UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

UTILIZAÇÃO DA REFLEXÃO COMPUTACIONAL PARA IMPLEMENTAÇÃO DE UM MONITOR DE SOFTWARE ORIENTADO A OBJETOS EM JAVA

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

FABIO CORDOVA DE SOUSA

BLUMENAU, DEZEMBRO/2002

2002/2-23

UTILIZAÇÃO DA REFLEXÃO COMPUTACIONAL PARA IMPLEMENTAÇÃO DE UM MONITOR DE SOFTWARE ORIENTADO A OBJETOS EM JAVA

FABIO CORDOVA DE SOUSA

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. Everaldo Artur Grahl

_____

Prof. Jomi Fred Hübner

_____

Prof. Marcel Hugo

AGRADECIMENTOS

Gostaria de agradecer, primeiramente aos meus pais pelo apoio e compreensão, e também ao professor Marcel pela ajuda dispensada a realização deste trabalho.

Não esquecendo dos colegas e amigos que fiz durante o curso. E aos meus professores, que de uma forma, ou de outra, foram os responsáveis pelo conhecimento que hoje eu possuo.

Enfim, eu gostaria de agradecer a todas aquelas pessoas, que durante este período, fizeram presentes ao meu lado.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS E TABELAS

RESUMO

Este trabalho descreve conceitos sobre a implementação de reflexão computacional através da orientação a objetos. A reflexão computacional é utilizada para adaptar e reutilizar sistemas, além de possuir grande aplicabilidade em sistemas com grande complexidade. No contexto deste trabalho, reflexão computacional é utilizada para monitorar o comportamento de um software orientado a objetos, empregando os meta protocolos existentes para a linguagem Java. A implementação é realizada utilizando a ferramenta Javassist e a API Reflect do Java.

ABSTRACT

This work describes models and concepts on implementation of computational reflection through the objects-oriented. The computational reflection, in the object model, is used to customize and to reuse systems, besides possessing great applicability in systems with great complexity. In the context of this work, computational reflection is used to add new behaviors of a software object oriented, in order to evaluate the goal existing protocols for the Java language, with the described theory.

1 INTRODUÇÃO

Com o aprimoramento de técnicas, conceitos e metodologias, surgem também novos paradigmas no desenvolvimento de software. Um desses paradigmas é a Orientação a Objetos que, cada dia mais, vem sendo utilizada e difundida. Segundo Winblad (1993), num futuro não muito distante, a Orientação a Objetos (OO) oferecerá grandes benefícios para três categorias de programadores: usuários, programadores comerciais e desenvolvedores em nível de sistemas. O mais dramático desses benefícios será para os criadores de sistemas, que precisarão adotar esta técnica de desenvolvimento, e suas ferramentas, a fim de tratar o aumento da complexidade e potencialidade dos softwares.

Conforme Winblad (1993), os desenvolvedores precisam fazer mudanças substanciais na maneira de analisar problemas e traduzi-los em programas. O paradigma da Orientação a Objetos é bastante diferente. Os mais experientes neste paradigma declaram: “Os programas orientados ao objeto são mais fáceis de escrever”. Dizem ainda que os conceitos são mais abstratos do que os tradicionais e podem ser difíceis de entender, a princípio. Com freqüência, quando alguns novatos fazem alterações em comandos, dizem: “Ele funciona, mas eu não sei porquê”. Existe uma curva de aprendizado inicialmente lenta, porém com um “ahá” ocorrendo no momento certo.

Considerando este fato, há a necessidade de ferramentas para o apoio do aprendizado dessa técnica. Esse suporte seria dado, por exemplo, por softwares que mostrassem como e quando objetos fossem instanciados, métodos utilizados e atributos acessados. Neste ponto, inclui-se a Reflexão Computacional, que serve como uma ferramenta para demonstrar, e até mesmo alterar, o conteúdo de softwares, usados como exemplos.

Segundo Lisboa (1997), Reflexão Computacional é toda a atividade de um sistema computacional realizada sobre si mesmo, e de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações em tempo real, podendo ser esta resolução tanto estrutural como comportamental.

Já Steel (1994) define como sendo a capacidade de um sistema computacional de interromper o processo de execução (por exemplo, quando ocorre um erro), realizar

computações ou fazer deduções no meta-nível e retornar ao nível de execução, traduzindo o impacto das decisões, para então, retomar o processo de execução.

Para tal, a reflexão computacional utiliza-se de objetos abertos (permitindo acessar, invocar e alterar um objeto sem estar delimitada à interface do mesmo) e de uma arquitetura de meta-níveis (composta por vários níveis).

A Reflexão Computacional segundo Barth (2000) pode ser dividida em duas partes distintas: a estrutural, que permite que o programa tenha a sua estrutura – tipo e número de componentes – alterada durante o processo de execução e/ou de compilação, e a comportamental, que atua indiretamente sobre aspectos de comportamento de um objeto, mantendo inalterada a estrutura do mesmo.

A linguagem de programação Java provê uma estrutura que suporta a reflexão computacional, não em todo o seu contexto, mas abrangendo um tipo de reflexão, a reflexão estrutural. Extensões para Java têm sido desenvolvidas para utilizar todos os recursos da reflexão computacional.

A proposta deste trabalho é desenvolver um software de apoio ao aprendizado de OO, utilizando a técnica de Reflexão Computacional para monitorar os objetos, métodos e atributos, através de uma interface gráfica. Um software de apoio como este será de muita utilidade na disciplina de Programação II (OO) do curso de Ciências da Computação da FURB ao permitir que os acadêmicos, após construírem seus exercícios e exemplos em Java, possam visualizar a execução dos objetos que construíram.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta de monitoração de softwares Java, usando a técnica de reflexão computacional comportamental, demonstrando a execução deste software em tempo real.

Os objetivos específicos são:

a) entender e aplicar as interfaces de programação para reflexão computacional em Java;

b) criar uma ferramenta de apoio ao ensino de Orientação a Objetos.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está organizado da seguinte maneira.

O capítulo 2 descreve a reflexão computacional(conceito, aplicação), e o protocolo de meta-objetos.

O capítulo 3 descreve o desenvolvimento do trabalho.

O capítulo 4 faz as considerações finais sobre o trabalho.

2 REFLEXÃO COMPUTACIONAL

Segundo Sizhong (2001), Reflexão Computacional refere-se à capacidade de um sistema raciocinar e agir sobre si mesmo. Mais especificamente, um sistema reflexivo é o único que provê a representação do seu comportamento, sensível a inspeção e adaptação, fazendo com que mudanças sejam visíveis no seu estado ou comportamento, e vice-versa. Segundo Dourish (1992), no domínio psicológico, “reflexão” refere-se ao processo de instropecção e análise. Por analogia, reflexão computacional é a habilidade de um sistema computacional de ponderar sobre o seu comportamento e eventualmente alterá-lo. Os elementos chaves são, primeiro, a representação com que o sistema assegura o seu comportamento, e segundo, a conexão causal entre a representação e o seu comportamento atual. A conexão causal pode ser de duas maneiras: não só alterando o comportamento reflexivo e mudanças equivalentes na representação, mas mudanças na sua representação serão também a causa de mudanças no seu comportamento.

Segundo Stankovic (1997) reflexão computacional, como conceito, não é novo em quase todos os sistemas que possuem alguma informação. Entretanto, identificando reflexão como chave da arquitetura principal e explorando isto em sistema em tempo real é muito novo. Em termos simples, reflexão pode ser definida como o raciocínio do processo computacional e agindo sobre o sistema. Normalmente, a computação resolve problemas com uma aplicação, por exemplo, filtrando os dados retornados. Adicionando a este sistema computacional procedimentos para configurar, policiar dinamicamente seus parâmetros, obtem-se uma estrutura com meta-níveis e possibilidades de reflexão. Conseqüentemente, usando uma arquitetura reflexiva, pode-se considerar que o sistema tem uma parte computacional e uma reflexiva. A parte reflexiva é a auto-representação, expondo o estado do sistema e a semântica da aplicação. Por exemplo, para filtrar processos escolhendo o filtro mais apropriado, configurando parâmetros do processo, consistindo a entrada de dados, entre outros.

Reflexão tem aparecido em sistemas de inteligência artificial, no contexto de sistemas baseados em regras e sistemas baseados em lógicas. Mais recentemente tem sido usada em linguagens orientadas a objetos e em banco de dados orientados a objetos incrementando suas

flexibilidades. Também pode ser usada em sistemas distribuídos como em sistemas em tempo real por suportar transparência e flexibilidade.

Conforme Maes (1987), Reflexão Computacional é o processo computacional envolvendo a consciência. Como nos humanos, reflexão depende da capacidade de raciocínio independente, e particularmente, raciocínio sobre os seus próprios processos. Um programa reflexivo tem a habilidade de metaprogramar: ele pode, sozinho, escrever programas. A capacidade de reflexão é um dos mais importantes componentes da inteligência artificial (I.A.) e pode ser relatada em outros aspectos da I.A. como a lógica difusa e redes neurais.

Quando um programa reflexivo executa, parece-se de certa maneira com a consciência humana. Ele leva em conta variáveis, assim como suas próprias condições, e informações contextuais. Por exemplo, pense nas operações envolvidas entre o caminho do seu carro até a sua casa. Se você ver um obstáculo no seu caminho, você pegará esta informação e adaptará para desviar, ou pegá-lo. Do mesmo modo, um programa reflexivo tem a habilidade de "pensar" sobre o que está acontecendo e alterar-se dependendo das circunstâncias.

Segundo Senra (2001), o termo reflexão nos 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.

Arquiteturas reflexivas compõem-se de dois níveis, meta-nível e o para-nível (nível base). Encontram-se no primeiro nível estruturas de dados e as ações que são executadas sobre o sistema objeto que está presente no nível base.

2.1 TIPOS DE REFLEXÃO

Segundo Souza (2001), 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 meta-nível 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 meta-nível.

Seguindo o raciocínio exposto anteriomente, a reflexão computacional pode ser particionada genericamente em dois tipos de funções: a introspecção e a intercessão, descritas nas subseções a seguir.

O programa que realiza a reflexão computacional, introspecção ou intercessão, é chamado de programa de meta-nível, enquanto o programa executado na computação reflexiva é chamado de programa de nível base (para-nível).

