UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

UTILIZAÇÃO DE REFLEXÃO COMPUTACIONAL PARA IMPLEMENTAÇÃO DE ASPECTOS NÃO FUNCIONAIS EM UM GERENCIADOR DE ARQUIVOS DISTRIBUÍDOS

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

FABRÍCIO JAILSON BARTH

BLUMENAU, AGOSTO/2000

2000/1-20

UTILIZAÇÃO DE REFLEXÃO COMPUTACIONAL PARA IMPLEMENTAÇÃO DE ASPECTOS NÃO FUNCIONAIS EM UM GERENCIADOR DE ARQUIVOS DISTRIBUÍDOS

FABRÍCIO JAILSON BARTH

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. Maurício Capobianco Lopes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Marcel Hugo

_____

Prof. Mauro Marcelo Mattos

“Qual será o absurdo de hoje que será a verdade de amanhã ?”.

(Alfred North Whitehead (1925))

Este trabalho é dedicado à minha família,
em especial a meu Pai e minha Mãe,
pelo apoio recebido ao longo de
toda a minha vida.

Agradecimentos

Gostaria de agradecer, principalmente, o professor Maurício, que concordou em realizar este trabalho, me auxiliando nas dúvidas que iam surgindo durante o período e sempre disposto a cobrar e otimizar os resultados.

Não posso esquecer dos colegas e amigos que fiz durante o período de quatro anos e meio que estive realizando o curso: o pessoal da minha sala, pessoas com que eu sempre pude contar para realizar trabalhos, trocar experiências e fazer festa ! O pessoal do Núcleo de Informática (meu primeiro protótipo de trabalho), o pessoal do Centro Acadêmico, o pessoal do Protem, o pessoal do voley, os meus amigos de outros cursos, a tartaruga que fugiu da minha casa e os meus professores, que de uma forma, ou de outra, foram os responsáveis por grande parte do conhecimento que hoje eu possuo.

Foram muitos sábados de Quake, saídas para rapel, congressos, rafting, muito vinho, cerveja, chopp, pascal, fila circular, polimorfismo, grafos, C, semáforos, noites sem dormir, às vezes por motivos de aula, outras não tão acadêmicas assim.

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

Sumário

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Abreviaturas

AST – *Abstract Syntax Tree*

MOP – *Metaobject Protocol*

UML – *Unified Modeling Language*

Resumo

Este trabalho descreve modelos e conceitos sobre implementação de reflexão computacional através da orientação a objetos. A reflexão computacional, no modelo de objetos, 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 adicionar novos comportamentos e aspectos não funcionais a um gerenciador de arquivos distribuídos, a fim de avaliar os meta protocolos existentes para a linguagem Java, com a teoria descrita. Exemplos descrevendo os modelos reflexivos, linguagens reflexivas, vantagens e desvantagens, bem como a implementação de novas características adicionadas a um gerenciador de arquivos são discutidas durante o trabalho. A implementação é realizada utilizando as ferramentas OpenJava e Javassist. Para situações em tempo de compilação é utilizado o metaprotocolo do pré-processador OpenJava, e para situações em tempo de execução é utilizado o metaprotocolo da ferramenta Javassist.

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 and not functional aspects to a distributed archives manager, in order to evaluate the goal existing protocols for the Java language, with the described theory. Examples describing the reflective models, reflective languages, advantages and disadvantages, as well as the implementation of new features added to a manager of archives are argued during the work. The implementation is carried through using the tools OpenJava and Javassist. For situations in compilation time metaprotocol of the OpenJava daily pre-processor is used, and for situations in execution time metaprotocol of the Javassist tool is used.

1. Introdução

1.1. Contextualização / Justificativa

Sistemas modernos requerem um alto grau de confiabilidade, disponibilidade, performance e tolerância a falhas, como por exemplo, centrais telefônicas, controle de trens, banco de dados e sistemas distribuídos.

À medida que estes sistemas vão evoluindo, o seu grau de complexidade evolui proporcionalmente, exigindo, dos desenvolvedores, o conhecimento de novas metodologias, conceitos e técnicas que possam suprir as dificuldades e simplificar o desenvolvimento.

Em busca de novas soluções, [KIC1991] propõe que, a separação dos aspectos funcionais dos não funcionais, em sistemas complexos, pode trazer aos desenvolvedores uma maior facilidade ao modelar e implementar tais sistemas. Aspectos funcionais de um sistema são todas as tarefas que levam a conclusão do objetivo principal do problema, enquanto que aspectos não funcionais são todas as tarefas que servem como suporte para que as tarefas principais cumpram seus objetivos.

Para dividir os aspectos funcionais dos não funcionais pode-se utilizar como padrão a arquitetura de meta níveis. Esta arquitetura detêm, em seus níveis inferiores, os objetos base (funcionais) e em seus níveis superiores, os metaobjetos (não funcionais), que têm o controle dos objetos base podendo modificar a sua estrutura e comportamento.

Tal atividade só é possível utilizando o conceito de reflexão computacional, que segundo [LIS1997] 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.

Já [STE1994], 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 retornar o processo de execução.

Para alcançar tal capacidade, o conceito de reflexão computacional se utiliza de objetos abertos (que através de uma meta interface permitem acessar, invocar e alterar um objeto sem estar delimitada a interface do mesmo) e de uma arquitetura composta por vários níveis, chamada de arquitetura de meta níveis.

Os objetos abertos possibilitam ao programador, não só ter acesso apenas à interface pública do objeto, mas também à interface privada, sua estrutura de dados e o comportamento interno. Esta capacidade torna possível que o programador ajuste o objeto da forma que desejar, tendo uma maior reusabilidade e poder de adaptabilidade do objeto.

Uma arquitetura de meta níveis é composta por vários níveis interligados através de um protocolo de comunicação entre os objetos. A utilização de uma arquitetura de meta níveis traz, como vantagens, a redução de complexidade, separação conceitual, reutilização e um maior grau de adaptabilidade para o sistema.

Para poder estudar e implementar tais conceitos escolheu-se um gerenciador de arquivos distribuídos [POS2000], que fornece, de forma transparente ao usuário, a localização física dos arquivos que deseja armazenar e recuperar, e permite a organização dos arquivos em estruturas de diretórios. Porém não trata de conceitos, como replicação, detecção de falhas no ambiente e gerenciamento de performance, conceitos estes que podem ser tratados como conceitos não funcionais deste sistema.

Atribuiu-se ao sistema descrito em [POS2000], através do conceito de reflexão computacional, conceitos não funcionais a um sistema gerenciador de arquivos distribuídos.

Durante o trabalho também foram estudadas as ferramentas OpenJava [TAT2000] e Javassist [CHI2000], ferramentas estas que permitem implementar conceitos de reflexão computacional sobre a linguagem Java em tempo de compilação e execução, respectivamente.

Assim, este trabalho permite a visualização, em grande parte, das características, vantagens e conceitos de reflexão computacional e das ferramentas OpenJava e Javassist, aplicados a um problema de adaptação de requisitos não funcionais em um sistema gerenciador de arquivos distribuídos.

Com isto, foi possível realizar um levantamento dos vários modelos, técnicas e ferramentas do conceito de reflexão computacional, verificando a real aplicabilidade do

conceito com exemplos reais e validar as vantagens levantadas, como: redução de complexidade, adaptabilidade e reutilização.

1.2. Objetivos

O objetivo principal do trabalho é modelar e implementar uma solução que adicione conceitos não funcionais a um sistema gerenciador de arquivos distribuídos, utilizando reflexão computacional em uma arquitetura de meta níveis.

Os objetivos secundários do trabalho são:

a) listar conceitos e formas de implementação sobre reflexão computacional, buscando um embasamento teórico sobre o assunto;

b) avaliar o funcionamento do pré-processador OpenJava e da ferramenta Javassist, tentando relacionar a teoria com a prática.

1.3. Organização do trabalho

Este trabalho é organizado do seguinte modo.

O capítulo 2 descreve conceitos, técnicas e aplicações de reflexão computacional no modelo de objetos e sua utilização em arquiteturas de meta-nível.

O capítulo 3 discute características e aspectos funcionais sobre o pré-processador OpenJava e a ferramenta Javassist.

O capítulo 4 apresenta o gerenciador de arquivos distribuídos, implementado em [POS2000].

O capítulo 5 discute e apresenta a especificação e implementação dos aspectos não funcionais, selecionados neste trabalho, que atuam sobre o gerenciador de arquivos distribuídos.

O capítulo 6 conclui as idéias principais do trabalho.

2. Reflexão Computacional

Segundo [MAE1987], reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações. Por sistema computacional entende-se um sistema baseado em um computador, com o objetivo de realizar computações e fornecer informações em um determinado domínio da aplicação. Neste conceito, a reflexão é definida como uma forma de introspecção, no qual o sistema tenta tirar conclusões sobre suas próprias computações, as quais podem ser afetadas posteriormente.

A idéia básica 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 ou não funcionais a módulos já existentes.

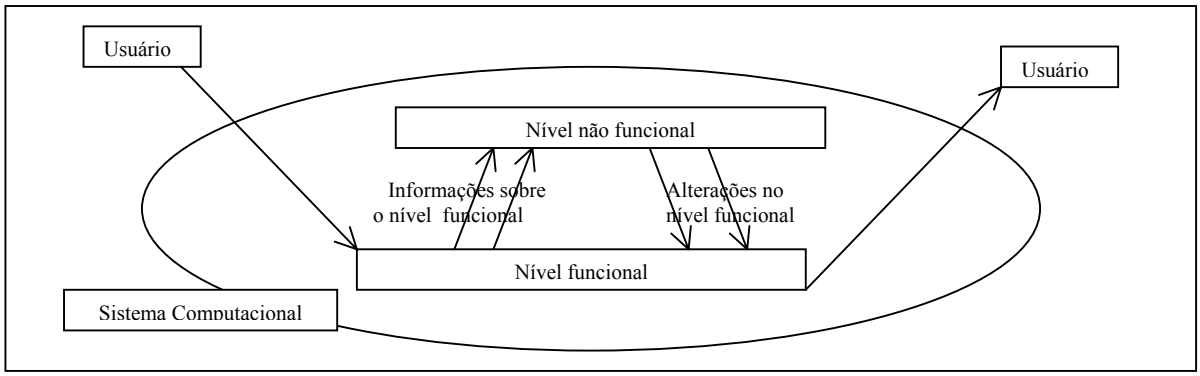
Segundo [LIS1997], a reflexão computacional define uma nova arquitetura de software. Este modelo de arquitetura é composto por um meta-nível, onde se encontram estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base. A separação de domínios é o aspecto mais atraente da arquitetura reflexiva, não apenas por possibilitar a construção de sistemas adaptáveis e incentivar a reutilização de componentes, mas principalmente por permitir ao programador da aplicação concentrar-se na solução do problema de programação específico do domínio da aplicação. Neste aspecto, arquiteturas de meta-nível têm sido adotadas para expressar propriedades não funcionais do sistema, como confiabilidade e segurança, de forma independente do domínio da aplicação.

Segundo [CYS1997], os requisitos não funcionais, ao contrário dos funcionais, não expressam nenhuma função a ser realizada pelo software, e sim comportamentos e restrições que este software deve satisfazer. Requisitos não funcionais a muito são mencionados em métodos de desenvolvimento, nomes como: restrições de software, condições de contorno, são algumas das denominações utilizadas. Apesar disso, raramente encontra-se um software em que os requisitos não funcionais tenham sido levados em consideração ou mesmo listados de maneira adequada.

A separação de domínios, que a reflexão computacional oferece, traz uma nova possibilidade de modelagem e implementação de características não funcionais ao sistema.

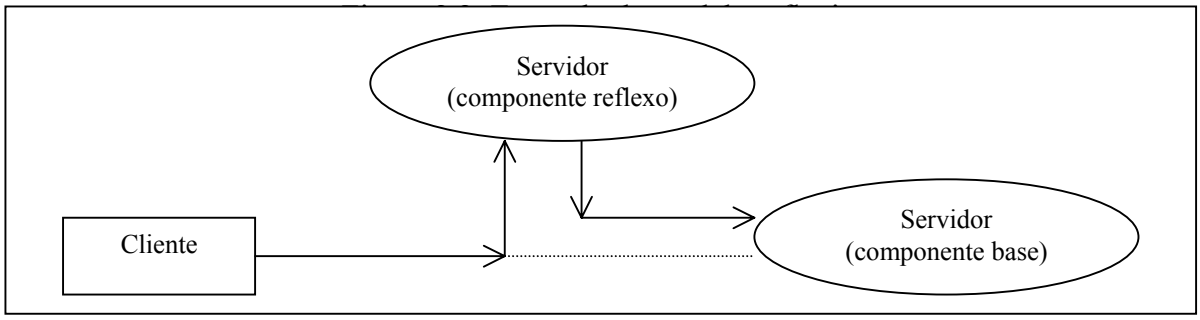
Na figura 2.1, pode-se visualizar, de forma genérica, o processo de reflexão em um sistema computacional. O sistema computacional é dividido em dois ou mais níveis computacionais. O ambiente externo (usuário) envia uma mensagem ao sistema computacional. Esta mensagem é tratada pelo nível funcional, que é responsável por executar corretamente a tarefa, e o nível não funcional realiza a tarefa de gerenciar o funcionamento do nível funcional.

Figura 2.1: Visualização genérica de um sistema computacional reflexivo



Como exemplo, pode-se utilizar uma aplicação de tempo real que implementa um sistema embarcado responsável por conduzir um objeto por uma esteira por determinado tempo. Neste caso o nível funcional é responsável pela lógica necessária para conduzir o objeto pela esteira e o nível não funcional é responsável por aspectos relativos a restrições temporais (visualizar se a tarefa será executada dentro do seu *deadline*), restrições de sincronização, exceções temporais e escalonamento em tempo real.

Outra forma de caracterizar reflexão computacional é demonstrado na figura 2.2, onde uma estrutura cliente requisita um serviço a uma estrutura servidor. A estrutura servidor é formada por um componente base e um componente reflexo. O componente reflexo possui o mesmo comportamento do que o componente base, porém ele funciona como um filtro, interceptando as informações ou invocações que deveriam ir diretamente ao componente básico, podendo alterar ou ajustar as características do componente base, antes de repassá-las.



Segundo [WU1997], a idéia básica sobre reflexão computacional está em:

a) separar as funcionalidades básicas das não funcionalidades básicas através de níveis arquiteturais;

b) as funcionalidades básicas devem ser satisfeitas pelos objetos da aplicação;

c) as não básicas devem ser satisfeitas pelos metaobjetos;

d) as capacidades não funcionais são adicionadas aos objetos da aplicação através de seus metaobjetos específicos;

e) o objeto base pode ser alterado estruturalmente e comportalmente em tempo de execução, ou compilação.

Atribuir a um sistema tal capacidade, significa dar-lhe, flexibilidade, para alterar e adaptar dinamicamente sua estrutura e comportamento, redução de complexidade, separação conceitual, reutilização, transparência e uma maior adaptabilidade.

Embora reflexão computacional possa ser utilizada em programação funcional e programação em lógica, é no modelo de objetos, com sua natural flexibilidade e facilidade de programação incremental, que ela tem mostrado a sua eficácia e elegância na obtenção de novas soluções no problema de programação.

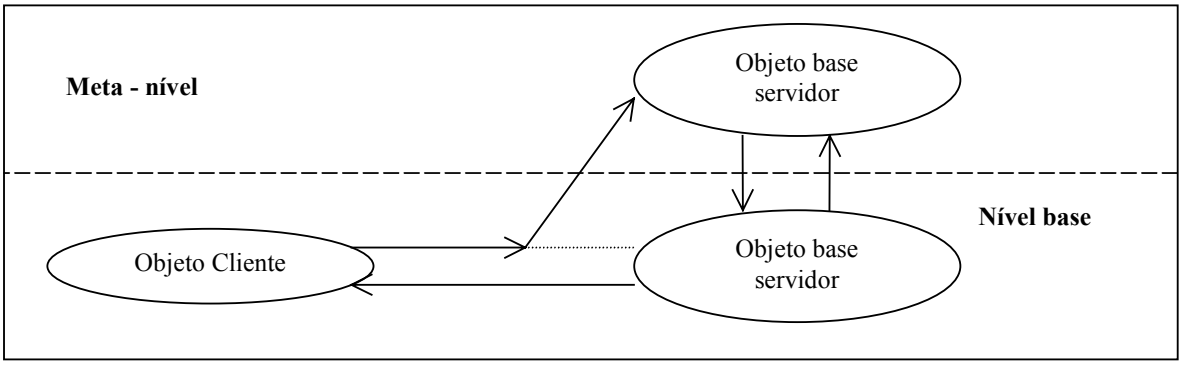
2.1. Reflexão computacional no modelo de objetos

Reflexão computacional está baseada na noção de definir um interpretador de uma linguagem de programação para a própria linguagem. 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 redefinidas por meio de metaclasses e metaobjetos, que estão dispostos em um nível diferente das classes e objetos. O nível no qual as metaclasses e metaobjetos estão dispostos é chamado de meta-nível.

Retornando ao exemplo da figura 2.2, pode-se analisar esta situação como sendo um objeto cliente que envia uma mensagem ao objeto servidor (objeto base), e sem o objeto cliente ter conhecimento, esta mensagem é interceptada pelo metaobjeto servidor, localizado no meta-nível (e associado ao objeto base servidor) (figura 2.3).

Figura 2.3: Exemplo de reflexão no modelo de objetos



Para realizar tal atividade o meta-nível precisa possuir algumas informações sobre o nível base, tais como:

a) classes de um objeto: fornece acesso ao nome da classe, identificando o tipo do objeto;

b) herança: fornece informações a respeito de classes ascendentes e descendentes;

c) propriedades: fornece informações sobre a estrutura de dados de uma classe e seus métodos, podendo incluir informações sobre restrições de acesso aos membros da classe (público, privado);

d) comportamento: a forma com que o objeto recebe e trata, ou executa, a mensagem recebida.

De forma mais genérica, pode-se dizer que os atributos e comportamentos das entidades do nível base tornam-se dados e valores, respectivamente, nas entidades do meta-nível.

Um sistema reflexivo consolida-se depois de completado 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 realizar tal serviço é necessário dispor de algum mecanismo que permite comunicação entre o objeto modificado e o objeto modificador, ou seja, entre o nível base e o meta-nível.

Esta comunicação é implementada através de uma interface que é definida através do *Metaobject Protocol* (MOP).

2.2. Protocolo de metaobjetos

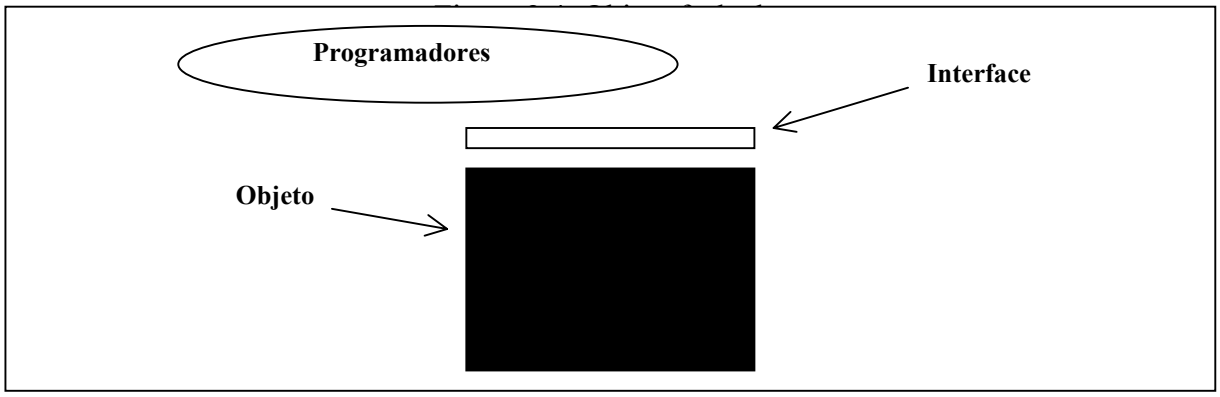
O *Metaobject Protocol*, também chamado de protocolo de metaobjetos, tem como responsabilidade, fornecer soluções para tais questões:

a) quais entidades devem ser transformadas em algo que possa sofrer operações no meta-nível ?

b) como o relacionamento entre o nível base e o meta-nível é implementado ?

c) quando o sistema passa para o meta-nível ?

Em sistemas orientados a objetos convencionais o analista faz a modelagem dos objetos de forma fechada, formando uma caixa preta com uma interface, a qual o programador tem acesso. A figura 2.4 mostra um objeto fechado com uma interface limitada.



Fonte: [KIC1996]

À medida que os programadores vão utilizando este objeto fechado, irão também encontrar maiores dificuldades para adaptar o objeto às suas necessidades, e em reutilizar este objeto para aplicações distintas.

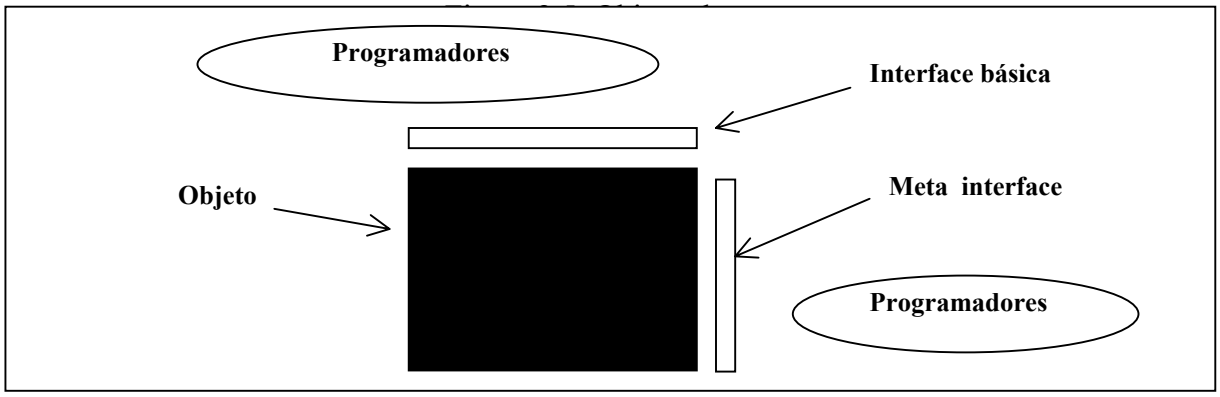
Em sistemas orientados a objetos convencionais, pode-se utilizar o conceito de herança para adaptar e reutilizar determinado objeto, redefinindo a implementação de operações já existentes, mas sem ter acesso a estrutura e comportamento interno do objeto.

Se ao objeto, apresentado na figura 2.4, fosse adicionado uma nova interface, que permitisse ao programador acessar e modificar a estrutura da implementação, problemas como falta de flexibilidade e adaptabilidade de aplicações seriam sanados, além de atribuir ao programador maior responsabilidade sobre a aplicação. Isto é chamado de implementação aberta, em inglês *Open Implementation.*

Segundo [KIC1996], implementações abertas possibilitam ao programador ter acesso à implementação, poder inspecionar, entender e modificar a implementação de acordo com as características que deseja implementar, obtendo maior funcionalidade e desempenho do componente. Tal nível de acesso, possibilita ao programador resolver muito dos seus problemas.

Quando um objeto contêm, além de sua interface básica, uma meta interface que dá acesso à sua implementação, este objeto é chamado de objeto aberto.

O modelo de um objeto aberto seria basicamente aquele visualizado na figura 2.4, com a adição de uma nova interface, chamada de meta interface (figura 2.5).

