uma estrutura serializada para um seleto conjunto de objetos escolhidos segundo as necessidades das aplicações. Cita-se, a título de exemplo, a aplicação do sistema Jamp [Val02] que personaliza o processo de serialização para definir contêineres móveis.

4.3 Exemplo de Serialização em Dispositivos Móveis

Para o exemplo a seguir utilizaram-se as bibliotecas desenvolvidas para reflexão e serialização em J2ME para transferir objetos de uma máquina para outra, em uma rede de dispositivos móveis.

Para utilizar reflexão computacional e serialização de objetos em J2ME com CLDC, é necessário importar as bibliotecas correspondentes, que devem estar presentes no ambiente: "**reflection.reflect.***" e "**middleware.***".

```
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import reflection.reflect.*; // Biblioteca de reflexão para J2ME/CLDC
import middleware.*; // Biblioteca de serialização para J2ME/CLDC
```

Programa 9: Uso de Reflexão e Serialização em J2ME

A classe cliente (Programa 10): (i) conecta-se ao servidor; (ii) cria uma instância da classe **Test**, utilizando **getConstructor** e **newInstance**; (iii) serializa o objeto criado, invocando, para isso, o método **marshal**; (iv) envia o objeto serializado ao servidor.

A classe do servidor também utiliza as bibliotecas do Arcabouço para prover reflexão e serialização em J2ME com a configuração CLDC, realizando os seguintes passos (Programa 11): (i) abre uma conexão para o cliente; (ii) recebe um objeto em sua forma serializada; (iii) reconstrói o objeto original utilizando, para isso, o método **unmarshal**; (iv) exibe os atributos do objeto recuperado, obtendo os mesmos através da invocação do método **getFields**.

Isso mostra que o uso de reflexão em conjunto com a serialização permite criar, transferir para outras máquinas - ou armazenar temporariamente - e, então, recuperar objetos de classes desconhecidas em tempo de compilação.

5. Compilador Reflexivo

5.1 Visão Geral

O compilador Reflexivo automatiza as tarefas de implementação de interfaces e métodos do Arcabouço. A idéia básica é utilizar reflexão computacional para realizar engenharia reversa e direta de classes Java e incluir uma fase intermediária de reestruturação das classes a ser realizada pelo Compilador Reflexivo, que torne

disponível recursos especiais em tempo de execução sem necessitar que os mesmos sejam providos de forma pelo ambiente.

```
.........
sc = (SocketConcection)Connector.open("socket://" + serverAddress);
os = sc.openOutputStream( );
String className = "Test";
Class cls = Class.forName(className);
Reflectable r = (Reflectable)cls.newInstance( );
Clazz cl = r.getClazz( );
Constructor c = cl.getConstructor(new Class[ ] (Integers.TYPE});
addText("invoking constructor " + className + "(11)");
r = (Reflectable) c.newInstance(new Object[ ] {new Integer(11) } );
cl = r.getClazz( );
addText("serializing object ...");
Marshalizable m = (Marshalizable) r;
ByteStream bs = new JbyteStream( );
m.marshal(bs);
addText("sending object ….");
os.write(className + "\n").getBytes( ));
os.write(bs.getBytes( );
os.close( );
……..
```

Programa 10: Exemplo de conexão (cliente)

```
.....
ServerSocketConnection scn = (ServerSocketConnection)
                                        Connector.open("socket://:"+ port);)
sc = (SocketConnection)scn.acceptAndOpen( );
is = sc.openInputStream( );
addTExt("receiving objet …"); ByteStream s = new JbyteStream( );
String ClassName = getClassName(is);
bs.setBytes(getBytes(is));
addText("building object …"); Class cls = Class.forName(className);
Marshalizable m = (Marshalizable) cls.newInstance( );
addLine("recovering object …");
m.unmarshal(bs);
addTExt("\nshowing object …."); Reflectable r = (Reflectable) m;
Clazz cl = r.getClazz( );
Field fields [ ] cl.getFieldss()
for (int i = 0; i < fields.length; i++) {
  Field f = fields[i]; addText(" " + f.getName( ) + " = " + f.get(r) + ";");
}
….
```

Programa 11: Exemplo de conexão (servidor)

5.2 Compilação e Execução

Considere uma classe bem simples **Test** (Programa 12) escrita em Java a seguir. Para que a classe **Test** possa usufruir capacidades reflexivas no ambiente J2ME com CLDC, deve-se submetê-la a uma fase de compilação especial.Para isto, em vez de utilizar diretamente o compilador padrão de Java, **javac**, usa-se o novo compilador (**rcmp**), que chama internamente o **javac**, e que foi especialmente desenvolvido para re-estruturar classes compiladas, acrescentando-lhes as funcionalidades requeridas (Figura 3)

```
public class Test{
   private int x;
   public Test(int x) {this.x = x}
   public int getDoubleX( ) {return 2*x;}
}
```

