UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE UM AMBIENTE DE MONITORAMENTO E APRESENTAÇÃO DE PROGRAMAS JAVA UTILIZANDO REFLEXÃO COMPUTACIONAL

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

ROMEU GADOTTI

BLUMENAU, NOVEMBRO/2002

2002/2-56

PROTÓTIPO DE UM AMBIENTE DE MONITORAMENTO E APRESENTAÇÃO DE PROGRAMAS JAVA UTILIZANDO REFLEXÃO COMPUTACIONAL

ROMEU GADOTTI

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof.

_____

Prof.

AGRADECIMENTOS

Agradeço inicialmente a meus pais, por todo o apoio e incentivo que sempre recebi, e que nesses últimos meses foram fundamentais.

Um agradecimento especial a meu orientador, professor Marcel Hugo, por toda a dedicação e colaboração para a realização e conclusão deste trabalho.

A todos os meus amigos, agradeço o apoio e a compreensão pelo tempo que não pude dispor para uma cervejinha.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
JVM	*Java Virtual Machine*
MOP	*Metaobject Protocol*
OO	Orientação a Objetos
POO	Programação Orientada a Objetos
UML	*Unified Modelling Language*

RESUMO

Este trabalho de conclusão de curso consiste na pesquisa da tecnologia de reflexão computacional aplicada no monitoramento de programas orientados a objeto escritos em Java. Baseando-se em recentes pesquisas e teses de professores e profissionais da área da ciência da computação, foram estudadas algumas extensões de bibliotecas que possibilitam a reflexão computacional no ambiente Java, tanto em seu aspecto estrutural quanto comportamental, assim como as classes reflexivas da própria Java. Como resultado obteve-se a implementação de um ambiente de representação da estrutura e comportamento das classes utilizadas na aplicação submetida à análise do protótipo.

ABSTRACT

This work about the conclusion of the course consists in research of technology of computational reflection to be applied in the monitoring of object-oriented programs written in Java. Supporting in recent researches and thesis of teachers and professionals in area of computation science, were studied some extensions of libraries that make possible the computational reflection in Java ambient, as much as in its structural and bearable aspect, as well as the reflexive classes of own Java. Like result, it obtained the implementation of an ambient of representation of the structure and behavior of classes, utilized in application subjected to analysis of the prototype.

1 INTRODUÇÃO

Segundo Coad (1991) o termo “orientado a objetos” pode não apresentar uma idéia clara da tecnologia de orientação a objetos, porque o nome “objeto” tem sido usado de formas diferentes. Na modelagem de informações, significa uma representação de alguma coisa real e um número de ocorrências desta coisa. Nas linguagens de programação baseadas em objetos, significa uma ocorrência em tempo de execução de algum processamento e valores, definidos por uma descrição estática chamada “classe”. Em 1993, Coad dirige-se à orientação a objetos como a revolução da programação. No mesmo ano, Winblad (1993), declara que se as técnicas de orientação a objeto são praticadas corretamente, a arquitetura de uma aplicação segue a estrutura do problema muito mais de perto. Isto faz com que o desenvolvimento, o uso e a manutenção de uma aplicação tornem-se fáceis e rápidas.

Senra (2001) afirma que a orientação a objetos (OO), um dos paradigmas ainda em ascensão na engenharia de software, já oferece ferramental para a luta contra a complexidade, coerente com a diretriz fundamental da Engenharia de Software: a busca de baixo acoplamento e alta coesão. Contudo, o paradigma de OO não representa uma solução completa e definitiva para todos os problemas envolvidos na produção de software.

Conforme Araújo (2000), as linguagens orientadas a objetos, como o Java, são aquelas que proporcionam mecanismos que ajudam a implementar o modelo orientado a objetos. Esses mecanismos são: polimorfismo, encapsulamento e a herança.

Segundo Winblad (1993), existem alguns problemas de entendimento do paradigma de orientação a objetos, o mais comum é a não distinção entre classes e objetos. Classes são gabaritos estáticos que existem somente no corpo de um programa-fonte. Objetos são entidades dinâmicas que aparecem como áreas de memória durante a execução de um programa. Estes problemas de entendimento podem ser parcialmente sanados com a utilização de um ambiente que apresente informações sobre a estrutura das classes de um aplicativo antes do mesmo ser executado, e após sua execução apresente os objetos instanciados e seu comportamento.

O protótipo proposto tem como objetivo principal a utilização da tecnologia de reflexão computacional para possibilitar o monitoramento estrutural e comportamental de qualquer programa escrito na linguagem de programação Java. Com a possibilidade do

monitoramento do programa Java, torna-se possível o processo automático de representação gráfica de seu comportamento durante a execução e, conseqüentemente, pode-se atingir objetivos mais específicos, como por exemplo, a apresentação e manipulação dos atributos de uma classe ou instância e a apresentação das mensagens passadas entre os objetos do programa.

O foco em Java deve-se ao fato de sua freqüente utilização no ensino da tecnologia de orientação a objetos. Por tratar-se de uma linguagem fortemente orientada a objetos e de alta legibilidade, tem-se um aumento significativo na facilidade de entendimento do código-fonte. Entre outras vantagens, como por exemplo, a de ser uma ferramenta gratuita, a linguagem Java destaca-se ainda por oferecer suporte à tecnologia de reflexão computacional.

A reflexão computacional, segundo a abordagem de metaobjetos, permite a separação entre a funcionalidade de uma aplicação (nível-base) e os aspectos de controle da mesma (metanível), ou seja, a reflexão computacional compreende a capacidade de um dado sistema computacional ter acesso a representação de seus próprios dados, e, sobretudo, de que quaisquer modificações realizadas sobre esta representação hão de refletir em seu estado e comportamento. Quando este conceito é mapeado para o domínio de linguagens de programação orientadas a objetos, o padrão de interação entre nível base e metanível é denominado de protocolo de metaobjetos (MOP).

O ambiente proposto será desenvolvido seguindo uma abordagem orientada a objetos e utilizando conceitos e técnicas da tecnologia de reflexão computacional. A especificação do ambiente será elaborada utilizando-se da linguagem de modelagem *Unified Modelling Language* (UML) e sua implementação dar-se-á na linguagem de programação Java.

1.1 MOTIVAÇÃO

Os benefícios que a utilização de reflexão computacional tem oferecido para várias áreas do desenvolvimento de sistemas complexos e robustos, como por exemplo, a possibilidade de refletir sobre seu próprio processamento e adaptar-se ou a possibilidade de monitorar e adaptar outros processos em seu beneficio, são uma das grandes motivações do desenvolvimento deste trabalho de conclusão de curso.

A adição de novas informações às poucas fontes de pesquisa, referentes ao tema deste trabalho, também motivaram a proposta e execução do mesmo.

1.2 OBJETIVO

Esta proposta de trabalho de conclusão de curso tem como objetivo o desenvolvimento de um protótipo de ambiente que através do monitoramento, destacando a apresentação e manipulação dos atributos de uma classe ou instância e a apresentação das mensagens passadas entre os objetos do programa. Este escrito na linguagem de programação Java.

Os objetivos específicos do trabalho são:

a) aprofundar os conceitos da tecnologia de orientação a objetos em Java;
b) analisar as classes do pacote *java.lang.reflect* e suas extensões;
c) aplicar a tecnologia de reflexão computacional para monitorar a execução de um programa escrito em Java;

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho está organizado da seguinte forma:

O capítulo 1 apresenta, objetivamente, uma introdução ao trabalho, suas motivações, seus objetivos e a organização do texto.

O capítulo 2 apresenta uma introdução a orientação a objetos e aspectos específicos da programação orientada a objetos no ambiente Java consideradas relevantes para este trabalho.

O capítulo 3 descreve conceitos, técnicas e aplicações de reflexão computacional no contexto de orientação a objeto.

O capítulo 4 descreve e demonstra a utilização das bibliotecas e extensões da classe de reflexão computacional da linguagem de programação Java, além de suas aplicações práticas.

O capítulo 5 apresenta a especificação e implementação do protótipo.

O sexto e último capítulo apresenta as principais idéias do trabalho, suas limitações e possíveis melhoramentos.

2 ORIENTAÇÃO A OBJETOS EM JAVA

Considerando que este trabalho de conclusão de curso não aborda o tema orientação a objetos como principal objeto de análise, é necessário um entendimento mínimo dos conceitos desta tecnologia para a compreensão do texto abaixo, e que podem ser obtidos em Coad (1991), Winblad (1993), Flanagan (2000) e Sintes (2002).

2.1 INTRODUÇÃO A POO (PROGRAMAÇÃO ORIENTADA A OBJETOS)

Segundo Sintes (2002) a POO estrutura um programa, dividindo-o em vários objetos de alto nível. Cada objeto modela um aspecto do problema, e estes interagem entre si para orientar o fluxo global do programa.

A programação orientada a objetos é bem diferente da programação tradicional, conhecida como procedural. Ela é fortemente baseada nos conceitos de encapsulamento, responsável pela divisão e independência de partes do programa; herança, responsável pela geração de uma classe partindo de outra previamente existente; e polimorfismo, que permite que um nome de método represente vários códigos diferentes.

Em seguida serão apresentadas as principais aplicações da POO na linguagem de programação Java, esta selecionada para o desenvolvimento deste trabalho por ser uma linguagem totalmente orientada a objetos. O próprio código fonte em Java é uma classe, responsável pela definição de objetos.

2.2 POO EM JAVA

Pretende-se neste capítulo, interar o leitor com as particularidades da programação orientada a objetos em uma linguagem específica, e principal foco do trabalho, a linguagem Java. A Java, como qualquer outra linguagem orientada a objetos, tem suas preferências por palavras reservadas e sintaxe das estruturas discutidas na tecnologia de OO. As menções feitas neste texto requerem um prévio conhecimento desta tecnologia, pois serão apenas para que haja uma maior compreensão da sintaxe Java em relação aos conceitos de OO.

2.2.1 CRIAÇÃO DE CLASSES

A palavra reservada *class*, que define uma classe, é o **encapsulamento** dos atributos e métodos, em outras palavras, é a classe, que em Java deve abrigar todo o código, isto é, nada pode ser escrito fora de uma classe. No quadro 1 pode-se visualizar um exemplo da criação de uma classe em Java.

QUADRO 1: CRIAÇÃO DE UMA CLASSE

```
public class Pessoa
{
    // membros da classe
}
```

2.2.2 DECLARAÇÃO DOS MEMBROS DA CLASSE

Na criação de uma classe tem-se uma palavra-chave, utilizada pela Java, denominada modificador de controle de acesso, que especifica quem pode acessar o método, atributo ou classe em particular. A Java define as regras de controle de acesso através das palavras-chave: *public, protected, private* e *package*. Este último é algo especial da Java que garante o acesso a todas as classes dentro de um pacote. Segundo Flanagan (2000), o acesso a pacotes não faz parte da linguagem Java em si, o controle do acesso geralmente é feito no nível das classes e dos membros de classe.

Os membros de uma classe podem ser de dois tipos distintos: membros de classe, que estão associados à própria classe, e membros de instância, que estão associados às instâncias individuais da classe, ou seja, aos objetos. Os membros de uma classe são definidos como campo ou atributo e método. Assim tem-se, atributo de classe e de instância, e método de classe e de instância. O modificador utilizado para fazer esta distinção é *static*, que define que um atributo ou método é de classe, sendo que todos os outros membros serão considerados de instância.

Outro modificador utilizado pela Java em classes e membros de classe é *final*, que indica que a classe, método ou o valor de um campo é final, isto é, um campo não mudará seu valor, uma classe não poderá herdar seus membros e um método não pode ser substituído. No

quadro 2 tem-se exemplos dos modificadores *static* e *final* na declaração de uma classe e seus membros.

QUADRO 2: CRIAÇÃO DE UMA CLASSE COM SEUS MEMBROS

```
public final class Pessoa
{
    private static final int maioridade = 18;
    private static int idade = 17;

    public static final void MaiorDeIdade()
    {
        if (idade >= maioridade)
            return true;
        return false;
    }
}
```

2.2.3 INSTANCIAÇÃO E DESTRUIÇÃO DE OBJETOS

A criação de uma instância de classe dá-se através do operador *new*, que cria a instância, mas não a inicializa. Isso acontece porque toda classe em Java tem pelo menos um construtor, o qual é um método que tem o mesmo nome da classe e cujo propósito é executar toda a inicialização de um objeto novo. Por outro lado, a finalização desse objeto instanciado é feita automaticamente pelo intérprete da Java quando este não é mais necessário. Porém, é possível escrever métodos finalizadores caso seja necessário. A Java permite apenas um finalizador por classe, e ele deve se chamar *finalize()*. No quadro 3 observa-se a criação de métodos construtores e finalizadores, e também a instanciação de objetos.

QUADRO 3: CRIAÇÃO E DESTRUIÇÃO DE OBJETOS

```
class Pessoa
{
    private String nome;
    private int idade;

    // construtor da classe Pessoa
    public Pessoa()
    {
```

```
        nome = "";
        idade = 0;
    }

    // finalizador da classe Pessoa
    protected void finalize()
    {
        super.finalize();
    }

    public static void main(String args[])
    {
        // criação de um objeto da classe Pessoa
        Pessoa p = new Pessoa();
    }
}
```

Segundo Flanagan (2000), a Java não dá garantias sobre quando os objetos serão destruídos através da coleta de lixo, ou sobre a ordem na qual os objetos serão coletados. Assim sendo, a Java não pode dar nenhuma garantia sobre quando (ou mesmo se) um finalizador será invocado. O intérprete Java pode sair sem fazer a coleta de lixo e assim alguns finalizadores podem nunca ser invocados.

2.2.4 SUBCLASSES E HERANÇA

Com exceção de *java.lang.Object*, toda classe definida em Java tem uma superclasse. Caso esta não for especificada com uma cláusula *extends*, a superclasse é a *java.lang.Object* (Fig. 1). Em Java, a palavra-chave *super()*, tem a mesma função que *this*, porém referencia os membros da superclasse, enquanto *this* referencia os membros da própria classe. No exemplo do quadro 4 *super* está chamando o construtor da superclasse, lembrando que esta chamada deve aparecer como a primeira declaração dentro do método construtor da classe em que está sendo executado.

QUADRO 4: ESTENDENDO UMA CLASSE

```
class Veiculo
{
    private String cor;
```

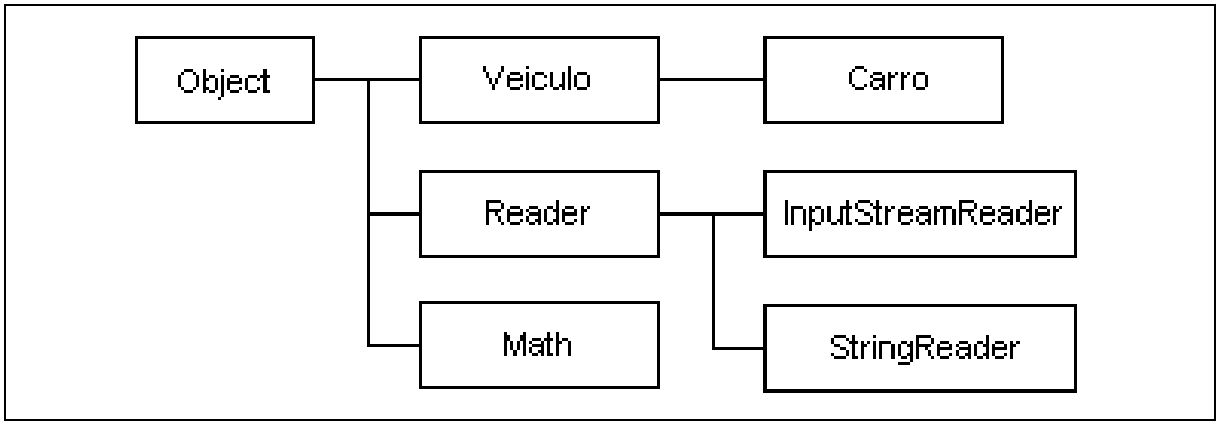
```
    private int anoFabricacao;

    // construtor da classe Veiculo
    public Veiculo(int anoFabricacao)
    {
        // this referencia o atributo da classe, que recebera a variável do construtor
        this.anoFabricacao = anoFabricacao;
    }
}

class Carro extends Veiculo
{
    private String tipoMotor;

    // o construtor da classe Carro chama o construtor da classe Veiculo
    public Carro(int anoFabricacao)
    {
        super(anoFabricacao);
    }
}
```

FIGURA 1: EXEMPLO DE HIERARQUIA DE CLASSES



Conforme Sintes (2002), atualmente, a linguagem Java fornece suporte apenas para herança simples. Porém, a herança múltipla pode ser conseguida através das interfaces (comentadas em mais detalhes adiante), que simulam esta estrutura da orientação a objetos. A palavra-chave *implements* descreve que uma determinada classe irá implementar uma interface. *extends* é a palavra-chave para estender uma classe ou interface Java, isto é, a classe estendida herda os campos e métodos da classe mãe. Uma classe não pode ser herdada caso o

modificador *final* seja encontrado em sua declaração, isto significa que ela não pode ser estendida nem ter subclasse. Com exeção de *java.lang.Object,* toda classe definida em Java tem uma superclasse (Fig. 1). Caso esta não for especificada com uma cláusula *extends*, a superclasse é a *java.lang.Object*.

A especificação Java declara que uma subclasse herda todos os campos e métodos de instância da superclasse acessível para ela. Se a subclasse estiver definida dentro do mesmo pacote da superclasse, ela herda todos os campos e métodos de instância não *private*. Se a subclasse for definida em um pacote diferente, porém, ela herda todos os campos e métodos de instância *protected* e *public*. Os campos e métodos *private* nunca são herdados; nem os campos e métodos de classe.

2.2.5 TROCA DE MENSAGENS ENTRE OBJETOS

Uma outra palavra-chave da Java é *static,* que torna um método acessível através da classe, sem ter que instanciá-la. Qualquer outro atributo ou método sem o modificador *static* é de instância. Os métodos determinam o comportamento dos objetos de uma classe. Quando um método é invocado, se diz que o objeto está recebendo uma mensagem (para executar uma ação). Programas complexos formam conjuntos de objetos que trocam mensagens entre si gerenciando inclusive os recursos do sistema.

O programa no quadro 5 exemplifica chamadas de métodos, para tal definindo um objeto que serve como contador. A implementação representa a contagem no atributo **num** que é um número inteiro. Os métodos são simples: **Incrementa** adiciona um ao contador em qualquer estado e **Comeca** inicializa a contagem em zero. **Decrementa** faz o oposto de **Incrementa**.

QUADRO 5: ENVIANDO MENSAGENS

```
public class Contador
{
    private int num;

    // incrementa o contador
    public void Incrementa()
    {
        // incrementa o contador
```

```
        num = num + 1;
    }

    // decrementa o contador
    public void Decrementa()
    {
        num = num – 1;
    }

    // inicializa o contador
    public void Comeca(int n)
    {
        num = n;
    }
}

public class Principal
{
    public static void Main(String args[])
    {
        // instanciado um objeto do tipo Contador
        Contador cont = new Contador();

        // enviando uma mensagem para o objeto comecar a contar a partir de 0
        cont.Comeca(0);

        // enviando uma mensagem para o objeto incrementar o contador
        cont.Incrementa();
    }
}
```

A sintaxe de chamadas de métodos do objeto deve ser feita usando o nome do objeto e o nome do método deste, separados por um ponto e seguido dos parênteses que podem conter zero ou mais argumentos.

2.2.6 POLIMORFISMO E SOBRECARGA

Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático (Sintes, 2002). A Java, assim como qualquer outra linguagem que suporta o polimorfismo, é denominada linguagem polimórfica.