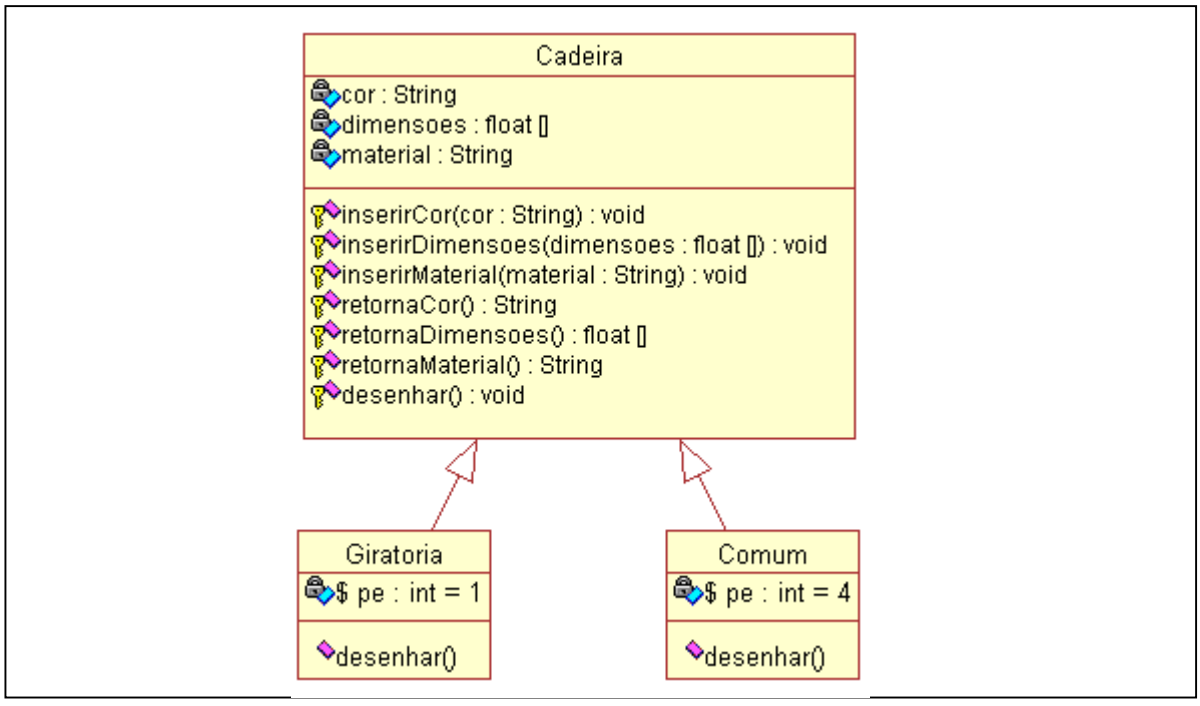


Fonte:[KIC1996]

Para fins de esclarecimento quanto ao real funcionamento de um objeto aberto, pode-se utilizar um exemplo de uma aplicação gráfica, onde, para desenhar uma cadeira, o programador utiliza um objeto fechado oriundo do diagrama apresentado na figura 2.6.

Neste diagrama, pode-se visualizar uma classe chamada *Cadeira* que possui duas sub-classes, a classe *Giratoria* e *Comum*. A classe *Giratoria* representa todas as cadeiras do tipo giratória e a classe *Comum* representa todas as classes com 4 pés. A classe *Cadeira* possui os atributos cor, dimensões e material, além de métodos para poder acessar estes atributos (métodos protegidos).

Figura 2.6: Diagrama de classes de um exemplo de objeto fechado



Para o programador, que irá utilizar estes objetos para realizar a implementação de todo o sistema gráfico, as operações possíveis são:

a) a execução do método *desenhar()* na classe *Giratória*, que traz como resultado o desenho de uma cadeira giratória com a cor, dimensão e material informados durante a execução da aplicação;

b) a execução do método *desenhar()* na classe Comum, que por sua vez, traz como resultado o desenho de uma cadeira comum com a cor, dimensão e material informados durante a execução da aplicação.

Logo, o programador possui um objeto que vem suprir as suas necessidades quanto ao desenho de cadeiras giratórias e de 4 pés, porém, se em algum momento ele desejar desenhar, por exemplo, uma cadeira que possua 3 pés, não será possível, pois o seu objeto não comporta a lógica necessária para realizar tal atividade e não é flexível e nem possui adaptabilidade suficiente para mudar o seu comportamento e realizar tal tarefa.

Se a este objeto fosse adicionado a meta interface que segue na tabela 2.1, poderia ser implementada uma adaptação da aplicação, utilizando-se a meta interface, para possibilitar que a aplicação desenhe cadeiras com três pernas.

Tabela 2.1: Métodos de uma meta interface de uma linguagem hipotética

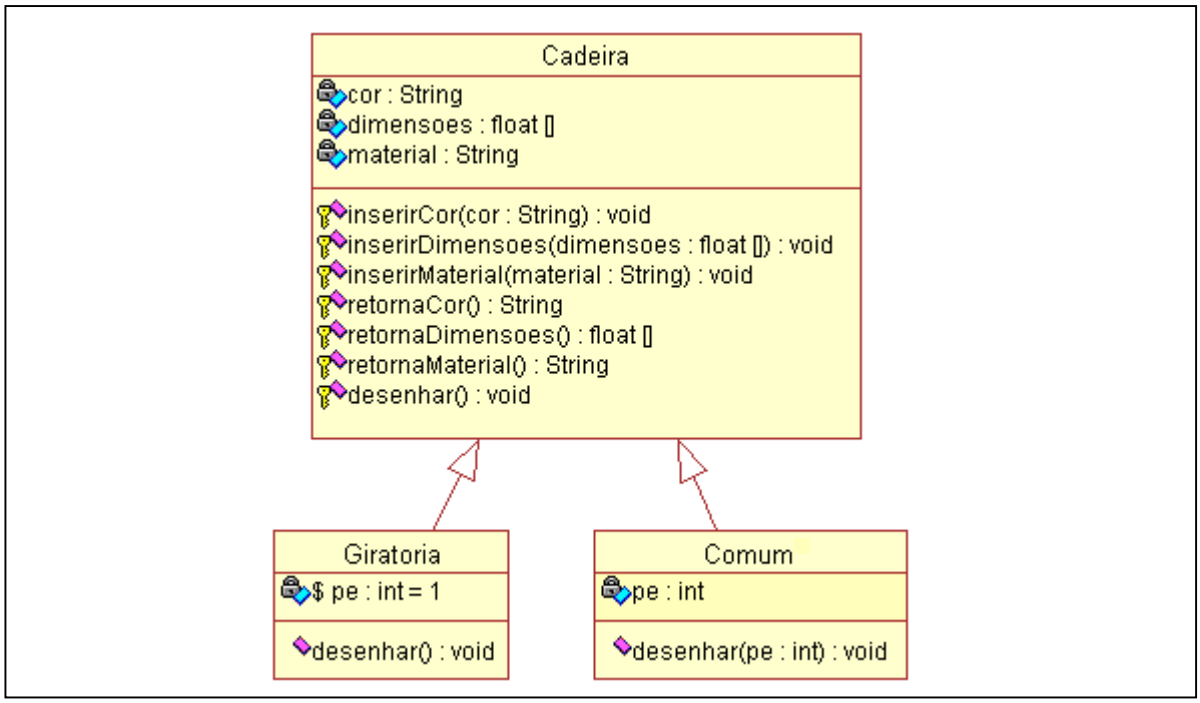
Método	**Descrição**
Classe obtemClasse (String classe)	Através do nome da classe o método retorna um objeto representando a classe
Classe [] retornaSuperClasse ()	Método que retorna um vetor de super classes da classe alvo.
Classe [] retornaSubClasse ()	Método que retorna um vetor de sub classes da classe alvo
Metodo [] retornaMetodos ()	Método que retorna um vetor de todos os métodos da classe alvo
Atributo [] retornaAtributo ()	Método que retorna um vetor de todos os atributos da classe alvo.
void alteraEstadoAtributo (Atributo atributo)	Método que permite alterar o estado (privado, público ou protegido) do atributo listado como parâmetro
void alteraNomeClasse ()	Método que permite alterar o nome da classe alvo.
void adicionaCorpoMetodo (String codigo, int linha)	Método que permite alterar o comportamento de um determinado método, adicionando determinada linha de código

O código da possível adaptação segue no quadro 2.1:

```
public Classe metaInterface () {
        Classe classe obtemClasse("Comum");
        // a variavel atributo recebe todos os atributos da classe Comum
        Atributo [] atributos = classe.retornaAtributo();
        for (int i=0; i<atributos.tamanho(); i++){
                if (atributos[i].nome = "pe"){
                        //se o nome do atributo for pe, ele deleta o atributo
                        // e cria outro atributo com estrutura diferente
                        classe.apagaAtributo("pe");
                        classe.adicionaAtributo(pe, int, private);
        }}
        // a variável metodos recebe todos os métodos da classe Comum
        Metodo []  metodos = classe.retornaMetodos ();
        for (int i = 0; i< metodos.tamanho(); i++) {
                if (metodos[i].nome == "desenhar"){
                        // o método que tiver o nome igual a desenhar recebe mais um parâmetro
                        // a sua chamada, do tipo inteiro e com  o nome de pe
                        metodos[i].adicionaParametro(int, pe);
                        // neste mesmo método, primeira linha, é adicionado novo código
                        metodos[i].adicionaCorpo ("this.pe = pe",0);
        }}
        //retorna a classe modificada para que a aplicação possa utilizá-la
        return classe;
}
```

Visualizando o diagrama da figura 2.6 pode-se verificar que o atributo *pe* era estático e possuía como valor o número 4 (4 pernas). Após a execução do código do quadro 2.1, o atributo passou a receber o valor informado pelo parâmetro do método. Além do método ter alterado seu comportamento, também foi modificada a estrutura da classe (figura 2.7).

2.7: Diagrama de classe do objeto transformado



Este tipo de situação é importante no momento em que se queira adaptar o sistema para determinadas tarefas, sem precisar refazer o modelo lógico, ou até mesmo adaptar em tempo de execução, sem alterar o código fonte. Também é importante para reduzir a complexidade, visto que funcionalidades diferentes estão separadas por níveis diferentes.

Outro exemplo, seria tornar todos os métodos da classe *Cadeira*, ou parte deles, públicos, ao invés de protegidos, para que se pudesse acessar diretamente a classe *Cadeira*, caso isso fosse necessário.

Ambos, meta interface e código, são gerados a partir de uma linguagem hipotética, criada para exemplificar alguns conceitos relatados neste trabalho.

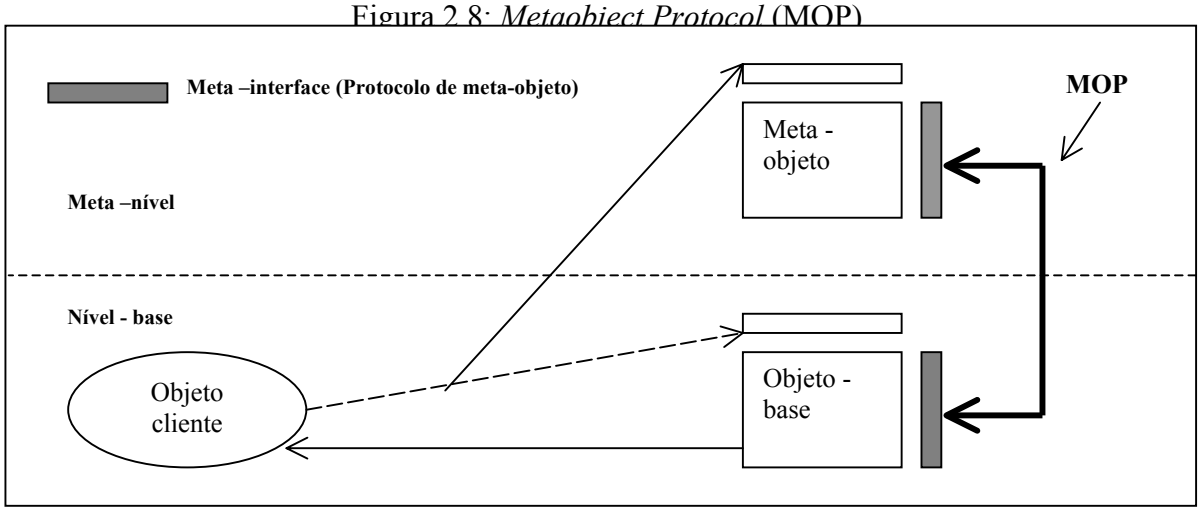
O programador continua utilizando principalmente a interface básica, mas ele também pode escrever programas separados usando a interface nova, ajustando a aplicação com as suas necessidades.

A interface básica fornece ao programador os métodos já implementados anteriormente, enquanto que a meta interface fornece informações sobre a estrutura da classe e seu comportamento, como aspectos de herança, nome da classe, tipo do objeto e propriedades. Estas informações estariam expostas na forma de métodos, como visto na tabela 2.1.

Este objeto aberto, além de proporcionar ao programador maior poder de adaptabilidade, flexibilidade e transparência pode também separar conceitos, reduzindo a complexidade da aplicação.

Se um modelo de implementação aberto for adicionado a uma arquitetura de meta níveis, pode-se utilizar a meta interface dos objetos abertos como interface entre o objeto base e o metaobjeto, formando o protocolo de metaobjetos. (figura 2.8).

Figura 2.8: *Metaobject Protocol* (MOP)



Segundo [KIC1991], o conjunto de métodos da meta interface formam o protocolo de metaobjetos, que pode ser dividido em:

a) protocolo de introspecção: permite examinar a estrutura da implementação inerente, através de uma interface apropriada. Um protocolo de metaobjetos introspectivo consiste de mecanismos que permitem obter informações sobre as classes, suas herdeiras, seus nomes e dados sobre a estrutura;

b) protocolo de invocação: fornece o acesso direto aos objetos e às operações sem precisar utilizar a interface básica do objeto, ou seja, é a capacidade que a metaclasse tem em invocar um método da sua classe de nível base de modo direto;

c) protocolo de intercessão: possibilita realizar mudanças de comportamento ou de estrutura do objeto base.

Continuando com a definição da linguagem hipotética, na tabela 2.2 pode-se visualizar uma listagem de métodos de introspecção, invocação e intercessão de uma meta interface.

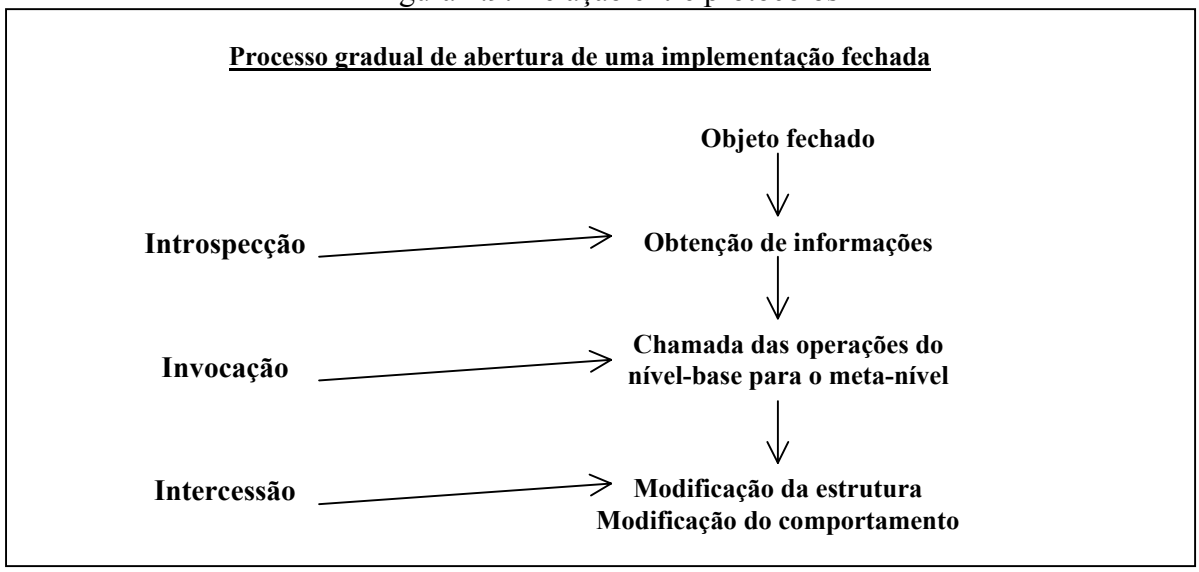
Tabela 2.2: Métodos e protocolos de uma linguagem estendida hipotética

Protocolo	Método
Introspecção	Classe obtemClasse (String classe)
Introspecção	Classe [ ] retornaSuperClasse ()
Introspecção	Classe [ ] retornaSubClasse ()
Introspecção	Metodo [ ] retornaMetodos ()
Introspecção	Atributo [ ] retornaAtributo ()
Invocação	realizaTraducao()
Invocação	associa (MetaClasse meta_classe, Classe classe)
Intercessão	void alteraEstadoAtributo (Atributo atributo)
Intercessão	void alteraNomeClasse ()

Intercessão	void adicionaCorpoMetodo (String codigo, int linha)
Intercessão	Classe criaClasse ()
Intercessão	void atribuiNomeClasse (String nome)
Intercessão	void atribuiSuperClasse (Classe classe)
Intercessão	void eliminaMetodo (Metodo metodo)

A relação entre os protocolos pertencentes ao MOP acontece de acordo com figura 2.9:

Figura 2.9: Relação entre protocolos



Fonte:[KIC1996]

A figura 2.9 pode ser associada aos estágios de um sistema reflexivo, onde a introspecção seria a forma de obter uma descrição abstrata do sistema, a invocação o método para utilizar esta descrição e realizar alguma manipulação e a intercessão o modo para modificar a descrição obtida.

A introspecção e a invocação fornecem o suporte para que aconteça a reificação ou materialização, que segundo [LIS1997], é o estágio que consiste na obtenção de uma descrição abstrata do nível-base tornando-a suficientemente concreta para o meta-nível, a fim de permitir operações sobre ela. Com base em [FER1989], [MAL1992] e [NAK1992], Lisboa define reificação do seguinte modo:

"*Reificação é a transformação de informações sobre a execução de um programa orientado a objetos em dados disponíveis ao próprio programa.*"

Os objetos reificados constituem as meta informações sobre as quais são realizadas as computações reflexivas. A escolha de qual objeto ou classe será reificado depende do objetivo da reflexão. De acordo com o objetivo pode-se determinar quais meta informações são

necessárias. Pode-se selecionar todos os objetos da aplicação, assim como apenas alguns: pelo nome, pelo método ou até mesmo pelo valor de determinado atributo.

Deve-se levar em consideração que esta divisão de protocolos é puramente conceitual, pois para que aconteça a reificação e reflexão é necessário a utilização de todos os protocolos de uma forma conjunta e muitas vezes difusa.

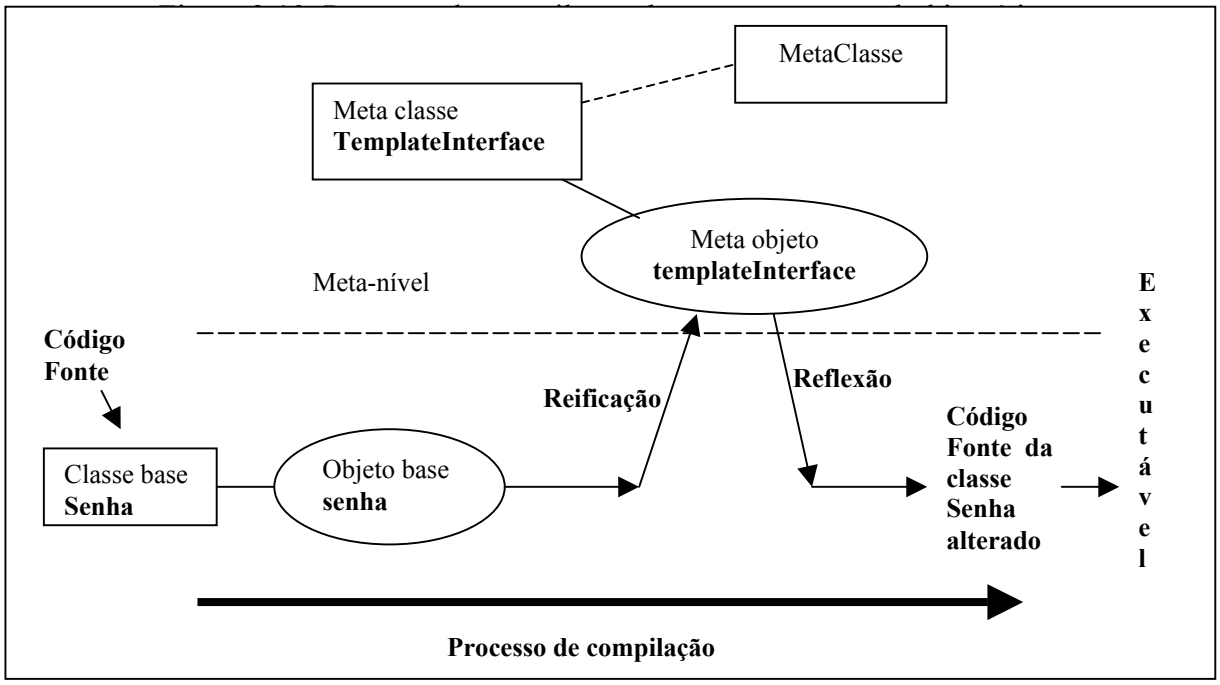
Com o objetivo de visualizar melhor o processo de reificação e reflexão, no quadro 2.2 é demonstrado a implementação de uma classe base e uma meta classe, que tem como objetivo se utilizar da reflexão computacional para construir *Templates*.

Quadro 2.2: Código fonte do exemplo de *Template* utilizando uma linguagem hipotética

```
//classe base
public class Senha ehInstancia templateInterface {
        String cor = "azul";
        String titulo = "Acesso ao sistema";
        String campos = ("Nome","Senha");
        String botoes =  ("Confirma");
        String conexao = "entradaDados";
}

// meta classe
public class templateInterface implementa MetaClasse {
        Metodo [] metodos;
        public templateInterface (Classe classeBase){
                metodos =  classeBase.retornaMetodos ();
        }
        public void realizaTraducao () {
                // definição de um novo comportamento para a classe base
                // de acordo com os atributos especificados na classe base
                // e a codificação existente neste método
}}
```

O programador de nível base precisa implementar uma interface tradicional para validação de usuário, ou seja, um campo com o nome do usuário, outro para a senha e um botão para confirmar. No código de nível base, implementado pelo programador de nível base, é necessário apenas adicionar algumas características da interface com os seus respectivos valores (classe Senha, quadro 2.2). Se o programador deseja implementar uma interface mais complexa, seria necessário apenas descrever um número maior de atributos para a classe base, pois a metaclasse está configurada de tal forma que o seu objetivo é ler todos os métodos da classe base, seu valor e de acordo com o número de atributos dimensionar o quadro com as coordenadas existentes no código da metaclasse.

Ao compilar este código fonte, ele passa pelo processo descrito na figura 2.10.



Feita a codificação da classe base, o programador pode então compilar tal classe. Ao iniciar o processo de compilação, o protocolo de metaobjetos hipotético cria um objeto para a classe base e um metaobjeto para a meta classe. Ambos objetos estão associados, devido ao programador ter determinado que a classe *Senha* teria como metaclasse a classe *TemplateInterface*. Ao acontecer a reificação, a estrutura e comportamento descrito na classe base irão se tornar dados manipuláveis na metaclasse, e logo após a sua manipulação, e possível alteração, serão refletidos na classe base, gerando outra codificação para a classe.

A reificação tem por objetivo revelar e tornar disponíveis ao meta-nível, e às vezes até mesmo ao programador da aplicação, informações a respeito dos componentes do programa que são conhecidas e tratadas normalmente apenas pelo compilador ou interpretador da linguagem de programação e pelo ambiente de execução.

As meta informações podem ser estáticas, quando se tratam de informações estruturais determinadas em tempo de compilação, ou dinâmicas, quando se tratam de informações sobre o processo de execução.

Durante o processo de reificação é que se deve escolher quais entidades devem ser transformadas em algo que possa sofrer operações no meta-nível, variando desde uma classe,

um conjunto de objetos, todo o sistema ou apenas um método de determinada classe. Isto vem a responder a primeira pergunta realizada anteriormente neste trabalho.