Programa 12: Teste do Arcabouço

Uma vez concluída a fase de compilação, é possível transportar a classe **Test.class** para o ambiente J2ME, considerando-se que a fase de pré-verificação transcorreu de forma satisfatória. Esta fase, que pode ser realizada utilizando o J2ME Wireless Toolkit 2.0, serve para atestar que classes compiladas podem executar corretamente na plataforma J2ME configurada com CLDC. O pré-verificador basicamente analisa os bytecodes dos arquivos **.class** para determinar se neles não há instruções inválidas como, por exemplo, as relacionadas com ponto flutuante (float e double), que não são suportadas em CLDC.

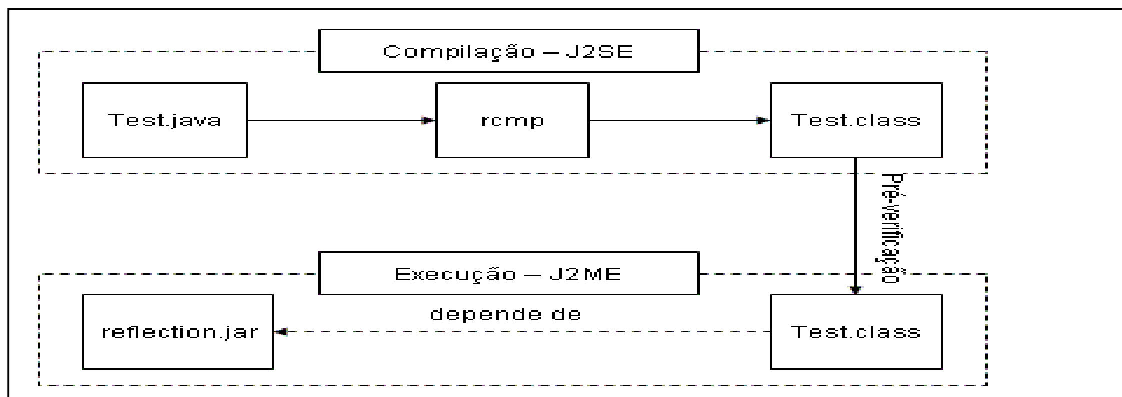

Figura 3. Ambiente de utilização do Compilador Reflexivo

Neste ponto, é possível analisar como se procede a execução de classes que necessitam de habilidades avançadas de reflexão computacional na plataforma J2ME. A

execução de Test.class fica condicionada à presença de uma biblioteca especial chamada reflection.jar (Figura 4), especialmente desenvolvida com base no Arcabouço. Esta biblioteca, que atende às necessidades de dispositivos móveis extremamente restritos em termos de recursos, possui dimensões reduzidas, aproximadamente 10 KB, compatíveis com esses dispositivos.

Considere a classe **ShowClassMIDlet**, por exemplo, que utiliza a biblioteca **reflection.jar** e as capacidades reflexivas adquiridas pela classe **Test** através da fase de compilação especial (Figura 4). A classe **ShowClassMIDlet** pode invocar **getClazz** (fluxo 1 no diagrama) em uma instância da classe **Test** e utilizar a biblioteca de reflexão (fluxo 2 no diagrama) para obter informações acerca da classe, exibindo-as em uma tela de dispositivo móvel; isso sem, no entanto, conhecer a classe em tempo de compilação.

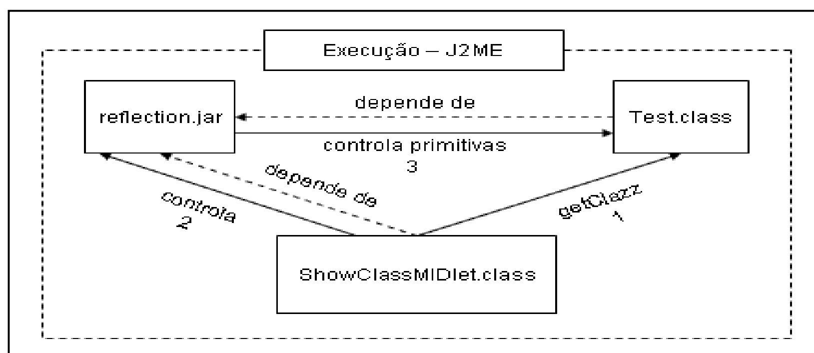

Figura 4: Fluxo de controle e dependências do ambiente

5.1 Descrição da Implementação do Compilador Reflexivo

O funcionamento do Compilador Reflexivo compreende quatro etapas principais: Pré-Compilação, Engenharia Reversa, Reestruturação das Classes e Engenharia Direta.