2.1.1 INTROSPECÇÃO

Segundo Souza (2001), a introspecção, também referenciada como reflexão estrutural, ou redirecionamento, refere-se ao processo de obter informação estrutural do programa e usá-la no próprio programa. Isso é possível através da representação, em meta-nível, dessa informação do programa, feita por um processo de reificação.

Segundo Barth (2000), introspecção ou reflexão estrutural de uma classe é toda atividade realizada em uma meta-classe, com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe.

2.1.2 INTERCESSÃO

Segundo Souza (2001) a intercessão, também referenciada como reflexão comportamental, ou ainda como interceptação, refere-se ao processo de alterar o comportamento do programa no próprio programa. Isso pode ser realizado, por exemplo, através do ato de interceder, intermediar, ou ainda interceptar operações de nível base – como operações de invocação de método, ou operações de estabelecimento / obtenção de valores de atributos -, que podem assim ser reificadas e conseqüentemente, manipuladas pelo meta-nível.

Conforme Barth (2000), por intercessão ou reflexão comportamental entende-se toda a computação que indiretamente atua sobre aspectos de comportamento de um objeto, mantendo inalterada a sua estrutura. A reflexão comportamental de um objeto é toda atividade realizada por um meta-objeto com o objetivo de obter informações e realizar transformações sobre o comportamento do objeto.

2.2 PROTOCOLO DE META-OBJETOS

Segundo Sobel (1996), num sistema orientado a objetos, o programa especifica o comportamento dos objetos de uma classe especificando um conjunto de métodos relacionados com essas classes (ou suas superclasses). O programa invoca métodos do objeto sem "saber" precisamente de onde as classes vem, e o sistema escolhe o mais apropriado para executar. O protocolo de meta-objetos estende ao sistema orientado a objetos dando a cada objeto o seu correspondente meta-objeto. E cada meta-objeto instanciado de sua meta-classe. O método especializado relacionado a metaclasse especifica o comportamento do meta-nível do par objeto-metaobjeto, onde o comportamento do meta-nível inclui, por exemplo o trabalho de herdar, instanciar, e o método de invocação. O protocolo de meta-objetos provê o poder, a flexibilidade de modificar o comportamento da linguagem.

Conforme Senra (2001), a interação entre o para-nível (nível base) e o meta-nível é regida pelo protocolo de meta-objetos ou MOP, que consiste em uma interface de acesso à meta-informação e manipulação do para-nível a partir do meta-nível. Existem quatro aspectos principais a serem considerados em um dado MOP: vinculação, reificação, execução e modificação. Por uma preferência de estilo, opta-se por rebatizar o termo modificação por intervenção mantendo a semântica original. Esses aspectos são definidos formalmente na

Tabela 1. Figurativamente, a vinculação pode ser entendida como a cola entre para-nível e meta-nível, a reificação como a leitura do para-nível pelo meta-nível, a intervenção como a escrita no para-nível pelo meta-nível e finalmente a execução como o uso do para-nível pelo meta-nível.

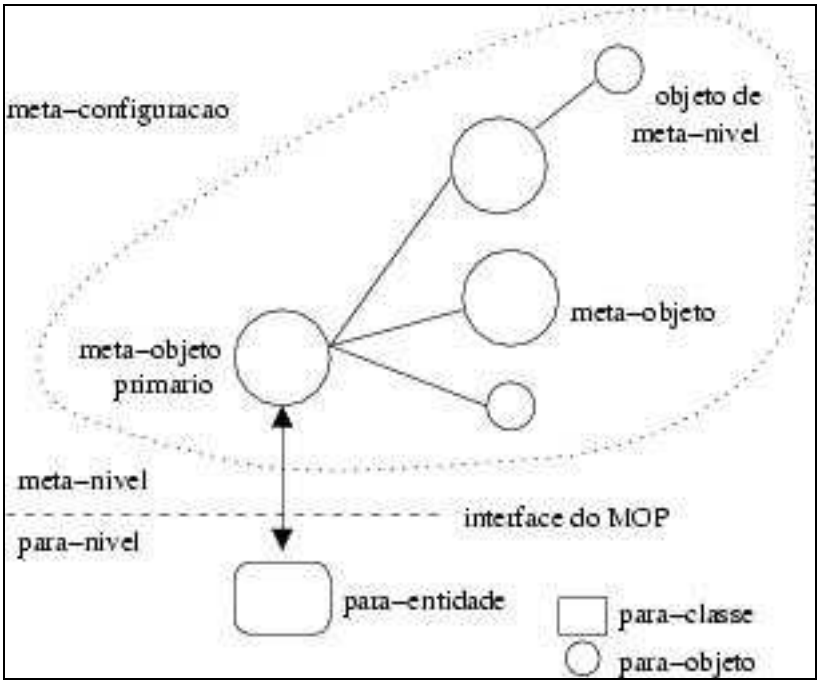
Tabela 1**:** Aspectos principais de um MOP

Vinculação	Reflete a natureza da ligação entre meta-entidades e para-entidades, como por exemplo qual é a cardinalidade do relacionamento meta-entidade/para-entidade, ou seja, quantas para-entidades podem estar subordinadas a uma única meta-entidade e, quantas meta-entidades podem gerenciar uma dada para-entidade. A vinculação pode ser estabelecida em tempo de compilação (estática) ou em tempo de execução (dinâmica), estabelecendo a jurisdição de uma meta-entidade no âmbito das para-entidades.
Reificação	É o processo de disponibilização de meta-informação oriunda do para-nível em um formato acessível ao meta-nível. Descreve tanto o estado das para-entidades quanto a dinâmica de troca de mensagens entre as mesmas.
Execução	Descreve a capacidade das meta-entidades de interagirem com o para-nível como se fossem para-entidades requisitando serviços.
Intervenção	Descreve a capacidade do meta-nível de controlar a estrutura, estado e comportamento do para-nível.

Segundo Senra (2001), classifica-se como reflexão estrutural a atuação de um dado aspecto do MOP sobre a estrutura das para-entidades. Complementarmente, classifica-se como reflexão comportamental a atuação de um aspecto do MOP sobre o comportamento das para-entidades. No domínio de orientação a objetos essa bi-dimensionalidade surge naturalmente pela própria definição de objeto como entidade que possui um estado e um comportamento associados.

A estrutura do MOP pode ser vista na figura 1.

Figura 1: Exemplo de uma arquitetura.



2.3 LINGUAGENS REFLEXIVAS

A seguir serão apresentadas algumas linguagens que possuem mecanismos de reflexão, com o intuito de contextualizar o leitor nas ferramentas existentes.

2.3.1 3-LISP

Segundo Senra (2001) 3-Lisp inovou pela introdução de um modelo de reflexão procedural baseado em uma arquitetura de computação seqüencial. Além de ter sido a precursora das linguagens imperativas não-orientadas a objetos com suporte a reflexão computacional, 3-Lisp já permitia reificação e intervenção de forma plena sobre todas suas estruturas, respeitando a propriedade da conexão causal e suportando reflexão em tempo de execução. A evolução do modelo reflexivo de 3-Lisp provavelmente foi comprometida por estar intimamente ligado a filosofia de programação em Lisp, por não contemplar o paradigma da orientação-a-objetos e principalmente pela ausência de uma separação clara entre para-nível e meta-nível.

3-Lisp, que foi construído sobre 2-Lisp, que por sua vez foi um dialeto de Lisp mais estrito muito similar a Scheme, segundo Senra (2001) foi a primeira linguagem construída

segundo o conceito de reflexão computacional, onde cada acesso reflexivo tinha sua própria estrutura interpretadora.

3-Lisp é um sistema reflexivo, construído por B.C. Smith, que usa a noção de torres reflexivas. Basicamente, este sistema consiste em uma torre infinita de interpretadores meta-circulares, mas estende-se a um mecanismo que provê ao usuário do programa acesso total às descrições da estrutura e operação do programa. Ganha-se acesso usando o que são chamados de procedimentos reflexivos: procedimentos que quando invocados, atuam não no nível que a invocação ocorreu, mas num nível superior, o nível onde o processador reflexivo está rodando sobre o programa.

2.3.2 3-KRS

Segundo Senra (2001), proposta por Pattie Maes quase cinco anos depois do advento de 3-Lisp, 3-KRS foi considerada a linguagem OO pioneira no suporte a reflexão. Assim como 3-Lisp, 3-KRS suportava reificação e intervenção de forma plena, sendo consistente com o princípio de conexão causal, o que se tornou possível pela a adoção de um modelo de representação uniforme onde todas entidades da linguagem eram representadas como objetos. Cada objeto era emparelhado com um respectivo meta-objeto construído tardiamente sob demanda.

Apesar de delimitar uma fronteira entre para-computação e meta-computação, o MOP de 3-KRS apresentava um acoplamento muito forte entre para-objeto e meta-objeto, dificultando assim a construção de bibliotecas de componentes para o meta-nível.

2.3.3 ABCL/R

ABCL/R é uma linguagem de programação reflexiva concorrente e interpretada. Isto inclui um poderoso e explícito protocolo de meta-objetos o estado o código

Segundo Senra (2001), quase concomitantemente ao surgimento de 3-KRS, ABCL/R foi concebida por Watanabe e Yonezawa com uma abordagem reflexiva radicalmente diferente de 3-KRS. Em ABCL/R cada para-objeto é implementado por uma torre independente de meta-objetos que o descrevem em nível estrutural e comportamental. O MOP de ABCL/R é baseado em troca de mensagens assíncronas que permite concorrência de

execução entre para-nível e meta-nível. Posteriormente, essa linguagem obteve continuidade através de ABCL/R2 e ABCL/R3. Esta última adotou um modelo de reflexão em tempo de compilação para evitar que haja sobrecarga relativa ao mecanismo de interceptação em tempo de execução. ABCL/R3 inovou com um compilador capaz de avaliar a definição de um dado meta-objeto para produzir uma meta-entidade geradora de código nativo que atuasse em tempo de execução. Essa abordagem foi denominada *partial evaluation*.

2.3.4 CLOS

Conforme Senra (2001), *Common Lisp Object System*, abreviado para CLOS, foi uma linguagem proposta em 1991 por Gregor Kiczales para se tornar o dialeto Common Lisp OO padrão. Historicamente, CLOS é a sucessora de FLAVORS (Symbolics) e LOOPS (Xerox). Atualmente há várias implementações de CLOS, entre as quais é relevante citar MCL, KCL, CL (Allegro e CMU), Ibuki, Lucid, Medley, Genera (Symbolics), CLOE, LispWorks (Harlequin) e PCL. Dentre todos esses sucessores de CLOS, PCL é a implementação que possui o MOP mais detalhado.