A segunda questão trata do relacionamento entre o nível base e o meta-nível: “como o relacionamento entre o nível base e o meta-nível é implementado ? ”. Para que este relacionamento possa ser implementado o metaobjeto deve possuir uma interface básica e uma meta interface. A interface básica oferece os mesmos serviços que o objeto base. Através desta interface o metaobjeto intercepta as mensagens, e pode ou não, transformá-las para então repassar a seu objeto base. Na meta interface, o metaobjeto pode comunicar-se com o seu objeto base assim como com outros metaobjetos, pois a sua meta interface faz parte do protocolo de metaobjetos (figura 2.8).

A terceira questão investiga sobre quem e como inicia o processo de reflexão. Segundo [MAE1987] existe duas soluções: a responsabilidade pode ser atribuída ao objeto de nível base (quadro 2.2), ou ser atribuída ao sistema (por exemplo, através do método associa (), executado em uma classe principal do sistema).

No caso da responsabilidade ser atribuída ao objeto de nível base, é necessário a existência de código no objeto base, mencionando seu metaobjeto. Caso contrário, se a responsabilidade for do sistema, quando ocorrer uma ação envolvendo o objeto base referente, o metaobjeto será ativado.

A realização da computação no meta-nível ocasionada pelo envio de uma mensagem a um objeto depende de informações dinâmicas obtidas durante o processo de execução, tais como:

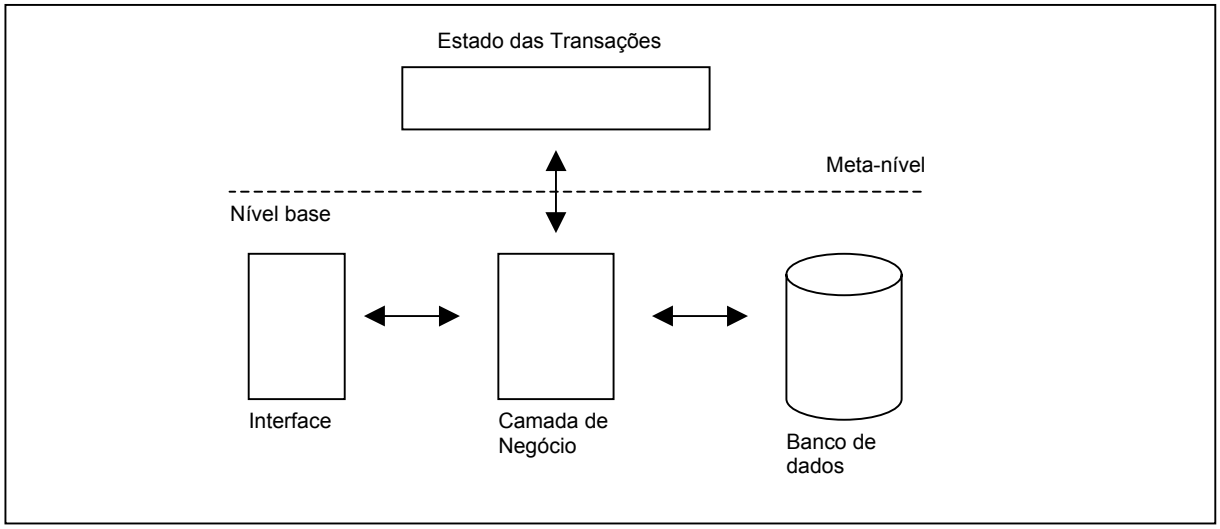
a) sobre o método: nome do método da aplicação destinatário da mensagem;

b) sobre a chamada: argumentos que o cliente forneceu na mensagem.

A reificação destes componentes possibilita que o metaobjeto reenvie a mensagem ao objeto da aplicação para a execução do método original, além de executar outras computações no meta-nível.

Outro exemplo da utilização de reflexão computacional, seria a implementação de uma aplicação composta por *n-camadas*, formada por uma camada de interface, de negócio e de acesso ao banco de dados, como descrito na figura 2.11.

Figura 2.11: Exemplo de aplicação de *n-camadas* reflexiva.



Com o objetivo de reduzir a probabilidade de falhas do sistema, a camada de negócio estaria associada a um meta-nível, que armazenaria, a cada transação, o seu estado. Caso houvesse alguma falha, uma exceção disparada no nível base, ou algo parecido, o meta-nível agiria sobre o nível base restaurando o seu contexto.

Este exemplo torna possível uma visualização bem clara da implementação de aspectos não funcionais utilizando reflexão computacional, pois no nível base é implementado todo o objetivo principal do sistema, ou seja, a sua funcionalidade, e no meta-nível é incorporado ao sistema a característica de persistência do estado dos objetos, que não é um objetivo principal do sistema, sendo, então, enquadrado como não funcionalidade do mesmo.

2.2.1. Protocolo de metaobjetos em tempo de compilação

Os protocolos de metaobjetos em tempo de compilação realizam o processo de reificação e reflexão em tempo de compilação, ou seja, a estrutura e comportamento das classes de nível inferior são modificados em tempo de compilação.

Segundo [CHI1996], as ferramentas que implementam o *Compile Time MOP* (protocolo de metaobjetos em tempo de compilação) são ferramentas que, além de sua biblioteca própria, possuem também um compilador próprio.

Além de possibilitarem a alteração de código fonte das classes de nível base, a maioria das ferramentas baseadas em *Compile Time MOP* geram classes adicionais que servem de suporte durante o processo de execução da aplicação.

Nos protocolos em tempo de compilação é comum que a responsabilidade de indicar as entidades reflexivas seja do nível base. Esta característica, aliada com a condição da reflexão que ocorre durante o processo de compilação do sistema, faz com que estes protocolos necessitem do código fonte das classes de nível base.

2.2.2. Protocolo de metaobjetos em tempo de execução

Os protocolos de metaobjetos em tempo de execução, como o próprio nome já diz, realizam a reificação e reflexão durante a execução do objeto. Ao contrário dos *Compile Time MOP's*, os *Run Time MOP's* não necessitam de compiladores, apenas do próprio protocolo que fornece a possibilidade de reflexão.

Durante a reflexão em tempo de execução, acontece a mudança de comportamento e estrutura do objeto sem alterar o seu código fonte. Diferentemente dos *Compile Time MOP's*, os *Run Time MOP's* não precisam do código fonte para realizar a reflexão. Geralmente a associação entre o objeto base e o metaobjeto é realizada pelo sistema, sem necessidade de implementar a chamada na classe base.

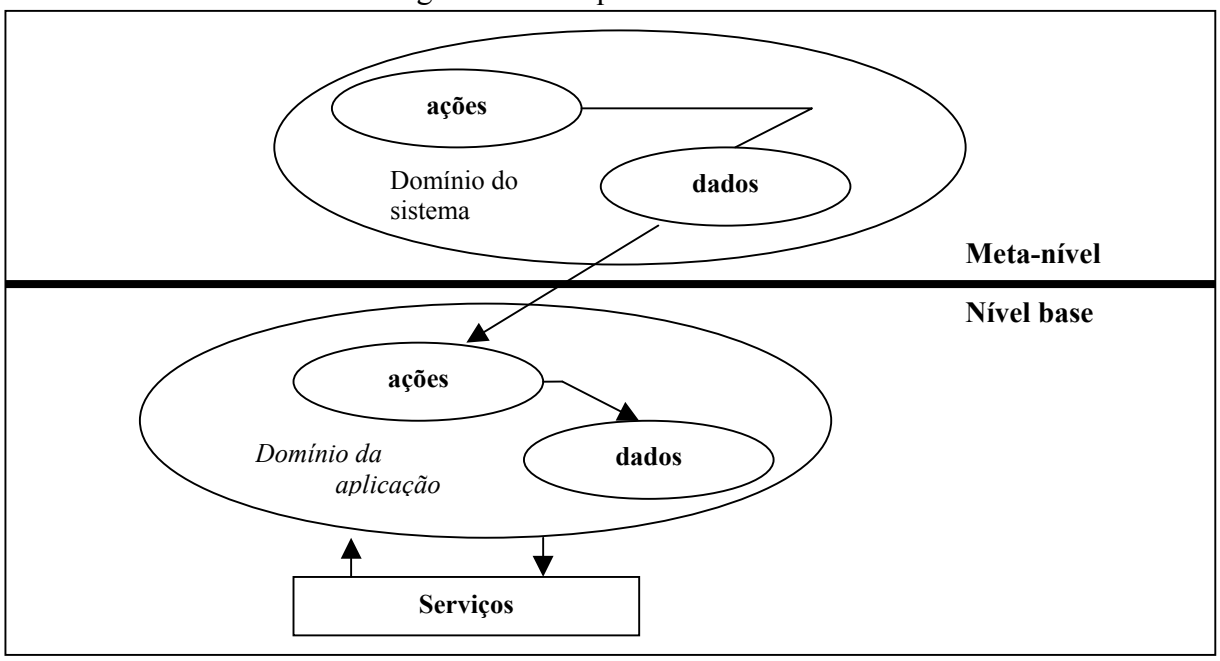
Maiores detalhes sobre protocolos de metaobjetos em tempo de compilação e execução podem ser encontradas em [CHI1996], [KLE1996] e [KIC1996].

2.3. Arquitetura Reflexiva

Segundo [BUZ1998], a reflexão computacional define uma arquitetura composta por meta níveis, onde se encontram as ações a serem realizadas sobre um sistema alvo localizado no nível base (ou da aplicação).

A figura 2.12 esquematiza a arquitetura reflexiva, mostrando que as ações no meta-nível são executadas sobre dados que representam a estrutura e comportamento do programa de nível base , enquanto que o programa de nível base executa ações sobre os seus próprios dados, com a finalidade de atender aos usuários de seus serviços.

Figura 2.12: Arquitetura reflexiva



Fonte: [LIS1997]

No nível base encontra-se as classes e objetos pertencentes ao sistema objeto. No meta-nível encontra-se as metaclasses e os metaobjetos.

A arquitetura reflexiva proporciona um novo tipo de abstração, uma abstração separada por níveis chamada de abstração reflexiva.

Segundo [LIS1997], a abstração reflexiva tem como vantagens:

a) redução de complexidade: no projeto do programa de nível base, o programador não precisa conhecer detalhes de implementação do ambiente de suporte, incluindo a linguagem de programação utilizada. Questões como – Como se implementa um objeto persistente ? – podem ter sua resposta delegada ao programa de meta-nível;

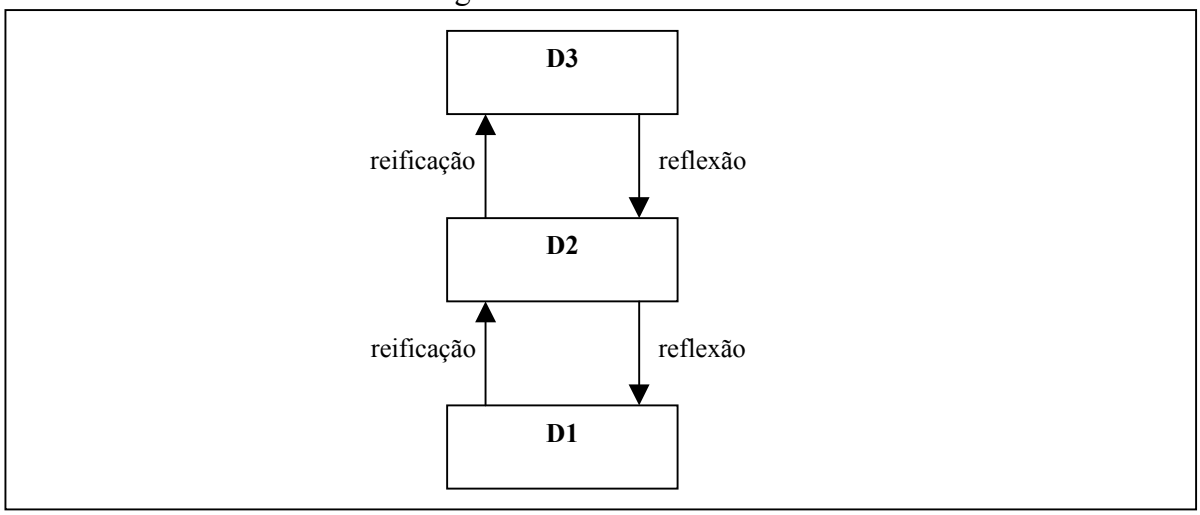
b) separação conceitual: significa que o programa de nível base somente trata com conceitos próprios do domínio da aplicação, relevantes à implementação das funcionalidades do programa. Os demais conceitos, a exemplo de requisitos não funcionais da aplicação, podem ser considerados como pertencentes ao domínio de meta-nível;

c) reutilização: considerando a separação de domínios, os componentes do programa são modelados como classes de objetos hierarquicamente separadas, permitindo a reutilização independente das classes do programa de nível base e do programa de meta-nível;

d) adaptabilidade: o programa de meta-nível pode estender e adaptar componentes do nível base, sem alterar a sua estrutura, reduzindo a possibilidade de introdução de falhas do programa de nível base.

A arquitetura reflexiva admite diversos meta níveis, caracterizando uma torre de reflexão. Cada nível da torre de reflexão constitui um domínio D*i*, considerado como domínio base do domínio D*i* + *1*, seu meta domínio situado no meta-nível imediatamente superior (figura 2.13).

Figura 2.13: Torre reflexiva



Na figura 2.13, pode-se visualizar uma arquitetura reflexiva composta por vários níveis, onde o domínio *D1* é considerado como sendo o nível base da aplicação, o domínio *D2* é considerado como o meta-nível do *D1* e o nível base do *D3*, e o domínio *D3* é o meta-nível do *D2*.

2.4. Reflexão estrutural

Segundo [FER1989], reflexão estrutural de uma classe **x** é toda a atividade realizada em uma metaclasse **Mcx**, com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe **x**.

Informações e transformações típicas de reflexão estrutural são:

a) obter informações sobre a classe **x**: sua ascendente, suas decendentes, suas instâncias, seus métodos e interfaces;

b) alterar a classe **x**: modificar seus atributos e seus métodos;

c) atuar sobre as classes: criando novas classes, eliminando classes existentes e renomeando classes.

A reflexão estrutural permite que o programa tenha a sua estrutura – tipo e número de componentes – alterada durante o processo de execução e/ou de compilação.

2.5. Reflexão comportamental

Por reflexão comportamental entende-se toda a computação que indiretamente atua sobre aspectos de comportamento de um objeto, mantendo inalterada a sua estrutura.

Segundo [MAE1987] reflexão comportamental de um objeto **O** é toda atividade realizada por um metaobjeto **Mox** com o objetivo de obter informações e realizar transformações sobre o comportamento do objeto **O**.

Deve-se notar que a reflexão comportamental, ao contrário da reflexão estrutural, não viola o encapsulamento do objeto, pois a reflexão estrutural pode substituir seus métodos por outros métodos, e mesmo alterar a classe de uma instância já existente.

2.6. Modelos de reflexão

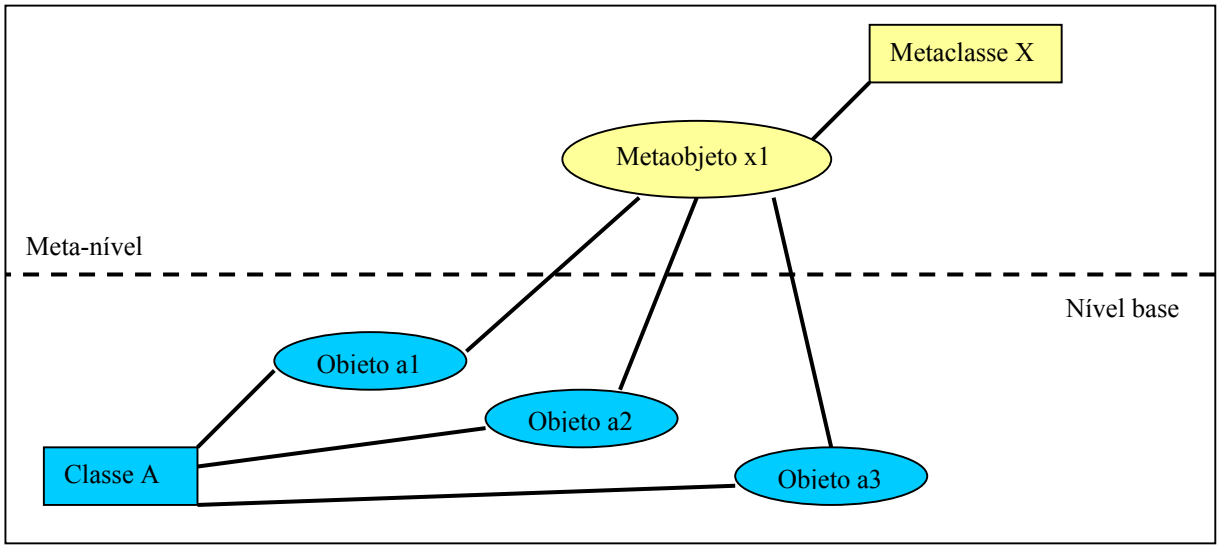
Segundo[BUZ1998] os modelos de reflexão computacional podem ser divididos em: modelo de metaclasses, modelo de metaobjetos e modelo de metacomunicações.

2.6.1. Modelo de metaclasses

O modelo de metaclasses é caracterizado pelo controle da reificação e reflexão ser realizado através de classes, ou seja, todo o objeto instanciado pela mesma classe terá o mesmo comportamento reflexivo de acordo com o que está modelado em sua metaclasse.

Na figura 2.14 pode-se visualizar uma ilustração que exemplifica o modelo de metaclasses.

Figura 2.14: Modelo de metaclasses



Este modelo perde em termos de flexibilidade, pois todos os objetos de uma classe são controlados por uma mesma metaclasse. No modelo de metaclasses, onde se tem acesso a estrutura de seus objetos, pode ocorrer tanto reflexão estrutural como comportamental. Por exemplo, ao executar algum método alternativo no lugar, ou além, do método original destinatário da mensagem, tem-se a reflexão comportamental. A reflexão estrutural ocorre apenas quando existe alteração de alguma propriedade descritiva na classe, tendo como conseqüência a alteração de todas as suas instâncias.

2.6.2. Modelo de metaobjetos

O modelo de metaobjetos se diferencia do modelo de metaclasses principalmente por sua associação por objetos e não por classes de objetos. Qualquer objeto da aplicação pode ser associado a um ou mais metaobjetos, que definem, implementam ou participam de diferentes formas da execução dos objetos de nível base. Segundo [LIS1997], sua principais características são:

a) individualidade: a cada objeto de nível base pode ser associado um metaobjeto. O objeto que for associado a uma metaobjeto é denominado de objeto reflexivo, podendo ser selecionadas apenas determinadas instâncias de classes para a realização de reflexão computacional, permanecendo não reflexivos os outros objetos instanciados a partir das mesmas classes;

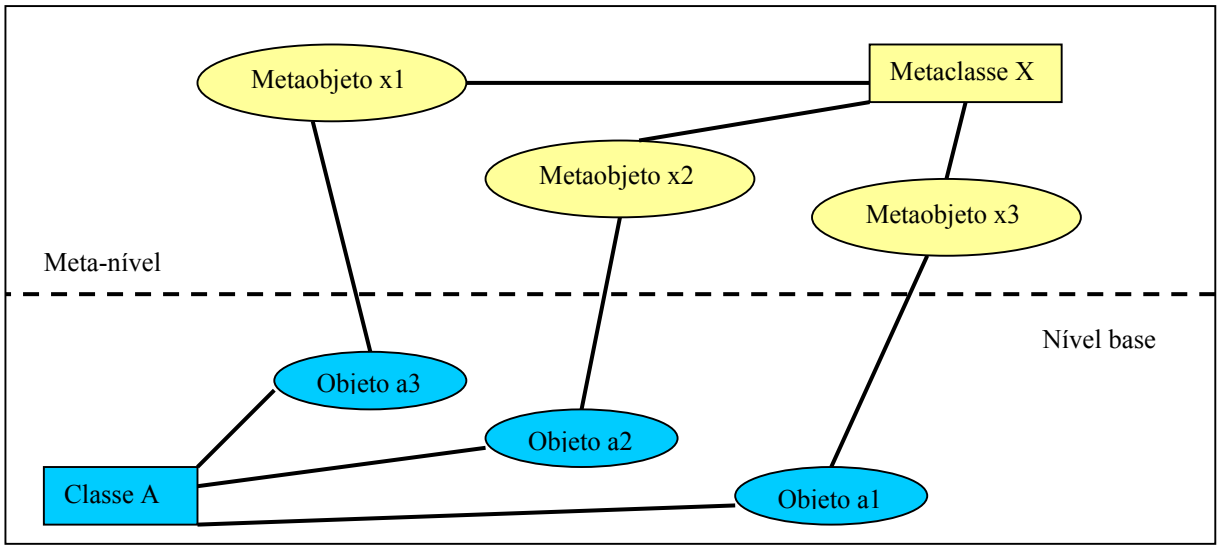
b) separação de classes: a classe do metaobjeto é distinta da classe de seu referente, permitindo que objetos de uma mesma classe sejam associados a diferentes metaobjetos; inversamente, metaobjetos instanciados a partir de uma mesma classe podem ser associados a objetos de nível base de diferentes classes;

c) associação dinâmica: a associação entre um metaobjeto e um objeto é feita em tempo de execução. Quando um objeto reflexivo é instanciado, ocorre também a instanciação de seu metaobjeto. Dependendo do protocolo de metaobjetos, é possível mudar o metaobjeto associado de forma dinâmica. Esta característica permite que um objeto assuma diferentes comportamentos ao longo do processo de execução, dependendo das classes dos metaobjetos sucessivamente associados a ele;

d) definição circular: um metaobjeto é um objeto, visto que é obtido por instanciação. Portanto, um metaobjeto pode ser tratado como um objeto comum e, inclusive, pode ter seu próprio metaobjeto. Esta característica permite a estruturação de uma torre de reflexão computacional.

Na figura 2.15 pode-se visualizar um diagrama que exemplifica o modelo de metaobjetos.

Figura 2.15: Modelo de metaobjetos

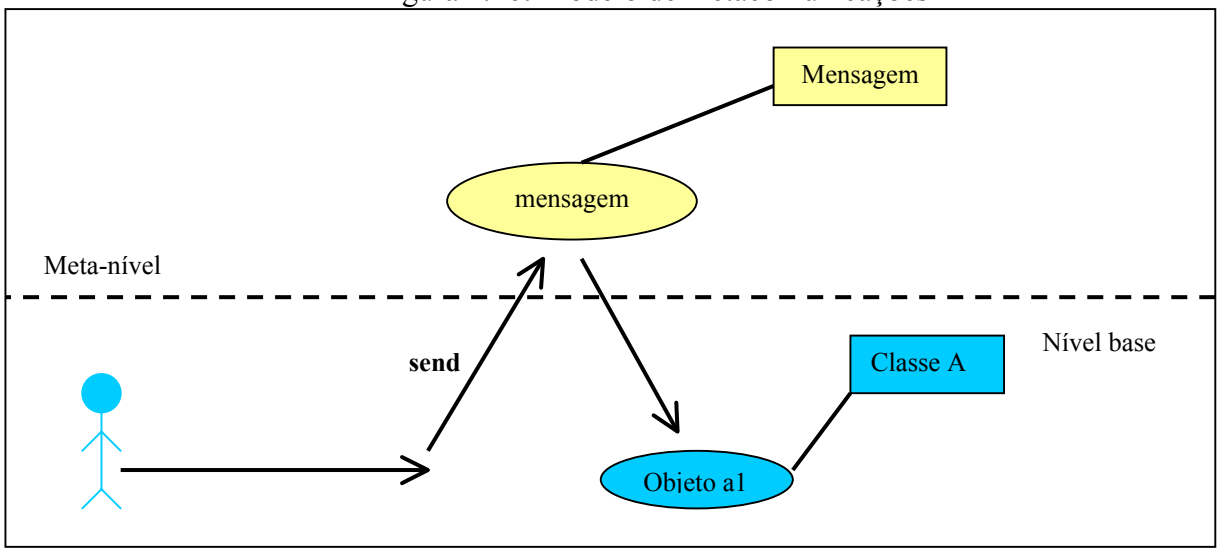


2.6.3. Modelo de metacomunicações

O modelo de metacomunicação acontece quando cada mensagem é materializada como um objeto, ou seja, existe uma classe **Mensagem** que representa as mensagens. Quando uma mensagem é enviada, uma instância da classe **Mensagem** é criada, e uma mensagem *send* é enviada para essa instância. A mensagem *send* possui todas as informações sobre a mensagem, como: a que objeto pertence, quais os seus parâmetros, qual o valor, quem está enviando, entre outros.