A Fase de Pré-Compilação: Durante esta primeira fase são produzidas, via javac, as classes em formato binário de bytecodes (arquivos .class). Estas classes são utilizadas nas fases seguintes para obtenção de informações e reestruturação, com a conseqüente geração de novas classes, agora, adaptadas para reter informação sobre as mesmas (meta-dados).

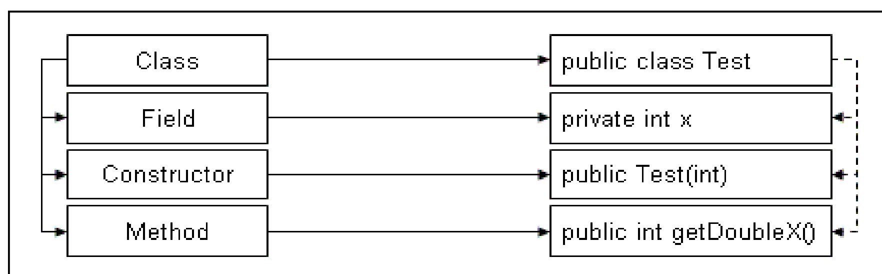

Figura 5. Mapeamento entre membros de classe e suas imagens

A Fase de Engenharia Reversa: Durante esta fase são obtidas informações acerca das classes compiladas na fase anterior e que se desejam portadoras de capacidades reflexivas. Dessa forma, é produzida uma representação interna de mais alto nível sobre

as mesmas. Tal representação corresponde às imagens das classes refletidas, i.e., as classes são refletidas pelas estruturas apropriadas de acordo com a API reflexiva de Java. Por exemplo, para as classes, atributos, construtores e métodos, temos os objetos do tipo **Class**, **Field**, **Constructo**r e **Method** (Figura 6), respectivamente.

A Fase de Reestruturação das Classes: A partir do mapeamento dos componentes das classes em uma representação interna de mais alto nível, é possível reorganizá-las acrescentando, atualizando, ou até mesmo removendo, determinados membros das classes, providenciando as requeridas funcionalidades, no caso, capacidades avançadas de reflexão computacional. Por exemplo, o Compilador Reflexivo desenvolvido para automatizar o Arcabouço altera a declaração das classes que se desejam portadoras de capacidades reflexivas, acrescentando-lhe a cláusula "**implements Reflectable**". Além disso, são criados os membros correspondentes aos existentes na interface **Reflectable**. Especificamente, membros dessa interface que representam métodos **get**, **set**, newInstance, **invoke** e **getClazz** são acrescentados com suas correspondentes assinaturas (Figura 7).

A Fase de Engenharia Direta: Terminada a fase de reestruturação das classes, deve-se re-gerar os arquivos **.class** correspondentes. Para isso, é novamente utilizada uma chamada ao compilador nativo da plataforma J2SE, javac.

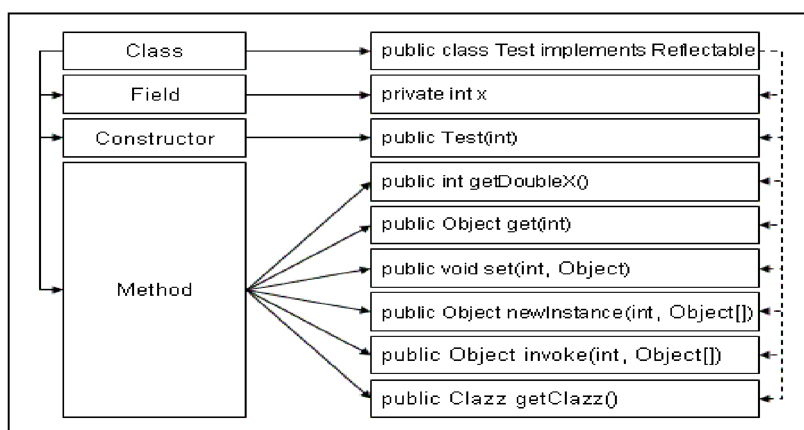

Figura 7: Classe Re-Estruturada

5.2 O Compilador Reflexivo Extensível

No compilador **rcmp**, as interfaces das classes a serem re-estruturadas estão encrustadas em seu código. Apresenta-se a seguir uma nova versão deste compilador, denominada **Rcx**, dotada de capacidade de extensibilidade.