Na sobrecarga um método diferencia-se do outro pela lista de parâmetros. Neste caso existe a necessidade da lista de parâmetros ser diferente. Quando uma classe é derivada de uma outra, ela pode redefinir um método (já definido na classe base) de forma a executar outras tarefas, mesmo que a definição seja idêntica (Quadro 6). A definição de qual método executar será feita levando-se em conta qual classe está sendo usada. (Souza, 2002).

QUADRO 6: DUAS FORMAS DE POLIMORFISMO

```
class Pessoa
{
    private String nome;
    private String endereco;
    private int anoNasc = 0;

    // nome de métodos iguais com lista de parâmetros diferente
    public void Cadastro(String nome, String endereco, int anoNasc)
    {
        this.nome = nome;
        this.endereco = endereco;
        this.anoNasc = anoNasc;
    }

    public void Cadastro(String nome, String endereco)
    {
        this.nome = nome;
        this.endereco = endereco;
    }
}


class Homem extends Pessoa
```

```
{
    private String nome;
    private String sobrenome;

    // nome do método e lista de parâmetros igual ao da classe base
    public void Cadastro(String nome, String sobrenome)
    {
        this.nome = nome;
        this.sobrenome = sobrenome;
    }
}
```

2.2.7 CLASSES E MÉTODOS ABSTRATOS

Em afirmativas feitas por Flanagam (2000), a Java permite definir um método sem implementá-lo declarando o método com o modificador *abstract*. Um método *abstract* não tem corpo. Ele simplesmente tem uma definição de assinatura seguida por um ponto e vírgula.

Toda classe que possui um método *abstract* é automaticamente *abstract* por si mesma e deve ser declarada como tal. Esta, quando declarada com este modificador, não pode ser instanciada, assim como suas subclasses não podem, se não implementarem todos os métodos *abstract* herdados. Quando uma subclasse de uma classe *abstract* não implementa todos os métodos que ela herda, ela passa a ser *abstract* também. E, finalizando, uma classe pode ser declarada *abstract* mesmo quando ela não tem realmente nenhum método *abstract*. Um exemplo pode ser observado no quadro 7.

QUADRO 7: UMA CLASSE ABSTRATA E SUA SUBCLASSE

```
public abstract class Shape
{
    public abstract double area();
    public abstract double circumference();
}

class Circle extends Shape
{
```

```
    public static final double PI = 3.14159265;
    protected double r;

    public Circle(double r)
    {
        this.r = r;
    }

    public double getRadius()
    {
        return r;
    }

    public double area()
    {
        return PI*r*r;
    }

    public double circumference()
    {
        return 2*PI*r;
    }
}
```

Fonte: Flanagan (2000);

2.2.8 INTERFACES

Assim como a classe, uma *interface* é um recurso básico da Java. Porém, ao contrário de uma classe, uma interface não define atributos e métodos, apenas fornece definições de métodos que podem ser implementados pelas classes. De modo geral, uma interface pode ser entendida como uma garantia de que um dado método será implementado na nova classe, o que não acontece com classes abstratas. Segundo Souza (2002), a Java define alguns pontos importantes sobre interfaces:

a) todas as classes que implementam uma interface devem obrigatoriamente definir o código de todos os métodos nela declarados;

b) pode-se declarar um objeto para ser de um tipo de interface, e criá-lo baseado em um tipo de classe que a implementou;

c) os métodos de uma interface devem ser públicos;

d) não é permitido instanciar objetos baseados em interfaces;

e) uma interface pode ser estendida para outras interfaces da mesma forma que as classes convencionais;

f) uma classe pode implementar mais de uma interface.

Assim como uma classe usa *extends* para especificar uma superclasse, *implements* é usada para nomear uma ou mais interfaces. *implements* é uma palavra-chave Java que pode aparecer em uma declaração de classe como no exemplo do quadro 8.

QUADRO 8: IMPLEMENTAÇÃO DE UMA INTERFACE

```
public interface Centered
{
    public void setCenter(double x, double y);
    public double getCenterX();
    public double getCenterY();
}

public class Rectangle
{
    protected double w, h;

    public Rectangle(double w, double h)
    {
        this.w = w;
        this.h = h;
    }

    public double area()
    {
        return w*h;
    }
}
```

```
public class CenteredRectangle extends Rectangle implements Centered
{
    private double cx, cy;

    public CenteredRectangle(double cx, double cy, double w, double h)
    {
        super(w, h);
        this.cx = cx;
        this.cy = cy;
    }

    // os métodos de Rectangle foram herdados, mas deve-se fornecer
    // implementações de todos os métodos da interface Centered
    public void setCenter(double x, double y)
    {
        cx = x;
        cy = y;
    }

    public double getCenterX()
    {
        return cx;
    }

    public double getCenterY()
    {
        return cy;
    }
}
```

Fonte: Flanagan (2000).

3 REFLEXÃO COMPUTACIONAL

A idéia do paradigma de reflexão computacional não é exatamente nova. Esta idéia originou-se em lógica matemática e recentemente, mecanismos de alto nível tornam o esquema de reflexão um aliado na adição de características operacionais a módulos já existentes (Barth, 2000).

Por analogia, o conceito de reflexão talvez seja melhor explicado pelo estudo de autoconsciência (*self-awareness*) da Inteligência Artificial: "Aqui estou andando rua abaixo na chuva. Uma vez que eu estou ficando ensopado, devo abrir meu guarda-chuva". Esse fragmento de pensamento é um exemplo que revela uma autoconsciência de comportamento e estado, que por sua vez leva à alteração nos mesmos comportamento e estado (Souza, 2001).

Segundo o ponto de vista da engenharia de software, reflexão computacional é uma ferramenta de divisão de interesses *(separation of concerns* em inglês), sendo assim, esta pode ser usada para permitir que programadores escrevam programas com um grande nível de abstração e com uma boa modularidade (Tatsubori, 2002).

Na programação convencional tem-se uma mistura de programação do aplicativo com complicados algoritmos de policiamento, dificultando assim, a compreensão, a manutenção, depuração e validação do programa. A separação de interesses pode ser vista como a reutilização dos algoritmos básicos de policiamento separados do domínio da aplicação. Desta forma tem-se um meta-espaço, permitindo ao programador focar mais especificadamente no domínio da aplicação. Esta funcionalidade adicional é provida através de uma biblioteca de meta-componentes, como será visto mais adiante.

Golm & Kleinoder (1998) definem reflexão como a capacidade de um sistema computacional refletir sobre suas próprias ações e ajustar-se conforme a variedade de condições a que é submetido.

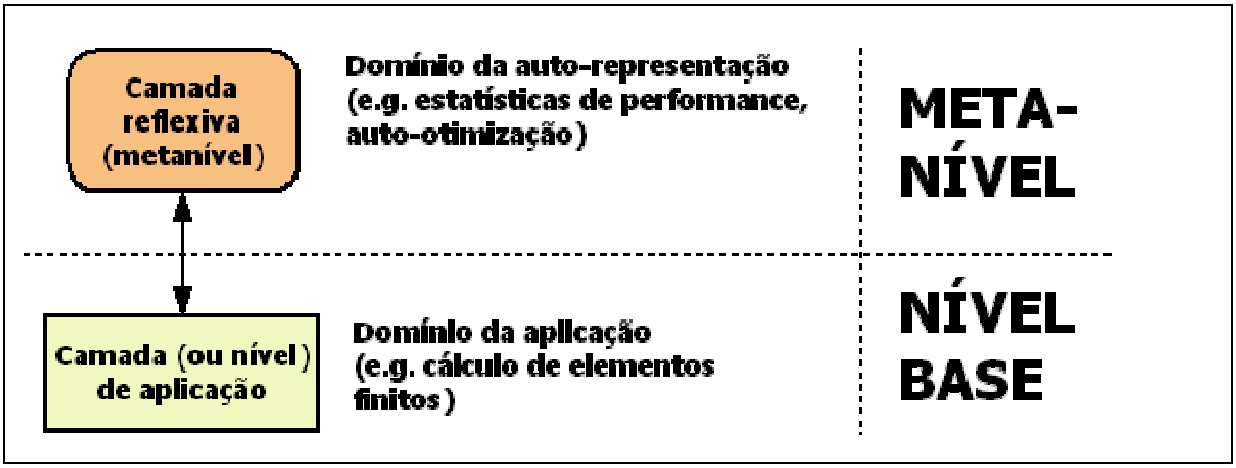
Segundo Senra (2001) o termo **reflexão** remete a dois conceitos distintos no domínio da linguagem natural. O primeiro conceito é reflexão como sinônimo de introspecção, ou seja, o ato de examinar a própria consciência ou espírito. O segundo descreve reflexão como uma forma de redirecionamento da luz. No domínio da Ciência de Computação, o binômio, reflexão computacional encerra ambas conotações: introspecção e redirecionamento. A

primeira denota a capacidade de um sistema computacional examinar sua própria estrutura, estado e representação. Essa trinca de fatores é denominada **meta-informação**, representando toda e qualquer informação contida e manipulável por um sistema computacional que seja referente a si próprio. Por sua vez, a segunda conotação, de redirecionamento, confere a um sistema computacional a capacidade da auto-modificação de comportamento. Ambos conceitos, redirecionamento e introspecção, são tipicamente materializados em linguagens de programação sob a forma de interceptação na execução de primitivas da linguagem. Portanto, a equação resultante é **reflexão computacional = meta-informação + interceptação**.

Sempre que o conceito de reflexão computacional for entoado com certeza trará junto consigo expressões como “domínio da aplicação”, “meta-domínio”, “nível base”, “metaníveis” e “arquiterura de metaníveis”. Suas diferenças são que as primeiras quatro expressões compõem a arquitetura de metaníveis. Denomina-se arquitetura de metaníveis qualquer arquitetura de software com características de análise de seu próprio comportamento, sua forma de execução e sua estrutura de dados, além de um nível para o desenvolvimento da aplicação, denominado nível base, e no qual chamadas de procedimento sejam automaticamente desviadas para um outro nível que se preocupe com as análises mencionadas anteriormente. A este nível dá-se o nome de metanível. Uma arquitetura de metaníveis é composta por vários níveis interligados através de um protocolo de comunicação (discutido na seção 3.1.1) entre os objetos.

Em Devegili (2000), uma arquitetura de metanível provê duas interfaces para dois níveis de funcionalidade distintos, porém conectados: uma interface de nível base que provê funcionalidade do domínio e uma interface de metanível que permite que o comportamento, a forma, ou a implementação da interface de nível base sejam manipulados, regulados ou influenciados. Na figura 2 pode-se ter uma visão mais concreta da definição de Devegili.

FIGURA 2: ARQUITETURA REFLEXIVA



Fonte: Devegili (2000)

Senra (2001) descreve a mesma arquitetura da figura 2, como sistemas computacionais reflexivos, subdivididos em dois domínios: **domínio da aplicação** (externo) e **meta-domínio** (interno). O domínio da aplicação é independente e irrestrito, enquanto o meta-domínio está sempre vinculado com o domínio da aplicação. Essa dicotomia leva arquiteturas reflexivas a serem organizadas em uma pilha de níveis de abstração. O domínio da aplicação é representado usualmente por um único nível sem características reflexivas denominado **nível base**.

O meta-domínio é representado por um ou mais níveis reflexivos denominados **metaníveis**. Cada metanível é responsável por manipular meta-informação do nível imediatamente inferior a si próprio, podendo tal nível ser o nível base (não reflexivo) ou outro metanível.

Segundo Barth (2000) um sistema reflexivo consolida-se depois de completados três estágios:

a) obter uma descrição abstrata do sistema tornando-a suficientemente concreta para permitir operações sobre ela;

b) utilizar esta descrição concreta para realizar alguma manipulação;

c) modificar a descrição obtida com os resultados da reflexão computacional, retornando a descrição modificada ao sistema.

Para a realização da técnica de reflexão computacional torna-se necessário o entendimento de duas partes que compõe esta tecnologia, a introspecção (*introspection*), também referenciada como reflexão estrutural (*structural reflection*), que refere-se ao processo de obtenção da informação estrutural do programa e sua utilização no próprio programa, e a intercessão (*intercession*), também referenciada como reflexão comportamental (*behavioral reflection*), ou ainda como interceptação, e refere-se ao processo de alterar o comportamento do programa no próprio programa.

3.1 REFLEXÃO EM ORIENTAÇÃO A OBJETOS

Apesar do conceito de reflexão computacional ser aplicado sobre diversos paradigmas de programação, este trabalho de conclusão de curso irá focar apenas a aplicação de reflexão computacional sobre o paradigma de orientação a objetos.

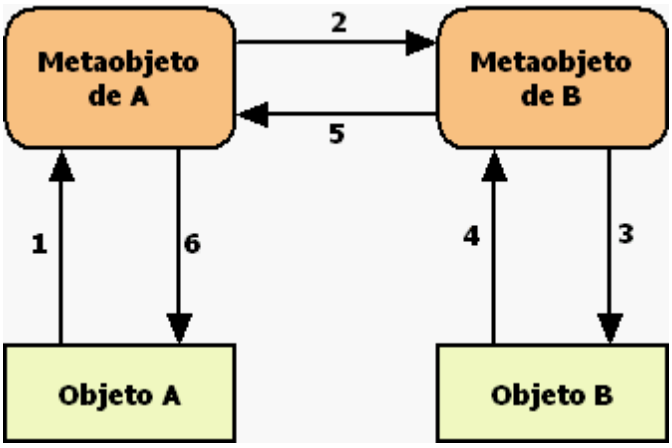
Senra (2001) prevê algumas vantagens na união entre reflexão computacional e OO, como a estruturação do meta-domínio e o gerenciamento da complexidade dos meta-níveis. O fruto imediato de tal união foi a criação do conceito de **metaobjeto**.

Um metaobjeto é um objeto, ou seja, possui um estado e um comportamento associados, como mostra a figura 3. Além disso, um metaobjeto reside no meta-domínio e está vinculado diretamente a um ou mais objetos que pertençam a uma camada de abstração inferior.

Em sistemas reflexivos cujo meta-domínio só possui uma camada, os objetos não-reflexivos referenciados por um metaobjeto são denominados **objetos de nível base**. Em contrapartida, considerando sistemas reflexivos cujo meta-domínio possua diversos metaníveis, o objeto vinculado a um metaobjeto de nível superior não pertence necessariamente ao nível base, uma vez que também possa ser um metaobjeto em relação a objetos de nível inferior ao seu. Este problema de nomenclatura ainda não foi resolvido, pois não existe um qualificador para se referenciar um objeto, realçando unicamente o fato de que ele seja subordinado a um outro objeto reflexivo, e independentemente de que ele próprio seja reflexivo ou não. Em suma, não existe um qualificativo para realçar subordinação sem mencionar a que nível pertence o objeto qualificado.

Segundo Devegili (2000), um sistema possui uma arquitetura de metaníveis orientada a objetos se é um sistema de metaníveis que especifica sua interface de metanível como um conjunto de objetos e protocolos que implementam alguns aspectos da interface do nível base.

FIGURA 3: ARQUITETURA REFLEXIVA EM ORIENTAÇÃO A OBJETOS

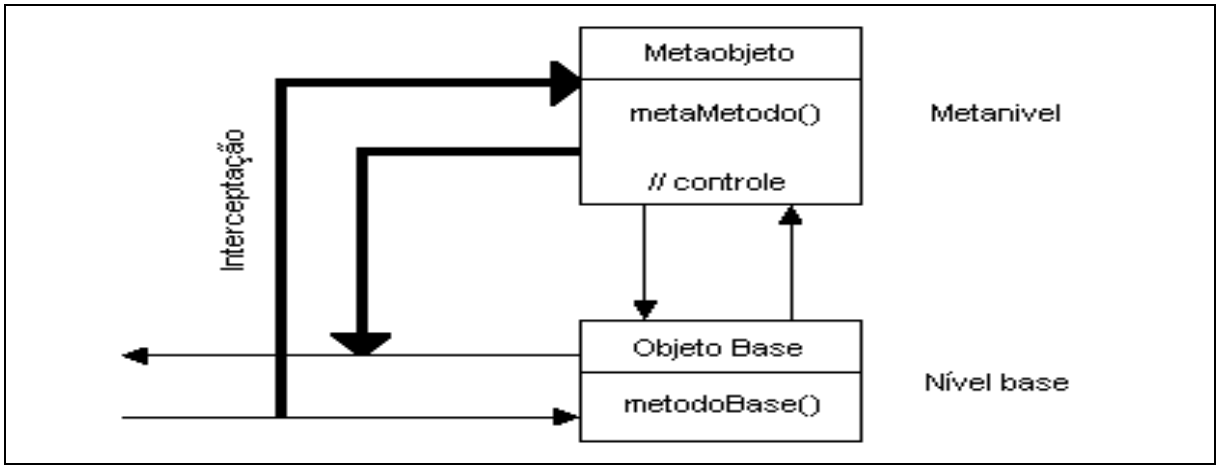


Fonte: Devegili (2000)

A reflexão, por si só, não provê flexibilidade ou capacidade incremental. A combinação de reflexão e orientação a objetos permite que se defina um conjunto de tipos de objetos e operações sobre eles. Este conjunto possui um conjunto de comportamentos denominado protocolo. Permite também que se defina um comportamento padrão denotado por classes e métodos padrão e que sejam feitos ajustes incrementais no comportamento padrão por meio de herança/especialização.

Essa abordagem possibilita a separação dos aspectos funcionais e não funcionais de uma aplicação, permitindo que os métodos e procedimentos da aplicação em si sejam tratados no nível base, enquanto o controle e o gerenciamento da aplicação são tratados no metanível, como mostra a figura 4. Dessa forma, a reflexão torna possível abrir a implementação de um sistema sem revelar detalhes desnecessários de sua implementação, fornecendo flexibilidade de implementação (Souza, 2001). O aspecto funcional de uma aplicação é responsável pela lógica e execução correta de uma determinada tarefa, e o nível não funcional realiza a tarefa de gerenciar o funcionamento do nível funcional através de restrições temporais, restrições de sincronização, exceções temporais e escalonamento em tempo real.

FIGURA 4: SEPARAÇÃO DOS ASPECTOS FUNCIONAIS E NÃO FUNCIONAIS

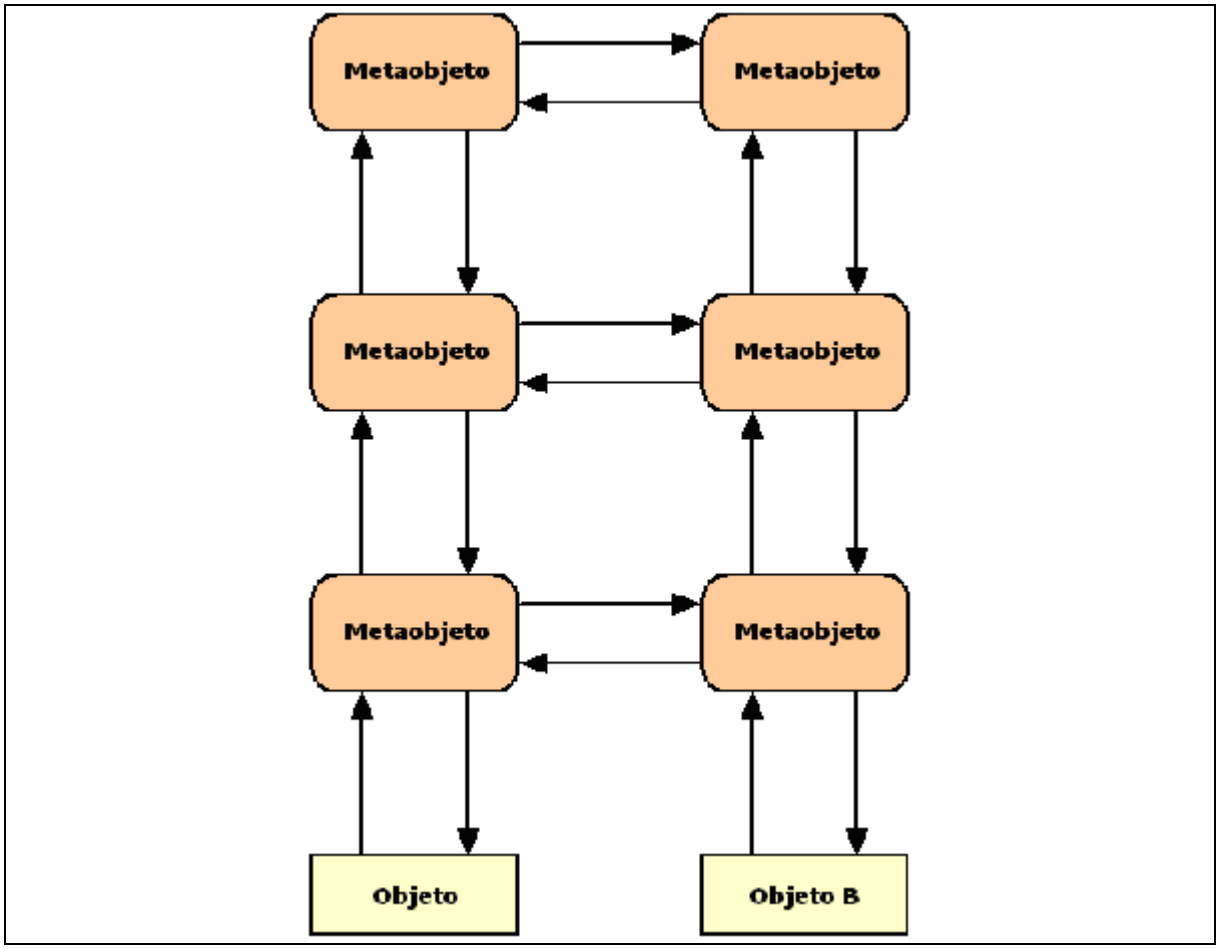


Fonte: Adaptado de Souza (2001).