Na figura 2.16 pode-se visualizar o modelo de metacomunicações.

Figura 2.16: Modelo de metacomunicações



Ao contrário dos outros modelos, onde para que haja reflexão é necessário acionar uma classe ou objeto reflexivo, no modelo de metacomunicações toda a mensagem do sistema é reificada. Além disso, outra diferença está em onde e como é realizado o processamento da reflexão. Em um modelo de metaclasses este processamento é realizado em uma metaclasse correspondente a classe ou objeto acionado. No modelo de metaobjetos o processamento é realizado no metaobjeto respectivo ao objeto reificado. Já no modelo de metacomunicações o processamento é realizado em um metaobjeto instanciado pela classe **Mensagem** de acordo com os valores recebidos pela mensagem *send.*

Este modelo, apesar de ser o mais flexível, é também o mais ineficiente, sendo que toda mensagem será interceptada pelo meta-nível, mesmo que esta mensagem não necessite da adição de um comportamento reflexivo.

2.7. Linguagens e ambientes reflexivos

Os modelos apresentados anteriormente correspondem apenas a uma classificação que não impõe limites rígidos a implementação de sistemas reflexivos. A realização da reflexão comportamental e estrutural depende apenas do protocolo de metaobjetos que a linguagem ou ambiente oferece. Neste item serão descritos, superficialmente, algumas linguagens e ambientes reflexivos existentes. As definições e descrições encontradas neste item foram formuladas com base em [LIS1997], [BUZ1998], [CHI1995], [WU1996], [CHI1996a] e [TAT1999].

2.7.1. Smalltalk

Em Smalltalk todas as classes são instâncias da classe *Object*, que é a raiz da hierarquia de classes do sistema Smalltalk. A partir da classe *Object* são derivadas as demais classes e cada classe tem a sua estrutura descrita em uma metaclasse. Cada metaclasse é criada junto com a definição da classe e possui o mesmo nome. Todas as metaclasses são instâncias de uma classe chamada *Metaclass*.

Em Smalltalk todas as entidades são objetos, inclusive as classes, podendo-se agir sobre a estrutura das mesmas. Cada objeto encapsula o seu estado e cada metaclasse fornece a estrutura de sua respectiva classe. As metaclasses fornecem um conjunto de métodos que dão acesso a implementação, tornando-o um objeto aberto.

2.7.2. OpenC++

A linguagem C++, por definição, não oferece facilidades reflexivas. Para possibilitar que a linguagem possua capacidades reflexivas é necessário modificar a implementação da linguagem ou implementar extensões baseadas em pré-processadores.

O pré-processador OpenC++ permite associar, de forma estática, metaobjetos a objetos da aplicação. No nível da aplicação, o programador pode escolher quais classes devem ser reflexivas, e, para cada uma, quais métodos e atributos devem ser controlados pelo metaobjeto a ele associado. A linguagem adota o modelo de metaobjetos, pois cada objeto tem um único metaobjeto que pode controlar o acesso a atributos e chamada de funções, sendo que um metaobjeto não pode ser compartilhado por mais de um objeto. Todas as metaclasses descendem da classe MetaObj.

A computação no meta-nível é realizada por redefinição dos métodos adquiridos da classe MetaObj e pelos métodos particulares (definidos ou herdados) da classe do metaobjeto em questão.

O protocolo de metaobjetos de OpenC++ foi incorporado através de um pré-processador que implementa um mecanismo de interceptação de mensagens. Uma metaclasse é na verdade um classe C++ comum, existindo apenas uma referência para a classe a qual ela está associada. Isto implica que a metaclasse não tem acesso direto aos atributos definidos pela sua classe associada.

2.7.3. CLOS

Common LISP Object System (CLOS) estende a linguagem LISP para o modelo de objetos. Em CLOS, todas características de uma classe (atributos e métodos) são herdados pela subclasse (ao contrário de outras linguagens orientadas a objetos onde somente os atributos públicos e protegidos são herdados e os privados não). Além disso CLOS não proporciona o ocultamento de nenhum método de classe, todos os métodos são públicos.

Uma questão importante, no que se refere a linguagem CLOS como uma linguagem reflexiva, é o fato dela permitir a redefinição de classe dinamicamente e possuir um nível composto por um conjunto de metaclasses e metaobjetos. O protocolo de metaobjetos em CLOS permite introspecção de objetos. Além disso, o protocolo de metaobjetos de CLOS é muito poderoso permitindo também a redefinição do comportamento da própria linguagem.

Segundo [KIC1996], o meta protocolo de CLOS proporciona um suporte completo para reflexão computacional.

2.7.4. Java

Na máquina virtual Java padrão, todas as classes tem como raiz a classe Object, pré-definida pela linguagem (assim como CLOS e Smalltalk). Java dá suporte para metainformação, permitindo introspecção de classes e objetos (como em CLOS e Smalltalk) através da *package java.lang.reflect*. Entretanto, a maquina virtual Java padrão não dá nenhum apoio direto para protocolos de metaobjetos.

Java permite a reflexão estrutural, mas não a comportamental. Através da *Java Reflection API* (*java.lang.reflect*) é possível: construir e instanciar novas classes e acessar e modificar métodos e atributos em tempo de execução.

A *Reflection API* é muito útil para aplicações que necessitam obter informações sobre classes e seus membros e usar estas informações em tempo de execução para localizar determinado objeto ou reutilizá-lo, como *JavaBeans* e aplicações distribuídas.

A fim de exemplificar a utilização da *package java.lang.reflect*, no quadro 2.3 está descrita uma classe que tem como objetivo gerar um arquivo com informações sobre os métodos e atributos de uma classe já compilada, a qual o programador não possui o seu respectivo código.

Quadro 2.3: Código fonte da classe *ObtemInformacoes* em Java

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

public class ObtemInformacoes {
    private Class classe; private Constructor [] construtor;
    private Method [] metodos; private Field [] atributos;

    public ObtemInformacoes (String nome) {
        try{
            classe = classe.forName(nome);
            construtor = classe.getDeclaredConstructors ();
            metodos = classe.getDeclaredMethods ();
            atributos = classe.getDeclaredFields ();

            File resultado = new File (nome + ".java");
            RandomAccessFile a_result = new RandomAccessFile(resultado, "rw");

            a_result.writeBytes("public class "+nome+" {\n");
            a_result.writeBytes("//Atributos\n");
            for (int i=0; i < atributos.length; i++) a_result.writeBytes(atributos[i].toString()+"\n");
            a_result.writeBytes("//Metodo construtor\n ");
            for (int i=0; i < construtor.length; i++) a_result.writeBytes(construtor[i].toString()+"{\n");
            a_result.writeBytes("//Metodos da classe "+nome+"\n");
            for (int i=0; i < metodos.length; i++) a_result.writeBytes(metodos[i].toString()+"{\n");
            a_result.close();
        }catch (Exception e){ System.out.println("Erro: "+e);}
    }
    public static void main (String args[]){ new ObtemInformacoes (args[0]); }
}
```

Ao iniciar o processo de execução, o usuário, através de parâmetro, informa o nome da classe que deseja obter tais informações, por exemplo: *java ObtemInformacoes Server*. Neste exemplo o usuário está executando a classe *ObtemInformacoes*, passando como parâmetro de entrada a classe *Server*, que retornará como saída o arquivo *Server.java*.

São poucas as linguagens de programação que oferecem bons mecanismos de reflexão. Segundo [LIS1997], [WU1996] e [TAT1999], uma nova possibilidade está sendo a linguagem Java, devido às suas características básicas e aos trabalhos desenvolvidos com o objetivo de estender a linguagem para que suporte os conceitos reflexivos.

Devido à linguagem Java não implementar os conceitos reflexivos de forma completa, estão sendo realizados alguns trabalhos com o objetivo de estender a linguagem Java para tal capacidade. No próximo capítulo serão vistos duas ferramentas que estendem a linguagem Java.

3. Mecanismos de extensão reflexivos para a linguagem Java

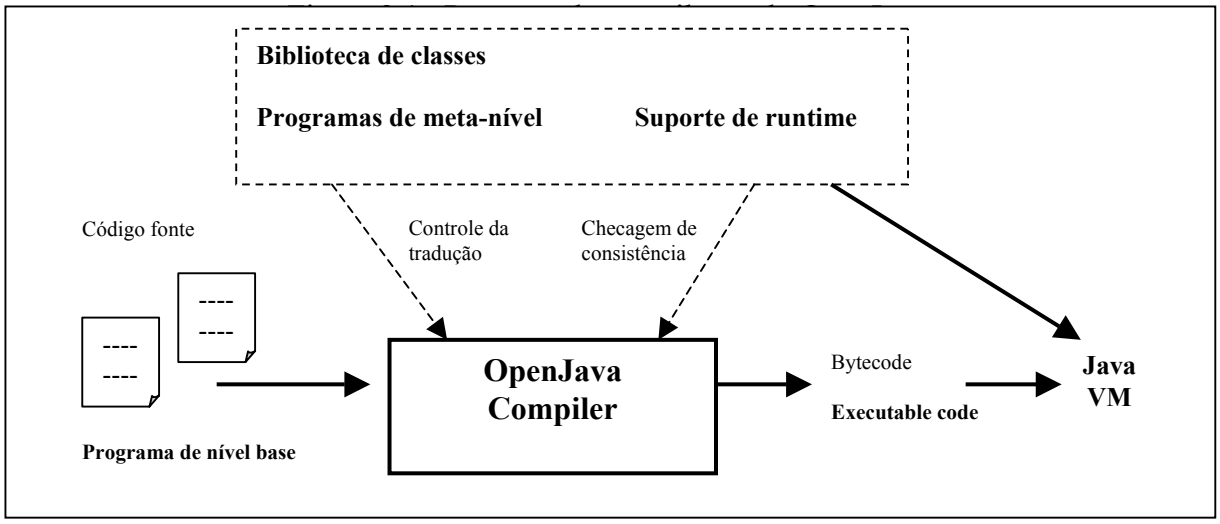
Neste capítulo serão apresentadas duas extensões da linguagem Java que possibilitam sua utilização em sistemas reflexivos. Ambas as ferramentas proporcionam reflexão estrutural e comportamental e possuem um meta protocolo. A diferença principal entre as duas ferramentas, é que uma proporciona reflexão em tempo de compilação e a outra em tempo de execução.

3.1. OpenJava

Segundo [TAT1999], 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++, pois ambas foram modeladas e implementadas seguindo o mesmo conceito.

Semelhante a outras linguagens baseadas em meta protocolos em tempo de compilação, o compilador do OpenJava, *ojc*, aceita programas de fontes escritos por programadores e gera *bytecodes*. A diferença do compilador regular Java para com o compilador do OpenJava, é que o compilador do OpenJava recorre a bibliotecas de meta-nível além de bibliotecas regulares.

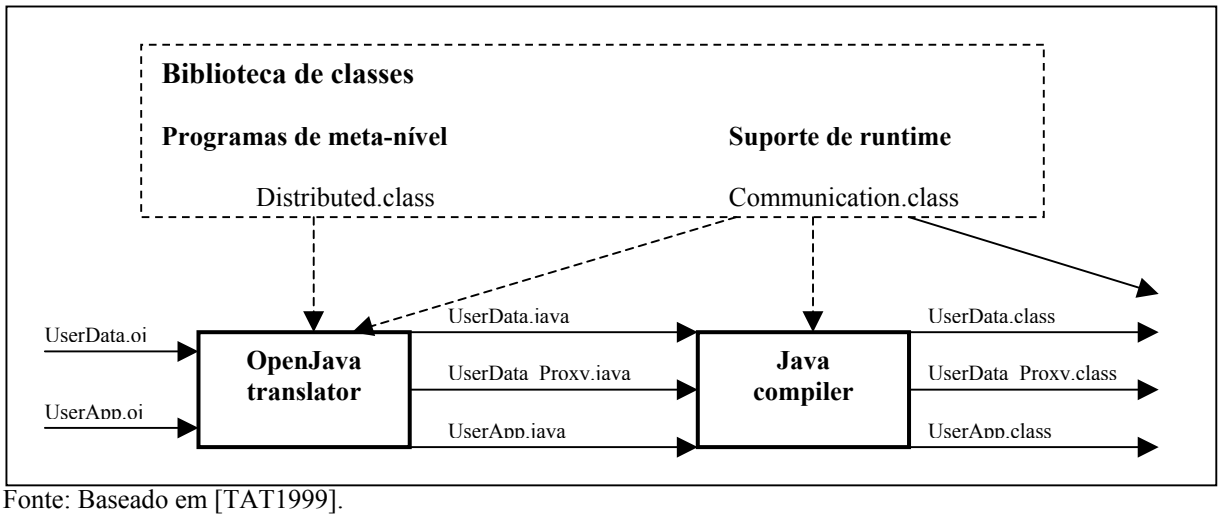
Uma avaliação do processo de compilação do OpenJava é visto na figura 3.1.



Fonte: Baseada em [TAT1999]

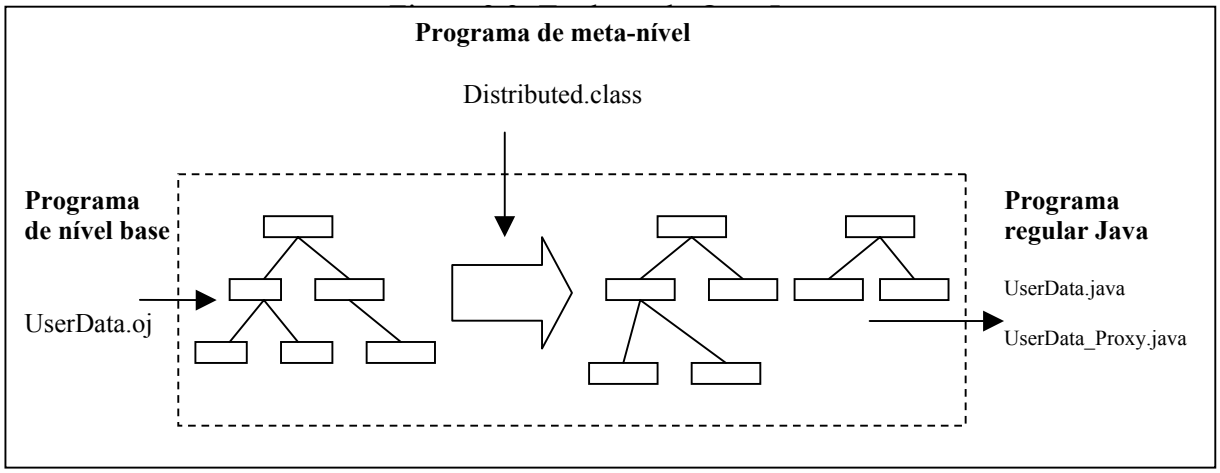
Interiormente, o compilador do OpenJava possui 2 módulos principais, um tradutor e um compilador de Java regular. O tradutor gera programas escritos em Java regular e então o compilador gera código de *bytecode* regular de acordo com o código fonte gerado pelo tradutor (figura 3.2).

Figura 3.2: Módulos do compilador OpenJava



Fonte: Baseado em [TAT1999].

Como pesquisa, o módulo mais importante deste sistema é o tradutor. Primeiro, o tradutor leva o programa fonte escrito no OpenJava (linguagem extendida) para o programa de meta-nível especificado naquele programa fonte e gera uma *Abstract Syntax Tree* (AST), árvore de sintaxe abstrata. Então transforma o AST de acordo com o programa de meta-nível. Finalmente, gera código fonte na linguagem Java regular do AST transformado. A figura 3.3 mostra este fluxo.



Fonte: Baseado em [TAT1999]

Segundo [TAT1999], optou-se realizar a implementação do OpenJava como sendo uma implementação de um *Compile Time MOP* (protocolo de meta objeto em tempo de compilação), ao invés de um *Run Time MOP* (protocolo de meta objetos em tempo de execução), devido à eficiência durante a execução do programa. Alguns autores alegam que o desempenho de um *Run Time MOP* é menor do que um *Compile Time MOP*, devido ao *overhead* em tempo de execução.

No quadro 3.1 é apresentado o código fonte de uma classe do nível base (*ClasseBase*) que está associada a uma classe do meta-nível (*MetaClasse*).

Quadro 3.1: Código fonte de um programa implementado em OpenJava

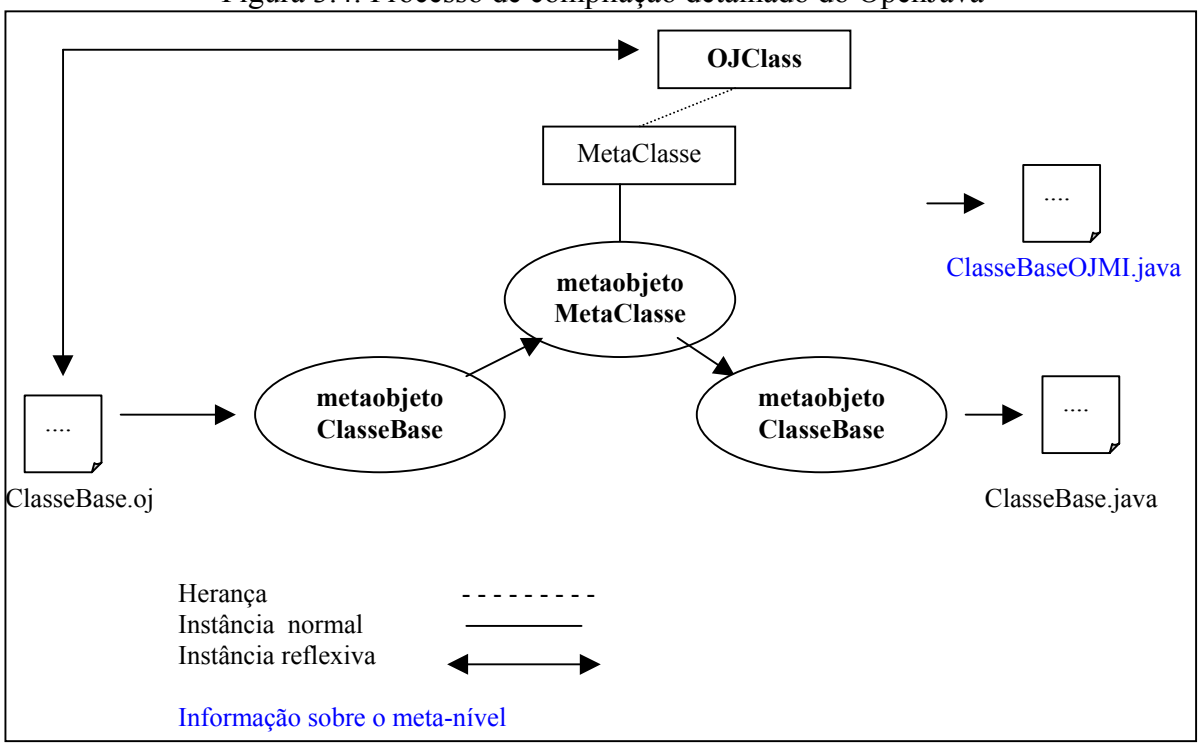
```
public class ClasseBase instantiates MetaClasse {
        public static void main (String[] args) {
                hello ();
        }
        static void hello () {
                System.out.println ("Nível base...");
}}

import openjava.mop.*;
import openjava.ptree.*;
public class MetaClasse instantiates Metaclass extends OJClass {
        public void translateDefinition () throws MOPException {
                OJMethod [] methods = geteclaredMethods ();
                for (int i = 0; i < methods.length; ++i) {
                        Statement printer = makeStatement ("System.out.println(\"Este é o método "+
                        methods[i]+"\" );");
                        Methods[i].getBody().insertElementAt (printer, 0);
}}}
```

A notação *ClasseBase instantiates MetaClasse* quer dizer que a classe *ClasseBase,* de nível base, é uma instância e está associada à classe *MetaClasse*, que é uma sub-classe da classe *OJClass*. Como resultado, toda a reflexão ao redor do objeto da classe *ClasseBase* será executada de acordo com a definição implementada na metaclasse *MetaClasse*.

A anotação *MetaClasse instantiates Metaclass extends OJClass*, especifica que a MetaClasse é herdeira da classe *openjava.mop.OJClass* e está associada à classe *openjava.mop.Metaclass*. Todas as metaclasses são herdeiras da classe *OJClass*, como no modelo do OpenC++ onde todas as metaclasses estão associadas à classe *MetaObj*. A classe *OJClass* fornece os métodos para acesso à implementação da classe base. Quando uma classe é herdeira da classe *OJClass* ela se torna uma metaclasse que sobrepõe os métodos da classe *OJClass*, segundo suas necessidades. Os programadores de meta-nível se utilizam de metaclasses herdeiras da classe *OJClass* para declarar um comportamento reflexivo específico para as classes base que irão sofrer reflexão através da determinada metaclasse.

A classe *openjava.mop.Metaclass* fornece a capacidade de, através do que está definido na metaclasse, poder gerar código no nível base.

Figura 3.4: Processo de compilação detalhado do OpenJava



Fonte: Baseado em [TAT1999].

Durante o processo de compilação, o OpenJava cria, para todas as classes associadas no processo, tanto em nível base como em meta-nível, um metaobjeto respectivo para cada

classe. São os metaobjetos que realizam o controle da reflexão e geração de código das novas classes durante o processo de compilação (figura 3.4).

No decorrer do processo de compilação, além de todo o processo de reificação e reflexão das classes pertencentes ao processo, também existe a geração de uma classe auxiliar que tem como objetivo guardar informações sobre o meta-nível. Como pode ser visto na figura 3.4, após a geração do arquivo *ClasseBase.java*, também foi criado o arquivo *ClasseBaseOJMI.java*, que guarda informações sobre o meta-nível correspondente à classe *ClasseBase*.

3.1.1. OpenJava API

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.

Os métodos para realizar a introspecção, modificação estrutural e modificação comportamental das classes, estão descritos na API do OpenJava que pode ser encontrada em [TAT2000].

A versão mais atual do OpenJava é a versão 1.0, com data de publicação na Internet do dia 20 de janeiro de 2000. Documentação sobre o OpenJava e a própria ferramenta podem ser encontrados em [TAT2000].

3.2. Javassist

Segundo [CHI1998], Javassist é uma ferramenta que visa facilitar o desenvolvimento de aplicações que utilizam a linguagem Java. Permite que programadores possam automatizar alguns tipos de definições de classe e permite a realização de reflexão computacional em tempo de execução.

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.

Neste trabalho será abordado apenas a capacidade reflexiva do Javassist, ou seja, apenas será utilizado o pacote javassist.reflect. O pacote javassist.reflect é composto pelas entidades descritas no quadro 3.2.

Quadro 3.2: Pacote javassist.reflect

Interface	
Metalevel	Interface que fornece o metaobjeto e a metaclasse do objeto respectivo
Classes	
ClassMetaobject	Corresponde a Metaclasse do modelo
Compiler	Possibilita realizar reflexão em *bytecodes*
Implement	Implementa mecanismos de reflexão
Metaobject	Corresponde ao metaobjeto do modelo
ReflectLoader	Permite a seleção dos objetos reflexivos
ReflectServer	Implementa uma classe reflexiva para aplicações distribuídas
Sample	Fornece um template para uma classe reflexiva
Exceções	
CannotCreateException	Lança uma exceção quando não é possível instanciar um metaobjeto
CannotInvokeException	Lança uma exceção quando não é possível invocar um método

Fonte: Baseado em [CHI2000]

No quadro 3.3 pode-se visualizar um trecho de código fonte de uma aplicação reflexiva utilizando a API do Javassist.

Neste exemplo a responsabilidade de definir o objeto reflexivo é do sistema, definido na classe *Main* através do método *makeReflective*, pertencente à classe *ReflectLoader*. Na classe *Main* define-se a classe que sofrerá reflexão e seu metaobjeto respectivo. A ferramenta Javassist adota, como modelo reflexivo, o modelo de metaobjetos. O metaobjeto cria um

invólucro sobre o seu objeto de nível base respectivo, interceptando todas as mensagens direcionadas ao objeto de nível base.

No exemplo do quadro 3.3, pode-se realizar duas formas de execução distintas: (1) através da linha de código: *java ClasseBase*, que não fornece a possibilidade de reflexão da classe base, pois apenas executa a classe de nível base sem mencionar a existência de uma relação com o meta-nível e (2) *java javassist.Loader Main*, que inicializa a classe Main fazendo com que a classe *ClasseBase* possa sofrer reflexão.

Quadro 3.3: Código fonte de uma aplicação reflexiva utilizando Javassist