Todos os elementos básicos da linguagem: classes, *slots* (variáveis de instância), funções genéricas, métodos, especializadores de métodos e combinações de métodos são objetos de primeira classe. Dessa forma CLOS suporta plena reificação, já a intervenção está sujeita a um conjunto de restrições que evita a quebra de protocolo OO no que tange a hierarquias de generalização/especialização e a interface pública de classes/objetos. Entretanto, o MOP de CLOS impõe que todos para-objetos derivados de uma dada classe só possam estar vinculados a meta-objetos derivados diretamente de uma mesma classe. Essa restrição representa uma desobediência ao princípio da separação entre domínios, ou seja, a existência de plena desvinculação entre os requisitos funcionais de para-nível e os requisitos gerenciais de meta-nível.

2.3.5 OPEN-C++

Conforme Senra (2001), Open-C++ foi elaborado por Shigeru Chiba unificando a estrutura do MOP de CLOS com o princípio de reflexão em tempo de compilação oriundo de Intrigue e Anibus.

O objetivo dessa linguagem era disponibilizar um mecanismo eficiente e transparente para implementação de extensões de C++ através do controle do processo de compilação, que por sua vez abrange os seguintes aspectos da linguagem subjacente: definição de classes, acesso a membros, invocação de funções virtuais e criação de objetos.

Por um lado essa abordagem provê ganho de eficiência por não incorrer em nenhuma sobrecarga em tempo de execução tal como suporte a interceptação. Por outro lado, há uma penalidade em custo temporal e espacial no processo de compilação.

Segundo Chiba (1998), Open C++ é uma linguagem extendida baseada no C++. A característica estendida do Open C++ é especificado por um programa de meta-níveis dado em tempo de compilação. Por distinção, programas regulares Open C++ são chamados de programas de nível base. Se não é dado nenhum meta-nível em tempo de compilação, um programa Open C++ é idêntico a um em C++.

O MOP de Open-C++ gerencia a tradução de meta-código (extensões em Open-C++) em código C++ puro através de duas fases. Na primeira fase o MOP gera meta-objetos transientes, que só existirão durante a compilação. Na segunda fase os meta-objetos são convidados a gerar fragmentos de código apropriados, que por sua vez podem ser tanto código C++ ou código Open-C++. A iteração entre as duas primeiras fases prossegue até que todos os fragmentos de código só possuam código C++. Por estar fundamentada na estrutura do MOP de CLOS (baseado em classe), Open-C++ também viola o princípio da separação entre domínios ao obrigar que para-objetos com mesma generalização (classe) estejam associados a meta-objetos também com mesma generalização. Outra deficiência de Open-C++ é restringir a vinculação de um meta-objeto a um único para-objeto. Deficiência esta que é amenizada pela criação de um sistema de propagação automática e customizável de meta-objetos através de hierarquias de generalização/especialização presentes no para-nível. Ou seja, subclasses (para-nível) podem herdar meta-classes associadas a sua classe base.

2.4 REFLEXÃO COMPUTACIONAL E PROTOCOLO DE META-OBJETOS EM JAVA

Conforme Sullivan (2001) reflexão computacional permite ao programa acessar a sua estrutura interna, o comportamento e também manipular programaticamente a sua estrutura,

desse modo modificando seu comportamento. A linguagem de programação Java permite algumas dessas capacidades reflexivas. Por exemplo, um programa Java pode perguntar sobre a classe de um objeto, os métodos dessa classe, e então invocar um desses métodos.

O protocolo de meta-objetos define a execução de uma aplicação em termos de comportamento implementados por uma classe. Por exemplo, a execução de um método dinâmico pode envolver o nome do método nas funções virtuais e pegar os valores dos argumentos usados pela chamada deste método, poderá determinar a maioria dos argumentos, e alterar a implementação do método. O programador pode sobrescrever o comportamento do método numa ordem que afete o que acontecerá quando a função for chamada.

Segundo Sullivan (2001), as capacidades reflexivas embutidas do Java ficam aquém do protocolo de meta-objetos por aspectos:

a) A reflexão em Java é "*read-only*". Por exemplo, um programa pode perguntar pelos métodos de uma classe, mas o programa não pode dinamicamente mudar os métodos dessa classe. Reflexão completa permite modificar qualquer uma das meta-informações que sofreram reflexão.

b) Java não permite criar ou alterar subclasses de meta-classes. Com o protocolo de meta-objetos completo, criar ou alterar as subclasses de meta-classes é uma maneira de mudar o comportamento da linguagem.

Java provê introspecção e não intercessão. Segundo Tatsoburi (1999), a *API Java Reflection* é refinada para introspecção em tempo de execução, especialmente na questão de segurança em tempo de execução, mas não possui nenhum mecanismo de intercessão.

2.5 API JAVA REFLECT

A *API Java Reflection*, que é constituída dos pacotes *java.lang.reflect* e *java.lang.class*, não possui uma capacidade reflexiva completa. Ela não permite alterar o comportamento do programa, suporta somente a introspecção do mesmo, por exemplo, inspecionar a definição de uma classe. Isto é aceitável levando-se em consideração que quanto maior for a capacidade reflexiva menor será a performance em tempo de execução. Existem muitas extensões desta API, mas sempre ocorre perda de performance.

Segundo Grand (1997), o núcleo da *API Java Reflection* provê uma pequena e segura API que suporta introspecção sobre as classes e objetos na Máquina Virtual Java corrente. Se permitido pela política de segurança, a API pode ser usada para:

a) construir instâncias de classse novas;

b) acessar e modificar atributos de objetos e classes;

c) invocar métodos e objetos de classes;

d) acessar e invocar elementos da matriz;

e) descobrir variáveis, métodos e construtores de qualquer classe.

Esta API define novas classes e métodos, a seguir, representados na figura 2:

a) três novas classes – *Field*, *Method* e *Constructor* – que refletem classes e interface de membros e construtores. Estas classes provêem: informações reflexivas abaixo do membro ou construtor; um tipo seguro com a finalidade de usar membros e construtores para operar objetos no Java;

b) Novo método da classe *Class* para a construção de novas instâncias das classes *Field*, *Method* e *Constructor*;

c) Uma classe – *Array* – que dispõem de métodos para construir e acessar dinamicamente matrizes do Java;

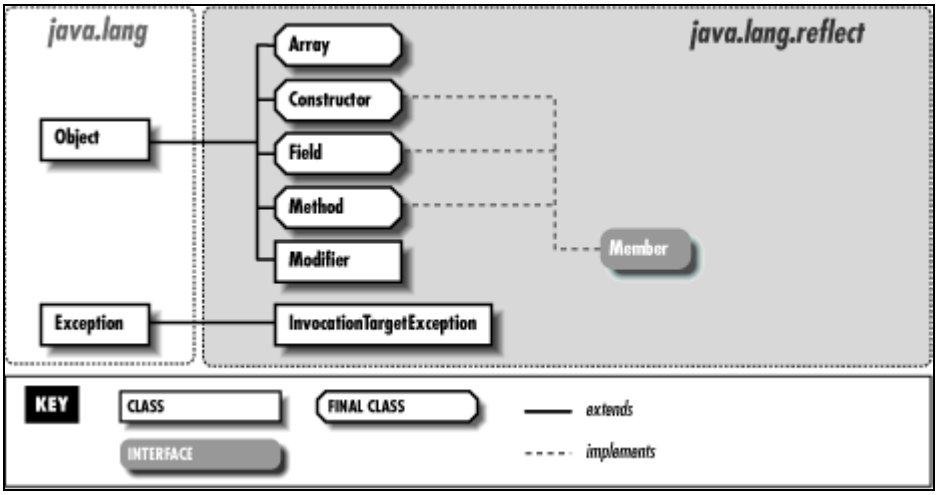
d) Uma nova e útil classe – *Modifier* – que ajuda a decodificar os modificadores de informação sobre as classes e seus membros;

e) Uma interface – *Member* – definição de parte dos métodos da classe;

f) Uma exceção – *InvocationTargetException* – lançada por um método ou construtor.

Informações sobre os métodos dos pacotes java.lang.class e java.lang.reflect são encontradas em Sun (1995).

Figura 2: Estrutura do pacote java.lang.reflect.



2.5.1 JAVA.LANG.CLASS

Esta classe contém elementos de uma classe *Class* representam tipos Java de uma forma que possam ser manipulados por um programa em execução.

Não possuem construtores públicos, pois cada classe nova que é carregada pelo *Java Virtual Machine* cria um objeto *Class*. Tais objetos não podem ser criados de outra forma.

2.5.2 JAVA.LANG.REFLECT.ARRAY

Esta classe contém métodos que permitem criar, atualizar ou questionar sobre os valores de uma matriz. Esta classe é usada para manipular os valores e não o tipo da matriz, que são representados pela classe *Class*. Todos os seus métodos são estáticos e se aplicam aos valores.

2.5.3 JAVA.LANG.REFLECT.CONSTRUCTOR

Esta classe provê acesso e informações sobre o construtor de uma classe declarada. Pode ser usado para criar e inicializar novas instâncias a partir de um construtor de uma classe que sofreu reflexão.

2.5.4 JAVA.LANG.REFLECT.FIELD

Esta classe representa os campos de uma classe. Provê acesso e informações sobre os campos de uma classe ou uma interface.

2.5.5 JAVA.LANG.REFLECT.MEMBER

Esta classe identifica informações sobre uma classe ou interface de um membro ou construtor.

2.5.6 JAVA.LANG.REFLECT.METHODS

Esta classe representa os métodos de uma classe. Provê acesso e informações sobre os métodos de uma classe ou uma interface. Pode ser usado para buscar informações de um método refletido, bem como para invocar um método de uma classe declarada.

2.5.7 JAVA.LANG.REFLECT.MODIFIER

Esta classe provê métodos estáticos e constantes que são usados para interpretar os modificadores de acesso de membros e classes.

Informações sobre os métodos da classe *java.lang.reflect.Modifier* podem ser encontradas em Sun (1995).

2.5.8 JAVA.LANG.REFLECT.INVOCATIONTARGETEXCEPTION

Um objeto dessa classe é criado pelo *Method.invoke()* ou *Constructor.newInstance()* quando uma exceção é lançada pela execução destes.