Barth (2000) diz que reflexão computacional está baseada na noção de definir um interpretador de uma linguagem de programação para a própria linguagem, e que no paradigma de objetos, isto significa representar toda a abstração do modelo de objeto em termos do próprio modelo de objetos. Como conseqüência, a representação de classes, métodos, atributos e objetos são redefinidos por meio de metaclasses e metaobjetos, que estão dispostos em um nível diferente das classes e objetos. As metaclasses e metaobjetos estão dispostos no metanível, comentado anteriormente.

Um problema, apontado por Devegili (2000), surgido na implementação de ambientes reflexivos é a sua inerente característica circular: se um metanível é a representação do domínio da aplicação, que nível representa o metanível? Poder-se-ía definir outro metanível para esta representação, e assim sucessivamente. Tem-se, portanto uma torre circular de metaníveis, em que cada círculo é o metanível do nível no círculo inferior. A este paradoxo dá-se o nome de torres de reflexão e pode ser visualizado na figura 5.

FIGURA 5: TORRES DE REFLEXÃO



Fonte: Adaptado de Devegili (2000).

Em sua dissertação Devegili (2000) aponta uma solução proposta por David Ungar (em um workshop sobre reflexão e arquiteturas de metanível em OO, em 1991): a utilização de programação subjetiva, em que a forma como um objeto responde a uma mensagem depende do contexto (ou da perspectiva) em que a mensagem foi enviada, como uma forma de eliminar a metacircularidade.

Um aspecto importante que determina a aplicabilidade e desempenho de uma meta arquitetura é a forma com que o código do nível base e o código do metanível estabelecem uma conexão e como o controle e a informação fluem entre os dois níveis.

A transferência do fluxo de execução do nível base para o metanível utiliza-se de dois passos:

a) materialização, que disponibiliza o estado e a estrutura de um objeto para que um metaobjeto consiga atuar sobre ele;

b) reflexão, que é o ato de um objeto ter seu estado, estrutura ou comportamento alterados por um metaobjeto.

É possível, neste ponto, verificar as duas propriedades de um metanível: acesso à representação no nível base e alteração desta representação.

A reificação (*reification*), ou materialização, é o ato de converter algo que estava previamente implícito, ou não expresso, em algo explicitamente formulado, que é então disponibilizado para manipulação conceitual, lógica ou computacional. Portanto, é através do processo de reificação que o nível meta obtém as informações estruturais internas dos objetos do nível base, tais como métodos e atributos. Contudo, o comportamento do nível base, dado pelas interações entre objetos, não pode ser completamente modelado apenas pela reificação estrutural dos objetos-base. Para tanto, é preciso intermediar as operações de nível base e transformá-las em informações passíveis de serem manipuladas (analisadas e possivelmente alteradas) pelo metanível (Souza, 2001).

Como o tema principal deste trabalho de conclusão de curso é a reflexão, não será necessário uma definição mais completa do segundo passo da transferência de fluxo entre os níveis do sistema.

3.1.1 METAOBJECT PROTOCOL (MOP)

No capítulo 3, Barth (2000) afirma que para consolidar-se um sistema reflexivo eram necessários três estágios. Contudo, para realizar tal serviço, é necessário dispor de algum mecanismo que permita a comunicação entre o objeto modificado e o objeto modificador, ou seja, entre o nível base e o metanível. Esta comunicação acontece através de uma interface definida pelo MOP. Conforme Devegili (2000), este protocolo, que permite a comunicação com os metaobjetos, pode ser dividido em três categorias:

a) protocolo explícito, onde os objetos do nível base podem comunicar-se com metaobjetos através de uma interface;

b) protocolo implícito, onde a invocação do metaobjeto é feita de forma transparente. Como as invocações de métodos entre objetos devem ser roteadas para metaobjetos, é necessária a implementação de interceptação de metanível, ou seja, a transferência de controle do nível base para o metanível. Esta interceptação pode ser feita por meio da alteração do interpretador ou da máquina virtual de uma certa linguagem ou por meio da utilização de um pré-processador em linguagens compiladas;

c) protocolo intermetaobjetos, que permite a comunicação entre metaobjetos. Apesar de similar ao protocolo explícito, este tipo de protocolo não é visível aos objetos base.

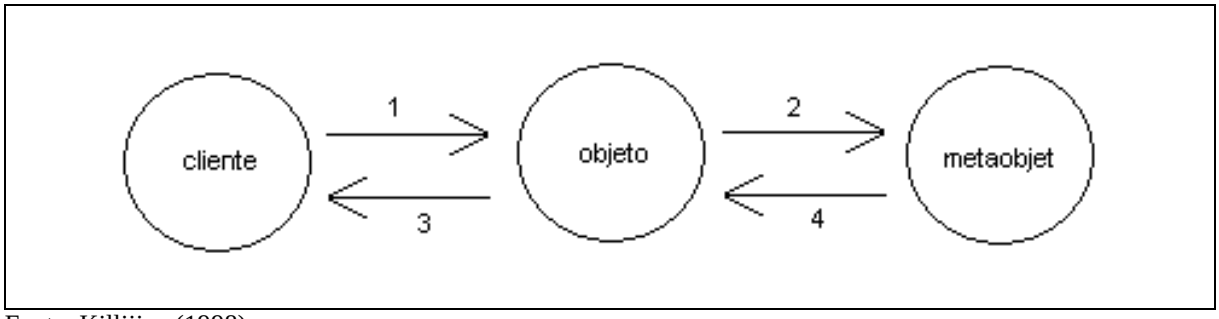
O custo desta comunicação entre nível-base e meta-nível é proporcional à quantidade de informação trocada entre os níveis. Metaobjetos diferentes têm exigências diferentes, tanto no tipo de informação manipulada quanto no modo em que esta é transmitida. Considera-se também como custo no processamento a passagem de controle de um método de nível-base para um método de meta-nível que tem o mesmo nome.

Conforme Souza (2001), o objetivo no desenvolvimento do protocolo de meta-objeto foi o de permitir que uma linguagem fosse extensível o suficiente para admitir a inclusão de novas características, mantendo ao mesmo tempo sua simplicidade, poder de programação, compatibilidade e desempenho com versões anteriores. Assim, o MOP permite a extensão de uma linguagem, que então passa a abrir sua abstração e implementação à intervenção do programador. Para isso, são usadas as técnicas de orientação a objetos e de reflexão computacional, que organizam uma arquitetura de metanível, onde é possível adicionar as características que estendem a linguagem de programação.

Um exemplo prático do funcionamento do MOP está representado na figura 6 e acontece durante a invocação de um método de objeto, feita pelo cliente, e utilizando os parâmetros exigidos (seta 1). O MOP apanha esta chamada, empacota os parâmetros e chama o método de do metaobjeto (seta 2). Desta forma o metaobjeto pode executar algumas ações antes de chamar o método do nível-base (seta 4). Após a execução do método de nível base o valor de retorno é empacotado e devolvido ao metaobjeto (seta 2), que novamente, pode

executar algumas ações antes de devolvê-lo ao objeto do nível-base (seta 4) que, por sua vez, devolve o resultado da operação ao cliente (seta 3).

FIGURA 6: MOP



Fonte: Killijian (1998).

4 REFLEXÃO COMPUTACIONAL EM JAVA

Java é uma linguagem de programação que suporta reflexão. A habilidade reflexiva da Java dá-se através da chamada da API de reflexão. Porém, é bastante restringida à introspecção, que é a habilidade de observação da própria estrutura de dados usada em um programa, como uma classe, por exemplo. Chiba (1998) afirma que a habilidade da API da Java para alterar o comportamento do programa ainda é muito limitada, ela só permite a um programa a instanciação de uma classe, ou a designação e captura de um valor de campo, ou a invocação de um método.

Na *Java Virtual Machine* (JVM) padrão, todas as classes tem como base a classe *Object*, pré-definida pela linguagem. Segundo Barth (2000), a Java dá suporte para metainformação, permitindo introspecção de classes e objetos através da *package java.lang.reflect*. Entretanto, a máquina virtual Java padrão não dá nenhum apoio direto para protocolos de metaobjetos.

Nos capítulos a seguir serão apresentados uma breve introdução a Java, em seguida a classe *java.lang.class* e a API de reflexão Java *java.lang.reflect*. Na seqüência tem-se uma breve apresentação da extensão reflexiva da Java, a API OpenJava e em seguida a API Javassist.

4.1 INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO JAVA

Para que não haja nenhum problema de entendimento posterior, será realizada uma pequena distinção entre a linguagem de programação Java, a JVM, a plataforma Java e o ambiente de programação Java.

Segundo Flanagan (2000) a linguagem de programação Java é uma linguagem avançada, orientada a objetos, na qual os aplicativos Java (incluindo os componentes *applets*, *servlets* e *JavaBeans*) são escritos. Sua sintaxe é semelhante à da C, porém evitando os recursos extremamente complexos que afundaram outras linguagens.

A Java foi projetada para ser pequena, simples e portável a todas as plataformas e sistemas operacionais. Esta portabilidade é obtida pelo fato da linguagem ser interpretada, ou

seja, o compilador gera um código independente de máquina chamado *bytecode*. No momento da execução o código fonte (**.java**) é compilado para *bytecode* (**.class**) que é interpretado por uma máquina virtual (JVM) instalada no computador. A JVM, ou intérprete Java, é a parte crucial de toda instalação Java, afirma Flanagan (2000).

A plataforma Java, ou ambiente de *runtime* Java, é o conjunto pré-definido de classes Java que existe em cada instalação Java. Estas classes estão disponíveis para uso por todos os programas Java e são organizadas em grupos relacionados, conhecidos como pacotes.

Um ambiente de programação serve para aprimorar e facilitar o desenvolvimento de programas escritos na linguagem Java. Pode-se citar como exemplos de ambientes Java o JBuilder (Armstrong, 1998) e o Visual J++ (Perry, 1997).

4.2 *JAVA.LANG.CLASS* E *JAVA.LANG.REFLECT*

O pacote *java.lang.reflect* contém as classes e interfaces que, juntamente com *java.lang.Class* compreendem a *Java Reflection API.*

Conforme Flanagan (2000), a classe *java.lang.Class* representa os tipos de dados da Java e, juntamente com as classes do pacote *Java.lang.reflect* dá aos programas Java a capacidade de introspecção (ou auto-reflexão). Uma classe Java pode olhar para si mesma, ou para qualquer outra classe, e determinar sua superclasse, os métodos que ela define e assim por diante. Após a obtenção de um objeto *Class* torna-se possível a execução de operações reflexivas sobre ele.

Através da API Java de reflexão *java.lang.reflect* um programador pode obter meta informações dos objetos Java, disponibilizadas pelo sistema de *runtime*, ou seja, pode ter acesso a informações de uma definição de classe, como campos e métodos desta. Adicionalmente, possibilita a chamada de métodos e o acesso a valores dos campo da classe. Estas capacidades são possíveis porque este tipo de meta-informação é mantida pelo sistema de *runtime* Java, permitindo assim, padrões de acesso consistentes (Lee, 1998).

A API de reflexão Java não provê capacidade refletiva completa, pois não possibilita alteração no comportamento do programa, suporta somente a introspecção, como por exemplo, a inspeção da definição de classe. Porém, várias extensões para a reflexão da API

Java foram propostas como mostra a seção 4.3. Com a API de Reflexão *java.lang.reflect*, programadores podem facilmente manipular classes desconhecidas, possibilitando a implementação de *browsers* de objeto, *JavaBeans* ou serialização de objeto. *JavaBeans* é um ótimo exemplo da aplicação prática das técnicas de reflexão computacional, onde tem-se um ambiente de programação visual que se utiliza de componentes para o desenvolvimento de aplicações. Este ambiente utilizará a técnica de reflexão computacional para obter as propriedades dos componentes da Java (classes) quando eles forem dinamicamente carregados.

No quadro 9 pode-se visualizar um pequeno exemplo da utilização do pacote *java.lang.reflect*, onde o objetivo é capturar todos os métodos declarados na classe que será passada para o programa como argumento. Inicialmente captura-se a classe a ser trabalhada com o método *forName()* da classe *Class*, em seguida, utiliza-se o método definido pela API *getDeclaredMethods()* para capturar todos os métodos declarados na classe, e as linhas seguintes mostram os métodos capturados na tela.

QUADRO 9: EXEMPLO DA UTILIZAÇÃO DO PACOTE *java.lang.reflect*

```
Import java.lang.reflect.*;

public class Collect
{
  public static void main(String args[])
  {
    try
    {
      Class c = Class.forName(args[0]);
      Method m[] = c.getDeclaredMethods();
      for (int i = 0; i < m.length; i++)
        System.out.println(m[i].toString());
    }
    catch (Throwable e)
    {
      System.err.println(e);
    }
  }
}
```

A API de reflexão da Java mantém um metaobjeto para cada um dos principais elementos da linguagem, como uma classe, método, ou campo. Tudo isso centra em torno da classe chamada *Class*. Há uma instância da classe *Class* para cada classe carregada no sistema de *runtime*. Através de código o usuário pode obter o metaobjeto associado invocando o método *getClass* do objeto. Uma vez obtido o metaobjeto inúmeros métodos tornam-se disponíveis para acessar informações contidas no metaobjeto.

O usuário, também através de código, pode ter acesso a campos ou métodos de um objeto através dos objetos de campo (*Field*) e dos objetos de método (*Method*). Por exemplo, podem ser obtidos o nome e o valor de um campo através do objeto *Field*. Semelhantemente, o nome de um método pode ser obtido e o método pode ser invocado pelo objeto *Method*. A classe *Class* abriga os métodos *getMethods*, *getMethod*, *getDeclaredMethods*, *getDeclaredFields*, *getFields*, e *getField* (Lee, 1998).

A API de reflexão Java também abriga uma classe estática chamada *Array* para as os vetores. Não existe uma instância de *Class* para um vetor, contudo esta tarefa, de manter a meta informação referente a vetores, recaiu sobre a classe *Array*, afirma Lee (1998).

4.2.1 CLASSLOADER

Pode-se encontrar em Souza (2001) a seguinte definição: “Um outro importante componente reflexivo Java é o carregador de classe (*class loader*), que transforma um *array* de bytes em uma classe. Esse processo acontece quando a JVM usa carregadores de classes para carregar arquivos **.class** e criar objetos. Como os carregadores de classe são instâncias de subclasses da classe *ClassLoader*, provida como API Java, os programadores podem definir novas subclasses da mesma em programas Java. Assim, em uma nova subclasse da *ClassLoader*, os programadores podem alterar o comportamento do programa pela modificação do *bytecode* carregado – gerando intercessão genérica de método. Embora o custo de carga e de modificação de *bytecodes* não seja pequeno, ainda assim é útil. Nesse processo, deve-se considerar a dificuldade de programação, pois a manipulação direta de *bytecodes* não é trivial”.

4.3 EXTENSÕES

Para suprir as deficiências da API *java.lang.reflect* citadas no capítulo 4.2, várias extensões dessa API para a reflexão em Java foram propostas. Para evitar a degradação de desempenho, a maioria destas extensões habilita a reflexão comportamental (*behavioral reflection*) restringida, permitindo apenas a alteração do comportamento de tipos específicos de operações como chamadas de método, acesso a campos, e criação de objetos. Os programadores podem selecionar algumas dessas operações e podem alterar seu comportamento. Os compiladores, ou sistemas de *runtime*, dessas extensões inserem ganchos nos programas, de forma que a execução das operações selecionadas seja interceptada. Sendo estas operações interceptadas, o sistema de *runtime* chama então um método ou objeto associado às operações ou objetos designados (metaobjetos). A execução da operação interceptada passa a ser implementada pelo método do metaobjeto, método este, que pode definir um novo comportamento das operações interceptadas.

Descrições de extensões da API *java.lang.reflect* específicas poderão ser observadas nas seções 4.3.1, 4.3.2 e 4.3.3, onde o objetivo principal foi proporcionar uma visão geral das extensões e disponibilizar referências para a continuidade da coleta de informações sobre as mesmas. Porém Javassist será uma exceção, por tratar-se da API utilizada para o desenvolvimento do protótipo proposto por este trabalho terá maior aprofundamento em sua apresentação.

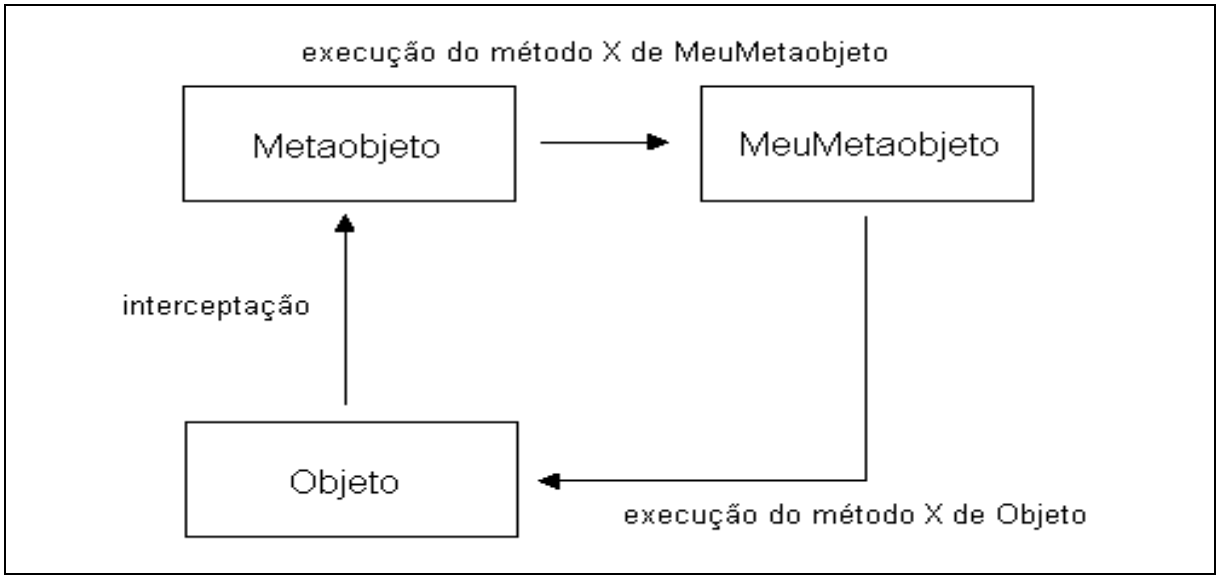
4.3.1 JAVASSIST

Javassist foi baseado em uma nova arquitetura de reflexão que pode ser implementada sem modificar o sistema de *runtime* existente. A Javassist como a maioria das extensões da API Java dispõe a reflexão comportamental, que é a habilidade de interceptar uma operação e alterar o comportamento dela, como uma chamada de método, por exemplo. A Javassist não é um sistema reflexivo em tempo de compilação. Segundo Chiba (1998), Javassist não é uma ferramenta reflexiva tão completa quanto as outras ferramentas da mesma linha, mas devido à sua API simplificada, Javassist é considerada como sendo uma ferramenta reflexiva de fácil utilização.