```
// Classe principal do sistema
import javassist.Loader; import javassist.Reflect.ClassMetaobject;
import javassist.Reflect.ReflectLoader;
public class Main{
        public static void main (String[] args) throws Throwable {
                Loader c1 = (Loader)Main.class.getClassLoader();
                ReflectLoader loader = new ReflectLoader();
                loader.makeReflective ("ClasseBase",MetaClasse.class,Metaobject.class);
                c1.addUserLoader (loader);
                c1.run ("ClasseBase",args);
}}

// Metaclasse
import javassist.*; import javassist.reflect.*;
public class MetaClasse extends Metaobject {
        public MetaClasse (Object classe, Object[] args) {
                super (classe, args);
                System.out.println ("Metaobjeto instanciado: "+classe.getClass().getName());
        }
        public Object trapMethodCall (int identifier, Object[] args) {
                System.out.println (" Alterando o método: "+ getMethodName(identifier) +" em
                "+getClassMetaobject().getName());
                return super.trapMethodCall (identifier, args);
}}

// Classe base
public class ClasseBase {
        public static void main (.......) { .....}
        public ClasseBase () { .... }
}
```

Na metaclasse codifica-se os métodos responsáveis pela reflexão. Todas as metaclasses são herdeiras da classe *Metaobject*.

Javassist está baseada em experiências sobre o OpenC++ e OpenJava. Porém, Javassist não é um sistema reflexivo em tempo de compilação, nem gera código fonte e não utiliza um AST. Em princípio, não necessita de arquivos fontes para prover reflexão computacional.

Segundo [CHI1998], Javassist não é uma ferramenta reflexiva tão completa quanto as ferramentas MetaXa [GOL1998], [KLE1996] e Dalang [WEL1998], mas devido à sua API simples, ao modelo que propõe e a não utilização de qualquer compilador ou ferramenta para suporte em tempo de execução, Javassist é considerada como sendo uma ferramenta reflexiva de fácil utilização.

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].

4. Gerenciador de arquivos distribuídos

Neste capítulo será apresentado o gerenciador de arquivos distribuídos implementado em [POS2000].

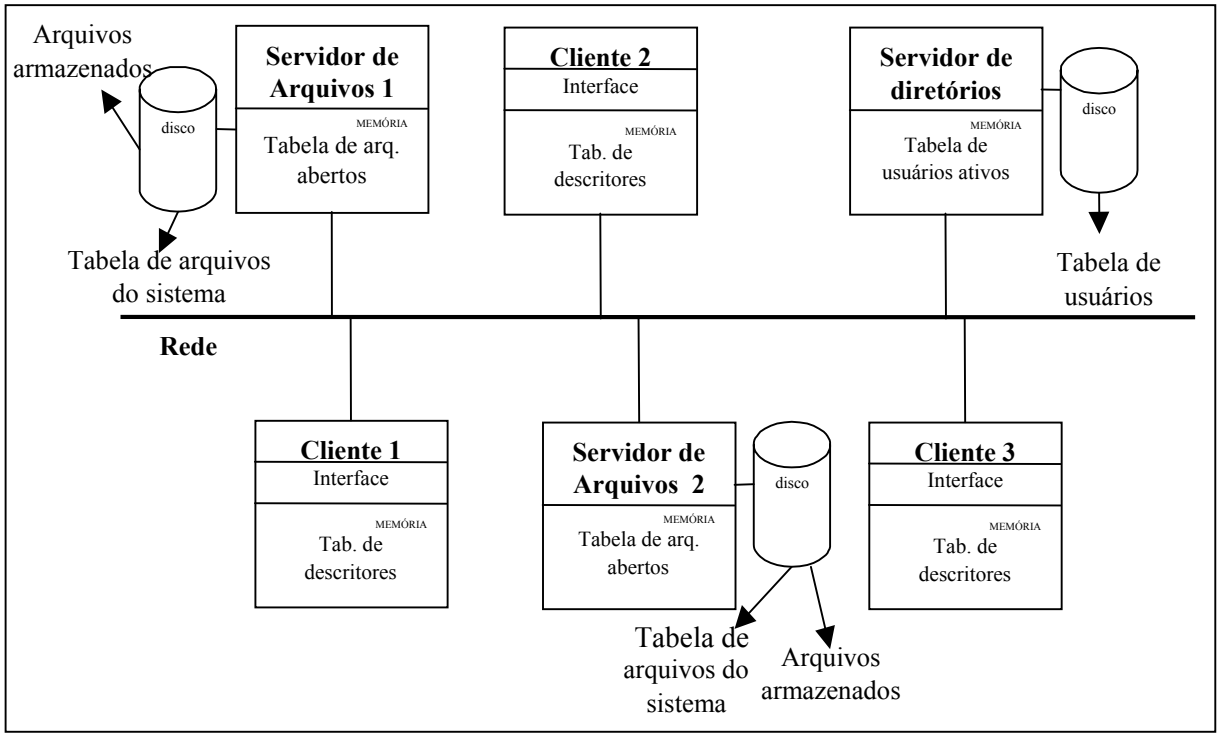
Segundo [POS2000], este gerenciador possibilita que o usuário acesse localmente ou remotamente os arquivos de forma transparente. A transparência de localização garante que o usuário não necessite se preocupar com a localização física do arquivo.

O gerenciador de arquivos tem uma estrutura do tipo cliente/servidor, podendo os clientes e servidores estarem na mesma rede local ou em redes de longa distância.

O gerenciador é constituído por três módulos: um cliente que roda em todas as estações de trabalho, um servidor de arquivos, que pode rodar em uma ou mais máquinas do sistema, e um servidor de diretórios, que é executado em uma das máquinas do sistema.

Na figura 4.1 pode-se visualizar a estrutura do gerenciador.

Figura 4.1: Estrutura geral do gerenciador de arquivos distribuídos



Fonte : [POS2000]

Os arquivos ficam armazenados nos servidores de arquivos, em qualquer ponto da rede e são acessados através da implementação de algumas primitivas que permitem criar, abrir, escrever ou inserir dados em arquivos, bem como fechar e excluir.

Na estrutura geral de arquivos, mantida pelos servidores de arquivos, ficam diversas informações como proprietário do arquivo, nome do arquivo, etc. Entre estas informações, um campo define as permissões para acesso ao arquivo, conforme o método *Unix* de proteção, utilizando os bits de proteção *rwx*, para o proprietário do arquivo, para seu grupo e para os outros usuários.

A manipulação de diretórios também é possível e é controlada através de um servidor de diretórios. Este servidor mantém uma estrutura que armazena o diretório de cada usuário, criando uma estrutura de diretórios em árvore. A estrutura de diretórios possui um diretório raiz chamado "/". A partir deste diretório, estão os diretórios de cada usuário e estes podem conter subdiretórios. O servidor de diretórios faz a localização de arquivos solicitados através das chamadas de sistema, retornando a localização do arquivo requerido. Algumas primitivas foram definidas para criar, excluir, listar e mudar de diretório.

A estrutura que o servidor de diretórios mantém, com informações de cada usuário que está conectado, possui também um campo denominado diretório corrente. Quando um usuário acessa o gerenciador de arquivos, por padrão, o diretório corrente é o */user*, sendo que *user* é o seu diretório pessoal. A partir do momento que ele utiliza o comando de mudança de diretório, este campo é atualizado para o diretório especificado.

Para facilitar a organização lógica dos dados, o protótipo implementa a estrutura de diretórios em árvore. Cada usuário tem um diretório pessoal e pode criar outros diretórios a partir deste, formando uma hierarquia em árvore.

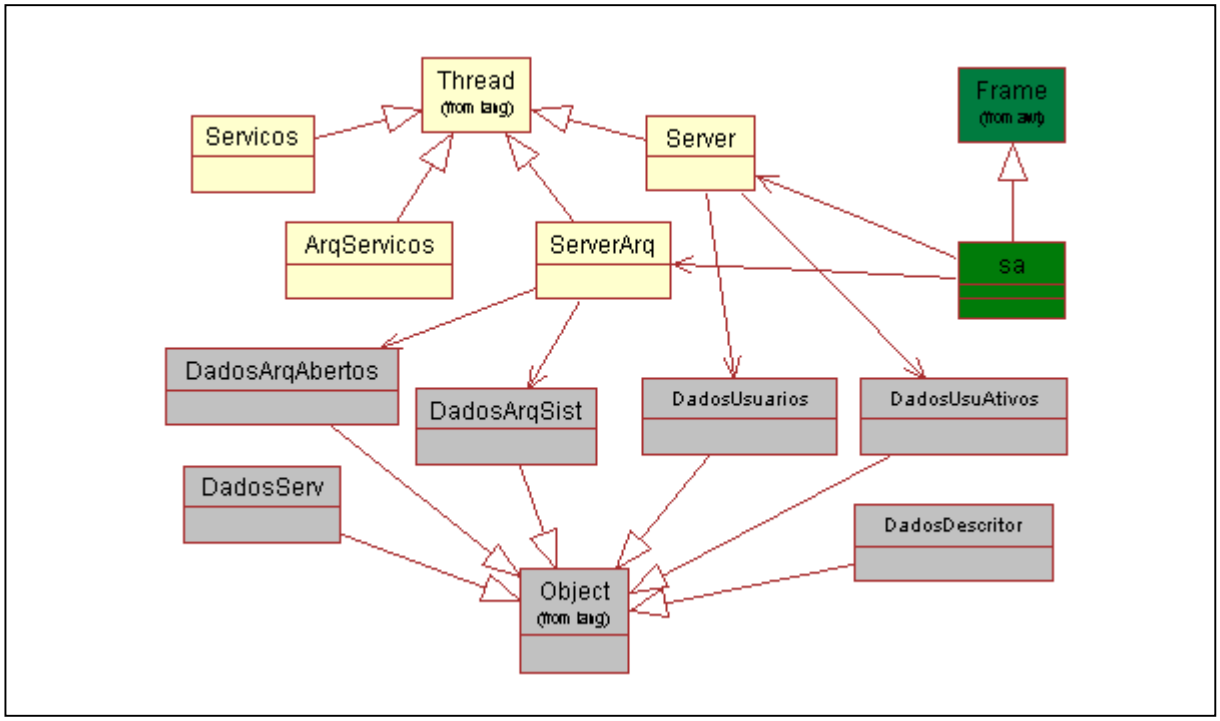
Ao especificar os nomes dos arquivos, o protótipo permite que seja utilizada a denominação absoluta ou relativa, já que o sistema tem como base um diretório raiz e também implementa o conceito de diretório corrente.

Como não é utilizado *cache* de blocos de dados, a consistência de informações é mantida através de controles com informações a respeito apenas de localização do arquivo, através de arquivos, tanto do cliente como do servidor. Atualizações em arquivos são feitas diretamente no servidor e, no momento da modificação ou inclusão de novas informações, o arquivo fica bloqueado para os demais usuários.

Para efeitos de performance, o módulo cliente mantém uma estrutura, que contém informações sobre os arquivos abertos. Essas informações apontam para uma entrada na tabela de arquivos abertos do servidor de arquivos. Uma operação realizada sobre um arquivo aberto não necessita mais do procedimento de localização. Para realizar operações de leitura, escrita e inserção no arquivo é necessário primeiramente abri-lo.

O gerenciador é implementado utilizando o modelo orientado a objetos e a linguagem Java. Na figura 4.2 é possível visualizar o diagrama de classes do gerenciador de arquivos, de acordo com a *Unified Modeling Language* (UML).

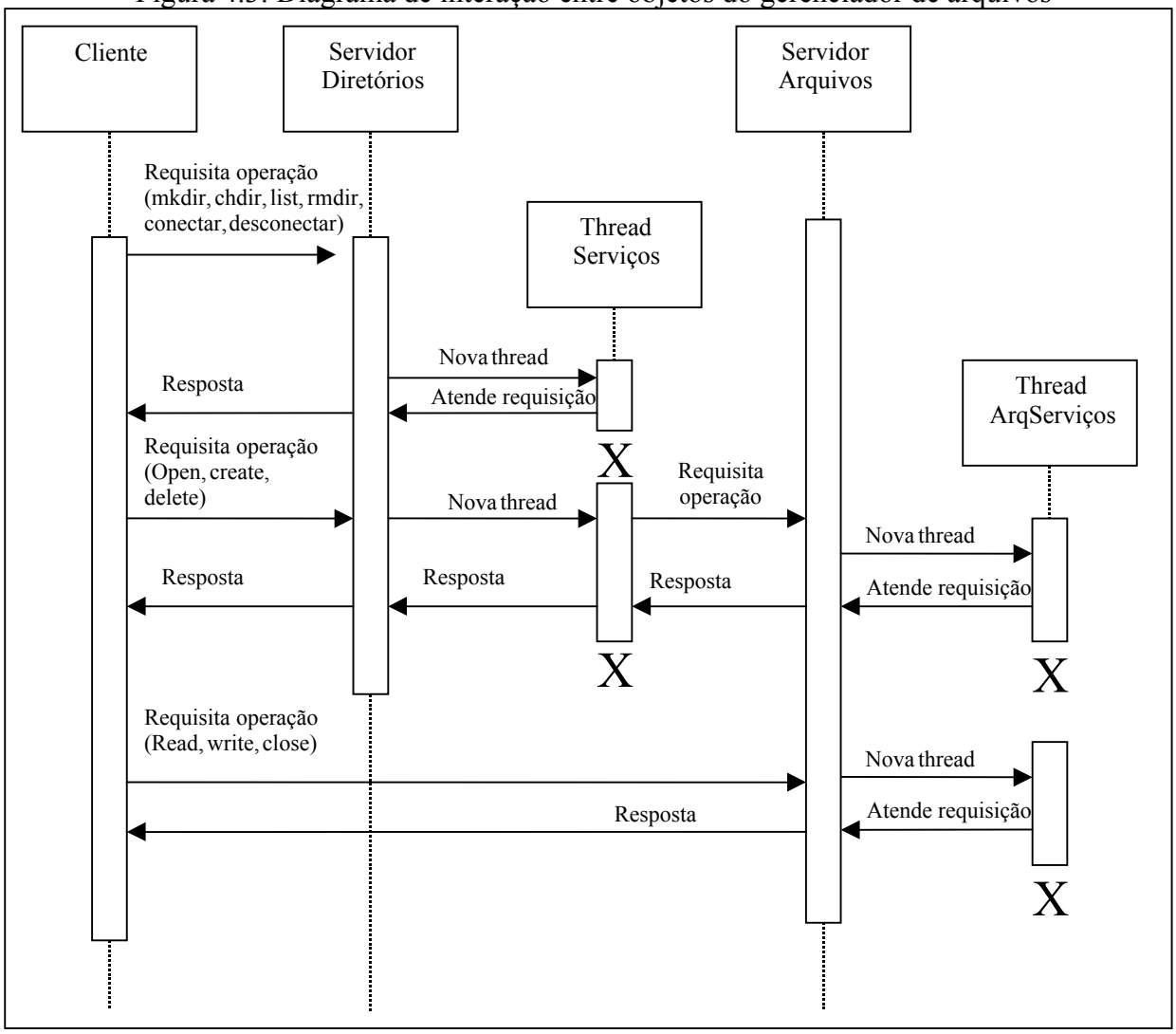
Figura 4.2: Diagrama de classes do gerenciador de arquivos distribuídos



No diagrama da figura 4.2, pode-se visualizar as classes *DadosUsuarios*, *DadosArqSist*, *DadosUsuAtivos*, *DadosDescritor, DadosArqAbertos* e *DadosServ* (cinza) herdeiras da classe *Object* do Java, que possuem informações sobre os usuários pertencentes ao sistema, sobre os usuários conectados, sobre os arquivos e sobre os diretórios. As classes *ServerArq, Server, ArqServicos* e *Servicos* (amarelo), herdeiras da classe *Thread* do Java, são, respectivamente, o servidor de arquivos, o servidor de diretórios, a classe responsável por executar as primitivas relacionadas a arquivos e a classe responsável por executar as primitivas relacionadas a diretórios. Também o diagrama apresenta a classe *sa* (verde) herdeira da classe *Frame* do Java, que realiza o papel de interface com o usuário.

Na figura 4.3 pode-se visualizar o diagrama de interação entre objetos do gerenciador de arquivos.

Figura 4.3: Diagrama de interação entre objetos do gerenciador de arquivos



Fonte: [POS2000]

Como primitivas básicas do gerenciador de arquivos, estão: criar, remover, mudar e listar o conteúdo de diretórios, criar, deletar, ler, escrever, fechar e abrir arquivos, além de conectar e desconectar usuários.

Como semântica de compartilhamento, os arquivos abertos para escrita são bloqueados para outros usuários e, para abrir um arquivo para leitura, é feita uma verificação para saber se o arquivo já está aberto para escrita.

Para cada primitiva desta existe um caso de uso especificando a sua funcionalidade, e, correspondente ao caso de uso, um diagrama de interação entre objetos. Para obter maiores detalhes sobre o funcionamento destas primitivas, tem-se como referência [POS2000].

Na versão atual do gerenciador de arquivos [POS2000], o comportamento para escolha de qual servidor de arquivos será utilizado para armazenar determinado arquivo criado é realizado de forma que, se um arquivo foi armazenado no *servidor1*, o próximo arquivo a ser criado será armazenado no *servidor2*, trocando de servidores a cada requisição de criação de um arquivo.

Quanto ao servidor de diretórios, existe apenas a possibilidade da execução de apenas um servidor de diretórios. Se por uma eventualidade este servidor parar de funcionar, todo o sistema é comprometido.

O gerenciador de arquivos distribuídos, apresentado neste capítulo, possui as funcionalidades básicas de um gerenciador de arquivos distribuído, ou seja, armazenar e recuperar arquivos de forma distribuída e transparente ao usuário. Porém o gerenciador não possui nenhum mecanismo de tolerância a falhas, sistema de *log* (histórico das ações) e outras ações que podem ser consideradas não funcionalidades de um sistema gerenciador de arquivos distribuídos, e podem ser adicionados ao mesmo, adaptando-o a novas possibilidades de utilização.

No capítulo seguinte será discutido e apresentado a especificação e implementação de novas funcionalidades adicionadas ao gerenciador de arquivos, utilizando o conceito de reflexão computacional.

5. Desenvolvimento

Com o objetivo de validar os conceitos vistos no capítulo 2 e as ferramentas descritas no capítulo 3, foi atribuído ao gerenciador de arquivos distribuídos [POS2000] novas tarefas que vem a adaptar funcionalidades já existentes e adicionar características não funcionais ao gerenciador.

Durante o trabalho, foram selecionadas tarefas que seriam relevantes para o estudo de reflexão computacional e para o próprio gerenciador de arquivos.

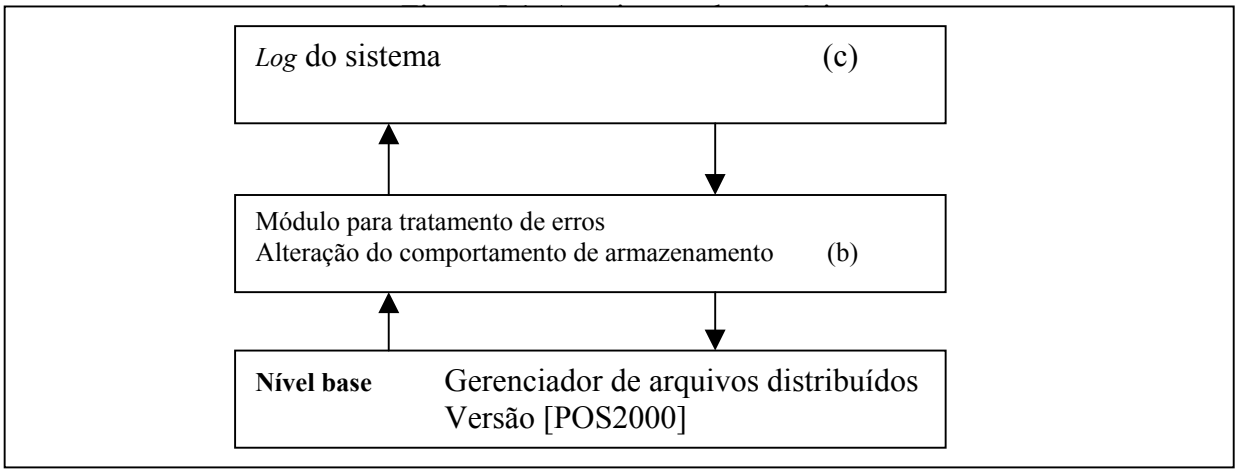
As tarefas, que foram implementadas, são: a alteração do comportamento de escolha para armazenamento de arquivos, a adição de um módulo para *log* do sistema e um módulo para tratamento de erros.

5.1. Modelagem e implementação do protótipo

Levando-se em consideração que este trabalho atua sobre o conceito reflexivo utilizando o modelo de objetos, e tendo-se em vista que não foi encontrada nenhuma referência sobre a existência de uma técnica para modelar sistemas reflexivos orientados a objetos, optou-se por utilizar o diagrama de classes e o diagrama de seqüência entre objetos da *Unified Modeling Language* (UML), para modelar o protótipo apresentado neste trabalho. Optou-se pelos diagramas da UML, por esta ser uma ferramenta de modelagem orientada a objetos unificada e com grande aceitação por parte da área acadêmica e comercial.

Tanto o diagrama de classes, como o diagrama de seqüência entre objetos sofreram pequenas alterações a fim de expressar o modelo reflexivo. No diagrama de classes adotou-se um traço horizontal entre as classes, para indicar a divisão de níveis, além de cores diferentes para as classes pertencentes a níveis diferentes. No diagrama de seqüência entre objetos, optou-se por uma configuração um pouco diferente da comum, com os objetos situados em níveis diferentes. Ambos os diagramas foram modelados utilizando a ferramenta *Rational Rose* [RAT2000].

A estrutura da arquitetura que o protótipo irá possuir está representado na figura 5.1.



No primeiro nível (nível base) tem-se a implementação que [POS2000] realizou, sem nenhuma alteração. As alterações a serem feitas no modelo são:

a) *alteração do comportamento de armazenamento*: as metaclasses dispostas no meta-nível (b) são responsáveis por alterar, em tempo de compilação, a estrutura e comportamento do nível base, a fim de otimizar o processo de escolha do servidor que será utilizado para armazenar determinado arquivo;

b) *módulo de tratamento de erros*: em tempo de compilação, as metaclasses dispostas no meta-nível (b) irão incluir, nas classes do nível base, dentro dos blocos de tratamento de exceções, chamadas para a metaclasse responsável por isolar os erros ocorridos durante a execução do sistema. Assim, em tempo de execução, quando ocorrer um erro, este é desviado para o meta-nível (b), juntamente com informações sobre o objeto em que ocorreu o erro;

c) *log do gerenciador*: em tempo de execução, as metaclasses dispostas no meta-nível (c) irão reificar todas as mensagens destinadas às classes do nível base e do nível (b), a fim de gerar um *log* de execução do sistema, permitindo a visualização de todas as atividades do gerenciador, inclusive os erros ocorridos durante a execução.

A implementação do protótipo será realizada utilizando as ferramentas OpenJava e Javassist. Para situações em tempo de compilação será utilizado o metaprotocolo do pré-processador OpenJava, e para situações em tempo de execução será utilizado o metaprotocolo da ferramenta Javassist. Assim, todos os níveis descritos acima estão interligados por um metaprotocolo, seja ele o metaprotocolo do OpenJava ou do Javassist.

Durante a implementação do protótipo foi utilizada a versão 1.2.2 da máquina virtual Java, devido ao fato da ferramenta Javassist somente admitir a versão 1.2.2 ou superior.

A versão do OpenJava utilizada será a versão 1.0, a mesma descrita no capítulo sobre ferramentas reflexivas da linguagem Java.

A versão do Javassist utilizada será a versão 0.6, versão beta teste, também a mesma versão descrita no capítulo sobre ferramentas que possibilitam a utilização de reflexão computacional na linguagem Java.

Nos itens abaixo, segue a modelagem e explanação detalhada, em separado, de cada tarefa implementada neste trabalho.

5.1.1. Escolha do Servidor para Armazenamento de Arquivos

Esta tarefa irá sobrepor o atual comportamento de escolha do servidor para armazenamento de arquivos. Tal comportamento é implementado nos métodos *retornaServidor* e *retornaDominio*, da classe base *Servicos*, descritos no quadro 5.1

Quadro 5.1: Comportamento atual dos métodos *retornaServidor* e *retornaDominio*