O compilador reflexivo extensível **RCx** possui a capacidade de construir classes reestruturadas com a definição da reestruturação podendo ser especificada de forma dinâmica. O compilador **RCx** (Figura 8) busca as declarações das interfaces e as definições de implementação das mesmas, respectivamente, em **interfaces** e **implementations**. Dessa forma, pode-se especificar para quais interfaces uma classe deverá ser reestruturada, sendo que a definição de tal reestruturação não se encontra mais incrustada no núcleo do compilador.

Esta abordagem aumenta o poder de expressão, porque dispensa a necessidade de se alterar o compilador para estender suas funcionalidades, sendo suficiente apenas acrescentar declarações de interfaces ao pacote **interfaces** e definir as regras de implementação das interfaces em **implementations**. Na ativação do compilador especificam-se as interfaces desejadas: **RCx** <*className*> <*interfacesList*>

Para o exemplo a seguir, foram especificadas a classe **Test** a ser reestruturada e as interfaces a serem implementadas: **Marshalizable** do pacote *middleware* e **Reflectable** do pacote **reflection.reflect**:

RCx *Test* middleware.Marshalizable reflection.reflect.Reflectable

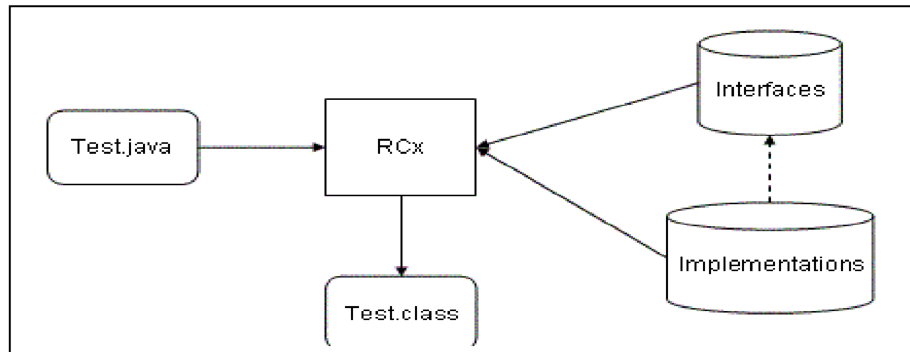

Figura 8: O Compilador Reflexivo Extensível

O compilador **Rcx** monta internamente, durante a fase de reestruturação, uma representação que reflete os métodos das interfaces **Reflectable** e **Marshalizable**, sendo que a abordagem utilizada para a construção desta representação baseada no uso do recurso de *dynamic proxy classes*.

O núcleo do compilador RCx possui aproximadamente 500 linhas distribuídas entre as classes de **RCx.java**, **JCompiler.java** e **RProxy.java**. A classe **JCompiler** agrega as principais funcionalidades que acessam diretamente o compilador de Java para a plataforma J2SE, javac. A classe **RCx** estende a classe **JCompiler** e realiza as principais etapas descritas na Subseção 4.3: pré-compilação, engenharia reversa e engenharia direta; invocando, também, a classe **RProxy** para realizar a fase inicial de reestruturação das classes. A classe **RProxy** (Figura 9) mapeia dinamicamente todos os métodos das interfaces pertencentes ao pacote **interfaces** que foram passadas ao compilador com os respectivos métodos das classes que contêm as definições de implementação pertencentes ao pacote **implementations**.