Através da utilização da API da Javassist programadores podem selecionar algumas operações e podem alterar o comportamento delas. Os compiladores, ou sistemas de *runtime*,

de extensões como Javassist inserem chamadas nos programas de forma que a execução das operações selecionadas seja interceptada. Quando desta interceptação, o sistema de *runtime* chama um método em um objeto associado com as operações ou os objetos designados (metaobjeto) e a execução da operação interceptada é implementada por aquele método. É possível definir uma versão própria destes metaobjetos para implementar um novo comportamento das operações interceptadas como mostra a figura 7. Mais informações sobre este processo podem ser observadas na seqüência deste capítulo.

FIGURA 7: ALTERAÇÃO DO COMPORTAMENTO DOS OBJETOS



Segundo Barth (2000), a Javassist possui uma API que possibilita ao programador uma abstração maior no desenvolvimento de suas aplicações e elaboração de novas estruturas genéricas. A API do Javassist é formada pelos seguintes pacotes:

a) *javassist*: núcleo da API do Javassist;

b) *javassist.reflect*: pacote que implementa as construções reflexivas da ferramenta;

c) *javassist.rmi*: pacote que possui um conjunto de métodos que visam facilitar a programação de aplicações distribuídas;

d) *javassist.tool*: pacote que fornece um compilador para adicionar novas características à linguagem.

Algumas das vantagens da Javassist é a boa divisão dos controles e do aspecto bem distribuído de seus processos, além da poderosa habilidade de introspecção parecida com a da API Java. Destaca-se também pela sua simplicidade de uso e sua forte conexão entre a manipulação de *bytecodes* e o protocolo de metaobjetos (MOP).

Em Javassist, uma classe é representada por um objeto de *CtClass*. O programador que deseja alterar a definição desta classe deve utilizar-se dos métodos do objeto de *CtClass*. Este objeto representa o *bytecode* de uma classe carregada pela JVM. Neste momento obtém-se o acesso à classe, possibilitando que o programa tenha acesso a sua estrutura.

```
CtClass c = new CtClass("Pessoa");
```

Esta linha de código cria um objeto de *CtClass* representando o *bytecode* da classe Pessoa. A classe *CtClass* dispõe de inúmeros métodos para realizar a introspecção e alteração da estrutura da classe. Mudanças na estrutura da classe Pessoa, por exemplo, são refletidas no *bytecode* representado pelo objeto de *CtClass* que carregou a classe. Na tabela 1 encontram-se os métodos utilizados para a realização da introspecção de classes Java.

TABELA 1: MÉTODOS DE CTCLASS UTILIZADOS NA INTROSPECÇÃO

Método	Descrição
`String getName()`	captura o nome da classe
`int getModifiers()`	captura os modificadores da classe
`boolean isInterface()`	retorna *true se* o objeto representa uma interface
`CtClass getSuperclass()`	captura a super classe
`CtClass[] getInterfaces()`	captura a interface
`CtField[] getDeclaredFields()`	captura todos os atributos declarados na classe
`CtMethod[] getDeclaredConstructors()`	captura todos os construtores declarados na classe
`CtMethod[] getDeclaredMethods()`	captura todos os métodos declarados na classe

A Javassist permite a definição de uma nova classe sem a necessidade da leitura de qualquer arquivo de classe. Esta opção torna-se útil se o programa precisa definir

dinamicamente uma nova classe. Para tanto, deve-se criar um novo objeto de *CtClass* como segue:

```
CtClass c2 = new CtNewClass();
```

O objeto c2 representa uma classe vazia que não possui métodos ou campos embora estes podem ser adicionados futuramente à classe pela API Javassist.

A informação sobre campos e métodos são providos através de outros objetos, já que *CtClass* serve apenas para capturar os campos e métodos da classe. Estes objetos são providos pelas classes *CtField* e *CtMethod* respectivamente. Os objetos de *CtField* são obtidos através do método de *CtClass getDeclaredFields*() e os objetos de *CtMethod* são obtidos através do método de *CtClass getDeclaredMethods*(). Existe ainda a classe *CtConstructor* que mantém as informações sobre os construtores da classe. Na tabela 2 encontram-se os métodos utilizados para a realização de introspecção em campos, métodos e construtores.

TABELA 2: MÉTODOS PARA INTROSPECÇÃO EM CAMPOS, MÉTODOS E CONSTRUTORES.

Métodos em *CtField*	Descrição
`String getName()`	captura o nome do campo
`CtClass getDeclaringClass()`	captura a classe na qual está declarado
`int getModifiers()`	captura os modificadores de acesso
`CtClass getType()`	captura o tipo do campo
Métodos em *CtMethod*	**Descrição**
`String getName()`	captura o nome do método
`CtClass getDeclaringClass()`	captura a classe na qual está declarado
`int getModifiers()`	captura os modificadores de acesso
`CtClass[] getParameterTypes()`	captura os tipos dos parâmetros
`CtClass[] getExceptionTypes()`	captura o tipo de exceção
`boolean isConstructor()`	retorna *true* se for um construtor de classe
`boolean isClassInitializer()`	retorna *true* se for o inicializador da classe
Métodos em *CtConstructor*	**Descrição**
`String getName()`	captura o nome do método
`CtClass getDeclaringClass()`	captura a classe na qual está declarado
`int getModifiers()`	captura os modificadores de acesso
`CtClass[] getParameterTypes()`	captura os tipos dos parâmetros
`CtClass[] getExceptionTypes()`	captura o tipo de exceção
`boolean isConstructor()`	retorna *true* se for um construtor de classe
`boolean isClassInitializer()`	retorna *true* se for o inicializador da classe

A diferença entre a API Javassist e a API de reflexão de Java padrão é que a Javassist provê vários métodos para modificar a estrutura da classe, como mostra a tabela 3. Estes métodos são categorizados em métodos para alterar os modificadores de classe, métodos para alterar a hierarquia de classe e métodos para adicionar novos membros à classe.

TABELA 3: MÉTODOS PARA ALTERAÇÃO

Métodos em *CtClass*	Descrição
`void bePublic()`	`marca a classe como pública`
`void beAbstract()`	`marca a classe como abstrata`
`void notFinal()`	`remove o modificador final`
`void setName(String name)`	`muda o nome da classe`
`void setSuperclass(CtClass c)`	`muda o nome da super classe`
`void setInterfaces(CtClass[] i)`	`muda as interfaces`
`void addConstructor(...)`	`adiciona um novo construtor`
`void addDefaultConstructor()`	`adiciona um construtor padrão`
`void addAbstractMethod(...)`	`adiciona um novo método abstrato`
`void addMethod(...)`	`adiciona um novo método`
`void addField(...)`	`adiciona um novo campo`
Métodos em *CtField*	**Descrição**
`void bePublic()`	`marcar o campo como público`
Métodos em *CtMethod*	**Descrição**
`void bePublic()`	`marca o método como público`
`void instrument(...)`	`modifica o corpo do método`
`void setBody(...)`	`substitui o corpo do método`

Segundo Chiba (1998) o *class loader* provido pela Javassist permite que um programa seja carregado para controlar a classe carregada por este *class loader*, isto é, se um programa é carregado pelo *class loader* da Javassist A e inclui uma classe B, então A pode interceptar a carga de B por auto-reflexão e modificar o *bytecode* de B. Para evitar recursão infinita, quando a carga de uma classe é interceptada, interceptação adicional é proibida. O método *Load*() de *CtClass* requer que um programa esteja carregado pelo *class loader* da Javassist embora os outros métodos trabalhem sem ele.

A API Javassist possibilita a reflexão comportamental (*behavioral reflection*) através de “ganchos” inseridos no programa quando as classes são carregadas. O metaobjeto associado a cada classe mantém métodos de interceptação que podem ser reescritos em classes estendidas da classe *Metaobject* (Quadro 10).

QUADRO 10: MÉTODOS DE INTERCEPTAÇÃO

```
public class MyMetaobject extends Metaobject
```

```
{
        public Object trapMethodcall(int identifier, Object[] args)
        {
                // código executado antes do código do nível base
        }

        public Object trapFieldRead(String fieldName)
        {
                // código executado antes do código do nível base
        }

        public void trapFieldWrite(String fieldName, Object value)
        {
                // código executado antes do código do nível base
        }
}
```

Os métodos apresentados no código do quadro 10 representam os métodos e atributos do objeto do nível base associado ao metaobjeto. Toda a chamada de método no objeto do nível base é delegada a *trapMethodcall*() no metaobjeto. O parâmetro *identifier* é um identificador que especifica qual o método do nível base que foi chamado e o parâmetro args é uma lista de parâmetros deste método. Da mesma forma, todo acesso a atributos do objeto do nível base, tanto para leitura como para modificação, é interceptado pelos métodos *trapFieldRead*() e *trapFieldWrite*(). Os parâmetros *fieldName* são identificadores que especificam que atributo foi acessado e o parâmetro *value* representa o novo valor que será gravado neste atributo.

O Javassist atualmente está na versão de número 0.6, e tornou-se disponível na Internet no dia 04 de abril de 2000. A documentação e o próprio Javassist estão disponíveis em [CHI2000].

Atualmente a API da Javassist encontra-se na versão 2.2. Esta versão tornou-se disponível no dia 01 de outubro de 2002 e pode ser adquirida juntamente com sua documentação no endereço http://www.csg.is.titech.ac.jp/~chiba/javassist/survey.

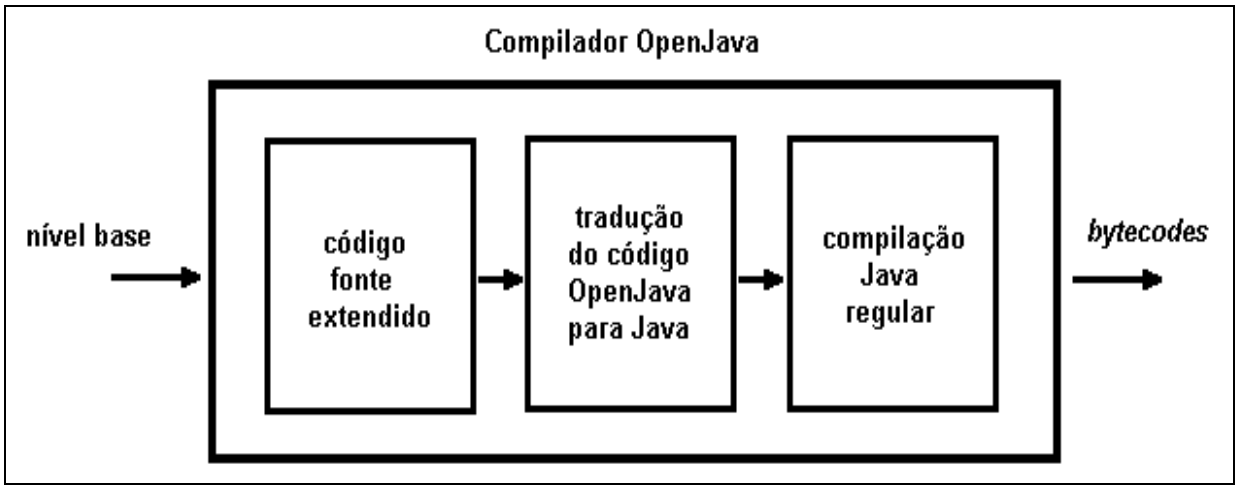
4.3.2 OPENJAVA

Segundo Barth (2000), OpenJava é uma linguagem extensível baseada em Java. As características estendidas do OpenJava são especificadas por um programa de meta-nível dado em tempo de compilação. A estrutura do OpenJava é idêntica a do OpenC++ (Kasbekar, 1998), pois ambas foram modeladas e implementadas seguindo o mesmo conceito.

O OpenJava é uma extensão da sintaxe da linguagem Java padrão, com o propósito de prover reflexão estrutural e comportamental. Em seu artigo Tatsubori (1998) descreve que as características estendidas do OpenJava são especificadas por um programa de nível meta em tempo de compilação. Assim, se não houver nenhuma declaração de nível meta, o OpenJava é idêntico ao Java.

Internamente, o compilador OpenJava opera em três estágios: o primeiro recebe o código fonte estendido do nível base (que contém as declarações dos metaobjetos), o segundo realiza o pré-processamento que traduz o código OpenJava para Java, e por fim realiza a compilação Java regular, gerando *bytecodes* para uma JVM padrão. Sua diferença em relação aos compiladores Java regulares está nas bibliotecas de metanível adicionada às bibliotecas padrão. Uma visão geral do processamento do compilador OpenJava é exibida na figura 8 (Souza, 2001).

FIGURA 8: O COMPILADOR OPENJAVA



Segundo Barth (2000), a parte mais importante da API do OpenJava é a metaclasse *OJClass*. Ela provê os métodos para o acesso à informação sobre a classe. Adicionalmente, as classes *OJField*, *OJMethod* e *OJConstructor* são importantes, pois representam os atributos,

métodos e métodos construtores da classe base. A classe *openjava.mop.OJClass* representa o objeto da classe. Através de seus métodos pode-se obter informações sobre a classe.

Informações adicionais definidas na API do OpenJava como, introspecção, modificação estrutural e modificação comportamental, podem ser encontradas em Tatsubori (2002), Tatsubori (1998) e Barth (2000).

4.3.3 METAXA

Segundo Golm & Kleinoder (1998), o metaXa conhecido formalmente como MetaJava, é um sistema reflexivo projetado para permitir reflexão estrutural e alguma forma de reflexão comportamental em sistemas baseados em Java. Ele consiste do programa de aplicação, do meta-sistema e de funções do sistema operacional subjacentes para conectar o nível base ao nível meta. Ou seja, a abordagem metaXa para reflexão não é baseada em linguagem, mas em sistema. Logo, não é a linguagem que é estendida, e sim a JVM, o que caracteriza o metaXa como um interpretador Java estendido com habilidades de intercessão em tempo de execução.

Sua abordagem para transferência de controle do nível base para o nível meta é feita através de eventos, que são lançados pelo nível base e entregues ao nível meta. O meta-sistema então pode avaliar os eventos e reagir de uma maneira específica. A passagem de evento é síncrona, logo a computação base é suspensa enquanto o meta-objeto processa o evento, descrevem Golm & Kleinoder (1998) em seu artigo "*metaXa and the Future of Reflection*".

Mais informações sobre o metaXa podem ser adquiridas em Golm & Kleinoder (1998) e Golm & Kleinoder (2000).

4.3.4 GUARANÁ

Souza (2001) descreve o Guaraná como uma arquitetura reflexiva independente de linguagem que visa alto grau de reutilização de código de nível meta, simplicidade, flexibilidade e segurança. Sua implementação foi efetivada através da modificação de uma implementação aberta de JVM, o que caracteriza sua abordagem para reflexão como a de MOP em Tempo de Execução. Ou seja, a implementação do Guaraná para Java também é baseada em um interpretador Java estendido. Nesse caso, nem a linguagem Java, nem o

formato dos *bytecodes* são alterados. Isso permite que qualquer aplicação Java existente possa tornar-se reflexiva através de sua execução nessa JVM modificada.

Uma vez que a essência da reflexão computacional consiste em interceptação e introspecção, as diretrizes da arquitetura Guaraná definem os limites sobre os quais será possível prover interceptação e introspecção, sem que sejam violados os atributos de segurança, reusabilidade, flexibilidade e simplicidade. Por conseguinte, o MOP da arquitetura Guaraná define mecanismos de interceptação e introspecção condizentes com estas diretrizes (Senra, 2001).

Possuindo diversas semelhanças com metaXa, o MOP de Guaraná suporta reflexão estrutural e comportamental. Os metaobjetos dessa arquitetura podem ser combinados através de *composers*, que permitem que vários metaobjetos sejam associados com um objeto de aplicação, habilitando a criação de blocos de construção de metaobjetos, usados para prover complexas configurações de metanível. Como os *composers* são metaobjetos, eles também podem ser combinados entre si, provendo um mecanismo de composição hierárquico (Souza, 2001).

Uma descrição completa sobre a arquitetura reflexiva Guaraná pode ser encontrada em Senra (2001).

5 DESENVOLVIMENTO DO TRABALHO

Neste capítulo serão apresentados os requisitos do problema a ser trabalhado, a especificação do protótipo, juntamente com suas limitações e alguns detalhes do processo de desenvolvimento.

5.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos identificados para este trabalho foram:

a) o protótipo deverá inicialmente apresentar informações sobre a estrutura das classes de um programa Java;
b) deverá também apresentar o comportamento dos objetos durante a execução deste programa, ou seja, apresentar a troca de mensagens dos mesmos.

5.2 ESPECIFICAÇÃO

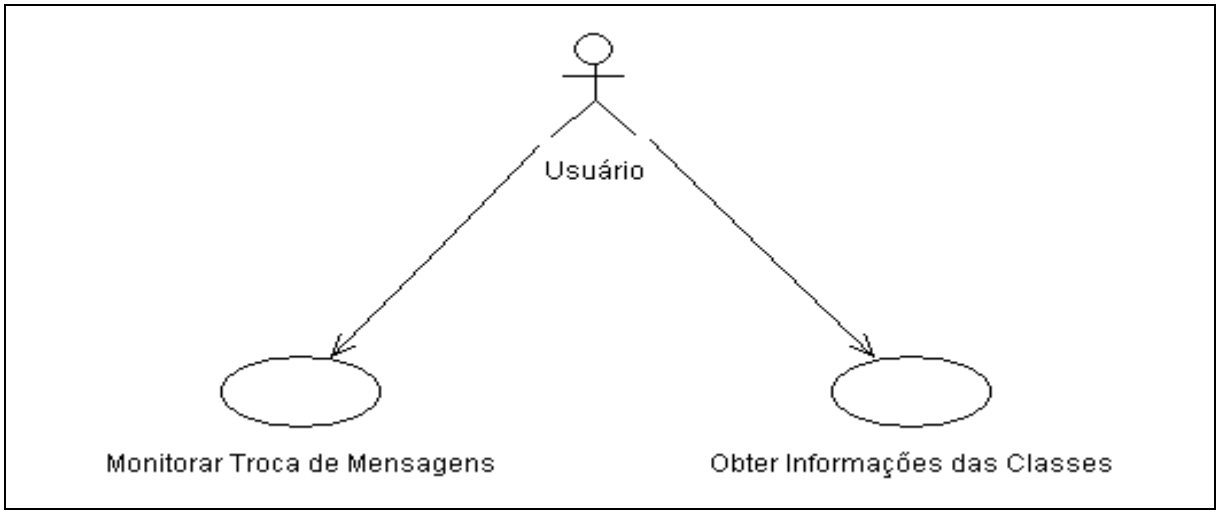
Para a especificação do protótipo foi utilizada a linguagem UML, através do diagrama de casos de uso, diagrama de classes e diagrama de seqüência. A ferramenta utilizada para a especificação foi o *Rational Rose C++ Demo 4.0.3*. Todos os diagramas descritos serão apresentados nos tópicos a seguir.