```
private String retornaServidor (String atual) {
  if (Servidor1 == atual)
       return Servidor2;
  else
       return Servidor1;
}
private String retornaDominio (String atual) {
  if (Servidor1 == atual)
       return Dominio2;
  else
       return Dominio1;
}
```

O comportamento atual é implementado seguindo uma lógica alternada de escolha do servidor de arquivos. Por exemplo, se em um primeiro momento o servidor de arquivos número 1 for utilizado para armazenar determinado arquivo, no segundo momento quem será utilizado será o servidor número 2.

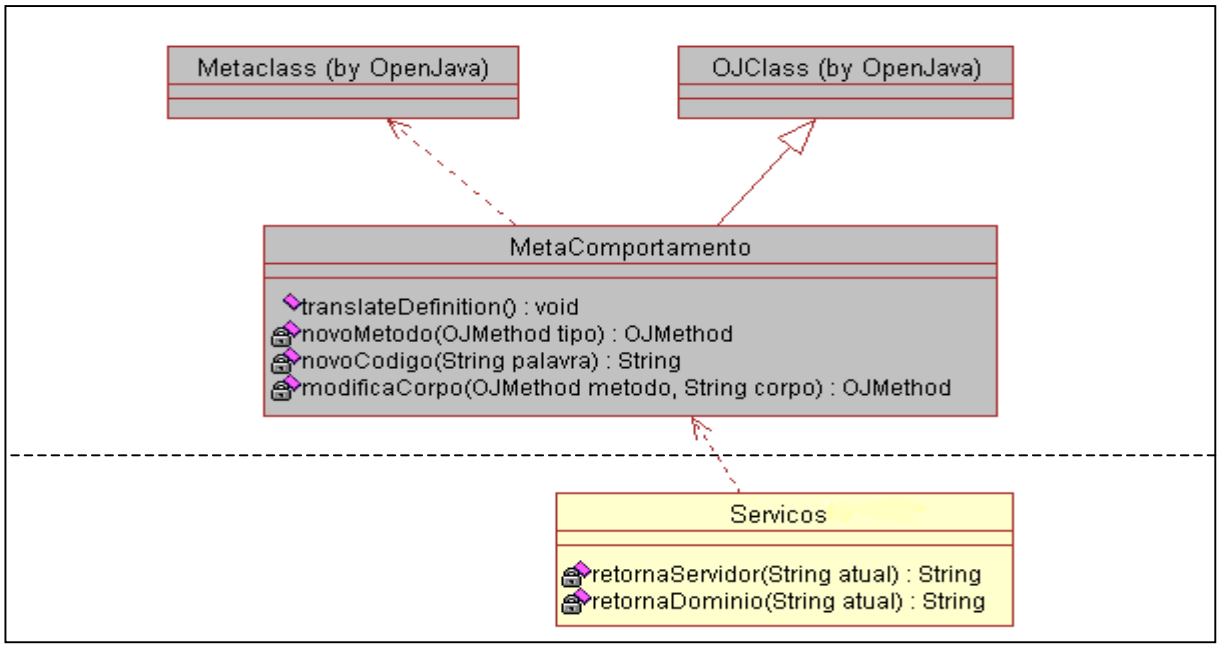
Através do atributo *atual* (quadro 5.1) se obtêm qual servidor foi utilizado por último para armazenar determinado arquivo. O atributo *atual* é comparado com o atributo *Servidor1*, se *atual* for igual ao *Servidor1*, então quer dizer que o último arquivo armazenado foi armazenado no *Servidor1* e o próximo será armazenado no *Servidor2*.

A fim de otimizar esta escolha, o novo comportamento irá admitir, como requisito, o espaço em disco livre de cada máquina.

Para isto foi criada, uma metaclasse denominada *MetaComportamento*, que irá, em tempo de compilação, sobrescrever os métodos responsáveis por decidir na escolha de qual servidor irá armazenar determinado arquivo.

Assim, como pode ser visto na figura 5.2, a classe que é responsável por descrever determinado comportamento (*Servicos*) no nível base é associada a classe *MetaComportamento*. É importante destacar que na classe *Servicos* foram apresentados apenas os métodos *retornaServidor* e *retornaDominio*, que sofreram modificações. Os demais foram omitidos.

Figura 5.2: Diagrama de classes da tarefa de alteração do comportamento



A metaclasse *MetaComportamento* (figura 5.2) possui os seguintes métodos:

a) *novoMetodo(OJMethod)*: retorna o método *espacoDisco*, que será um novo método pertencente a classe base *Servicos,* tendo, como responsabilidade, retornar a máquina que detêm o maior espaço livre em disco. O atributo do tipo *OJMethod* é passado como parâmetro para este método, para que o mesmo possa utilizar as características do método passado para criar o método novo;

b) *novoCodigo(String)*: responsável por retornar o novo código que irá sobrepor o código existente nos métodos *retornaServidor* e *retornaDominio*. O atributo do

tipo *String* é passado como parâmetro para identificar uma particularidade que diferencia o método *retornaServidor* do método *retornaDominio*;

c) *modificaCorpo(OJMethod,String)*: responsável por adicionar à classe base o método alterado. Este método recebe como parâmetros, as características do novo método (*OJMethod*) e o corpo do novo método (*String*);

d) t*ranslateDefinition()*: responsável por realizar a reflexão sobre os métodos reflexivos.

Visto que esta tarefa é apenas realizada em tempo de compilação, não se tem um diagrama de seqüência entre objetos desta tarefa, uma vez que tal diagrama é utilizado para representar o sistema em tempo de execução.

A metaclasse *MetaComportamento* é codificada utilizando-se pacotes da linguagem Java e do pré-processador OpenJava. No diagrama da figura 5.2, observa-se que a metaclasse *MetaComportamento* herda características da classe *OJClass* (como toda metaclasse em OpenJava) e é instanciada a partir da classe *Metaclass*. O nome do arquivo deve conter como extensão as letras *oj*, para que possa ser compilado pelo compilador do OpenJava.

Na classe base *Servicos*, é necessário alterar a assinatura da classe, de *class Servicos extends Thread*, para *class Servicos extends Thread instantiates MetaComportamento*, e também alterar a sua extensão para *oj*.

Para compilar tais classes deve-se digitar os seguintes comandos:

a) para a metaclasse *MetaComportamento*: ojc MetaComportamento.oj, ou java openjava.ojc.Main MetaComportamento.oj, que traz o mesmo resultado, gerando os arquivos MetaComportamento.java, MetaComportamento.class, MetaComportamentoOJMI.java e MetaComportamentoOJMI.class;

b) para a classe *Servicos*: ojc Servicos.oj, ou java openjava.ojc.Main Servicos.oj, que gera os arquivos Servicos.java, Servicos.class, ServicosOJMI.java e ServicosOJMI.class.

Deve-se tomar o cuidado em compilar primeiro a metaclasse e depois a classe, pois a classe é quem especifica a associação.

Após compilada a metaclasse e a classe base, os métodos *retornaServidor* e *retornaDominio* já terão sido alterados, e o método *espacoDisco* já haverá sido adicionado na classe base. No quadro 5.2 pode-se verificar tal modificação.

Quadro 5.2.: Novo comportamento dos métodos transformados

```
public String retornaServidor(java.lang.String oj_param0 ){
   if (espacoDisco( Dominio1 ) <= espacoDisco( Dominio2 )){
       return Servidor1;
   } else {
       return Servidor2;
}}

public long espacoDisco( java.lang.String oj_param0 ){
   //comportamento do método espacoDisco
   return espaco; }

public String retornaDominio( java.lang.String oj_param0){
   if (espacoDisco( Dominio1 ) <= espacoDisco( Dominio2 )){
       return Dominio1;
   } else {
       return Dominio2;
}}
```

O código fonte da metaclasse *MetaComportamento* e os métodos modificados da classe *Servicos* podem ser visualizados de forma mais detalhada no anexo A.

Caso deseja-se que a classe *Servicos* volte a ter o comportamento anterior, pode-se, através do arquivo Servicos.oj, restaurar o velho comportamento.

5.1.2. *Log* do sistema

Esta tarefa tem como objetivo gerar um *log* do sistema, retornando, através de uma janela de visualização, a descrição das ações realizadas em determinado período. O *log* do sistema é implementado através de um metaprotocolo em tempo de execução.

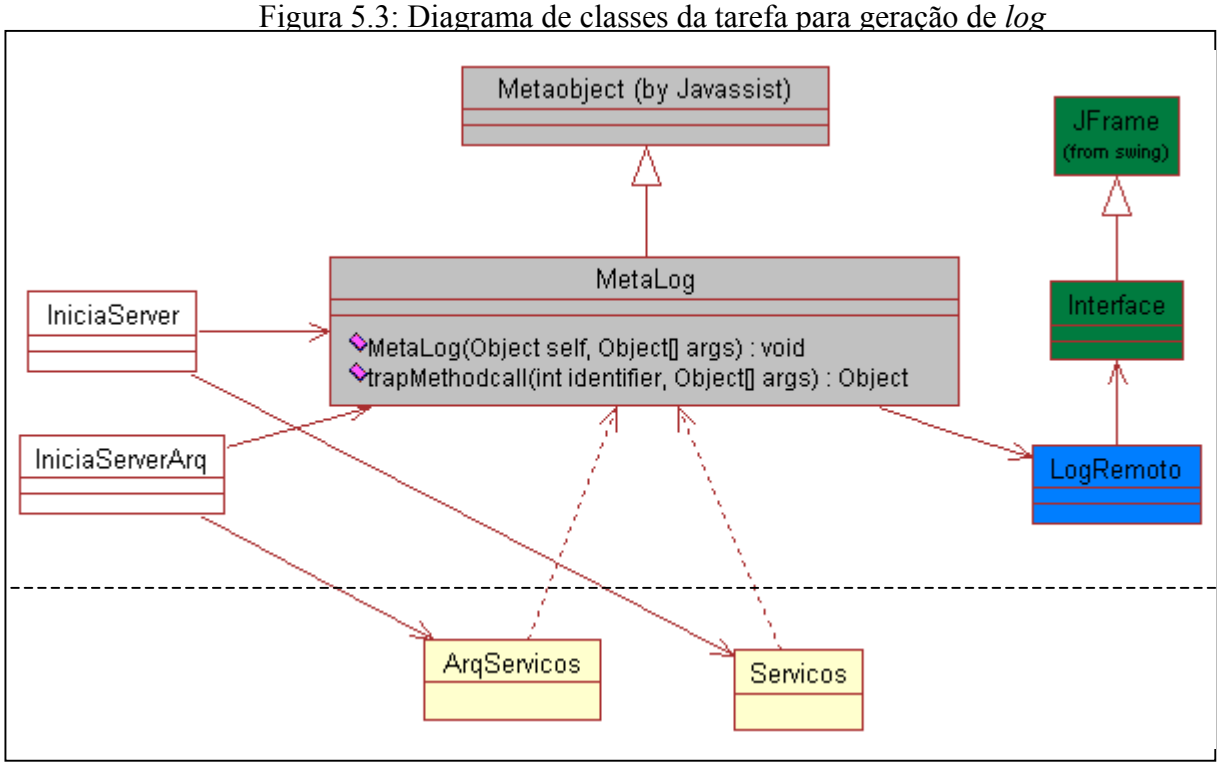
Na figura 5.3 pode ser visualizado o diagrama de classes desta tarefa. Neste diagrama tem-se a metaclasse *MetaLog,* que é a metaclasse responsável por captar todas as atividades e repassá-las às interfaces de destino.

As classes *IniciaServer* e *IniciaServerArq* detêm a responsabilidade de realizar a associação entre as classes de nível base com as classes de meta-nível.

A classe *LogRemoto* é associada com a classe *MetaLog* para possibilitar que todos os eventos ocorridos no ambiente distribuído sejam anotados.

A classe *Interface*, que é herdeira da classe *JFrame* da máquina virtual Java, serve como meio de visualização do *log* do sistema.

Figura 5.3: Diagrama de classes da tarefa para geração de *log*



A metaclasse *MetaLog* possui os métodos:

a) *MetaLog(Object self, Object[] args)*: método construtor que é acionado toda vez que um novo objeto de uma classe base for instanciado. Como parâmetros do método construtor, tem-se: o objeto que está sofrendo a reificação (*Object self*) e os argumentos do próprio objeto (*Object[] args*);

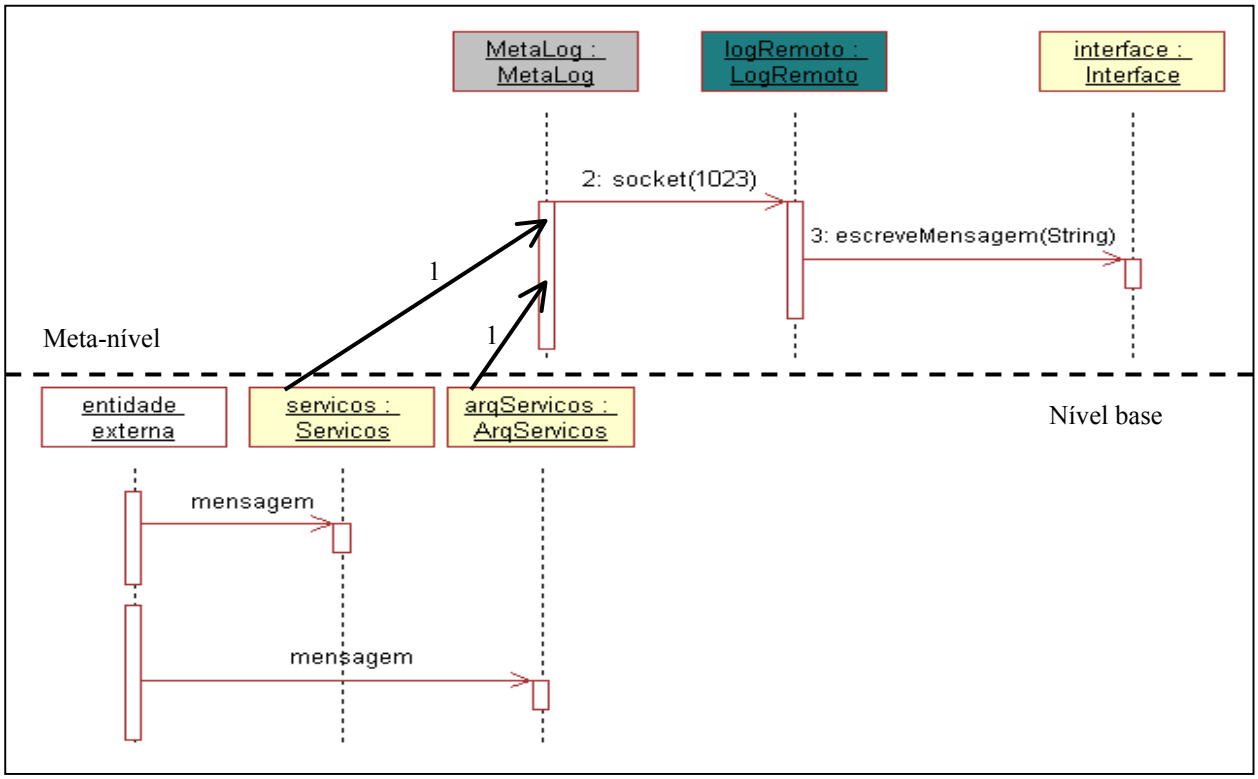
b) *trapMethodcall (int identifier, Object[] args)*: intercepta toda mensagem direcionada a uma classe base, ou seja, toda vez que for acionado um método da classe base. Possui como parâmetros o identificador do método (*int identifier*) e os parâmetros do método (*Object[] args*).

Assim a metaclasse *MetaLog* intercepta todas as mensagens direcionadas às suas classes do nível base, inclusive a mensagem para instância de objeto. Através da interceptação de todas as mensagens, a metaclasse obtêm informações sobre as classes de nível base e envia estas informações para a classe *LogRemoto*. Após a classe *LogRemoto* ter recebido as informações, ela redireciona a mesma informação para a classe *Interface*, que torna público os resultados.

Como pode ser visualizado no diagrama da figura 5.3, a metaclasse *MetaLog* é herdeira da classe *Metaobject*, característica comum a todas as metaclasses que utilizam o metaprotocolo da ferramenta Javassist.

Na figura 5.4 pode-se visualizar o diagrama de seqüência entre objetos da tarefa de *log* do sistema.

Figura 5.4: Diagrama de seqüência entre objetos da tarefa de *log* do sistema



Na figura 5.4 é possível visualizar a seqüência entre o nível base e o meta-nível. Todas as mensagens que são direcionadas aos objetos de nível base são interceptadas pelos metaobjetos e os dados reificados são encaminhados para o objeto *Interface*. Toda mensagem encaminhada aos objetos de nível base são interceptadas e reificadas pelas setas de número 1, a fim de fornecer aos metaobjetos informações sobre o estado atual dos objetos de nível base. Após reificadas as mensagens o meta-nível trata-as e encaminha ao objeto *LogRemoto*, que em seguida aciona o método *escreveMensagem* da classe *Interface*.

No código descrito no quadro 5.3, pode-se verificar que a classe *IniciaServer* associa a classe *Servicos* com a metaclasse *MetaLogServerArq* (*loaderServicos.makeReflective("Servicos", MetaLogServerArq.class, ClassMetaobject.class);*), e logo após determina que a classe *Server* inicie a sua execução (*classe.run("Server",args);*).

```
public class IniciaServer {
  public static void main(String[] args) throws Throwable {
    Loader classe = (Loader)IniciaServer.class.getClassLoader();
    ReflectLoader loaderServicos = new ReflectLoader();
    ReflectLoader loaderErro = new ReflectLoader();
    loaderServicos.makeReflective("Servicos", MetaLog.class
                                  ,ClassMetaobject.class);
    loaderErro.makeReflective("MetaErro", MetaLog.class ,ClassMetaobject.class);
    classe.addUserLoader(loaderServicos);
    classe.addUserLoader(loaderErro);
    classe.run("Server", args);
  }}
```

No quadro 5.4 pode-se visualizar o código fonte descrito no método construtor da metaclasse *MetaLog*, método que é executado toda vez que uma classe do nível base é instanciada.

Quadro 5.4: Método construtor da metaclasse *MetaLog*

```
public MetaLog(Object self, Object[] args) throws CannotInvokeException {
  super(self, args);
  try{
    File arq = new File("vazio");
    String separador = arq.separator;
    DirTrabalho = args[1].toString();
    File arqhost = new File(DirTrabalho + separador + "host");
    RandomAccessFile rhost = new RandomAccessFile(arqhost,"r");
    String dadosHost = (rhost.readLine()).trim();
    StringTokenizer token = new StringTokenizer(dadosHost,"/");
    dadosHost = token.nextToken();
    dadosHost = token.nextToken();

    st = new Socket(dadosHost,1023);
    toServer = new PrintStream(st.getOutputStream());
    fromServer = new DataInputStream(st.getInputStream());
    toServer.println("Inicializando " +   self.getClass().getName()+"\n");