A classe *InvocationTargetException* possui somente um método, *getTargetException().*

2.6 JAVASSIST

Para estender a capacidade reflexiva do Java existem as ferramentas reflexivas, como o OpenJava, Dalang, MetaXa, Guaraná (que é nacional) e o Javassist.

Segundo Chiba (2000) Javassist é um ferramenta de programação de assistência ao programador Java. Esta classe possibilita ao programador automatizar alguns tipos de definições de classe e permite a reflexão comportamental em tempo de execução.

Os seguintes pacotes fazem parte do Javasist (*Java programming assistant)*:

a) javassist: é uma biblioteca de classes para a edição do *bytecode* em Java; isto permite ao programa Java definir novas classes em tempo de execução e modificar uma classe quando o *Java Virtual Machine* (JVM) carrrega.

b) javassist.bytecode: esta classe provê a edição de um arquivo *class* em baixo nível. Isto permite aos usuários ler e modificar por exemplo uma instrução *bytecode*. Os usuários desta biblioteca precisam conhecer as especificações do arquivo *class* e do *bytecode* Java;

c) javassist.expr: esta biblioteca contêm classes para modificar o corpo dos métodos;

d) javassist.prepoc: é um pré-processador em tempo de compilação;

e) javassist.reflect: esta classe permite ao meta-objeto interceptar métodos chamados e campos acessados por um objeto Java. Uma instância da classe que sofreu reflexão é associada em tempo de execução a um meta-objeto e a uma meta-classe, que controlam o comportamento do objeto instanciado;

f) javassit.rmi: esta biblioteca permite *applets* acessar remotamente objetos executedos no *web server* com a sintaxe regular Java;

g) javassist.runtime: nesta biblioteca inclui-se classes de suporte que são requisitadas por classes modificadas com o Javassist. A maioria das classes modificadas não requerem classes de suporte;

h) javassist.web: provê um simples *web server* para as outras bibliotecas.

3 DESENVOLVIMENTO DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta de monitoração de softwares Java, usando a técnica de reflexão computacional comportamental, demonstrando a execução deste software em tempo real.

Faz-se necessário que antes de usar o sistema exista um diretório com as classes da aplicação que se deseja monitorar. Esta aplicação deve ser de baixa complexidade, ou seja, que possua somente um método *main* e que não esteja em pacotes, pois trata-se de um monitor de intuito acadêmico.

Com o início da execução do monitor, este faz a leitura dos arquivos com a terminação *.class*, que são arquivos compilados do Java. Para cada arquivo que é lido pelo monitor, verifica-se a existência entre seus métodos do método *main()*, pois é a partir dele que o monitor iniciará a execução da aplicação, e em seguida é feita a reflexão do mesmo.

Com as classes necessárias (arquivos *.class*) sendo reflexivas, o monitor consegue interceptar a comunicação com/entre os objetos, sendo eles alterados, lidos ou no momento de sua instanciação. Após o monitor detectar está comunicação é mostrado ao usuário, através de uma interface gráfica, as informações do objeto que acabou de ser criado, como por exemplo, a classe, os métodos, campos e seus valores.

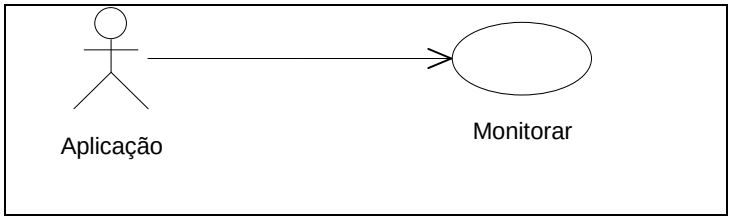
3.1 ESPECIFICAÇÃO

Os diagramas foram modelados baseados na *Unified Modeling Language* (UML), e utilizou-se para tal a ferramenta *Rational Rose.*

3.1.1 DIAGRAMA USE CASE

O sistema monitora a aplicação fazendo a reflexão das classes, representado pela figura 3.

Figura 3: Diagrama Use Case.



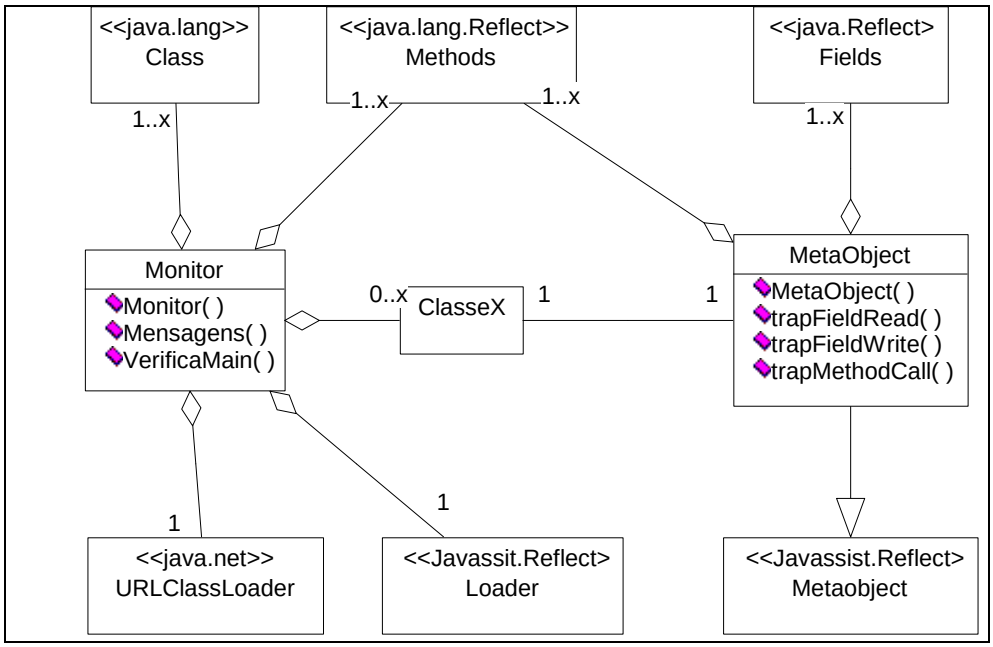
3.2 DIAGRAMA DE CLASSES

A classe Monitor possui apenas três métodos: Monitor() que é o construtor e o método que faz a reflexão das classes, Mensagens() que é o responsável por passar as mensagens do meta-objeto para a interface gráfica, e o método VerificaMain() que verifica entre os métodos de uma classe o método *main()*, que será responsável pela execução da aplicação que sofrerá a reflexão.

Para a classe dos meta-objetos - a classe MetaObject - foram implementados três métodos de interceptação, e um construtor. Foi incluso entre os métodos da classe MetaObject, a chamada para o método Mensagens() da classe Monitor.

A classe ClasseX representa as classes que sofrerão reflexão.

Figura 4: Diagrama de Classes da Apliacação.



3.3 DIAGRAMA DE SEQÜÊNCIA

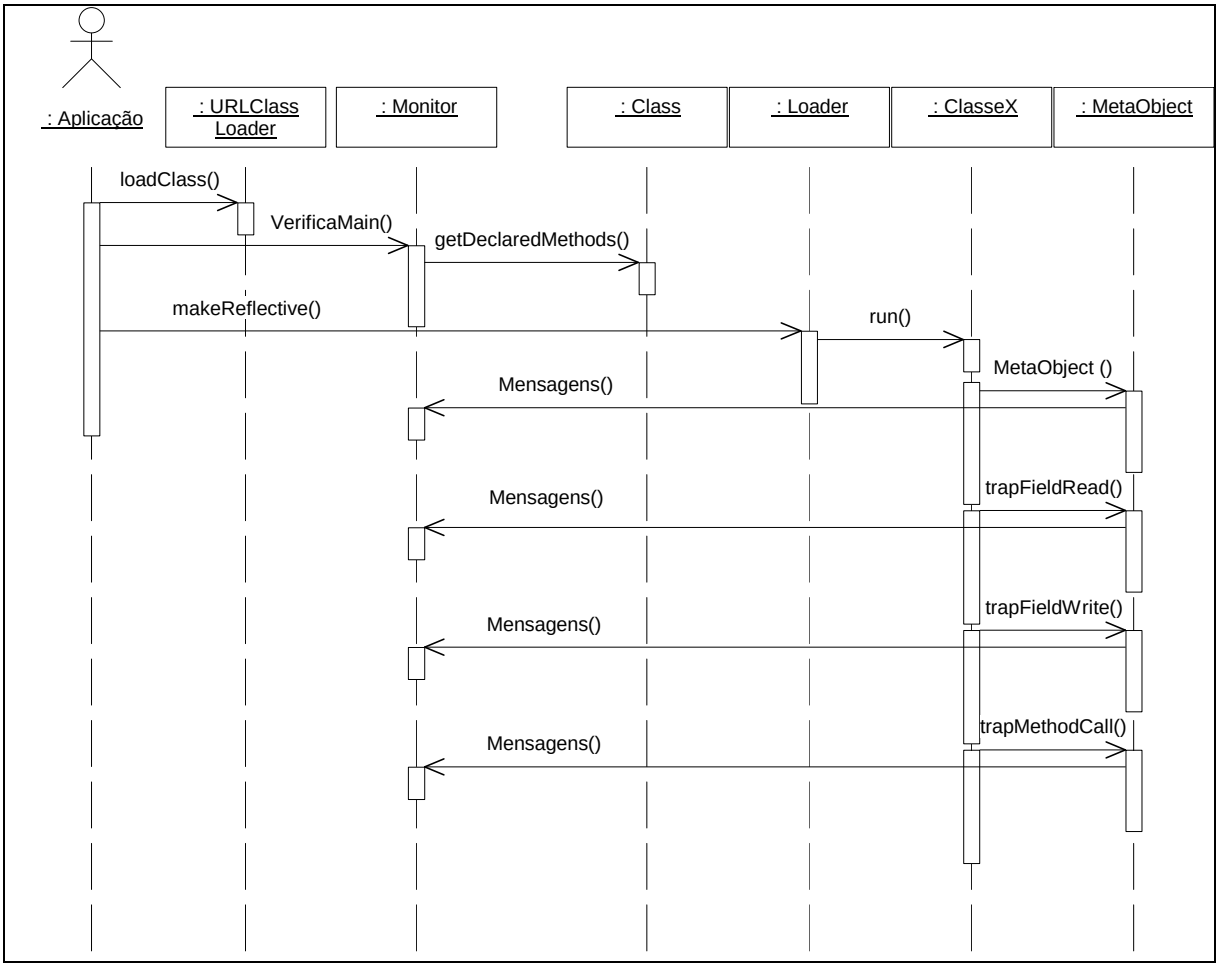
O sistema lê os arquivos *.class* de um diretório fixo chamado de “C:\Prototipo”. É necessário que todos os arquivos *.class* que forem utilizados na aplicação estejam no diretório. É feita a reflexão desses arquivos utilizando o método *makeReflective()* que pertence a classe *Javassist.Reflect*, representado na figura 4.