5.2.1 DIAGRAMA DE CASO DE USO

Na modelagem deste protótipo foram observados dois casos de uso, mostrados na figura 9:

a) obter informações das classes: o usuário obtém todas as informações referentes às classes da aplicação;
b) monitorar a troca de mensagens: o usuário tem a possibilidade de monitorar o comportamento dos objetos da aplicação.

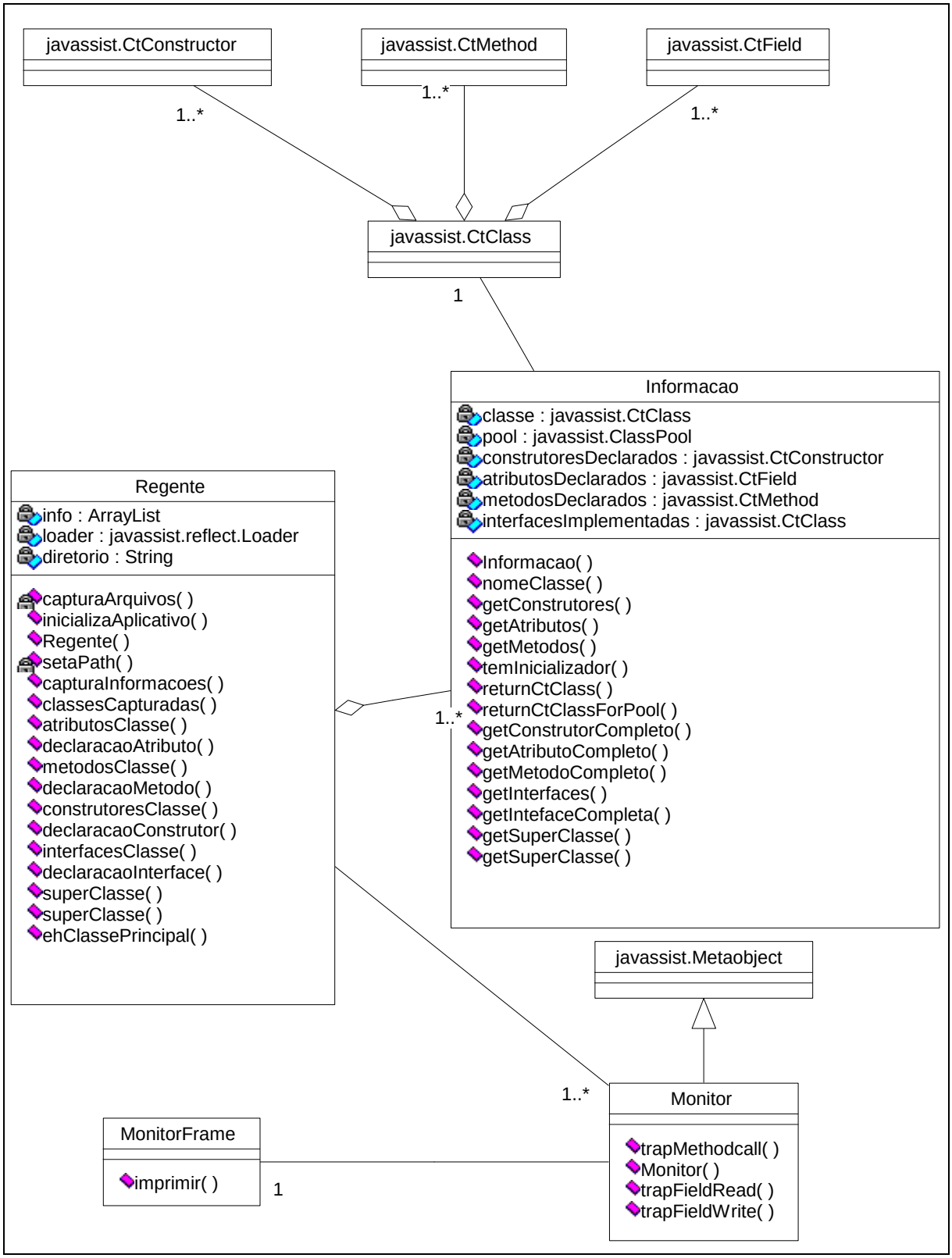
FIGURA 9: DIAGRAMA DE CASO DE USO



5.2.2 DIAGRAMA DE CLASSES

Na figura 10 tem-se o diagrama de classes que fornecerá uma visão geral das classes do modelo proposto e logo a seguir uma descrição completa de cada uma dessas classes.

FIGURA 10: DIAGRAMA DE CLASSES



a) Regente: esta pode ser considerada a principal classe do processo de reflexão do programa Java, pois ela é responsável pelo gerenciamento de todas as operações

reflexivas como tornar as classes do programa Java reflexivas (prontas para serem monitoradas) e apresentar todas as informações requisitadas das estruturas das classes do programa submetido ao protótipo;

b) Informacao: classe que armazena e seleciona informações referentes às classes informadas pela classe Reflexão. Esta classe manipula diretamente a classe *CtClass* da Javassist obtendo diversas informações como métodos, atributos, construtores, interfaces e outras. Tendo estas informações a classe Informacao as arranja de uma forma mais amigável para manipulação;

c) Monitor: responsável pela interceptação dos métodos e alterações em valores dos atributos de objetos das classes do nível base. A classe Regente torna uma classe do nível base reflexiva associando-a a classe Monitor. Assim, sempre que um objeto da classe base for instanciado, um objeto da classe Monitor será instanciado pela Javassist, tornando-se um metaobjeto. Monitor estende a classe *Metaobject* nativa da Javassist;

d) MonitorFrame: esta classe tem a responsabilidade de imprimir todas as mensagens enviadas pela classe Monitor.

As classes *CtClass*, *CtMethod*, *CtField* e *CtConstructor* fazem parte da biblioteca Javassist. Cada uma delas representa uma classe ou um membro dela, como *CtField*, por exemplo, que representa um atributo de uma classe.

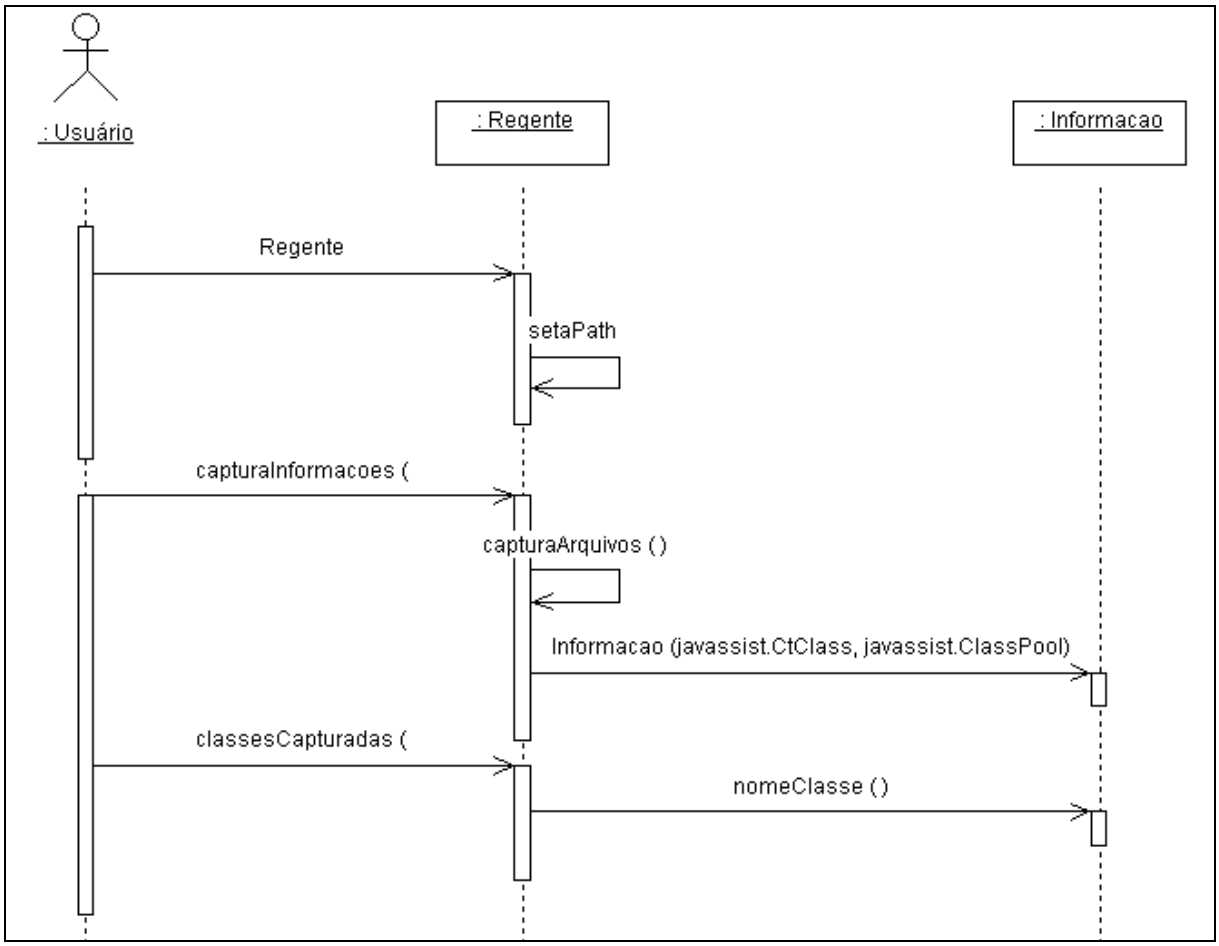
A classe *Metaobject* também mencionada no diagrama de classes representa o metaobjeto associado a um objeto do nível base. Este metaobjeto será invocado toda vez que ocorrer uma mudança no comportamento no objeto do nível base associado a ele.

5.2.3 DIAGRAMA DE SEQÜÊNCIA

Para cada caso de uso definido foi gerado um diagrama de seqüência correspondente, o que significa que foram desenvolvidos dois diagramas de seqüência, e que serão devidamente analisados.

O primeiro diagrama de seqüência, por ser bastante extenso foi dividido em 3 (três) partes que serão analisadas nas figuras a seguir (Fig. 11, 12 e 13) juntamente com o segundo diagrama.

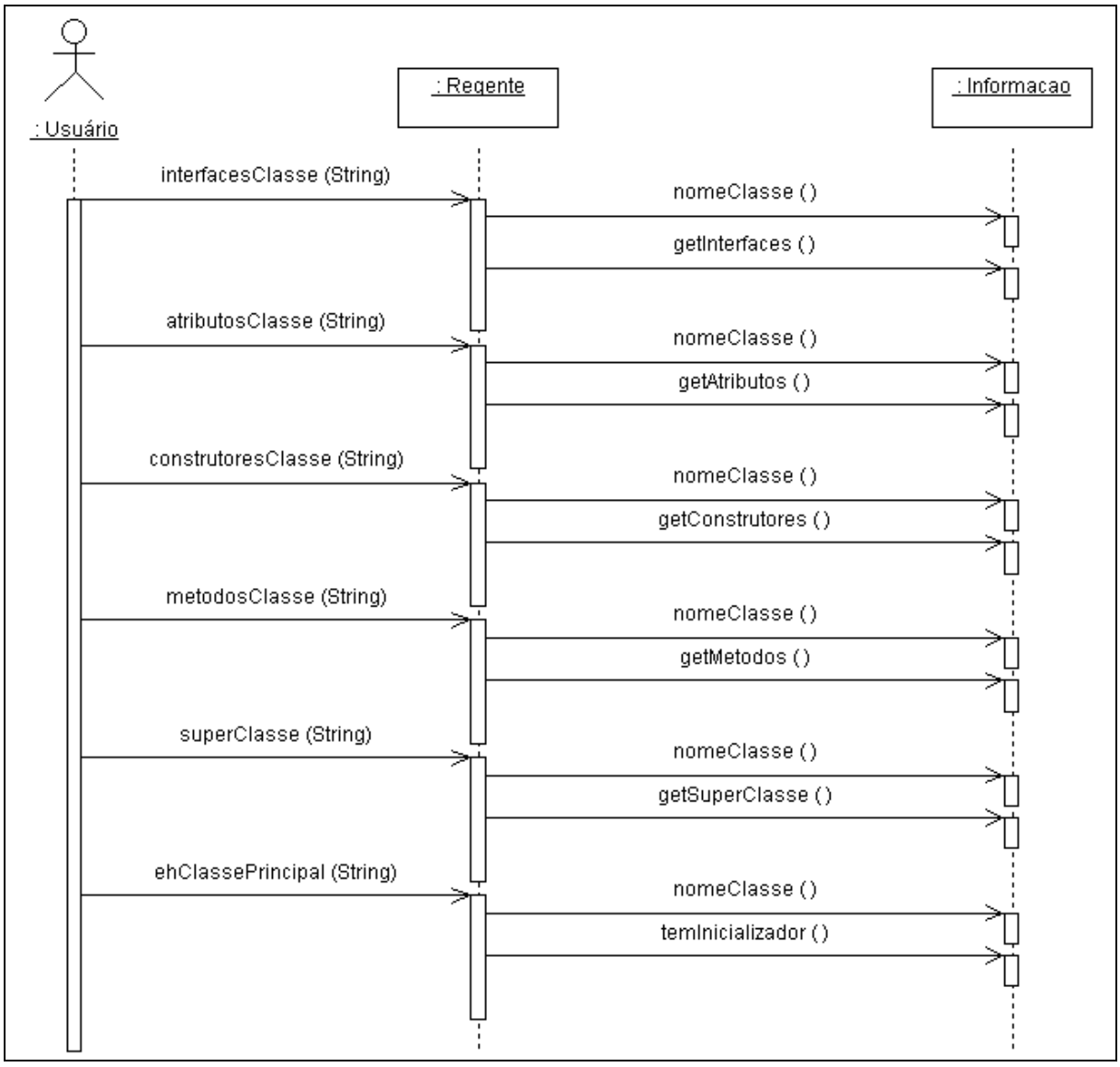
FIGURA 11: DIAGRAMA DE SEQUÊNCIA – OBTER INFORMAÇÕES DAS CLASSES (PARTE 1)



O monitoramento da aplicação analisada pelo protótipo só torna-se possível após a execução desta primeira etapa de obtenção das informações das classes. Como mostra a figura 11, inicialmente um objeto da classe Regente é instanciado tendo como parâmetro uma *string* que representa o caminho até o diretório onde se encontra o aplicativo. O construtor da classe Regente ao receber esta *string* chama o método setaPath(String) que a irá inserir no *ClassPath* da Java. Em seguida o método capturaInformacoes() é chamado na classe Regente, este chama o método capturaArquivos(), da própria classe, que possui a função de capturar todos os arquivos **.class** do diretório passado no construtor e também instancia um objeto da classe Informacao. Na seqüência o método classesCapturadas() é invocado pela interface na classe

Regente. Este irá buscar o nome das classes capturadas no diretório e as retornará em um vetor de *strings.*

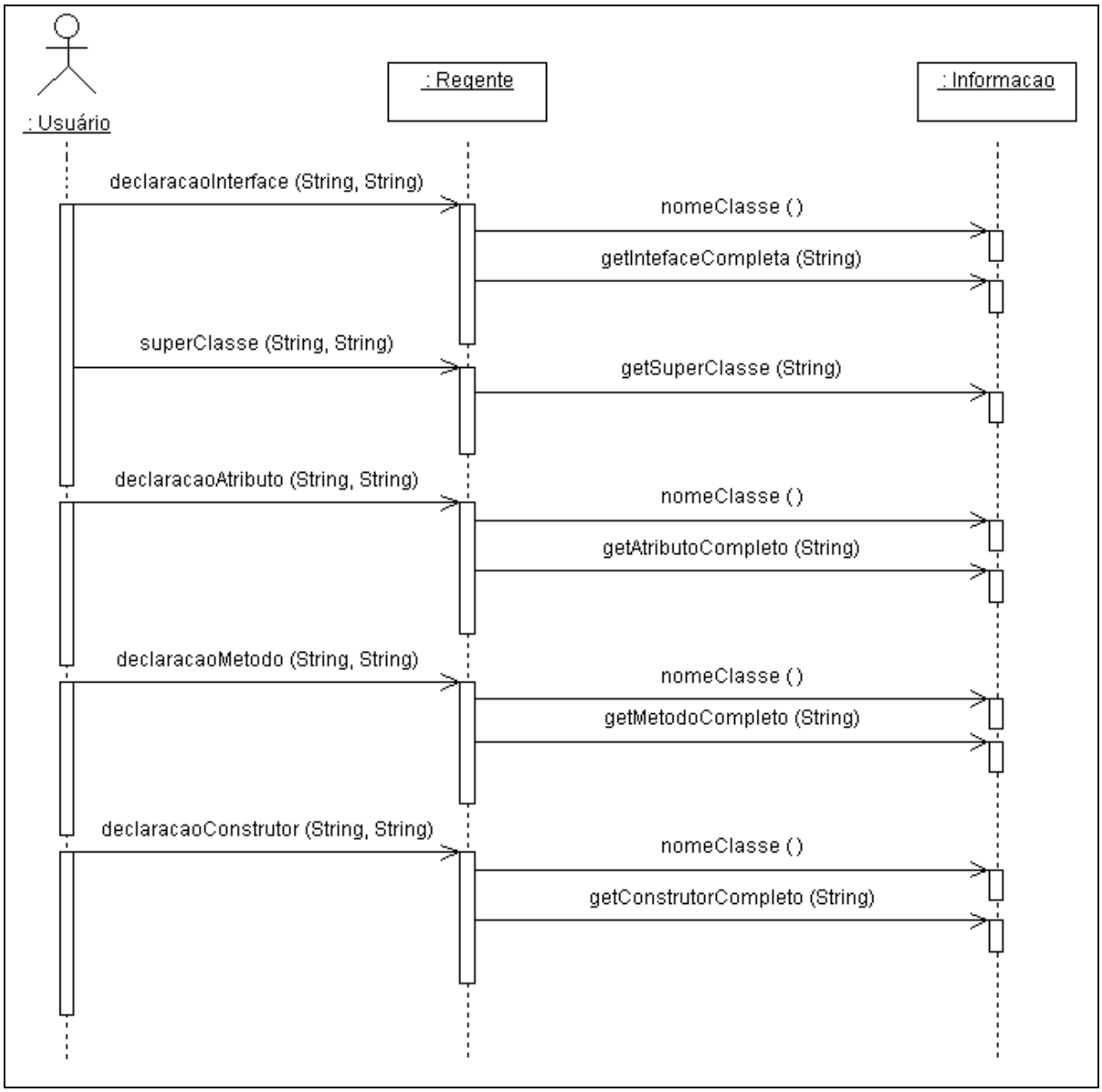
FIGURA 12: DIAGRAMA DE SEQUÊNCIA – OBTER INFORMAÇÕES DAS CLASSES (PARTE 2)



Nesta parte do diagrama (Fig. 12) a interface requisita as interfaces, atributos, construtores, métodos e superclasse da classe selecionada pelo usuário. Os métodos funcionam de forma idêntica: primeiro o método da classe Regente é invocado, interfacesClasse(String) por exemplo, trazendo consigo o nome da classe como parâmetro. Este irá buscar o objeto da classe Informacao correspondente à classe passada como parâmetro e as interfaces implementadas por ela. A única exceção é o método

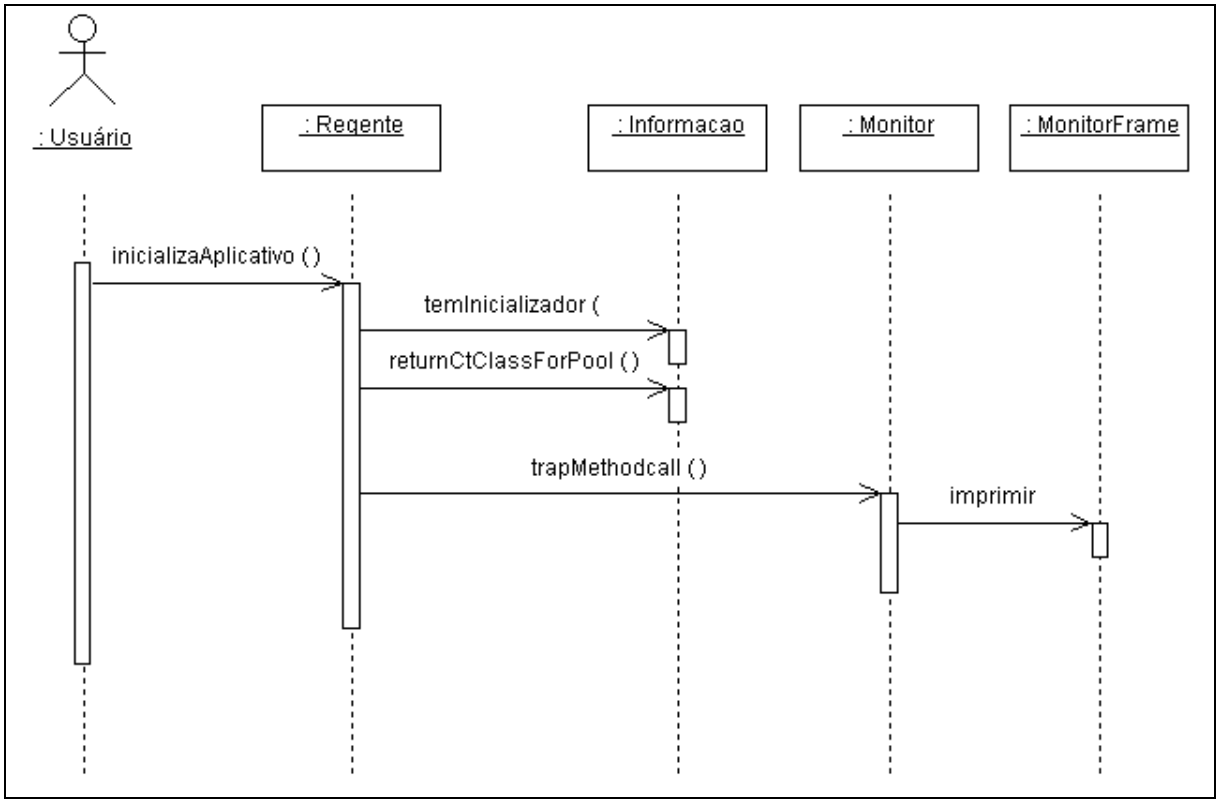
ehClassePrincipal(String) que tem a finalidade de verificar se a classe cujo nome foi passado como parâmetro tem o inicializador de classe *main*.