    toServer.close();
    fromServer.close();
    st.close();
 }catch(Exception e){ System.out.println("ERRO: "+e); }
}
```

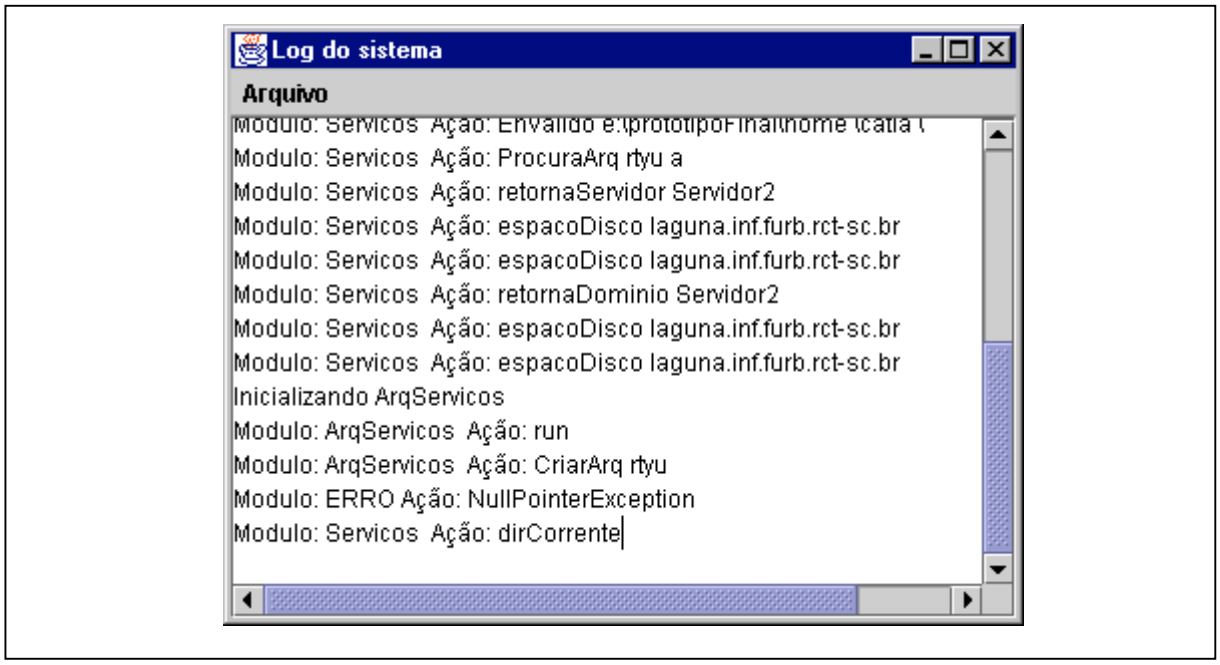
No quadro 5.4, a parte em cinza é responsável por determinar em que máquina da rede está a classe *LogRemoto*, e a parte em verde é responsável por estabelecer uma comunicação com a classe *LogRemoto* e enviar os dados da classe de nível base.

Toda chamada realizada nas classes de nível base são desviadas para a metaclasse *MetaLog*, e esta, por sua vez, através de uma porta de comunicação, via *Socket*, com a classe *LogRemoto,* envia uma mensagem disponibilizando, ao usuário, através da classe *Interface,* o resultado do *log*.

Ao executar o sistema, deve-se iniciar a execução do servidor de diretórios: *java javassist.Loader IniciaServer*. Em seguida, na mesma máquina deve-se iniciar a execução do LogRemoto: *java LogRemoto*, pois a visualização do *log* fica restrita a máquina servidora de diretórios. Após isto pode-se iniciar o servidor de arquivos: *java javassist.Loader IniciaServerArq* e o módulo cliente *java sa*.

No momento em que for iniciada a execução do LogRemoto, a interface visualizada na figura 5.5 já é instanciada, e a medida do tempo vai sendo acrescentada por novas mensagens, que visam demonstrar a situação do sistema gerenciador de arquivos.

Figura 5.5.: Interface do *log* do sistema



O código fonte das classes envolvidas no processo está descrito de forma completa no anexo B deste trabalho.

5.1.3. Módulo para tratamento de erros

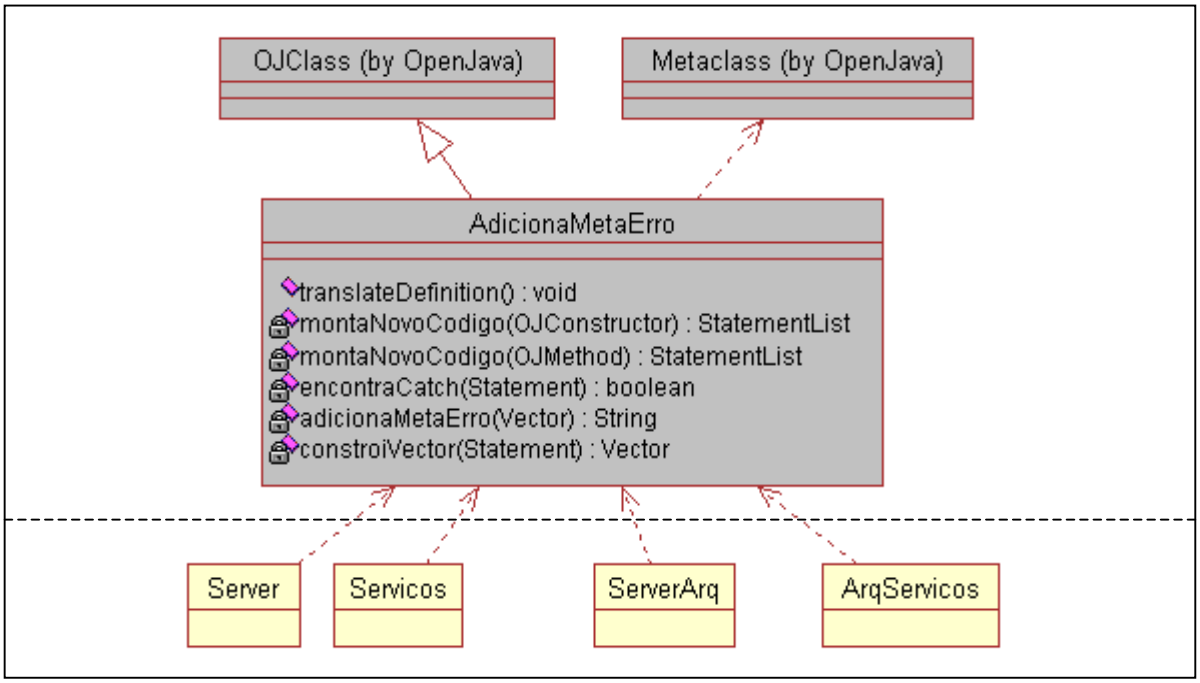
Esta tarefa tem como objetivo a construção de um módulo que trate os erros ocorridos durante a execução do sistema, de forma separada do gerenciador de arquivos.

O módulo, além de retornar o erro ocorrido, retorna o estado do objeto onde o erro ocorreu e, devido às características reflexivas que o módulo possui, pode-se alterar o estado do objeto para dar continuidade ou terminar a execução da aplicação.

A reflexão do sistema acontece em dois estágios: um no momento da compilação e o outro durante a execução. Assim, para cada modelo foi construído um diagrama de classes.

Na figura 5.6 pode-se visualizar o diagrama de classes, para alterar as classes de nível base em tempo de compilação.

Figura 5.6: Diagrama de classes do módulo de erro em tempo de compilação



No diagrama da figura 5.6, pode-se visualizar a metaclasse *AdicionaMetaErro*, que possui a responsabilidade de adicionar, a todas as classes de nível base, a capacidade de quando ocorrer um erro, poder instanciar um objeto da classe *MetaErro.*

A metaclasse *AdicionaMetaErro*, possui os métodos:

a) *translateDefinition()*: responsável por realizar a reflexão nas classes de nível base;

b) *montaNovoCodigo (OJConstructor atual)*: método que através do parâmetro passado, o método construtor da classe, identifica se existe ou não um bloco para tratamento de exceção no corpo do método, caso exista: faz com que outros métodos sejam executados para alterar o corpo do método, caso contrário: apenas retorna o corpo do método igual ao que recebeu pelo parâmetro;

c) *montaNovoCodigo (OJMethod atual)*: método com a mesma funcionalidade do método anterior, porém recebe como parâmetro, ao invés do método construtor da

classe, os métodos declarados pela classe. Este método possui a mesma lógica que o anterior;

d) *encontraCatch (Statement atual)*: método que retorna um valor lógico. Caso venha a encontrar um bloco de tratamento de exceções inserido no Statement que foi passado como parâmetro, retorna true, caso contrário retorna false;

e) *constroiVector (Statement atual)*: retorna um Vector após dividir o Statement, passado como parâmetro. Cada elemento do Vector é uma palavra que pertence ao Statement passado pelo parâmetro;

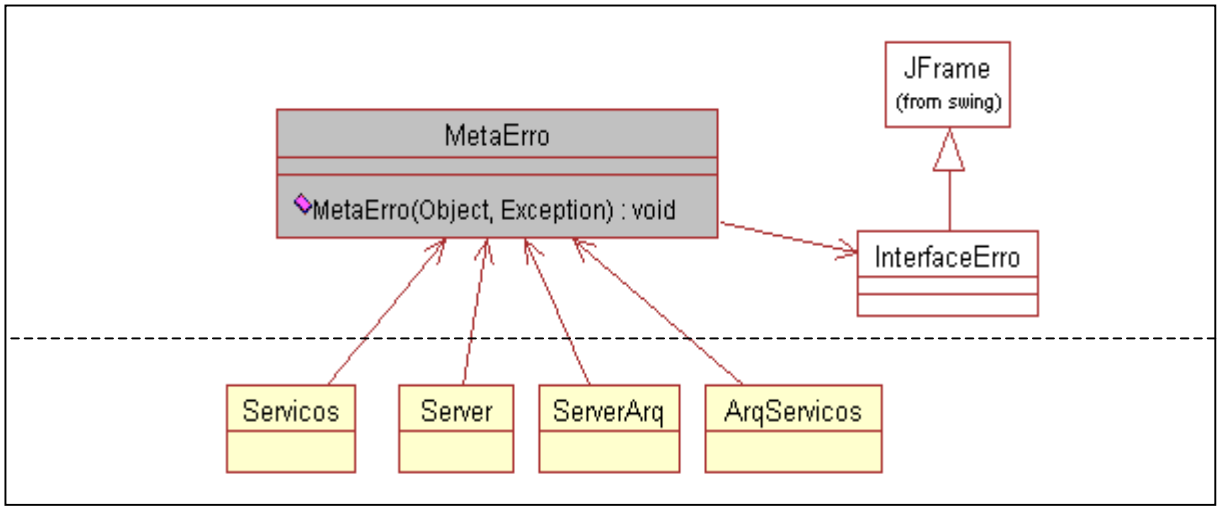
f) *adicionaMetaErro (Vector atual)*:método que recebe como parâmetro um Vector que é resultado do retorno do método *constroiVector(Statment atual)*. Este método adiciona na área de tratamento de exceções a chamada para a classe *MetaErro* (*new MetaErro(this,e)*).

Quando encontrado um bloco de tratamento de exceção, é posto em seu bloco uma chamada para a classe *MetaErro*, que em tempo de execução, caso haja um erro, poderá ser instanciada.

O diagrama de classes, em tempo de execução, do módulo de tratamento de erros, possui uma particularidade. A composição do diagrama em tempo de execução depende do que foi realizado em tempo de compilação, podendo haver, ou não, as associações descritas entre as classes do nível base para com o meta-nível. Caso, em tempo de compilação não for encontrado nenhum bloco de tratamento de erros em determinada classe do nível base, esta classe não será associada com a metaclasse em tempo de execução.

Na figura 5.7 pode-se visualizar o diagrama de classes, em tempo de execução, do módulo de detecção de falhas.

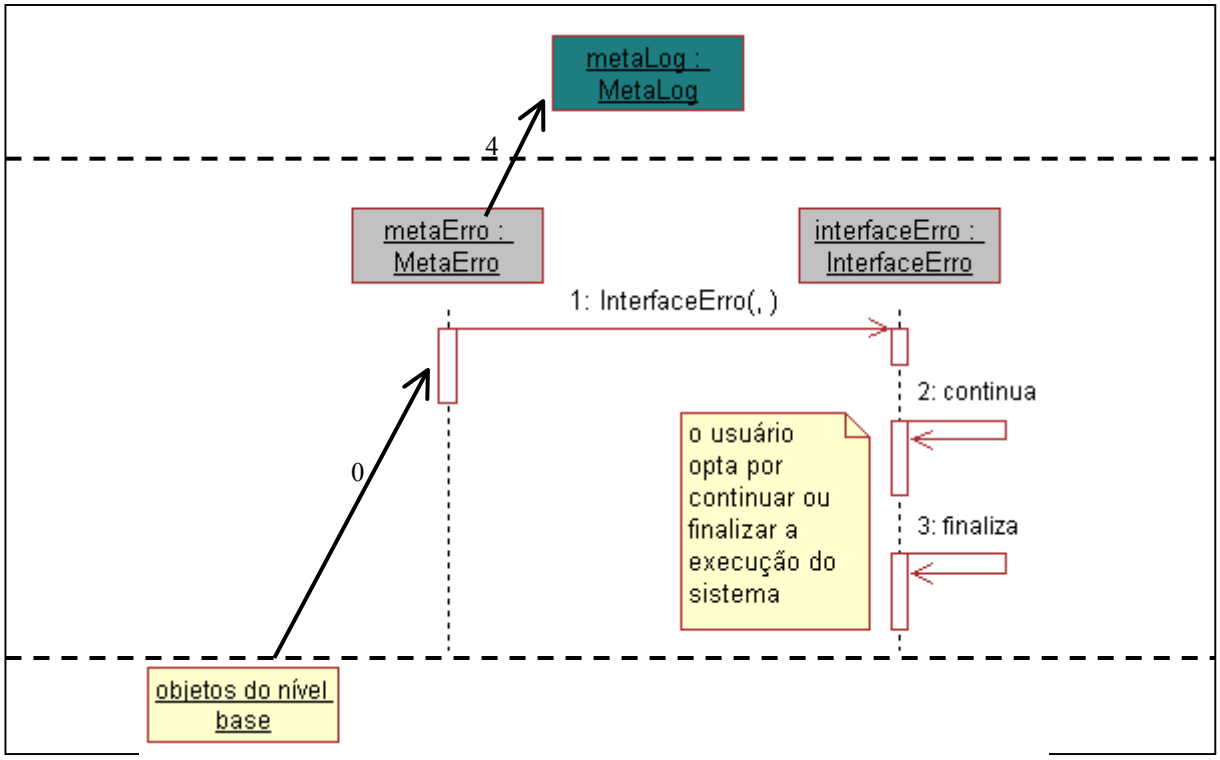
Figura 5.7: Diagrama de classes do módulo de erros em tempo de execução



A classe *MetaErro* possui apenas um método construtor, pois quando ocorre um erro a classe *MetaErro* é instância e é no método construtor que ocorre todos os controles para futura visualização dos dados na classe *InterfaceErro*.

A fim de expressar o funcionamento do módulo de tratamento de erros, na figura 5.8 pode-se visualizar o diagrama de seqüência entre objetos. O diagrama demonstra a execução do sistema a partir do momento em que ocorrer um erro na execução.

Figura 5.8.: Diagrama de seqüência entre objetos do módulo de erro



No diagrama da figura 5.8, pode-se visualizar:

a) o momento em que as classes do nível base instanciam a classe *MetaErro* (0), ou seja, no momento em que ocorre algum tipo de erro;

b) a reificação do objeto da classe *MetaErro* para a sua metaclasse *MetaLog* (4), no momento em que ocorre a instância de um objeto da classe *MetaErro*;

c) a passagem dos valores para a classe *InterfaceErro* (1), para que os mesmos sejam visualizados pelo usuário;

d) a opção em que o usuário tem de continuar (2) com a execução do sistema, ou finalizar (3).

Durante a compilação do sistema é utilizado o metaprotocolo do pré-processador OpenJava, com o objetivo de embutir código no sistema gerenciador de arquivos a fim de realizar chamadas, em tempo de execução, para a classe *MetaErro*.

As chamadas são embutidas dentro dos blocos de levantamento de exceções da linguagem Java, como pode ser visto no quadro 5.5.

Quadro 5.5: Chamada da classe *MetaErro*

```
try{

  //codigo com possibilidades de erro
}catch (Exception e){

  new MetaErro(this,e);
}
```

Após transformado o sistema, quando ocorrer qualquer espécie de erro, a classe de nível base onde tiver acontecido o erro, executa uma instância da classe *MetaErro*, que não se utiliza de qualquer tipo de metaprotocolo, mas utiliza-se apenas das capacidades da API do próprio JDK (java.lang.reflect) para realizar introspecção sobre a classe onde ocorreu o erro e sobre a exceção (quadro 5.6).

Quadro 5.6: Método construtor da classe *MetaErro*

```
public MetaErro(Object self, Exception e){
     nomeClasse = self.getClass().getName();
     try{
       vitima = self.getClass();
       atributos = vitima.getDeclaredFields();
       erro = e.toString();
       for(int i=0; i<atributos.length; i++){
         nomeAtributos.addElement(atributos[i].getName());
         valorAtributos.addElement(atributos[i].get(self));
       }
       interfaceErro = new    InterfaceErro(nomeClasse,erro,nomeAtributos,
                          valorAtributos);
     }catch(Exception excessao){
       System.out.println(excessao);
     }
}
```

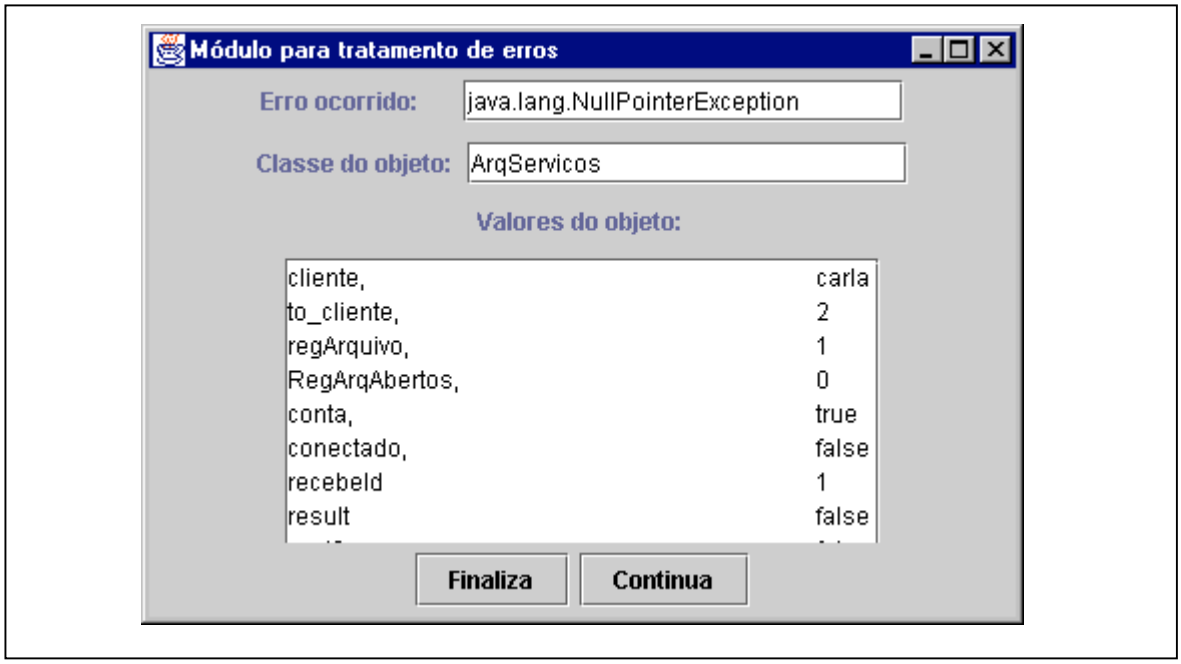
A execução desta tarefa ocorre primeiro em tempo de compilação, onde é necessário alterar a extensão de todas as classes de nível base de *.java* para *.oj*, compilar a metaclasse *AdicionaMetaErro*: java openjava.ojc.Main AdicionaMetaErro.oj, e compilar todas as classes *Server, Servicos, ServerArq* e *ArqServicos* utilizando o mesmo compilador.

Deve-se levar em consideração que a classe *Server* é alterada duas vezes em tempo de compilação, e deve-se tomar o cuidado de compilar a classe primeiro utilizando a metaclasse *MetaComportamento* e depois a metaclasse *AdicionaMetaErro*.

Depois de compiladas as classes pode-se executá-las normalmente utilizando a máquina virtual Java padrão.

Na figura 5.9. pode-se visualizar o funcionamento da classe *InterfaceErro*.

Figura 5.9: Interface do módulo de tratamento de erros



A classe *MetaErro* age localmente, sendo classe base da *MetaLog*, por isso todo erro que ocorrer em qualquer dos módulos será notificado na interface da classe *MetaLog*.

Ao ocorrer um erro, o usuário tem, à sua disposição, o erro que ocorreu, em que módulo e o valor dos atributos do objeto, podendo tomar duas atitudes: finalizar a execução do sistema, apertando no botão “Finaliza”, ou continuar com a execução do sistema, ignorando o erro, pressionando no botão “Continua”.

Este módulo apenas se utiliza da capacidade introspectiva da linguagem Java, ou seja, apenas obtêm valores, não os modificando.

Como nas outras tarefas, o código completo das classes pertencentes a esta tarefa esta disponível no anexo C.

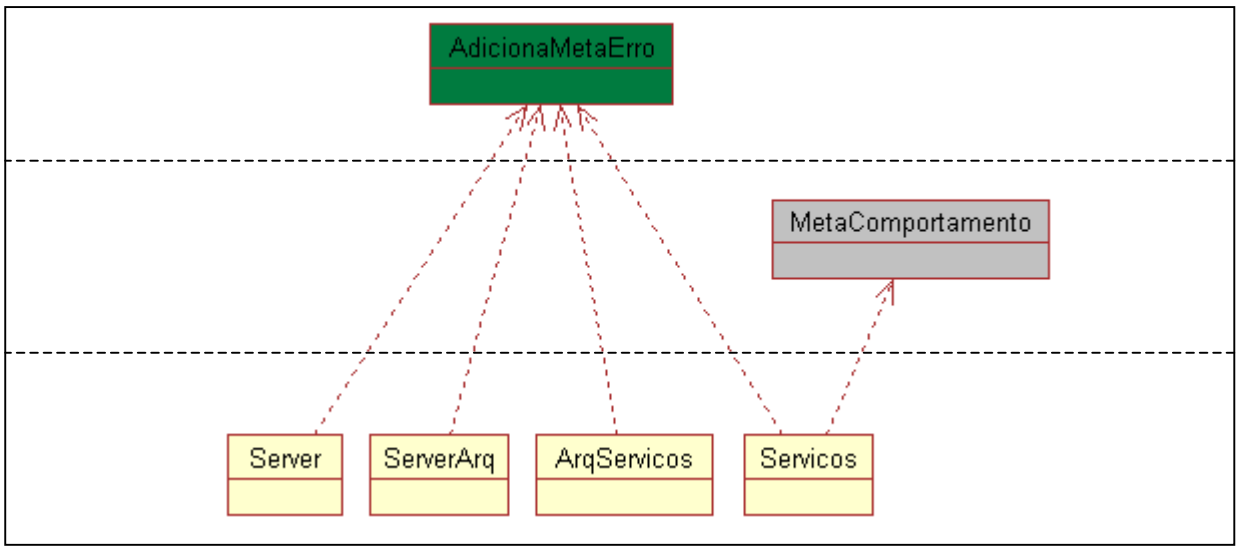
5.2. Diagramas de classes consolidados

Para uma melhor visualização do sistema, são apresentados os diagramas de classes consolidados, ou seja, que agregam todas as alterações realizadas no sistema. Estes diagramas estão separados em um diagrama que descreve o que acontece em tempo de compilação e o outro em tempo de execução.

Ambos os diagramas são formados a partir dos diagramas anteriores e respeitando as camadas estabelecidas pela arquitetura da aplicação (figura 5.1).

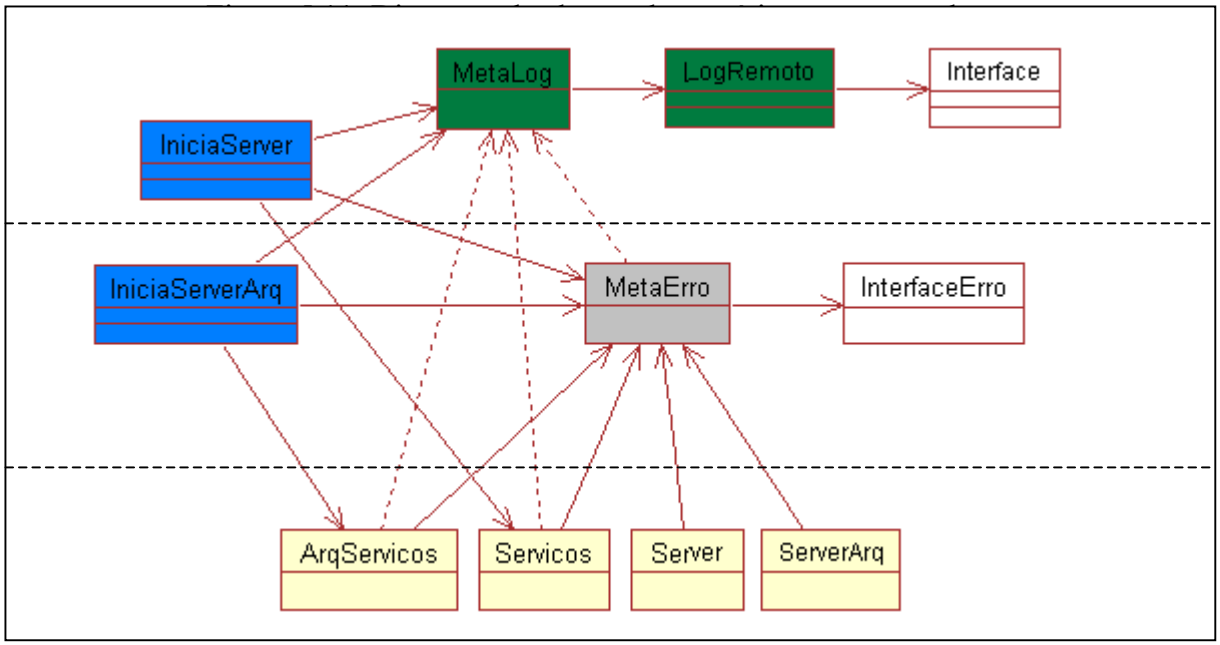
Na figura 5.10, pode-se visualizar o diagrama de classes, para o tempo de compilação, do protótipo.

Figura 5.10: Diagrama de classes do protótipo em tempo de compilação



No diagrama da figura 5.10, pode-se visualizar as classes de nível base e as metaclasses *MetaComportamento* e *AdicionaMetaErro*. As classes *Server, ServerArq* e *ArqServicos* são classes base da metaclasse *AdiconaMetaErro*, enquanto que a classe *Servicos* é classe base das metaclasses *AdicionaMetaErro* e *MetaComportamento*.

O diagrama apresentado na figura 5.11 apresenta a composição do protótipo em tempo de execução.



O nível base é composto pelas classes do gerenciador de arquivos, implementado em [POS2000], com algumas modificações devido às transformações acontecidas em tempo de compilação. No segundo nível estão dispostas as metaclasses do módulo de tratamento de erros. No terceiro, e último, nível estão dispostas as metaclasses que realizam o *log* do sistema. A metaclasse *MetaErro*, além de ser metaclasse, também é classe base da metaclasse *MetaLog*. A associação entre a classe base *MetaErro* e a sua metaclasse *MetaLog* é realizado nas classes *IniciaServerArq* e *IniciaServer*.

6. Conclusões e Sugestões

Durante este trabalho foram apresentados conceitos e ferramentas que possibilitam implementar sistemas reflexivos. No decorrer do trabalho foi possível destacar as características, vantagens e desvantagens dos diversos modelos e protocolos.

Este trabalho pode ser utilizado por pessoas que queiram ter um fundamento básico sobre reflexão computacional, serviu para adicionar novas funcionalidades ao gerenciador de arquivos, que proporcionam uma melhor performance, e alguns serviços extras para um melhor controle do sistema.

Através do protótipo implementado e dos exemplos listados, pode-se perceber que são verdadeiras as vantagens sobre a utilização de reflexão computacional em sistemas orientados a objetos, listadas no capítulo 2, como: alto poder de adaptação, alto grau de reusabilidade e redução de complexidade.

Em teoria, o modelo reflexivo, aliado a arquitetura de meta-níveis, 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 metaprotocolo que vem a atender de forma restrita algumas necessidades de sistemas reflexivos.

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. Neste trabalho utilizou-se o pré-processador OpenJava, que realiza reflexão em tempo de compilação, e a ferramenta Javassist que realiza reflexão em tempo de execução. Pôde-se verificar que a utilização de um metaprotocolo reflexivo em tempo de compilação é mais adequado para adaptar e reutilizar sistemas, enquanto que, um metaprotocolo em tempo de execução é mais adequado para auxiliar na implementação de sistemas que exigem uma certa tolerância a falhas.

Ainda sobre as ferramentas utilizadas, pode-se notar que o pré-processador OpenJava fornece um conjunto extenso de classes e métodos na sua API, mas por outro lado a sua utilização é de certa forma complexa, enquanto que a API da ferramenta Javassist é simples de se utilizar, mas ao mesmo tempo muito limitada.

A respeito do protótipo implementado, os objetivos propostos foram atingidos. Conseguiu-se implementar aspectos não-funcionais ao gerenciador, conseguiu-se listar conceitos e formas de implementação sobre reflexão computacional e também avaliar o funcionamento do pré-processador OpenJava e da ferramenta Javassist.

Como restrições deste trabalho, tem-se o módulo de tratamento de erros que não trata todos os erros e também não fornece a possibilidade de restaurar o sistema após uma falha grave.

Sugestões para trabalhos futuros são: estudar e avaliar mais profundamente o pré-processador OpenJava, assim como outras metaprotocolos, linguagens e ferramentas reflexivas. Pode-se, também, visar a implementação de objetos persistentes, réplicas de servidores, armazenamento de transações e recuperação de estados utilizando reflexão computacional sobre a implementação do gerenciador de arquivos, além da implementação de sistemas tolerantes a falhas ou de tempo real que utilizem um modelo reflexivo. Pode-se também adicionar ao módulo de tratamento de erros, além da capacidade de introspecção, a capacidade de intercessão, com o objetivo de possibilitar que o módulo possa realmente tratar o erro e permitir com que o sistema continue executando sem atingir a integridade dos dados.

Anexo A: Código fonte das classes pertencentes a tarefa de alteração do comportamento

Este anexo contêm o código fonte da metaclasse *MetaComportamento*.