Antes de torná-los reflexivos fez-se necessário verificar entre os métodos de cada classe a existência do método main(), feito através do método VerificaMain(), pois sendo a partir desta que se dará a inicialização da aplicação. As outras classes necessitam tornar-se reflexivas antes da execução da mesma. Para esta verificação foi preciso carregar a classe com o uso do método *loadClass()*.

Usando o método *makeReflective()* a classe que sofreu a reflexão tem sua estrutura alterada. O Javassist inclui nos método(s) e construtor(es) da classe com os métodos *setWrapper()* e *instrument()* do Javassist, chamadas para os métodos *trapMethodCall()*, *trapFieldWrite()* e *trapFieldRead()*.Métodos esses que foram utilizados na classe *MetaObject* para fazer o monitoramento do objetos no nível-base.

Com a aplicação iniciada, todo e qualquer comunicação com os objetos pode ser interceptada, como mostra a figura 5, pois quando uma classe torna-se reflexiva, pelo Javassist, ela tem o seu formato original alterado. Na estrutura da classe são incluídas chamadas para os métodos de interceptação que estão nos meta-objetos, estes que são responsáveis pelo monitoramento. Monitoramento que se dá incluindo nos métodos do meta-objeto chamadas para o método Mensagens() que está implementado na classe Monitor(). Isto porque a comunicação sempre ocorre a partir do meta-objeto para o objeto. Este método denominado Mensagens() instancia um objeto da classe MostraClasses() que é responsável pela representação gráfica.

Figura 5: Diagrama de Seqüência da Aplicação.



3.4 IMPLEMENTAÇÃO

3.4.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O uso do Javassist para tornar as classes reflexivas mostrou-se simples e funcional. Pois para tornar uma classe reflexiva foi necessário usar um *loader* da classe Javassist.Reflect do próprio Javassist e o método makeReflective(), como mostra o quadro 1. O código completo está no Anexo A.

Quadro 1: Método makeReflective tornando classe reflexiva.

```
cr = new javassist.reflect.Loader();
String classe = nome do arquivo .class;
if (cr.makeReflective(classe,"MetaObject","javassist.reflect.ClassMetaobject"))
    listMode2.addElement("Tornou reflexiva!");
```

A comunicação entre o meta-objeto com o objeto do nível base é implementada no próprio meta-objeto. Como exemplo pode-se colocar uma chamada para um método ou simplesmente imprimir o resultado no próprio meta-objeto, como mostra quadro 2. O mesmo acontece com os outros métodos da classe. O código completo está no Anexo B.

Quadro 2: Método construtor da classe Meta-Objeto.

```
public class MetaObject extends Metaobject {

  public MetaObject(Object self, Object[] args) {
    super(self, args);
        //chamada para o método da classe base
        reflexao3.Monitor.Mensagens("** Objeto Construído: "
                                    + self.getClass().getName());
        //imprime no próprio metaobjeto
        System.out.println("** Objeto Construído: "
                           + self.getClass().getName());

    }

}
```

3.4.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Para demonstrar a operacionalidade do sistema foi escolhido um exemplo simples, conforme quadro 3, com três classes: Partido, Vereador e ExXE05_Lista4 que estão representadas na figura 6. O código completo do exemplo está no Anexo C.

Quadro 3: Enunciado do exemplo

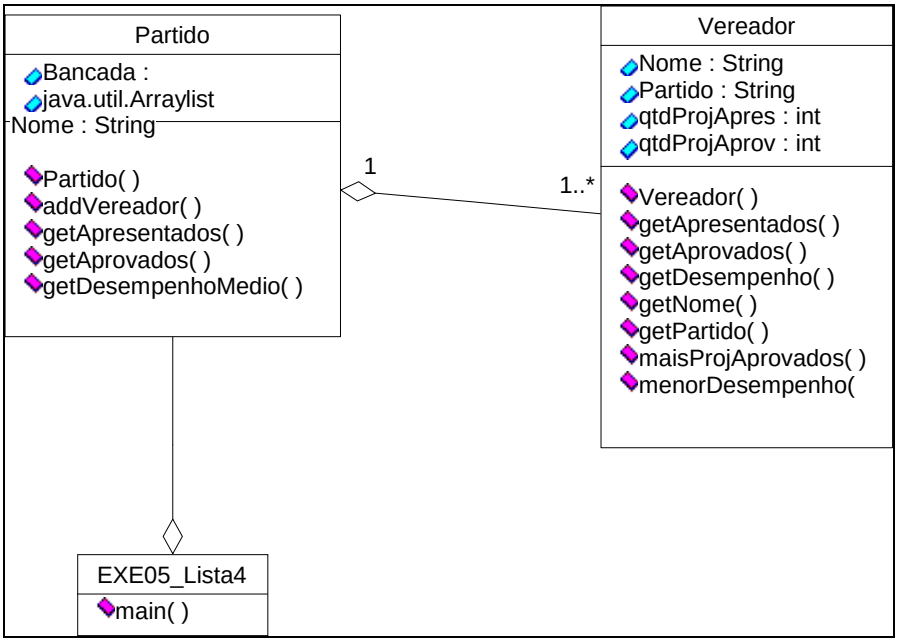
A Câmara Municipal de Vereadores de Blumenau pretende realizar uma estatística sobre o desempenho dos seus parlamentares durante a última legislatura. Para cada um dos 21 vereadores, ela possui o nome, partido (número e nome), quantidade de projetos apresentados, quantidade de projetos aprovados.
O desempenho é calculado da seguinte forma:
(projetos aprovados / projetos apresentados) * índice de trabalho.
Se não apresentou nenhum projeto, o desempenho é 0 (zero). O índice de trabalho é definido pela seguinte tabela:

Projetos apresentados	Índice de trabalho
01 – 05	0,80
06 – 10	1,00
11 – 17	1,08
acima de 17	1,22

Escreva um programa Java orientado a objetos que leia os dados disponíveis pela Câmara e imprima o nome, partido e desempenho do vereador.
Ao final, imprima :

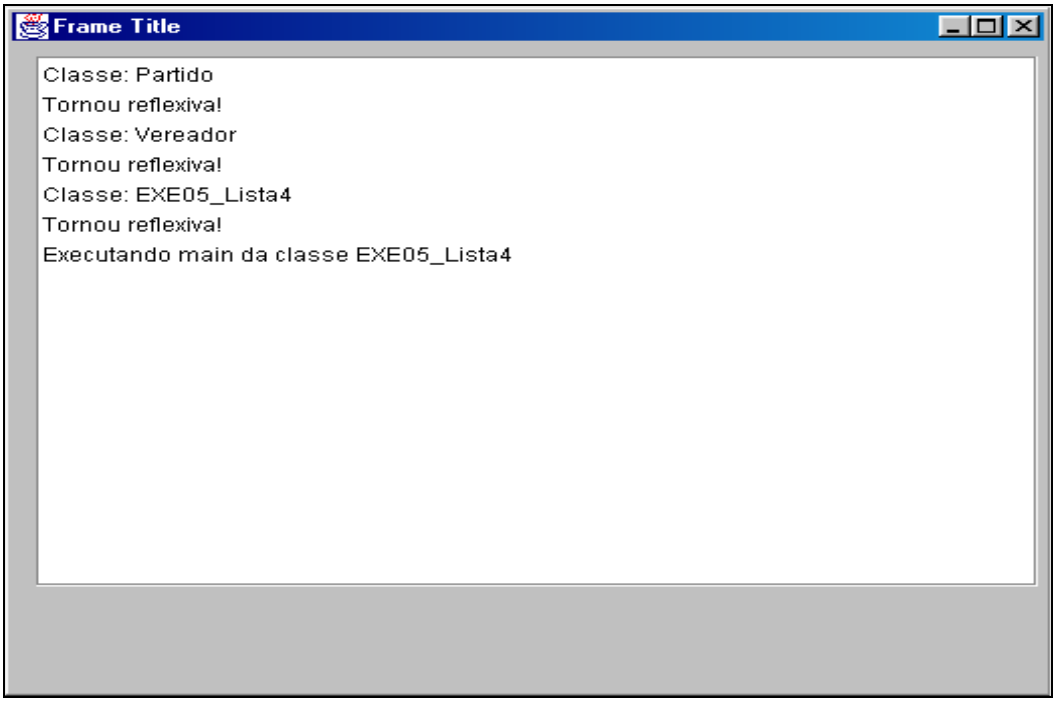
1. o total de projetos apresentados e de aprovados na câmara;
2. o nome, partido e desempenho do vereador com mais projetos aprovados;
3. o nome, partido e desempenho do vereador com menor desempenho;
4. o nome, partido e desempenho dos vereadores cujo desempenho seja maior que o desempenho médio de toda a câmara;
5. a média de desempenho por partido político;
6. o total de projetos apresentados e de aprovados por partido político.

Figura 6: Diagrama de classes das classes exemplo.