FIGURA 13: DIAGRAMA DE SEQUÊNCIA – OBTER INFORMAÇÕES DAS CLASSES (PARTE 3)



Esta é a terceira e última parte do primeiro diagrama de seqüência. Tem-se aqui a busca pela declaração completa dos membros da classe inspecionada. Na figura 13 pode-se observar que a interface faz várias requisições ao objeto da classe Regente que as repassa aos objetos da classe Informacao.

FIGURA 14: DIAGRAMA DE SEQUÊNCIA – MONITORAR TROCA DE MENSAGENS



O diagrama de seqüência da figura 14 trata do monitoramento do comportamento dos objetos da aplicação analisada. Para que exista o monitoramento, as classes da aplicação devem ser previamente carregadas pelo objeto de Regente, o que ocorreu no diagrama da figura 11. Em determinado momento da execução do protótipo a interface irá requisitar ao Regente a inicialização do aplicativo, como mostra o método inicializaAplicativo(). Este irá verificar junto à classe Informacao qual das classes carregadas possui o método inicializador *main* e buscará desta o *ClassPool* que possui informações referentes a ela. Após isto, pelo processo de reflexão comportamental é invocado o construtor da classe Monitor que passará a invocar o método imprimir(String) da classe MonitorFrame sempre que interceptar uma mudança no comportamento dos objetos do aplicativo.

5.3 IMPLEMENTAÇÃO DO PROTÓTIPO

A implementação do protótipo foi realizada no ambiente de programação Borland JBuilder 7 Enterprise Trial, escolhido por disponibilizar condições de desenvolvimento do protótipo proposto.

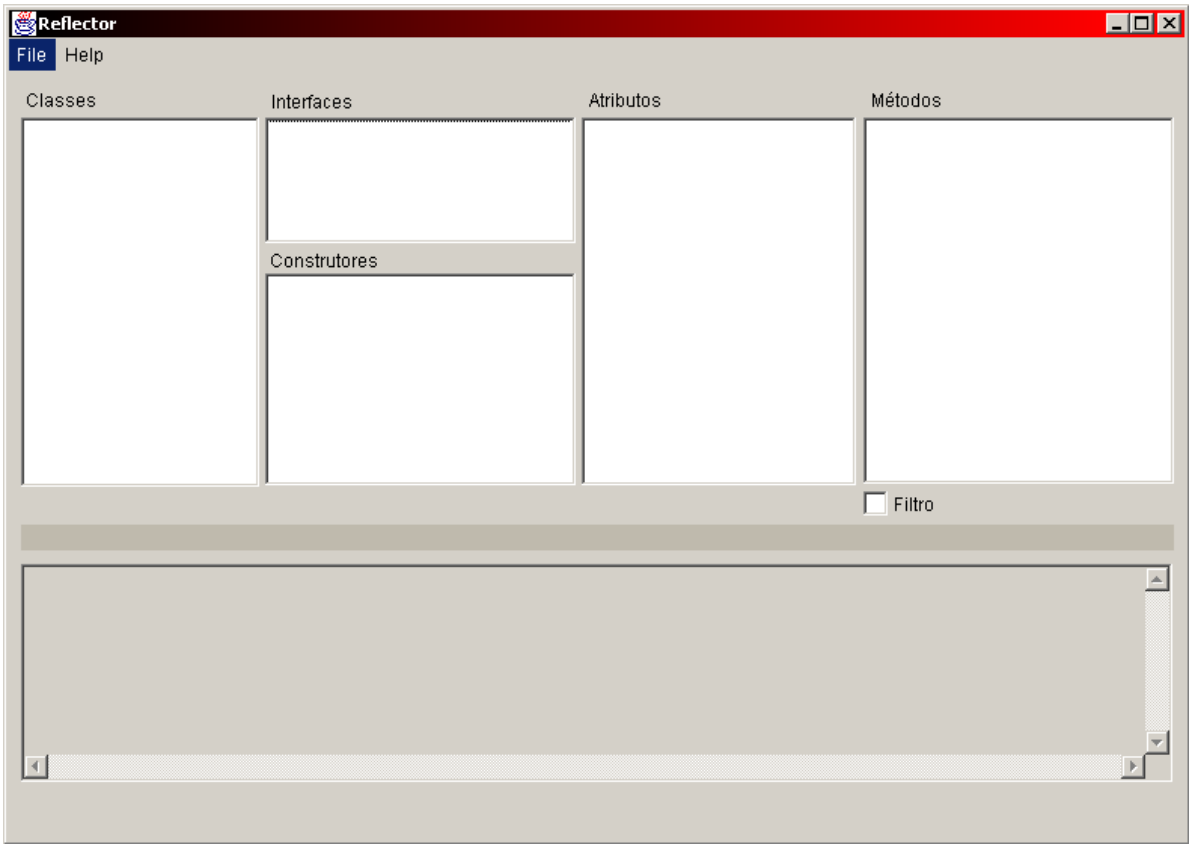
Inicialmente a implementação seria realizada no ambiente Microsoft Visual J++ 6.0, mas devido a problemas de compatibilidade da biblioteca de classes da Javassist optou-se pelo ambiente JBuilder.

Todo o processo de desenvolvimento foi baseado nos conceitos de orientação a objetos além dos conceitos e técnicas de reflexão computacional para possibilitar o monitoramento do aplicativo Java.

Neste tópico serão apresentadas algumas das partes mais relevantes da implementação do protótipo, assim como as telas e funcionamento do mesmo, levando-se em consideração a validação dos conceitos descritos nos capítulos anteriores.

A tela principal permite ao usuário selecionar o aplicativo que sofrerá inspeção. Esta tela possui uma lista para cada um dos membros mais importantes da estrutura de um programa orientado a objetos, como atributos, métodos, interfaces e outros, como pode ser verificado na figura 15. O menu é constituído pelas opções Abrir..., que dispõe os arquivos e diretórios para seleção, Rodar Aplicativo, que inicializa o aplicativo para monitoramento de seu comportamento, Sair, para sair do protótipo, Filtro, que também aparece como um botão de checagem abaixo da lista de métodos e serve para excluir da lista os métodos internos da Java para uma melhor visualização.

FIGURA 15: TELA PRINCIPAL



Ao iniciar a inspeção de um aplicativo, a classe Regente manterá um vetor de objetos de Informacao responsáveis pela extração de informações das classes capturadas do aplicativo. Durante a inspeção, cada classe do aplicativo é associada a um *ClassPool*, que gerencia a obtenção de um arquivo de classe assim como sua modificação, e a um *CtClass*, que representa uma classe na Javassist. Cada *CtClass* está contido dentro de um *ClassPool* que recebe o caminho do diretório em que se encontra a classe associada a ele (*ClassPath*) antes da carga da classe inspecionada pela classe *Loader* da Javassist. No quadro 11 pode-se acompanhar o código para tornar uma classe reflexiva. Para tanto utiilzou-se o método *makeReflective()* que recebe como parâmetro o nome da classe que se deseja inspecionar, o nome da classe que irá monitorar o comportamento dos objetos e a classe da Javassist responsável pela interceptação de mensagens.

QUADRO 11: TORNANDO UMA CLASSE REFLEXIVA

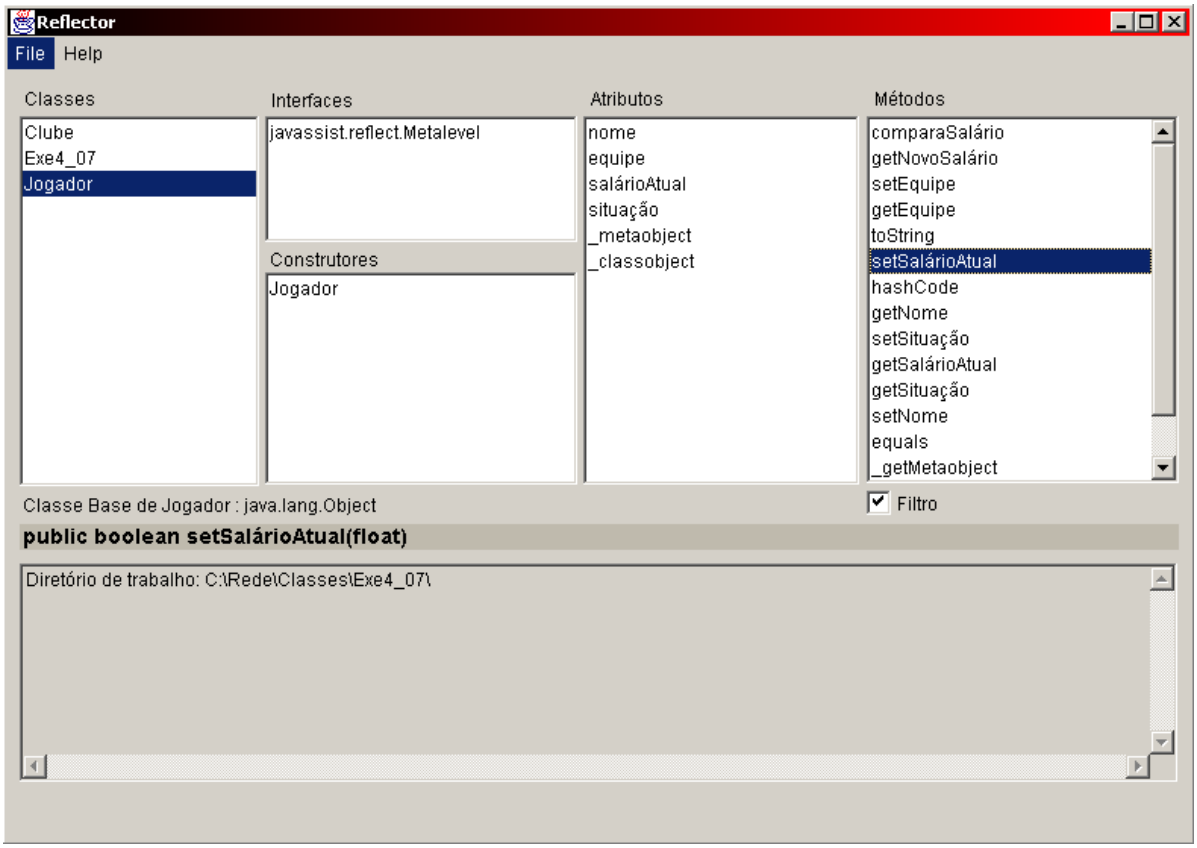
```
ClassPool pool = ClassPool.getDefault();
for (int i = 0; i < arquivos.size(); i++)
{
```

```
        String nomeClasse = ((File)arquivos.get(i)).getName().substring(0,
              ((File)arquivos.get(i)).getName().lastIndexOf("."));
        try
        {
            // seta caminho da classe
            pool.insertClassPath(diretorio);

            // torna a classe reflexiva
            loader.makeReflective(pool.get(nomeClasse).getName(), "reflectorapp.Monitor",
                    "javassist.reflect.ClassMetaobject");
        }
        catch (Throwable t) {
            throw t;
        }
}
```

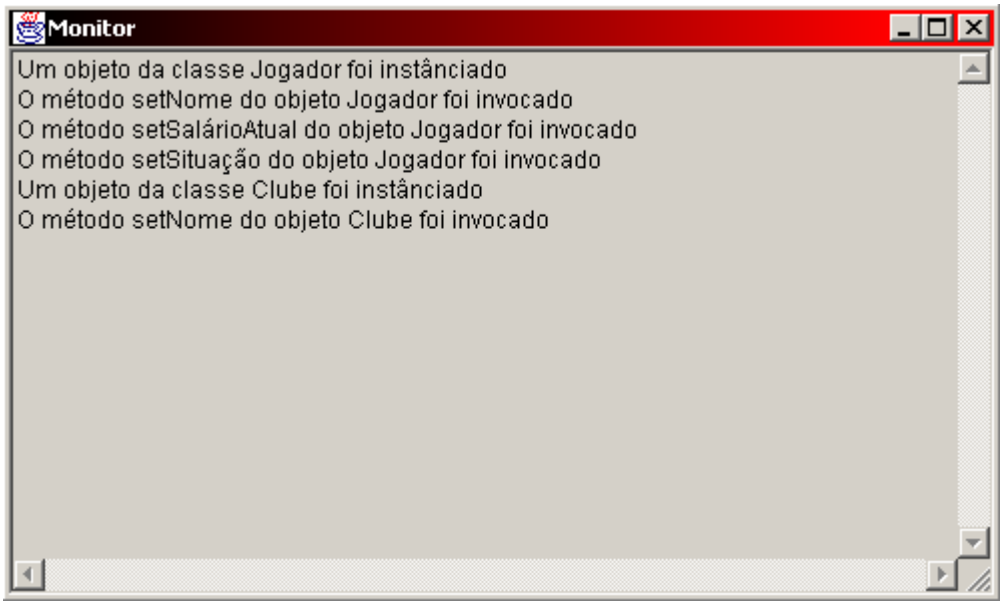
As informações referentes à classe são dispostas na tela como mostra a figura 16. Quando o usuário seleciona um membro da classe em uma das listas esta é mostrada juntamente com seus modificadores, tipos de retorno e tipos de parâmetros.

FIGURA 16: TELA COM INFORMAÇÕES CARREGADAS



No momento em que o usuário inicializa o aplicativo, o protótipo habilita a tela de monitoramento, que apresentará mensagens informando novas instâncias de classes e mudanças em seu comportamento, como mostra a figura 17. Esta tela é ativada pela classe Monitor, a qual entrará em atividade no momento em que o primeiro objeto da classe principal, ou seja, a classe que possui o método *main*, do aplicativo entrar em atividade.

FIGURA 17: TELA DE MONITORAMENTO



A inicialização do aplicativo dar-se-á através da linha de código descrita abaixo.

```
loader.run(s, new String[] {});
```

Para que a execução do aplicativo ocorra de forma correta o objeto *loader* deve ter sido inicializado anteriormente como mostrado no quadro 11. Os parâmetros informados ao método *run* são o nome da classe principal (aquela que possui o *main*) e seus parâmetros, se houver.

5.3.1 INCLUSÃO DA BIBLIOTECA JAVASSIST

Para que fosse possível utilizar os recursos do mecanismo de reflexão no JBuilder, foram realizados os seguintes procedimentos:

a) durante o processo de criação da aplicação, preencheu-se o campo *Required Libraries* com o caminho até o pacote de classes Javassist;

b) em cada uma das classes que usaram os benefícios das classes de reflexão da biblioteca Javassist, importou-se as classes do pacote *javassist* e do pacote *javassist.reflect*.

5.4 APLICATIVO QUE SOFRERÁ REFLEXÃO (EXEMPLO)

Para fins de demonstração, foi elaborado um aplicativo muito simples, que contenha em seu código técnicas de orientação a objetos para que os dados de suas classes possam ser inspecionados e seus objetos monitorados pelo protótipo desenvolvido.

5.4.1 ESTUDO DE CASO

O sindicato dos jogadores profissionais de futebol negociou o aumento de salários de toda a categoria, para todos os clubes, de acordo com a tabela seguinte (Tabela 4):

TABELA 4: ESTUDO DE CASO

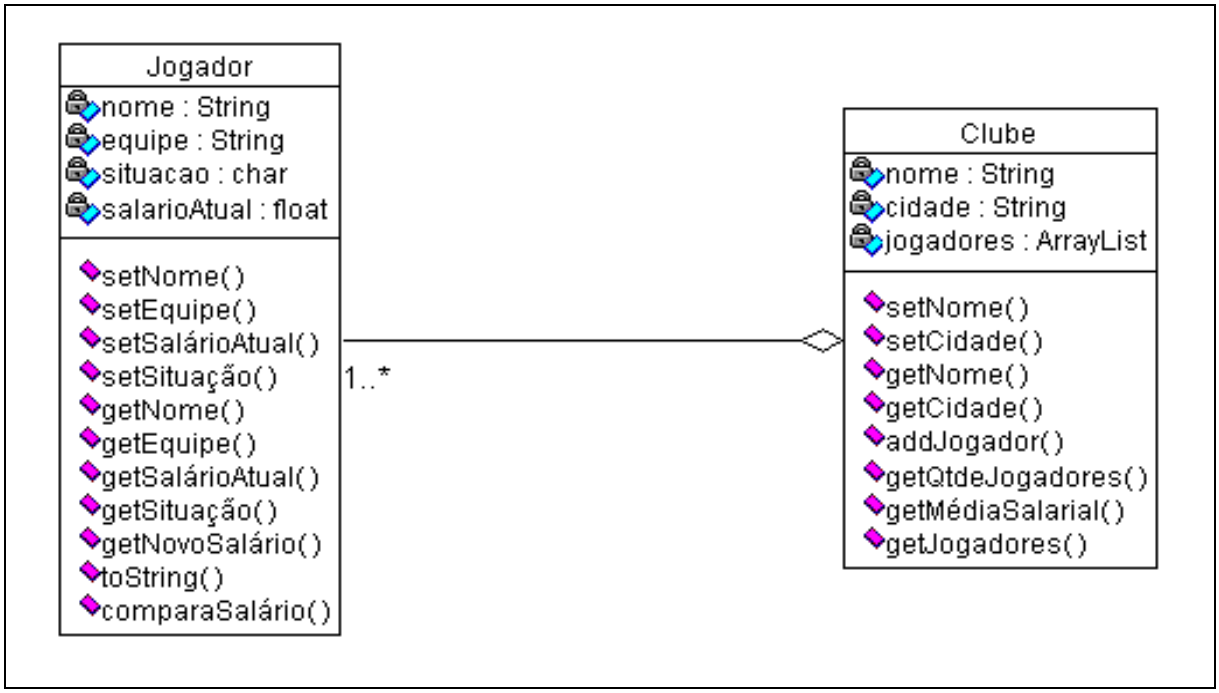
Salário atual (R$)	**Aumento (%)**
0 - 9.000	20
9.001 - 13.000	10
13.001 - 18.000	5
acima de 18.000	0

O programa Java, orientado a objetos, lerá o nome, situação (Titular ou Reserva), equipe (clube) e o salário de cada jogador (até que nome seja “fim”) e imprimirá o nome, salário atual e novo salário. Ao final, também imprimirá o total dos salários atuais e dos novos salários; o nome, salário atual e novo salário do jogador com menor salário atual; a participação percentual dos novos salários dos titulares e dos reservas sobre o total de novos salários; o nome, salário atual e novo salário dos jogadores com novo salário acima da média de novos salários; e os dados de todos os clubes, como, nome do clube e cidade-sede, média salarial e quantidade de jogadores.