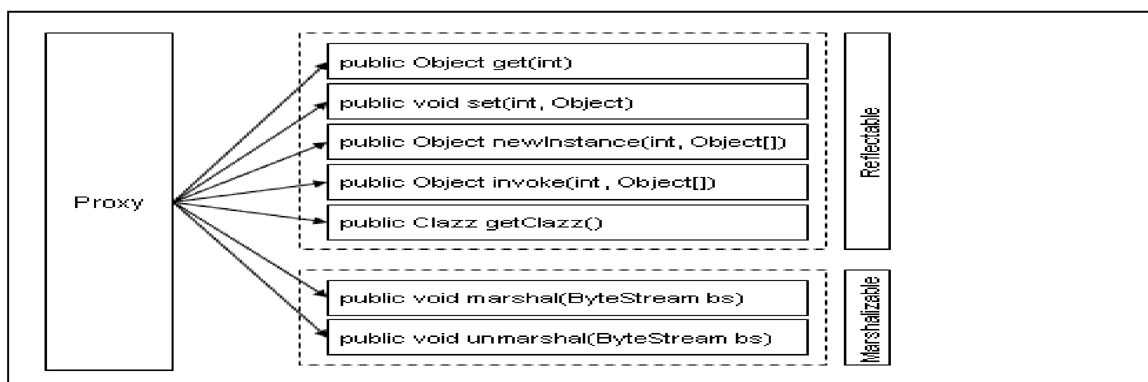

Figura 9. Proxy para as interfaces Reflectable e Marshalizable

Segundo o exemplo, os métodos das interfaces **Reflectable** e **Marshalizable** em **interfaces** ficam associados, respectivamente, aos métodos das classes **ReflectableImpl** e **MarshalizableImpl** em **implementations**, respeitada essa convenção de nomes.

Dessa forma, quando o compilador gerar a classe final já reestruturada, as definições de implementação das interfaces são executadas dependendo das associações baseadas nas assinaturas dos métodos e das convenções de nomes.

6 Conclusão

A união de um arcabouço de reflexão computacional de dimensões reduzidas que seja compatível com as limitações dos atuais dispositivos móveis e um mecanismo de automatização de sua implementação traz o poder e simplicidade do recurso de reflexão computacional à plataforma J2ME com a configuração CLDC.

Telefones celulares com a capacidade de executar aplicações Java já não são mais projetos utópicos e puramente acadêmicos; e tornam-se, dia após dia mais populares com a previsão de superar em número de conexões à Internet os tradicionais computadores *desktop* em 2 ou 3 anos. Da mesma forma, prevê-se que redes móveis *ad-hoc* se tornem o principal paradigma de redes de computadores de um futuro próximo. Este cenário da atualidade e as perspectivas tão promissoras para a Computação Móvel valorizam técnicas de transporte de sistemas mais sofisticados para configurações mais limitadas.

O presente trabalho mostra como, via recurso de reflexão computacional, realizar a implementação de mobilidade de conjuntos de objetos, contêineres, em em contextos de configurações limitadas, como J2ME com CLDC.

É também apresentada a implementação de um Compilador Reflexivo, o qual foi desenvolvido para ser extensível, permitindo, assim, adicionar dinamicamente interfaces e descrições de implementações, sem necessidade de se alterar o núcleo do compilador, haja vista que as regras para a implementação das interfaces ficam em módulos externos, que podem ser adicionados, removidos ou alterados de forma independente.

Há ainda várias outras funcionalidades, que podem ser acrescentadas em trabalhos futuros, como por exemplo: (a) desenvolver um mecanismo de segurança para restringir os membros das classes a serem acessados através de reflexão; (b) acrescentar as interfaces **Array** e **Modifie**r juntamente com as respectivas classes que as implementam, **JArray** e **Jmodifier**, ao arcabouço de reflexão proposto, adaptando também o compilador reflexivo para gerar as correspondentes reestruturações nas classes; (c) acrescentar o mecanismo de *proxy classes* ao arcabouço de reflexão, com a correspondente adaptação do compilador reflexivo; (d) adaptar o mecanismo de serialização para prover a funcionalidade de evolução de classes.

7 Referências Bibliográficas

[Arn00] ARNOLD, Ken, GOSLING, James, HOLMES, David, **The Java Programming Language,** third edition, Addison-Welsley, 2000.

[Nic00] NICHOLAS, Kassem, Design Entreprise Applications with Java 2 Platform, Entreprise Edition, Addison-Wesley, 2000.

[Rig01]RIGGS, Roger; TAIVALSAARI, Antero; VANDENBRINK, Mark; HOLLIDAY, Jim. **Programming Wireless Devices with the Java 2 Platform Micro Edition**. Addison-Wesley, 2001.

[Rob98]ROBSON, Geraldo; LOUREIRO, Antônio Alfredo. **Introdução à Computação Móvel**. Décima Primeira Escola de Computação, 1998.

[Rom00] ROMAN, Gruia-Catalin; PICCO, Gian Pietro; MURPHY, Amy L. **Software Engineering for Mobility: A Roadmap**. In A. Finkelstein, editor, The Future of Software Engineering, pages 241-258. ACM Press, 2000.

[Sta01]STANDAGE, Tom. **The Internet, untethered**. The Economist, October 2001.

[Val02]VALENTE, Marco Túlio de Oliveira. **Mobilidade e Coordenação de Aplicações em Redes sem Fio**. (Tese de Doutorado) Belo Horizonte, 2002.

[Var00]VARSHNEY, Upkar; VETTER, Ronlado. **Emerging Mobile and Wireless Networks**. Communications of the ACM, 43(6):73-81, June 2000.