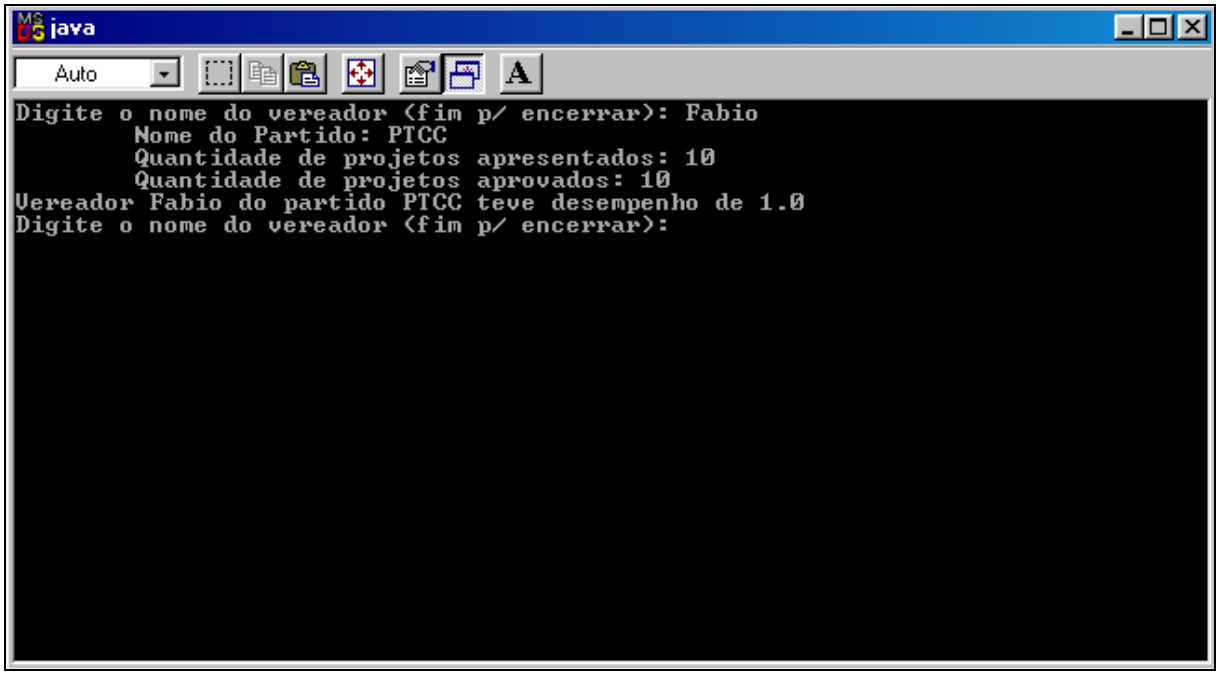
Após o início do sistema, com todos os arquivos.class pertencentes a sistema no diretório “c:\prototipo”, a primeira tela apresentada, como mostra a figura 7, informa qual classe está sofrendo reflexão e qual a que será executada, bem como algum possível erro.

Figura 7: Tela Inicial do Sistema.



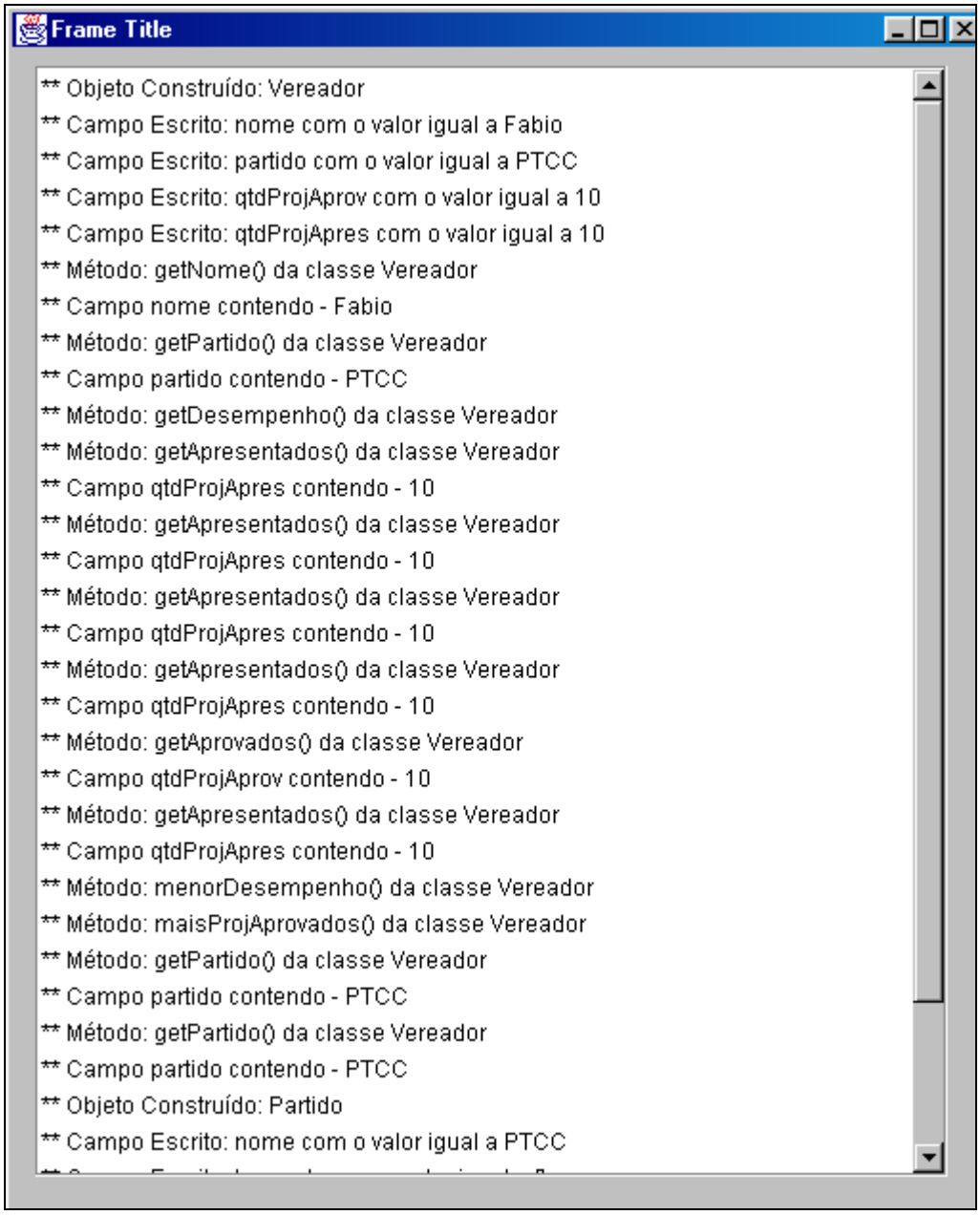
Em seguida é aberta a aplicação normalmente em outra janela, como mostra a figura 8.

Figura 8: Aplicação sendo executada.



As informações sobre a aplicação são interceptadas e mostradas em tempo de execução. Quando ocorre esta interceptação elas são mostradas em outra tela, como mostra a figura 9.

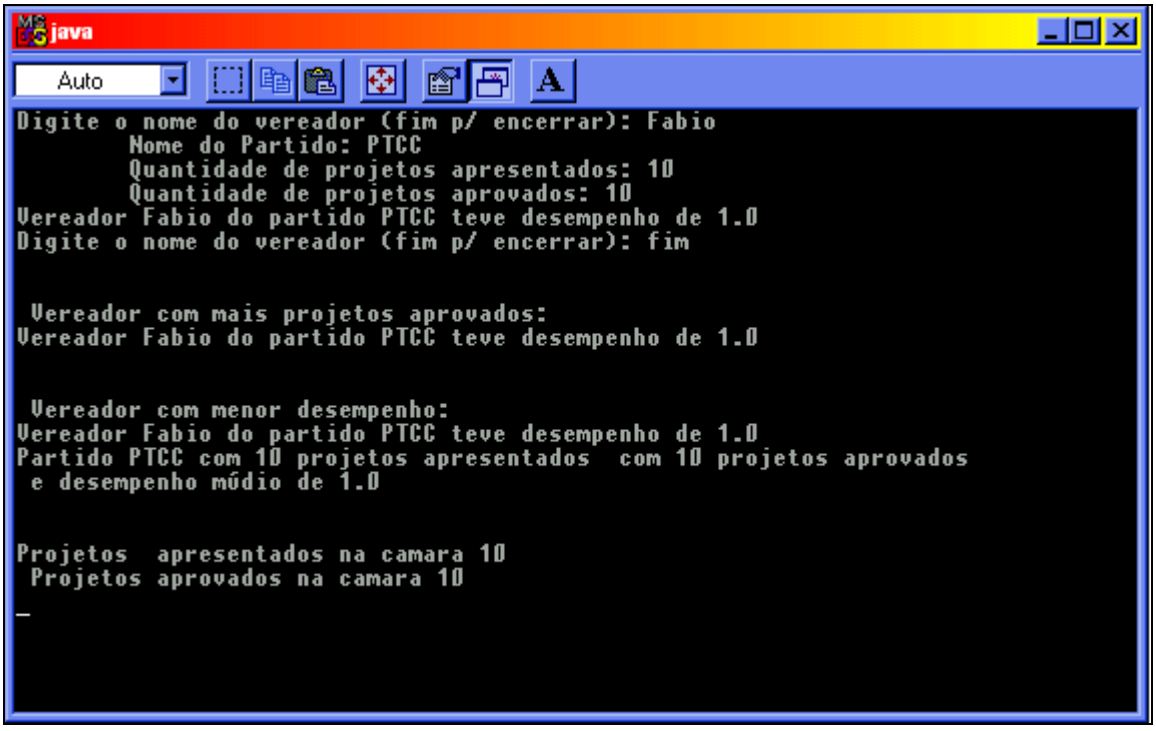
Figura 9: Interceptação das mensagens.



Como mostra a figura 9, pode-se verificar que um objeto Vereador foi instanciado com o campo nome com o valor igual a Fabio, o campo partido com o valor igual a PTCC, o campo qtdProjAprov com o valor igual a 10 e o qtdProjApres com o valor igual a 10. Em seguida foi utilizado o método getPartido() e leu o campo nome com o valor igual a Fabio.

A figura 10 mostra o fim da aplicação mostrada como exemplo.

Figura 10: Final da aplicação exemplo.



4 CONCLUSÕES

Durante este trabalho foram apresentados conceitos e ferramentas que possibilitam implementar sistemas reflexivos. Foi possível destacar as características de modelos e protocolos. Este trabalho pode ser utilizado por pessoas que queiram ter um fundamento básico sobre reflexão computacional.

Através do protótipo implementado e do exemplo listado, pode-se perceber que são verdadeiras as vantagens sobre a utilização de reflexão computacional em sistemas orientados a objetos, entre as quais incluem-se o poder e a flexibilidade de modificar o comportamento da linguagem.