O diagrama de classes aparece na figura 18, onde são encontradas as classes que se deseja implementar no sistema de estudo de caso.

O código-fonte Java deste exemplo pode ser visualizado no anexo 4. Nota-se que estas classes não sofreram qualquer modificação prévia em suas estruturas para tornarem-se reflexivas. Esta tarefa é completamente executada pelo protótipo.

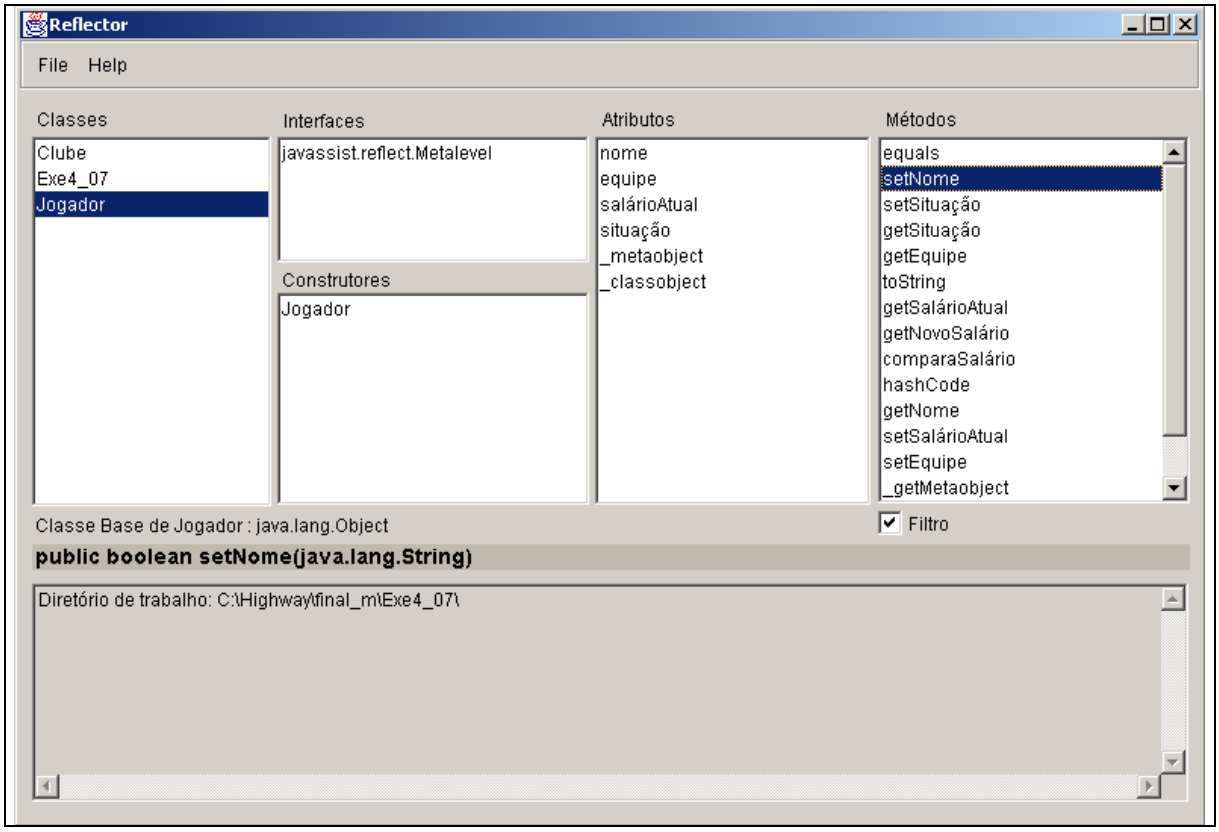
FIGURA 18: DIAGRAMA DE CLASSES DO ESTUDO DE CASO



5.4.2 RESULTADOS

Como demonstra a figura 19, as classes do aplicativo foram inspecionadas corretamente pelo protótipo desenvolvido. As classes apresentadas pela figura 18 estão na lista de classes do protótipo juntamente com a classe Exe4_07 que refere-se a classe de interação com o usuário e por esse motivo não foi representada no diagrama de classes. Verificando-se a classe Jogador no diagrama da figura 19 encontram-se 4 (quatro) atributos: nome, equipe, situacao e salarioAtual, que estão dispostos na lista de atributos da classe jogador mostrada pela figura 19.

FIGURA 19: INSPEÇÃO DAS CLASSES



Continuando a verificação dos resultados chega-se ao monitoramento da aplicação analisada. A aplicação e o protótipo serão apresentados em dois estágios de sua execução possibilitando um acompanhamento mais detalhado do processo.

A figura 20 mostra o aplicativo de teste em execução. No momento em que o usuário informa o nome do jogador o monitor do protótipo entra em atividade informando que um objeto da classe Jogador foi instanciado e que o método setNome() deste objeto foi chamado como apresenta a figura 21.

FIGURA 20: APLICATIVO DE TESTE (PARTE 1)

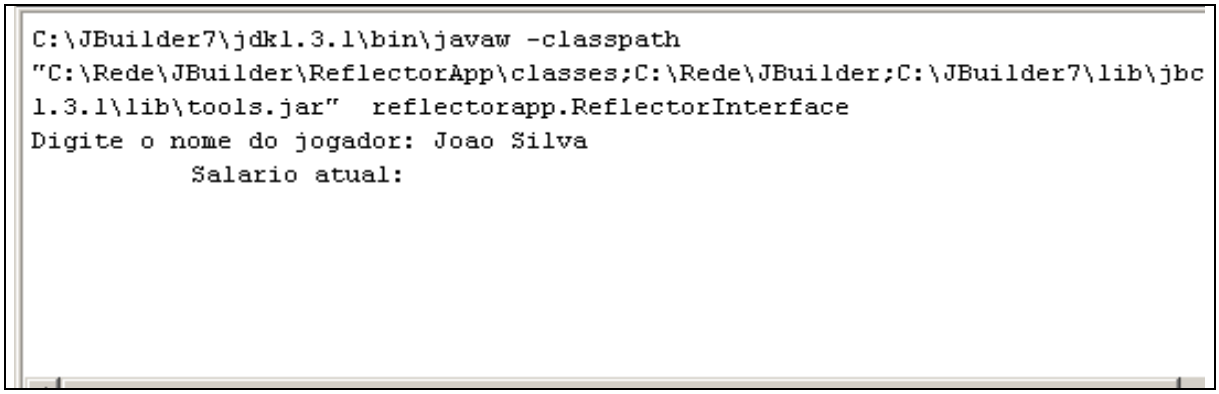
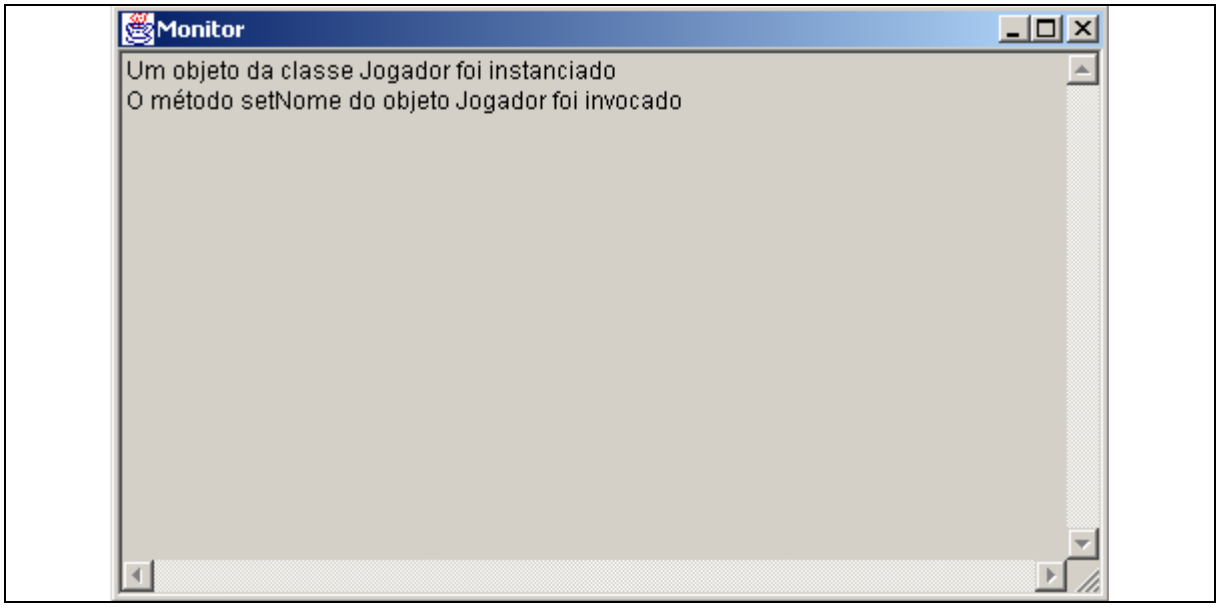


FIGURA 21: MONITORAMENTO (PARTE 1)



O usuário prossegue com a entrada dos dados. Ao informar o nome da equipe na qual o jogador atua (Fig. 22) o monitor do protótipo novamente entra em atividade informando que um objeto da classe Clube foi instanciado e que o método setNome() deste objeto foi chamado como apresenta a figura 23. Nesta figura pode-se verificar também que o monitor interceptou as chamadas aos métodos setSalárioAtual() e setSituação(), dados informados pelo usuário anteriormente como mostra a figura 22.

FIGURA 22: APLICATIVO DE TESTE (PARTE 2)

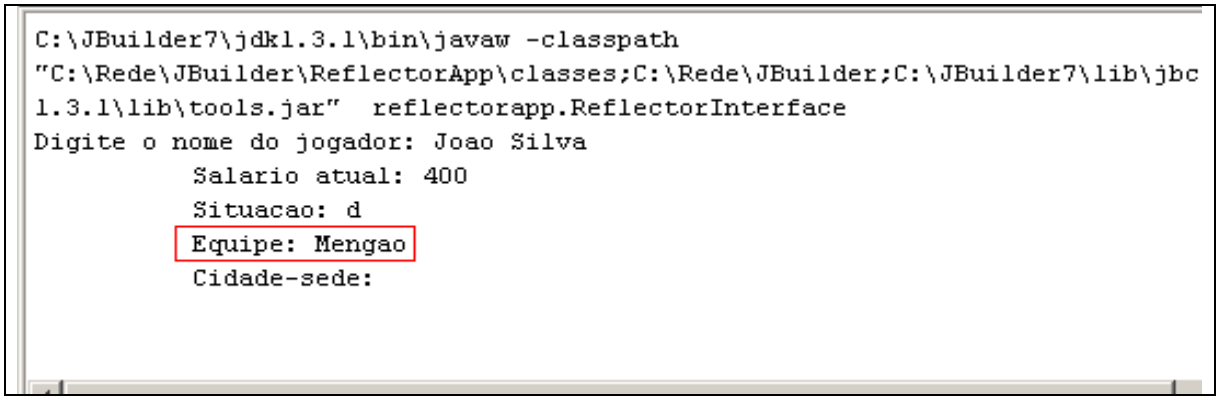
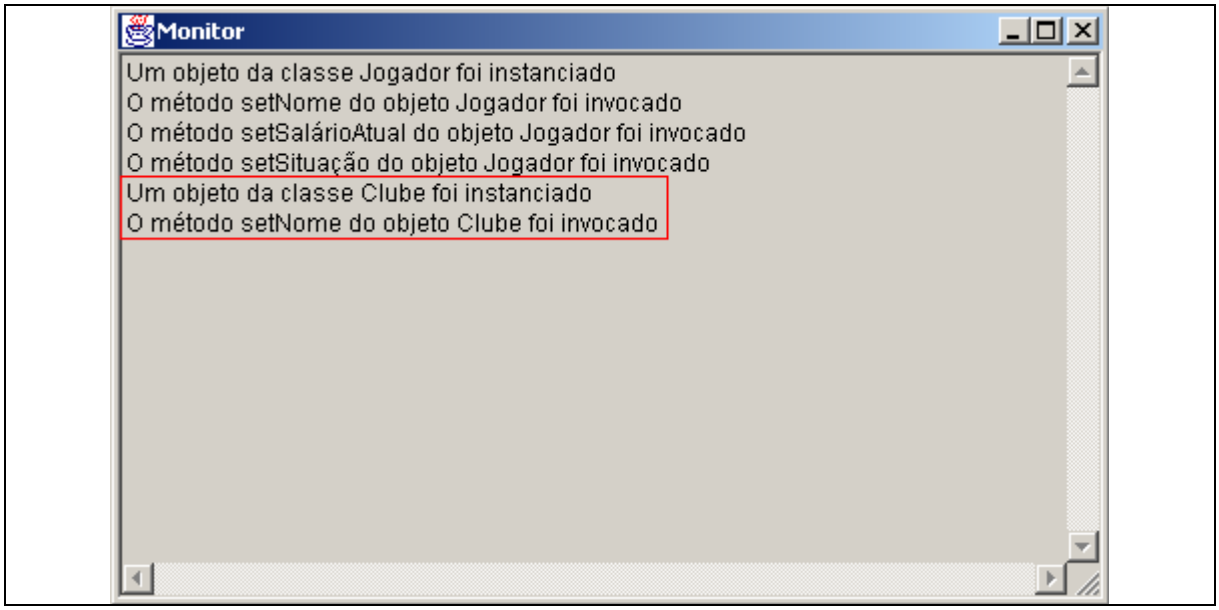


FIGURA 23: MONITORAMENTO (PARTE 2)



6 CONCLUSÕES

O estudo realizado por este trabalho de conclusão de curso sobre as tecnologias de reflexão computacional e orientação a objetos serviu para demonstrar a facilidade de manipulação e introspecção de classes Java com o auxílio da biblioteca Javassist. Apesar da tecnologia não ser muito recente, seus conceitos e funcionalidades estão vindo à tona apenas nos últimos anos. Diversos estudos, como de Shigero Chiba (2001), vêm sendo realizados com o intuito de prover características reflexivas às plataformas em geral, sem sobrecarregar as funcionalidades, eficiência e desempenho das mesmas.

Todos os objetivos pré-estabelecidos foram alcançados. Neste trabalho utilizou-se das vantagens oferecidas pelas tecnologias de reflexão computacional e orientação a objetos, como a possibilidade de monitoramento dos objetos, para o desenvolvimento do protótipo apresentado no capítulo 5. Este não visou atender completamente os conceitos de reflexão computacional apresentados no capítulo 3, mas sim, demostrar de forma genérica como interceptar as mensagens dos objetos, possibilitando mudanças em seu comportamento. O aprofundamento dos conhecimentos da tecnologia de orientação mostrou-se essencial para a conclusão deste trabalho por tratar-se da apresentação de estruturas desta tecnologia e do monitoramento dos objetos dessas estruturas. Foram estudadas também algumas extensões da biblioteca de reflexão Java para utilização da técnica de reflexão comportamental, sendo a Javassist selecionada, por ser de fácil manipulação e por atender a todas as necessidades do protótipo.

À medida que ambientes e linguagens de programação reflexivas tornarem-se mais difundidos, é bastante provável que a utilização de reflexão computacional será tão comum quanto a utilização de orientação a objetos. O desenvolvimento deste protótipo mostrou que as técnicas de reflexão podem ser facilmente utilizadas, mesmo no dia-a-dia do desenvolvimento de software para reduzir a complexidade e aumentar a facilidade de manutenção. Isto ocorre em um exemplo simples onde se dispõem de classes que necessitam ser instanciadas conforme determinado comportamento do programa. Desta forma é possível definir, em tempo de execução, qual classe será instanciada em um código mais simples e, desta forma, de fácil manutenção. Vale lembrar que a reflexão computacional fica em um nível diferente do nível operacional, mas ela não resolve os problemas do domínio da aplicação.

6.1 LIMITAÇÕES

Devido à restrição de tempo o ambiente não dispôs de uma apresentação gráfica mais agradável para o usuário, gerando dessa forma uma deficiência na visualização do comportamento dos objetos e das estruturas das classes analisadas.

Como o título deste trabalho já sugere, o protótipo não permite reflexão em estruturas de classes de aplicativos desenvolvidos em outras linguagens a não ser a Java.

Compreende de forma generalizada os conceitos de reflexão comportamental, pois apenas executa o monitoramento do comportamento dos objetos sem alterar seu comportamento padrão como pregam os conceitos descritos no capítulo 3.

6.2 EXTENSÕES

Como sugestão para continuação deste trabalho inclui-se:

a) melhorar o ambiente gráfico do protótipo;
b) aperfeiçoar o protótipo de forma que este possa atender por completo os conceitos apresentados no capítulo 3;
c) utilizar o modelo apresentado para o desenvolvimento de uma ferramenta específica, como um gerenciador de erros de aplicação ou uma ferramenta de execução de testes automatizados.