Em teoria, o modelo reflexivo visa facilitar o trabalho do desenvolvedor, porém o conjunto de modelos teóricos, não é totalmente correspondido na prática. Pôde-se perceber, durante o trabalho, que, devido ao conceito de reflexão computacional ser algo novo, as ferramentas disponíveis não satisfazem de forma completa a teoria descrita. Linguagens de programação não correspondem totalmente aos modelos, e ferramentas, como o Javassist, possuem um conjunto de facilidades e um meta-protocolo que vem a atender de forma restrita algumas necessidades de sistemas reflexivos, como por exemplo a falta de comunicação do objeto para com seu meta-objeto.

Outro aspecto que deve ser levado em consideração é em que momento realizar a reflexão, se em tempo de compilação ou em tempo de execução. Utilizou-se neste trabalho o Javassist, uma ferramenta que realiza reflexão em tempo de execução, devido à natureza do protótipo.

Embora a utilização do Javassist não seja complicada, o seu aprendizado mostrou-se um tanto quanto custoso, pois o material disponível era escasso. Não foram encontrados exemplos, nem outras fontes para auxiliar na implementação. O material para o auxílio foi encontrado em Chiba (2000).

A respeito do protótipo implementado, os objetivos propostos foram parcialmente atingidos, pois não foi possível demonstrar graficamente a execução dos programas. Contudo,

ainda pode ser uma ferramenta útil no ensino de Orientação a Objetos, pois permite que a execução de um programa seja acompanhada e sua dinâmica seja melhor compreendida.

4.1 EXTENSÕES

Este trabalho propôs-se a monitorar aplicações simples, ou seja, classes que não estejam em pacotes e que haja um único *main*(). Como sugestão pode-se citar a continuação deste trabalho, com o intuito de monitorar aplicações complexas.

Outra limitação está na portabilidade do protótipo. Apesar de ter sido desenvolvido em Java, está preso ao sistema operacional Windows por procurar um diretório “c:\prototipo”.

Ainda como sugestões para futuros trabalhos pode-se citar o estudo e a avaliação de outros meta-protocolos, linguagens e ferramentas reflexivas.

ANEXO A: CÓDIGO FONTE DAS CLASSES PERTENCENTES AO MONITOR

Este anexo contém fonte da classe Monitor.

```
package reflexao;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javassist.*;
import javax.swing.*;

public class Monitor {

  static javassist.reflect.Loader cr = new javassist.reflect.Loader();
  static MostraClasses frame2 = new MostraClasses();

  public static void Monitorar(java.net.URLClassLoader cl)  {

    File directory = new File("c:\\prototipo");
    String classname = null;

    //busca lista do arquivos contidos no diretorio
    String [] files = directory.list();
    String main = null;

    for (int i=0;i<files.length;i++) {
      // busca somente os arquivos .class
      if (files[i].endsWith(".class")) {
        // remove a extensao .calss
                    classname = files[i].substring(0,files[i].length()-6);
        //veririca se a classe não possui o método main
        if (VerificaMain(cl,classname)) // se for true é porque não existe método main nesta
classe{
          try{
            //torna a classe reflexiva
            if (cr.makeReflective( classname , "reflexao.MetaObject",
"javassist.reflect.ClassMetaobject"))
              {
                reflexao.Frame1.Imprime("Classe: " + classname);
```

```
            reflexao.Frame1.Imprime("Tornou reflexiva!");
           }
          } catch( NotFoundException JANFE ){
           reflexao.Frame1.Imprime("Javassist - NotFoundException ..." +
JANFE.getMessage().toString() );
           continue;
          } catch(CannotCompileException JACCE ) {
           reflexao.Frame1.Imprime("Javassist - CannotCompileException ..." +
JACCE.getMessage().toString());
           continue;
          } catch(NullPointerException JANPE ){
           reflexao.Frame1.Imprime("Javassist - NullPointerException ..." +
JANPE.getMessage().toString());
           continue;
          }
       else
       {
          main = classname;
       }
     }
   }

   try{
     String[] args = {};
     //torna a classe reflexiva
     if (cr.makeReflective( main , "reflexao.MetaObject", "javassist.reflect.ClassMetaobject"))
     {
       reflexao.Frame1.Imprime("Classe: " + main);
       reflexao.Frame1.Imprime("Tornou reflexiva!");
       reflexao.Frame1.Imprime("Executando main da classe " + main);
       cr.run(main,args);
       reflexao.Frame1.Imprime("Terminou Execução.");
     }
   } catch( Throwable crT ){
     reflexao.Frame1.Imprime("cr - Throwable " + crT );
   }

 }

 public static void Mensagens(String s)
 {
   frame2.setVisible(true);
   frame2.Imprime(s);
 }

 public static boolean VerificaMain(java.net.URLClassLoader cl,String classe)
 {
```

```
    Class c=null;
    Method[] m = null;

    try {
      //carrega a classe para verificar a existencia do método main
      c = cl.loadClass(classe);
      m = c.getDeclaredMethods();
    } catch (ClassNotFoundException cCNFE) {
      Mensagens("VerificaMain - c - ClassNotFoundException ..." +
cCNFE.getMessage().toString());
    } catch (ArrayIndexOutOfBoundsException cAIOFBE){
      Mensagens("VerificaMain - c - ArrayIndexOutOfBoundsException ... " +
cAIOFBE.getMessage().toString());
    } catch (NullPointerException cNPE){
      Mensagens("VerificaMain - c - NullPointerException ... " + cNPE.getMessage() );
    }

    for (int q=0;q<m.length;q++) {
      if (m[q].getName().toString().equals("main"))
        return false;
    }
    return true; // não há main
  }
}

package reflexao;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.BoxLayout;
import java.util.*;
import java.io.*;

public class MostraClasses extends JFrame {
  JPanel contentPane;
  JScrollPane jScrollPane1 = new JScrollPane();
  static DefaultListModel listMode1 = new DefaultListModel();
  JList jList1 = new JList(listMode1);

  public MostraClasses() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
```

```
  }

  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

  private void jbInit() throws Exception {
    this.getContentPane().setLayout(null);
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(516, 664));
    this.setTitle("Frame Title");
    jScrollPane1.setBounds(new Rectangle(12, 9, 477, 581));
    jList1.setModel(listMode1);
    contentPane.add(jScrollPane1, null);
    jScrollPane1.getViewport().add(jList1, null);

  }
  public static void Imprime(String s)
  {
    listMode1.addElement(s);
  }

}

package reflexao;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javassist.*;
import javax.swing.*;

public class Frame1 extends JFrame {
  JPanel contentPane;
  JScrollPane jScrollPane2 = new JScrollPane();
  static DefaultListModel listMode2 = new DefaultListModel();
  JList jList2 = new JList(listMode2);
  java.net.URLClassLoader cl=null;

  /**Construct the frame*/
  public Frame1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

```
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  /**Component initialization*/
  private void jbInit() throws Exception,SecurityException,ClassNotFoundException{

    //setIconImage(Toolkit.getDefaultToolkit().createImage(Frame1.class.getResource("[Your
Icon]")));
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(519, 438));
    this.setTitle("Frame Title");
    jScrollPane2.setBounds(new Rectangle(12, 9, 496, 336));
    contentPane.add(jScrollPane2, null);
    jScrollPane2.getViewport().add(jList2, null);
  }
  /**Overridden so we can exit when window is closed*/
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

  public void Inicio()  {

    // mudar o classpath
    Properties pr = System.getProperties();
    String classPath = pr.getProperty("java.class.path");
    classPath = "c:\\prototipo;"+classPath;
    pr.setProperty("java.class.path",classPath);
    System.setProperties(pr);

    try {
      java.net.URL[] url = {(new java.net.URL("file:///C:/prototipo/"))};
      cl = new java.net.URLClassLoader(url, this.getClass().getClassLoader());
    } catch (java.net.MalformedURLException clMFURL) {
      Imprime("VerificaMain - cl - Malformed ..." + clMFURL.getMessage().toString());
    }

    reflexao.Monitor.Monitorar(cl);

  }
```

```
  public static void Imprime(String s){
        listMode2.addElement(s);
        }

}
```

ANEXO B: CÓDIGO FONTE DAS CLASSES PERTENCENTES AO META-OBJETO

Este anexo contém fonte da classe MetaObject.

```
package reflexao;

import javassist.reflect.Metaobject;
import javassist.*;
import javassist.reflect.*;
import java.lang.reflect.*;

public class MetaObject extends Metaobject {
  public MetaObject(Object self, Object[] args) {
          super(self, args);
       reflexao.Monitor.Mensagens("** Objeto Construído: " + self.getClass().getName());
   }

   public Object trapFieldRead(String name) {
      try{
        Field campo = super.getObject().getClass().getField(name);
        campo.setAccessible(true);
        reflexao.Monitor.Mensagens("** Campo " + campo.getName() + " contendo - " +
campo.get(super.getObject()).toString() );
      } catch( NoSuchFieldException camposNSFE ){
        reflexao.Monitor.Mensagens("teste - campos - NoSuchFieldException ..." +
camposNSFE );
      } catch( IllegalAccessException camposIAE ){
        reflexao.Monitor.Mensagens("teste - campos - IllegalAccessException ..." +
camposIAE );
      } catch( NullPointerException camposNPE ){
        reflexao.Monitor.Mensagens("teste - campos - NullPointerException ..." + camposNPE
);
      }
      return super.trapFieldRead(name);
   }

   public void trapFieldWrite(String name, Object value) {
         reflexao.Monitor.Mensagens("** Campo Escrito: " + name + " com o valor igual a " +
value.toString() );
         super.trapFieldWrite(name, value);
   }
```

```
    public Object trapMethodcall(int identifier, Object[] args)
            throws Throwable
    {
            reflexao.Monitor.Mensagens("** Método: " + getMethodName(identifier)
                                    + "() da classe " + getClassMetaobject().getName());
            return super.trapMethodcall(identifier, args);
    }
}
```

ANEXO C: CÓDIGO FONTE DAS CLASSES PERTENCENTES AO EXEMPLO UTILIZADO

Este anexo contém o código fontes das classes Vereador, Partido e EXE05_Lista4.

```
import java.io.*;
import java.util.*;


public class EXE05_Lista4
{
	public static void main(String[] args) throws IOException
	{
		BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));
		Vereador atualVereador, vereadorMaisAprovados = null ,
vereadorMenorDesempenho = null;
		Partido atualPartido;
		HashMap camara = new HashMap();
		System.out.print("Digite o nome do vereador (fim p/ encerrar): ");
		String aux2, aux3, aux4, aux1 = teclado.readLine();
		while (!aux1.equalsIgnoreCase("fim"))
		{
			System.out.print("\tNome do Partido: ");
			aux2 = teclado.readLine().toUpperCase();
			System.out.print("\tQuantidade de projetos apresentados: ");
			aux3 = teclado.readLine();
			System.out.print("\tQuantidade de projetos aprovados: ");
			aux4 = teclado.readLine();
			atualVereador = new Vereador( aux1, aux2, Integer.parseInt(aux4),
Integer.parseInt(aux3));
			System.out.println("Vereador "+ atualVereador.getNome() +
								" do partido " +
atualVereador.getPartido() +
								" teve desempenho de " +
atualVereador.getDesempenho());

			if (atualVereador.menorDesempenho(vereadorMenorDesempenho))
				vereadorMenorDesempenho = atualVereador;

			if (atualVereador.maisProjAprovados(vereadorMaisAprovados))
				vereadorMaisAprovados = atualVereador;

			atualPartido = (Partido) camara.get(atualVereador.getPartido());
	//busca no hashmap se há algum partido com esse nome
```

```
                        if (atualPartido == null)         //se não localizou
                        {
                                atualPartido = new Partido(atualVereador.getPartido());
        //criar um novo partido
                                camara.put(atualPartido.getNome(), atualPartido);
        //guardar o partido no hashmap
                        }
                        atualPartido.addVereador(atualVereador);     //agrega o vereador no
partido

                        System.out.print("Digite o nome do vereador (fim p/ encerrar): ");
                        aux1 = teclado.readLine();
                }               //fim do while

        System.out.println("\n\n Vereador com mais projetos aprovados: ");
        System.out.println("Vereador "+ vereadorMaisAprovados.getNome() +
                                        " do partido " + vereadorMaisAprovados.getPartido()
+
                                        " teve desempenho de " +
vereadorMaisAprovados.getDesempenho());

        System.out.println("\n\n Vereador com menor desempenho: ");
        System.out.println("Vereador "+ vereadorMenorDesempenho.getNome() +
                                        " do partido " +
vereadorMenorDesempenho.getPartido() +
                                        " teve desempenho de " +
vereadorMenorDesempenho.getDesempenho());

        //listar sequencialmente os partidos
        Iterator percorre;  //declara um objeto  de iterator(iterador / sequenciador / repetidor)
        percorre = camara.values().iterator(); //hashmap camara é transformado em collection
pelo
                                                                //método values,
e o collection permite a varredura
                                                                //sequencial pelo
Iterator

        int totAprov = 0 , totApres = 0;
        while (percorre.hasNext())                      //itens 5 e 6
        {
                atualPartido = (Partido) percorre.next();
                System.out.println("Partido " + atualPartido.getNome() +
                                        " com " + atualPartido.getApresentados() + "
projetos apresentados " +
                                        " com " + atualPartido.getAprovados() + "
projetos aprovados " +
                                        "\n e desempenho médio de " +
atualPartido.getDesempenhoMedio());
```

```
                totAprov += atualPartido.getAprovados();
                totApres += atualPartido.getApresentados();
        }       //fim do while
        System.out.println("\n\nProjetos  apresentados na camara " + totApres);
        //item 1
        System.out.println(" Projetos aprovados na camara " + totAprov ) ;




        }
}       //fim da classe EXE05_Lista4

import java.io.*;
import java.util.*;


public class Partido
{
        //atributos
        public String nome;
        public ArrayList bancada;      //container que vai conter(agregar) os objetos de
Vereador

        //construtor
        public Partido( String nome )
        {
                this.nome = nome;
                bancada = new ArrayList();
        }

        //métodos
        public String getNome()                         {return nome;}

        public void addVereador(Vereador um)
        {
                bancada.add(um);
        }

        public int getAprovados()
        {
                int soma = 0;
                Vereador parlamentar;

                for (int x=0; x < bancada.size() ; x++ )
                {
                        parlamentar = (Vereador) bancada.get(x);
```

```
                soma += parlamentar.getAprovados();
            }
            return soma;
        }

        public int getApresentados()
        {
            int soma = 0;
            Vereador parlamentar;

            for (int x=0; x < bancada.size() ; x++ )
            {
                parlamentar = (Vereador) bancada.get(x);
                soma += parlamentar.getApresentados();
            }
            return soma;
        }

        public float getDesempenhoMedio()
        {
            float soma = 0;
            Vereador parlamentar;

            for (int x=0; x < bancada.size() ; x++ )
            {
                parlamentar = (Vereador) bancada.get(x);
                soma += parlamentar.getDesempenho();
            }
            return (soma / bancada.size());
        }
}               //fim da classe partido

import java.io.*;
import java.util.*;

public class Vereador
{
            //atributos
            public String nome, partido;
            public int qtdProjAprov, qtdProjApres;

            //construtor
            public Vereador(String nome, String partido, int aprov, int apres)
            {
                        this.nome = nome;
                        this.partido = partido;
                        if ( aprov > 0 )
                                this.qtdProjAprov = aprov;
```

```
                        else
                                this.qtdProjAprov = 0;
                        if ( apres > 0 )
                                this.qtdProjApres = apres;
                        else
                                this.qtdProjApres = 0;
        }

        //Métodos
        public String getNome()                 {return nome;}
        public String getPartido()              {return partido;}
        public int getApresentados()  {return qtdProjApres;}
        public int getAprovados()               {return qtdProjAprov;}

        public float getDesempenho()
        {
                if (getApresentados() == 0)
                        return 0.0f;

                float indice;
                if (getApresentados() > 17)
                        indice = 1.22f;
                else
                        if (getApresentados() > 10)
                                indice = 1.08f;
                        else
                                if (getApresentados() > 5)
                                        indice = 1f;
                                else
                                        indice = 0.08f;

                return ((float)getAprovados() / getApresentados() * indice);
        }

        public boolean maisProjAprovados(Vereador outro)
        {
                if (outro == null || this.getAprovados() > outro.getAprovados())
                        return true;
                else
                        return false;
        }

        public boolean menorDesempenho(Vereador outro)
        {
                return (outro == null || this.getDesempenho() > outro.getDesempenho());
        }
};                      //fim da classe vereador
```

REFERÊNCIAS BIBLIOGRÁFICAS

ARMSTRONG, Eric **Jbuilder 2 bible.** Foster City: IDG Books Wordwide, 1998.

BARTH, Fabrício J.. **Utilização de Reflexão Computacional para Implementação de Aspectos não Funcionais em um Gerenciador de Arquivos Distribuídos**, 2000. 90 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CHIBA, Shigeru. **Welcome do Javassist 2.** Tokyo, Abr [2000?]. Disponível em: <http://www.hlla.is.tsukuba.ac.jp/>. Acesso em: 15 nov., 2002.

CHIBA, Shigeru. **Open C++ Tutorial**, Tokyo: University of Tsukuba, 1998.

DOURISH, Paul. **Computational Reflection and CSCW Design.** Relatório de pesquisa na Rank Xerox Research Centre, Cambridge, mar. 1992.

FURLAN, José D. **Modelagem de objetos através da UML - The Unifield Modeling Language**. São Paulo: Makron Books, 1998.

GRAND, Mark; KNUDSEN, Jonathan. Java Fundamental Classes Reference. O´Reilly, 1997.

HUGO, Marcel **Página das disciplinas Programação II e Análise e Programação Orientada a Objetos**. Blumenau, [2001]. Disponível em: <http://home.furb.br/marcel/>. Acesso em : 10 abr, 2001.

LISBÔA, M. **Arquiteturas de meta-nível**. In: XI Simpósio Brasileiro de Engenharia de Software. Fortaleza, 1., 1997. **Anais…** Fortaleza: UFC, 1997. p. 21-28.

MAES, P. **Concepts and Experiments in Computational Reflection**. In OOPSLA '87 Proceedings. 1987, Orlando. Anais... Orlando: 1987, p. 147-155.

NAUGHTON, Patrick **Dominando o JAVA :[guia autorizado da Sun Microsystems]**. São Paulo: Makron Books, 1997.

SENRA, Rodrigo Dias A.. **Programação Reflexiva sobre o Protocolo de Meta-Objetos Guaraná**, Campinas, nov. [2001]. Disponível em: <http://www.ic.unicamp.br/~921234/dissert/dissert.html>. Acesso em: 15 nov. 2002.

SIZHONG, Yang; JINDE, Liu. **RECOM: A Reflective Architecture of Middleware.** China: College of Computer Science and Engineering, UEST of China, 2001.

SOBEL, Jonathan M.; FRIEDMAN, Daniel P. **An Introduction to Reection - Oriented Programming**, In: Reflection '96, 1996, San Francisco. **Anais…** Indiana: Indiana Univerisy.

SOUZA, Cristina Verçosa Pérez Barrios de. **Habilitação de Reflexão Computacional através dos Mecanismos de Montagem e Implantação da Plataforma J2EE** 2001. 15 f. Programa de Pós-Graduação em Informática Aplicada - Pontifícia Universidade Católica do Paraná, Curitiba.

STEEL, L. **Beyond objects**. In: European Conference on object-oriented programming. Bologna. 1., 1994. **Anais…** Bologna: Lecture notes in computer science n. 821, 1994. p. 1 – 11.

SULLIVAN, Gregory T.. **Aspect-Oriented Programming using Reflection and Metaobject Protocols**. Massachusetts: Massachusetts Institute of Technology, 2001.

SULLIVAN, Gregory T.. Aspect-Oriented Programming using Reflection. In: OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay. **Anais…** Tampa Bay: Massachusetts Institute of Technology, 2001.

SUN Microsystems **Java Standard Edition Platform Documentation,** [S.1.], [1995]. Disponível em : <http://java.sun.com/docs/ >. Acesso em: 15 nov., 2002.

STANKOVIC, John A.; Ramamritham, Krithi. **A Reflective Architecture for Real-Time Operating Systems**. Charlottesville: University of Virginia, 1997.

TATSUBORI, Michiaki.**An Extension Mechanism for the Java Language.** 1999. 64 f. Tese de Mestrado - Universidade de Tsukuba, Tokyo.

WINBLAD, Ann L.; EDWARDS, Samuel D.; HING, David R. **Software Orientado ao Objeto**. São Paulo: Makron Books, 1993.