ANEXO 1 – CLASSE INFORMACAO

Classe responsável pela reflexão estrutural da classe inspecionada.

```
import javassist.*;

public class Informacao
{
  private CtClass classe;
  private ClassPool pool;

  public Informacao(CtClass c, ClassPool pool)
  {
    this.classe = c;
    this.pool = pool;
  }

  public CtClass returnCtClassForPool() throws Throwable
  {
    try
    {
      return pool.get(nomeClasse());
    }
    catch (java.lang.Throwable t)
    {
      throw t;
    }
  }

  public CtClass returnCtClass()
  {
    return classe;
  }

  public String nomeClasse()
  {
    return classe.getName();
  }

  // informações sobre construtores
  private CtConstructor[] construtoresDeclarados = null;

  public String[] getConstrutores() throws Throwable
  {
    construtoresDeclarados = classe.getConstructors();
    String[] construtores = new String[construtoresDeclarados.length];

    for (int i = 0; i < construtoresDeclarados.length; i++)
      construtores[i] = construtoresDeclarados[i].getName();

    return construtores;
  }

  public String getConstrutorCompleto(String nomeConstrutor) throws Throwable
  {
    if (construtoresDeclarados != null)
    {
      for (int i = 0; i < construtoresDeclarados.length; i++)
```

```
    {
      if (construtoresDeclarados[i].getName().equals(nomeConstrutor))
      {
        String construtor = Modifier.toString(
                  construtoresDeclarados[i].getModifiers());
        construtor += " " + construtoresDeclarados[i].getName() + "(";
        CtClass[] parametros = construtoresDeclarados[i].getParameterTypes();

        for (int j = 0; j < parametros.length;)
        {
          construtor += parametros[j++].getName();
          if (j < parametros.length)
            construtor += ", ";
        }
        construtor += ")";
        return construtor;
      }
    }
  }
  return null;
}

// informações sobre atributos
private CtField[] atributosDeclarados = null;

public String[] getAtributos() throws Throwable
{
  atributosDeclarados = classe.getDeclaredFields();

  String[] s = new String[atributosDeclarados.length];
  String temp;

  for (int i = 0; i < atributosDeclarados.length; i++)
    s[i] = atributosDeclarados[i].getName();

  return s;
}

public String getAtributoCompleto(String nomeAtributo) throws Throwable
{
  if (atributosDeclarados != null)
  {
    for (int i = 0; i < atributosDeclarados.length; i++)
    {
      if (atributosDeclarados[i].getName().equals(nomeAtributo))
      {
        String atributo = Modifier.toString(
                  atributosDeclarados[i].getModifiers());
        atributo += " " + atributosDeclarados[i].getType().getName();
        atributo += " " + atributosDeclarados[i].getName();

        return atributo;
      }
    }
  }
  return null;
}

// informações sobre metodos
private CtMethod[] metodosDeclarados = null;

public String[] getMetodos() throws Throwable
{
  metodosDeclarados = classe.getDeclaredMethods();
  String[] metodos = new String[metodosDeclarados.length];
```

```
  for (int i = 0; i < metodosDeclarados.length; i++)
    metodos[i] = metodosDeclarados[i].getName();

  return metodos;
}

public String getMetodoCompleto(String nomeMetodo) throws Throwable
{
  if (metodosDeclarados != null)
  {
    for (int i = 0; i < metodosDeclarados.length; i++)
    {
      if (metodosDeclarados[i].getName().equals(nomeMetodo))
      {
        String metodo = Modifier.toString(metodosDeclarados[i].getModifiers());
        metodo += " " + metodosDeclarados[i].getReturnType().getName();
        metodo += " " + metodosDeclarados[i].getName() + "(";

        CtClass[] parametros = metodosDeclarados[i].getParameterTypes();

        for (int j = 0; j < parametros.length;)
        {
          metodo += parametros[j++].getName();
          if (j < parametros.length)
            metodo += ", ";
        }

        metodo += ")";
        return metodo;
      }
    }
  }
  return null;
}

// verifica se a classe possui o metodo main
public boolean temInicializador() throws Throwable
{
  String[] metodos = getMetodos();

  for (int i = 0; i < metodos.length; i++)
  {
    if (metodos[i].indexOf("main") != -1)
      return true;
  }

  return false;
}

// informações sobre interfaces implementadas
private CtClass[] interfacesImplementadas = null;

public String[] getInterfaces() throws Throwable
{
  interfacesImplementadas = classe.getInterfaces();
  String[] interfaces = new String[interfacesImplementadas.length];

  for (int i = 0; i < interfacesImplementadas.length; i++)
    interfaces[i] = interfacesImplementadas[i].getName();

  return interfaces;
}

public String getIntefaceCompleta(String nomeInterface) throws Throwable
```

```
  {
    if (interfacesImplementadas != null)
    {
      for (int i = 0; i < interfacesImplementadas.length; i++)
      {
        if (interfacesImplementadas[i].getName().equals(nomeInterface))
        {
          String s = Modifier.toString(interfacesImplementadas[i].getModifiers());
          s += " " + interfacesImplementadas[i].getName();

          return s;
        }
      }
    }
    return null;
  }

  // retorna a super classe de uma classe
  public String getSuperClasse() throws Throwable
  {
    CtClass superClasse = classe.getSuperclass();
    return superClasse.getName();
  }

  // retorna super classe de uma interface
  public String getSuperClasse(String nomeInterface) throws Throwable
  {
    CtClass[] interfacesDeclaradas = classe.getInterfaces();

    for (int i = 0; i < interfacesDeclaradas.length; i++)
    {
      if (interfacesDeclaradas[i].getName().equals(nomeInterface))
        return interfacesDeclaradas[i].getSuperclass().getName();
    }
    return null;
  }
}
```

ANEXO 2 – CLASSE REGENTE

Classe responsável pela gerência das informações das classes inspecionadas e monitoramento de objetos instanciados.

```
import javassist.*;
import javassist.reflect.*;
import java.util.*;
import java.lang.*;
import java.io.*;

public class Regente
{

  private ArrayList info = new ArrayList();
  private javassist.reflect.Loader loader = new javassist.reflect.Loader();
  private String diretorio;

  public Regente(String diretorioClasses) throws Exception
  {
    try
    {
      File file = new File(diretorioClasses);

      if (file.exists())
        setaPath(file.getPath());
    }
    catch (Exception e)
    {
      throw e;
    }
  }

  private void setaPath(String diretorioClasses)
  {
    //
    // gravar o diretório das classes a serem analisadas no ClassPath da Java
    //
    Properties pr = System.getProperties();
    String classPath = pr.getProperty("java.class.path") + ";" + diretorioClasses;
    pr.setProperty("java.class.path", classPath);
    System.setProperties(pr);

    //
    // guardar o diretório das classes
    //
    diretorio = diretorioClasses;
  }

  public void capturaInformacoes() throws Exception
  {
    ArrayList arquivos = capturaArquivos();

    if (arquivos.size() > 0)
    {
      ClassPool pool = ClassPool.getDefault();

      for (int i = 0; i < arquivos.size(); i++)
```

```
    {
      String nomeClasse = ((File)arquivos.get(i)).getName().substring(0,
            ((File)arquivos.get(i)).getName().lastIndexOf("."));

      try
      {
        //
        // seta caminho da classe
        //
        pool.insertClassPath(diretorio);


        //
        // torna a classe reflexiva
        //
        loader.makeReflective(pool.get(nomeClasse).getName(),
           "reflectorapp.Monitor", "javassist.reflect.ClassMetaobject");


        //
        // carrega o array de objetos de informacao
        //
        info.add(new
           Informacao(pool.get(((File)arquivos.get(i)).getName().substring(0,
           ((File)arquivos.get(i)).getName().lastIndexOf("."))), pool));
      }
      catch (javassist.NotFoundException nf)
      {
        throw nf;
      }
      catch (javassist.CannotCompileException cc)
      {
        throw cc;
      }
    }
  }
}

private ArrayList capturaArquivos()
{
  ArrayList arqs = new ArrayList();

  File file = new File(diretorio);

  if (file.exists())
  {
    File[] files = file.listFiles();

    for (int i = 0; i < files.length; i++)
    {
      if (files[i].getName().indexOf(".class") != -1)
        arqs.add(files[i]);
    }
  }

  return arqs;
}

public void inicializaAplicativo() throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).temInicializador())
    {
    String s = ((Informacao)info.get(i)).returnCtClassForPool().getName();
```

```
    if (loader != null)
      loader.run(s, new String[] {});
    }
  }
}

public boolean ehClassePrincipal(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
    {
      if (((Informacao)info.get(i)).temInicializador())
        return true;
    }
  }

  return false;
}

public String[] classesCapturadas()
{
  String[] classes = new String[info.size()];

  for (int i = 0; i < info.size(); i++)
    classes[i] = ((Informacao)info.get(i)).nomeClasse();

  return classes;
}

public String[] atributosClasse(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getAtributos();
  }

  return null;
}

public String declaracaoAtributo(String nomeClasse, String nomeAtributo)
     throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getAtributoCompleto(nomeAtributo);
  }

  return null;
}

public String[] metodosClasse(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
       return ((Informacao)info.get(i)).getMetodos();
  }

  return null;
}

public String declaracaoMetodo(String nomeClasse, String nomeMetodo)
```

```
      throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getMetodoCompleto(nomeMetodo);
  }

  return null;
}

public String[] construtoresClasse(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getConstrutores();
  }

  return null;
}

public String declaracaoConstrutor(String nomeClasse, String nomeConstrutor)
     throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getConstrutorCompleto(nomeConstrutor);
  }

  return null;
}

public String[] interfacesClasse(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getInterfaces();
  }

  return null;
}

public String declaracaoInterface(String nomeClasse, String nomeInterface)
      throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
      return ((Informacao)info.get(i)).getIntefaceCompleta(nomeInterface);
  }

  return null;
}

public String superClasse(String nomeClasse) throws Throwable
{
  for (int i = 0; i < info.size(); i++)
  {
    if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
    {
    return ((Informacao)info.get(i)).getSuperClasse();
  }
```

```
    }

    return null;
  }

  public String superClasse(String nomeClasse, String nomeInterface)
        throws Throwable
  {
    for (int i = 0; i < info.size(); i++)
    {
      if (((Informacao)info.get(i)).nomeClasse().equals(nomeClasse))
         return ((Informacao)info.get(i)).getSuperClasse(nomeInterface);
    }

    return null;
  }
}
```

ANEXO 3 – CLASSE MONITOR

Classe responsável pelo monitoramento do objeto da classe inspecionada.

```
import javassist.*;
import javassist.reflect.*;

public class Monitor extends javassist.reflect.Metaobject
{
  private static MonitorFrame mf = new MonitorFrame();

  public Monitor(Object self, Object[] args) throws CannotInvokeException
  {
    super(self, args);

    mf.setSize(500, 300);
    mf.show();

    mf.imprimir("Um objeto da classe " + self.getClass().getName() +
                " foi instânciado\n");
  }

  public Object trapMethodcall(int identifier, Object[] args) throws Throwable
  {
    mf.imprimir("O método " + getMethodName(identifier) + " do objeto " +
                getObject().getClass().getName() + " foi invocado\n");
    return super.trapMethodcall(identifier, args);
  }

  public Object trapFieldRead(String name)
  {
    mf.imprimir("O atributo " + name + " do objeto " +
                    getObject().getClass().getName() + " foi lido\n");
    return super.trapFieldRead(name);
  }

  public void trapFieldWrite(String name, Object value)
  {
    mf.imprimir("O atributo " + name + " do objeto " +
                    getObject().getClass().getName() + " foi alterado\n");
    super.trapFieldWrite(name, value);
  }
}
```

ANEXO 4 – CLASSES DO ESTUDO DE CASO

Classes utilizadas para testar a funcionalidade do protótipo.

```
import java.io.*;
import java.util.*;
public class Exe4_07
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader teclado = new BufferedReader(new
             InputStreamReader(System.in));
        HashMap clubes = new HashMap();
        Clube clubeAtual;
        Jogador jogadorAtual, pobreCoitado=null;
        String s;
        float totalAtual, totalNovosRes, totalNovosTit;
        totalAtual= totalNovosRes= totalNovosTit = 0;
        int totalJog = 0;

        System.out.print("Digite o nome do jogador: ");
        s = teclado.readLine();
        while (!s.equalsIgnoreCase("fim"))
        {   jogadorAtual = new Jogador();
            jogadorAtual.setNome(s);
            System.out.print("\tSalario atual: ");
            s = teclado.readLine();
            jogadorAtual.setSalárioAtual(Float.parseFloat(s));

            System.out.print("\tSituacao: ");
            s = teclado.readLine();
            s.toUpperCase();
            jogadorAtual.setSituação(s.charAt(0));

            System.out.print("\tEquipe: ");
            s = teclado.readLine();
            s.toUpperCase();
            clubeAtual = (Clube) clubes.get(s);
            if (clubeAtual == null) // não encontrou clube com o nome digitado
            {   clubeAtual = new Clube();
                clubeAtual.setNome(s);
                System.out.print("\tCidade-sede: ");
                s = teclado.readLine();
                clubeAtual.setCidade(s);

                clubes.put(clubeAtual.getNome(), clubeAtual);
            }

            clubeAtual.addJogador(jogadorAtual);
            jogadorAtual.setEquipe(clubeAtual.getNome());
            System.out.println(jogadorAtual);

            // itens 1 e 3
            totalJog++;
            totalAtual += jogadorAtual.getSalárioAtual();
            if (jogadorAtual.getSituação() == 'T') // titular
                totalNovosTit += jogadorAtual.getNovoSalário();
            else
                totalNovosRes += jogadorAtual.getNovoSalário();
```

```
            // item 2
            if (jogadorAtual.comparaSalário(pobreCoitado))
                pobreCoitado = jogadorAtual;

            System.out.print("Digite o nome do jogador (fim para encerrar): ");
            s = teclado.readLine();
        } // fim do while de leitura de jogadores

        // item 1
        System.out.println("Total de salários atuais = R$ "+totalAtual);
        System.out.println("Total de novos salários  =
              R$ "+(totalNovosTit+totalNovosRes));

        // item 2
        System.out.println("\nJogador com menor salário atual "+pobreCoitado);

        // item 3
        System.out.println("Participação % nos novos salários");
        System.out.println("Titulares =
             "+totalNovosTit/(totalNovosTit+totalNovosRes)*100+"%");
        System.out.println("Reservas  =
             "+totalNovosRes/(totalNovosTit+totalNovosRes)*100+"%");

        // itens 4 e 5
        ArrayList al;
        float média = (totalNovosTit+totalNovosRes) / totalJog;
        Iterator c = clubes.values().iterator();
        while (c.hasNext())
        {   clubeAtual = (Clube) c.next();
            System.out.println("\n=============== Clube "+clubeAtual.getNome()+
                    " =============");
            System.out.println("Cidade-sede: "+clubeAtual.getCidade());
            System.out.print("Média salarial = R$ "+clubeAtual.getMédiaSalarial());
            System.out.println("\t Quantidade de
                    jogadores:"+clubeAtual.getQtdeJogadores());
            al = clubeAtual.getJogadores();
            for (int i=0;i<al.size();i++)
            {   jogadorAtual = (Jogador) al.get(i);
                if (jogadorAtual.getNovoSalário() > média)
                    System.out.println(jogadorAtual);
            } // fim do for
        }// fim do while

    } // fim do main
} // fim da classe



import java.util.ArrayList;

public class Clube
{   // Atributos
    private String nome, cidade;
    private ArrayList jogadores = new ArrayList();

    // Métodos
    public boolean setNome(String str)
    {   nome = str;
        return true;
    }
    public boolean setCidade(String str)
    {   if (str == null || str.equals(""))
            return false;
        cidade = str;
```

```
        return true;
    }

    public String getNome()     { return nome;}
    public String getCidade()   { return cidade;}

    public boolean addJogador(Jogador pernaDePau)
    {   jogadores.add(pernaDePau);
        return true;
    }

    public int getQtdeJogadores()
    {   return jogadores.size();
    }

    public float getMédiaSalarial()
    {   float total = 0;
        Jogador j;
        for (int i=0; i<jogadores.size(); i++)
        {   j = (Jogador)jogadores.get(i);
            total += j.getSalárioAtual();
        }
        return (total / jogadores.size());
    }

    public ArrayList getJogadores()     { return jogadores; }
}


public class Jogador {
    // Atributos
    private String nome, equipe;
    private float salárioAtual;
    private char situação;

    // Métodos
    public boolean setNome(String str)
    {   nome = str;
        return true;
    }
    public boolean setEquipe(String str)
    {   equipe = str;
        return true;
    }
    public boolean setSalárioAtual(float valor)
    {   if (valor >=0) {
            salárioAtual = valor;
            return true;
        }
        else
            return false;
    }
    public boolean setSituação(char c)
    {   if (c == 'T' || c == 'R') {
            situação = c;
            return true;
        }
        return false;
    }

    public String getNome()         { return nome;}
    public String getEquipe()       { return equipe;}
    public float getSalárioAtual()  { return salárioAtual;}
    public char getSituação()       { return situação;}
```

```
    public float getNovoSalário()
    {   float indice;
        if (salárioAtual < 9001)
            indice = 1.20f;
        else
            if (salárioAtual < 13001)
                indice = 1.10f;
            else
                if (salárioAtual < 18001)
                    indice = 1.05f;
                else
                    indice = 1;
        return (salárioAtual * indice);
    }

    public String toString()
    {   return ("Nome: "+nome+" da equipe "+equipe+
                "recebe R$ "+salárioAtual+
                " receberá R$ "+this.getNovoSalário());
    }

    // retorna true caso salário deste jogador for menor que o do outro
    public boolean comparaSalário(Jogador outro)
    {   if (outro == null ||
            outro.getSalárioAtual() > this.getSalárioAtual())
            return true;
        else
            return false;
    }
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

ARAÚJO, Diego Lucas de; REIS, Félix Gonçalves dos. **Desenvolvendo com Java 2**. Rio de Janeiro: Book Express, 2000.

ARMSTRONG, Eric. **JBuilder 2 bible**. Foster City: IDG Books Worldwide, 1998.

BARTH, Fabrício Jailson. **Utilização de reflexão computacional para implementação de aspectos não funcionais em um gerenciador de arquivos distribuídos**. 2000. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CESTA, André Augusto. **Tutorial: A linguagem de programação Java**. São Paulo, ago 96. Disponível em: <http://www.dcc.unicamp.br>. Acesso em: 18 ago. 2002.

CHIBA, Shigeru. **That are the best join points?** Tokyo, 2001. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/>. Acesso em: 01 out. 2002.

CHIBA, Shigeru. **Javassist – a reflection-based programming wizard for Java**. Tsukuba, 1998. Disponível em: <http://www.is.tsukuba.ac.jp/~chiba/>. Acesso em: 12 jun. 2002.

COAD, Peter; YOURDON, Edward. **Análise baseada em objetos.** Rio de Janeiro: Campus, 1991.

DEVEGILI, Augusto Jun. **Tutorial sobre reflexão em orientação a objetos**. Florianópolis, abr 2000. Disponível em: <http://www.uvm.edu/~dewey/reflection_manual/>. Acesso em: 23 jun. 2002.

FLANAGAN, David. **Java:** o guia essencial. Rio de Janeiro: Campus, 2000.

GOLM, Michael, KLEINODER, Jurgen. **Jumping to the meta level**. Erlangen, Jan. 2000. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/>. Acesso em: 28 ago. 2002.

GOLM, Michael, KLEINODER, Jurgen. **metaXa and the future of reflection**. Erlangen, jan. 1998. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/>. Acesso em: 28 ago. 2002.

KASBEKAR, Mangesh; NARAYANAN, Chandramouli; DAS, Chita R.. **Using reflection for checkpointing concurrent object oriented programs**. Pennsylvania, 1998. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/>. Acesso em: 28 ago. 2002.

KILLIJIAN, Marc-Olivier; FABRE, Jean-Charles; RUIZ-GARCIA, Juan-Carlos. **Development of a metaobject protocol for fault-tolerance using compile-time reflection**. Cedex, 1998. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/>. Acesso em: 28 ago. 2002.

LEE, Arthur H.; SHIN, Ho-Yun. **Building a persistent object store using the java reflection API**. Seoul. 1998. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/>. Acesso em: 28 ago. 2002.

PERRY, Greg M. **Usando Visual J++**. Rio de Janeiro: Campus, 1997.

SENRA, Rodrigo Dias Arruda. **Programação reflexiva sobre o protocolo de meta-objetos Guaraná,** São Paulo, nov. 2001. Disponível em: <http://www.ic.unicamp.br/~921234/dissert/node5.html>. Acesso em: 12 maio 2002.

SINTES, Anthony. **Aprenda programação orientada a objetos em 21 dias**. São Paulo: Makron Books, 2002.

SOUZA, Cristina Verçosa Peres Barrios de. **Uso de reflexão computacional em aplicações da plataforma J2EE**. Curitiba, 2001. Disponível em: <http://www.ppgia.pucpr.br/~cristina/Dissertacao.html>. Acesso em: 16 jul. 2002.

TATSUBORI, Michaki. **Welcome to OpenJava**. Ago. 2002. Disponível em: <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/> Acesso em: 07 ago. 2002.

TATSUBORI, Michaki. **OpenJava: A Class-based Macro System for Java**. Tsukuba, 1998. Disponível em: <http://www4.informatik.uni-erlangen.de/Publications/> Acesso em: 28 ago. 2002.

WINBLAD, Ann L.; EDWARDS, Samuel D.; KING, David R. **Software orientado ao objeto**. São Paulo: Makron Books, 1993.