```
import openjava.mop.*;
import openjava.ptree.*;
import openjava.syntax.*;

public class MetaComportamento instantiates Metaclass extends OJClass {

  /*
  @ Metodo que realiza a reflexao da classe base
  */
  public void translateDefinition() throws MOPException {

     OJMethod [] metodos = getMethods (this);
     OJMethod novo;
     for (int i=0; i<metodos.length; ++i){
        if (metodos[i].getName().equals("retornaServidor")){
           novo = modificaCorpo (metodos[i],novoCodigo("Servidor"));
           removeMethod(metodos[i]);
           addMethod(novo);
        }
        if (metodos[i].getName().equals("retornaDominio")){
           novo = modificaCorpo (metodos[i],novoCodigo("Dominio"));
           addMethod(novoMetodo(metodos[i]));
           removeMethod(metodos[i]);
           addMethod(novo);
        }
     }
  }

  /*
  @ Metodo que retorna o novo metodo adicionado a classe
  */
  private OJMethod novoMetodo (OJMethod tipo){
    OJMethod metodo = new OJMethod
    (this,
     tipo.getModifiers(),
     OJSystem.LONG,
     "espacoDisco",
     tipo.getParameterTypes(),
     tipo.getExceptionTypes(),
     null);

    try{

    OJField atributoArq = new OJField (this,
    OJModifier.forModifier(OJModifier.PRIVATE),forName("java.net.Socket"),"st");

    OJField atributoRArq = new OJField
    (this,
     OJModifier.forModifier(OJModifier.PRIVATE),
     forName("java.io.PrintStream"),
     "toServer");

    OJField atributoEspD = new OJField
    (this,
       OJModifier.forModifier(OJModifier.PRIVATE),
       forName("java.io.DataInputStream"),
       "fromServer");

      addField(atributoArq);
      addField(atributoRArq);
```

```
        addField(atributoEspD);

        TypeName tipoExcessao = new TypeName("Exception");
        Parameter parametroExcessao = new Parameter(tipoExcessao,"e");
        CatchBlock blocoCatch = new CatchBlock
          (parametroExcessao,new StatementList(makeStatement("return 0;")));
        CatchList listaCatch = new CatchList(blocoCatch);

        StatementList codigo = new StatementList
          (makeStatement("st = new Socket(oj_param0,1024);"));
        codigo.add
          (makeStatement("toServer = new PrintStream(st.getOutputStream());"));
        codigo.add
          (makeStatement("fromServer = new DataInputStream(st.getInputStream());"));
        codigo.add
          (makeStatement("return Long.valueOf(fromServer.readLine()).longValue();"));

        TryStatement declaraTry = new TryStatement(codigo,listaCatch);
        codigo = new StatementList(makeStatement(declaraTry.toString()));
        metodo.setBody(codigo);

      }catch(Exception e){
         System.out.println("Erro: "+e);
      }
      return metodo;
    }

    /*
    @ Metodo que descreve o novo comportamento dos metodos reflexivos
    */
    private String novoCodigo (String palavra){
      return   "if (espacoDisco(Dominio1) <= espacoDisco(Dominio2))
                return "+palavra+"1; else return "+palavra+"2;";
    }

    /*
    @ Metodo que reifica as caracteristicas e altera o
    comportamento dos metodos reflexivos
    */
    private OJMethod modificaCorpo (OJMethod metodo, String corpo){
    OJMethod modificado = new OJMethod
    (this,
      metodo.getModifiers(),
      metodo.getReturnType(),
      metodo.getName(),
      metodo.getParameterTypes(),
      metodo.getExceptionTypes(),
      null);

    try{
      StatementList codigo = new StatementList(makeStatement(corpo));
      modificado.setBody (codigo);
    }catch(Exception e){
      System.err.println("Erro: "+e);
    }
    return modificado;
    }
}
```

Anexo B: Código fonte das classes pertencentes a tarefa de implementação do *log* do sistema

Este anexo contêm os códigos das classes *IniciaServer.java, IniciaServerArq.java, MetaLog.java, LogRemoto.java, Interface.java* e *IniciaMetaErro.java.*

```
import javassist.Loader;
import javassist.reflect.ClassMetaobject;
import javassist.reflect.ReflectLoader;

public class IniciaServer {
  public static void main(String[] args) throws Throwable {
    Loader classe = (Loader)IniciaServer.class.getClassLoader();
    ReflectLoader loaderServicos = new ReflectLoader();
    ReflectLoader loaderErro = new ReflectLoader();
    loaderServicos.makeReflective("Servicos", MetaLog.class
                                  ,ClassMetaobject.class);
    loaderErro.makeReflective("MetaErro", MetaLog.class ,ClassMetaobject.class);
    classe.addUserLoader(loaderServicos);
    classe.addUserLoader(loaderErro);
    classe.run("Server", args);
  }
}
```

```
import javassist.Loader;
import javassist.reflect.ClassMetaobject;
import javassist.reflect.ReflectLoader;

public class IniciaServerArq {
  public static void main(String[] args) throws Throwable {
    Loader cls = (Loader)IniciaServerArq.class.getClassLoader();
    ReflectLoader loader = new ReflectLoader();
    ReflectLoader loaderErro = new ReflectLoader();
    loader.makeReflective("ArqServicos", MetaLog.class, ClassMetaobject.class);
    loaderErro.makeReflective("MetaErro", MetaLog.class, ClassMetaobject.class);
    cls.addUserLoader(loader);
    cls.addUserLoader(loaderErro);
    cls.run("ServerArq", args);
  }
}
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Interface extends JFrame {
  private BorderLayout borderLayout1 = new BorderLayout();
  private JMenuBar menuBar1 = new JMenuBar();
  private JMenu menuFile = new JMenu();
  private JMenuItem menuFileExit = new JMenuItem();
  private JScrollPane jScrollPane1 = new JScrollPane();
  private JTextArea jTextArea1 = new JTextArea();
  private JMenuItem menuFileSave = new JMenuItem();

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

```
  }

  private void jbInit() throws Exception {
    this.setTitle("Log do sistema");
    this.getContentPane().setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    menuFile.setText("File");
    menuFile.setLabel("Arquivo");
    menuFileExit.setText("Sair");
    menuFileExit.setLabel("Sair");
    menuFileSave.setText("Salvar");
    menuFileSave.setLabel("Salvar");
    menuFileSave.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        openDialog(e);
      }
    });
    menuFileExit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          fileExit_ActionPerformed(e);
      }
    });
    menuFile.add(menuFileSave);
    menuFile.add(menuFileExit);
    menuBar1.add(menuFile);
    this.setJMenuBar(menuBar1);
    this.getContentPane().add(jScrollPane1, BorderLayout.CENTER);
    jScrollPane1.getViewport().add(jTextArea1, null);
    this.show();
  }

  private void fileExit_ActionPerformed(ActionEvent e) {
    System.exit(1);
  }

  private void openDialog(ActionEvent e) {
  }

  public void adiciona(String linha){
    jTextArea1.append(linha+"\n");
  }
}



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

public class LogRemoto {

  private ServerSocket s = null;
  private Socket st;
  private PrintStream toClient;
  private DataInputStream fromClient;
  private Interface interfaces;

  public static void main (String args[]){
    new LogRemoto();
  }

  public LogRemoto(){
    try{
      interfaces = new Interface();
      s = new ServerSocket(1023);
      conecta();
    }catch(IOException e){
      System.out.println("Erro: "+e);
    }
  }

  public void conecta(){
    while (true){
      try{
        st = s.accept();
        toClient = new PrintStream(st.getOutputStream());
        fromClient = new DataInputStream(st.getInputStream());

        String mensagem = fromClient.readLine();
```

```
          escreveMensRemota(mensagem);
          fromClient.close();
          toClient.close();
          st.close();
        }catch (IOException e){
          System.out.println("Erro: "+e);
        }
      }
    }

    public void escreveMensRemota(String aux){
      interfaces.adiciona(aux);
    }
}



import javassist.*;
import javassist.reflect.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class MetaLog extends Metaobject {

    private Socket st;
    private PrintStream toServer;
    private DataInputStream fromServer;
    private String DirTrabalho;

    public MetaLog(Object self, Object[] args) throws CannotInvokeException {
      super(self, args);
      try{
        File arq = new File("vazio");
        String separador = arq.separator;
        DirTrabalho = args[1].toString();
        File arqhost = new File(DirTrabalho + separador + "host");
        RandomAccessFile rhost = new RandomAccessFile(arqhost,"r");
        String dadosHost = (rhost.readLine()).trim();
        StringTokenizer token = new StringTokenizer(dadosHost,"/");
        dadosHost = token.nextToken();
        dadosHost = token.nextToken();

        st = new Socket(dadosHost,1023);

        toServer = new PrintStream(st.getOutputStream());
        fromServer = new DataInputStream(st.getInputStream());
        toServer.println("Inicializando " + self.getClass().getName()+"\n");

        toServer.close();
        fromServer.close();
        st.close();
      }catch(Exception e){
        System.out.println("ERRO: "+e);
      }
    }


    public Object trapMethodcall(int identifier, Object[] args) throws
CannotInvokeException {

      try{
        String temp = "";
        File arq = new File("vazio");
        String separador = arq.separator;
        File arqhost = new File(DirTrabalho + separador + "host");
        RandomAccessFile rhost = new RandomAccessFile(arqhost,"r");
        String dadosHost = (rhost.readLine()).trim();
        StringTokenizer token = new StringTokenizer(dadosHost,"/");
        dadosHost = token.nextToken();
        dadosHost = token.nextToken();

        st = new Socket(dadosHost,1023);

        toServer = new PrintStream(st.getOutputStream());
        fromServer = new DataInputStream(st.getInputStream());
        temp = temp +"Modulo: "+getClassMetaobject().getName()+
                     "  Ação: "+getMethodName(identifier)+" ";
        for (int i=0; i<args.length ; i++)
```

```
          temp = temp +args[i].toString()+" ";
        toServer.print(temp);
        toServer.close();
        fromServer.close();
        st.close();
      }catch(Exception e){
        System.out.println("ERRO: "+e);
      }
      return super.trapMethodcall(identifier, args);
    }
}
```

Anexo C: Código fonte das classes pertencentes ao módulo de tratamento de erros.

Este anexo possui o código fonte das classes *MetaErro.java, AdicionaqMetaErro.oj* e *InterfaceErro.java*.

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

public class MetaErro{

  private InterfaceErroAux interfaceErroAux;
  private String nomeClasse;
  private Class vitima;
  private Field [] atributos;
  private String erro;
  private Vector nomeAtributos = new Vector();
  private Vector valorAtributos = new Vector();

  public MetaErro(Object self, Exception e){
    try{
    nomeClasse = self.getClass().getName();
      vitima = self.getClass();
      atributos = vitima.getDeclaredFields();
      erro = e.toString();
      for(int i=0; i<atributos.length; i++){
        nomeAtributos.addElement(atributos[i].getName());
        valorAtributos.addElement(atributos[i].get(self));
      }
      interfaceErroAux = new
            InterfaceErroAux(nomeClasse,erro,nomeAtributos,valorAtributos);
    }catch(Exception excessao){
      nomeClasse = "Thread";
      try{
        vitima = self.getClass();
        erro = e.toString();
        Vector nomeAtributos = new Vector();
        Vector valorAtributos = new Vector();
        nomeAtributos.addElement("Acesso ilegal");
        valorAtributos.addElement("Acesso ilegal");
        interfaceErroAux = new
            InterfaceErroAux(nomeClasse,erro,nomeAtributos,valorAtributos);
      }catch (Exception e2){
        System.out.println("Erro interno: "+e2);
      }
    }
  }
}
```

```
import openjava.mop.*;
```

```
import openjava.ptree.*;
import openjava.syntax.*;
import java.util.*;

public class AdicionaMetaErro instantiates Metaclass extends OJClass {

  /*
  @ Metodo que realiza a reflexao da classe base
  */
  public void translateDefinition() throws MOPException {
    OJConstructor [] construtor = getConstructors (this);
    OJMethod [] metodos = getDeclaredMethods ();
    for(int i=0; i<construtor.length; i++)
      construtor[i].setBody(montaNovoCodigo(construtor[i]));
    for(int i=0; i<metodos.length; i++)
      if (!metodos[i].getName().equals("main")){
        OJMethod aux = new OJMethod
        (this,
        metodos[i].getModifiers(),
        metodos[i].getReturnType(),
        metodos[i].getName(),
        metodos[i].getParameterTypes(),
        metodos[i].getParameters(),
        metodos[i].getExceptionTypes(),
        null);
        aux.setBody(montaNovoCodigo(metodos[i]));
        removeMethod(metodos[i]);
        addMethod(aux);
      }
  }

  private StatementList montaNovoCodigo(OJConstructor atual){
    StatementList codigoVelho = new StatementList();
    StatementList codigoNovo = new StatementList();
    try{
      codigoVelho = atual.getBody();
      for (int j=0; j<codigoVelho.size(); j++)
        if(encontraCatch(codigoVelho.get(j)))
          codigoNovo.add
            (makeStatement(adicionaMetaErro(constroiVector(codigoVelho.get(j)))));
        else
          codigoNovo.add(codigoVelho.get(j));
    }catch(Exception e){
      System.out.println(e);
    }
    return codigoNovo;
  }

  private StatementList montaNovoCodigo(OJMethod atual){
    StatementList codigoVelho = new StatementList();
    StatementList codigoNovo = new StatementList();
    try{
      codigoVelho = atual.getBody();
      for (int j=0; j<codigoVelho.size(); j++)
        if(encontraCatch(codigoVelho.get(j)))
          codigoNovo.add
            (makeStatement(adicionaMetaErro(constroiVector(codigoVelho.get(j)))));
        else
          codigoNovo.add(codigoVelho.get(j));
    }catch(Exception e){
      System.out.println(e);
    }
    return codigoNovo;
  }

  private boolean encontraCatch (Statement atual){
    try{
      StringTokenizer palavra = new StringTokenizer(atual.toString());
      int aux_int = palavra.countTokens();
      for (int j=0; j<aux_int; j++)
        if (palavra.nextToken().equals("catch"))
          return true;
    }catch(Exception e){
      System.out.println(e);
    }
    return false;
  }

  private String adicionaMetaErro (Vector atual){
```

```
    String aux = "";
    int i=0;
    String varException = "e";
    while (i<atual.size()){
      aux = aux + atual.elementAt(i).toString()+"  ";
      if(atual.elementAt(i).equals("catch")){
        for(int j=1; j<=5; j++){
          aux = aux + atual.elementAt(i+j).toString()+ "  ";
          if(j==3)
            varException = atual.elementAt(i+j).toString();
        }
        aux = aux + "new MetaErro(this," + varException + ");  " ;
        i=i+5;
      }
      i=i+1;
    }
    return aux;
  }

  private Vector constroiVector (Statement atual){
    Vector retorno = new Vector();
    try{
      StringTokenizer palavra = new StringTokenizer(atual.toString()," ");
      int aux_int = palavra.countTokens();
      for (int j=0; j<aux_int; j++)
        retorno.addElement(palavra.nextToken());
    }catch(Exception e){
      System.out.println(e);
    }
    return retorno;
  }
}
```

```
import javax.swing.*;
import java.util.*;
class InterfaceErroAux extends JFrame{
  public InterfaceErroAux
    (String nomeClasse, String erro, Vector nomeAtributos, Vector valorAtributos) {
        InterfaceErro interfaceErro = new InterfaceErro
        (this,
        "Módulo para tratamento de erros",
        true,
        nomeClasse,
        erro,
        nomeAtributos,
        valorAtributos);
  }
}
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class InterfaceErro extends JDialog {
  JPanel jPanel1 = new JPanel();
  JPanel jPanel2 = new JPanel();
  JPanel jPanel3 = new JPanel();
  JPanel jPanel4 = new JPanel();
  JPanel jPanel5 = new JPanel();
  JPanel jPanel6 = new JPanel();
  JPanel jPanel7 = new JPanel();
  JLabel jLabel1 = new JLabel();
  JTextField jTextField1 = new JTextField(20);
```

```
JLabel jLabel2 = new JLabel();
JTextField jTextField2 = new JTextField(20);
JButton jButton1 = new JButton();
JButton jButton2 = new JButton();
JScrollPane painel1 = new JScrollPane();
JScrollPane painel2 = new JScrollPane();
JTextArea textoNome = new JTextArea();
JTextArea textoValor = new JTextArea();

public InterfaceErro
(Frame parent, String title, boolean modal, String nomeClasse, String erro,
Vector nomeAtributos, Vector valorAtributos) {
   super(parent, title, modal);
   try  {
     jbInit(nomeClasse,erro,nomeAtributos,valorAtributos);
     pack();
   }
   catch (Exception e) {
     e.printStackTrace();
   }
}

private void jbInit
(String nomeClasse, String erro, Vector nomeAtributos, Vector valorAtributos)
throws Exception {

  jPanel1.setLayout(new BorderLayout());
  jPanel3.setLayout(new BorderLayout());
  jPanel5.setLayout(new BorderLayout());
  jPanel2.setLayout(new BorderLayout());
  jLabel1.setText("Erro ocorrido:");
  jLabel1.setFont(new Font("Dialog", 1, 12));
  jTextField1.setColumns(25);
  jTextField1.setText(erro);
  jLabel2.setText("Classe do objeto:");
  jLabel2.setFont(new Font("Dialog", 1, 12));
  jTextField2.setColumns(25);
  jTextField2.setText(nomeClasse);
  jButton1.setText("Continua");
  jButton1.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
      continua(e);
    }
  });
  jButton2.setText("Finaliza");
  jButton2.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
      finaliza(e);
    }
  });
  getContentPane().add(jPanel1);
  jPanel1.add(jPanel2, BorderLayout.NORTH);
  jPanel2.add(jLabel1, BorderLayout.WEST);
  jPanel2.add(jTextField1, BorderLayout.EAST);
  jPanel1.add(jPanel3, BorderLayout.CENTER);
  jPanel3.add(jPanel5, BorderLayout.NORTH);
  jPanel5.add(jLabel2, BorderLayout.WEST);
  jPanel5.add(jTextField2, BorderLayout.EAST);
  jPanel3.add(jPanel7, BorderLayout.CENTER);
  jPanel7.setLayout(new BorderLayout());
  jPanel7.add(painel1, BorderLayout.WEST);
  jPanel7.add(painel2, BorderLayout.EAST);
  textoNome.setColumns(24);
  textoValor.setColumns(24);
  painel1.getViewport().add(textoNome);
  painel2.getViewport().add(textoValor);
  for(int i=0; i<nomeAtributos.size(); i++){
     textoNome.append(nomeAtributos.elementAt(i).toString()+"  \n");
     if(valorAtributos.elementAt(i) != null)
       textoValor.append(valorAtributos.elementAt(i).toString()+" \n");
     else
       textoValor.append("null  \n");
  }
  jPanel1.add(jPanel4, BorderLayout.SOUTH);
  jPanel4.add(jButton1, null);
  jPanel4.add(jButton2, null);
  this.setSize(600,350);
  this.show();
}
```

```
  void continua(MouseEvent e) {
    this.hide();
  }

  void finaliza(MouseEvent e) {
    System.exit(0);
  }

}
```

Referências Bibliográficas

[BUZ1998] BUZATO, L., RUBIRA, C. **Construção de sistemas orientados a objetos confiáveis.** Livro da XI Escola de Computação. Rio de Janeiro, RJ. Julho, 1998.

[CHI1995] CHIBA, S. **A metaobject protocol for C++.** ACM SIGPLAN Notices, v.30, n.10, p.482-501, October 1995. Trabalho apresentado no OOPSLA, 1995, Austin, Texas.

[CHI1996] CHIBA, S. **OpenC++ Programmer's Guide for Version 2.** Outubro, 1998. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/~chiba/. Data de aquisição: fevereiro, 2000.

[CHI1996a] CHIBA, S. **A study of compile-time metaobject protocol.** Outubro, 1998. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/~chiba/. Data de aquisição: fevereiro, 2000.

[CHI1998] CHIBA, S. **Javassist – A reflection-based programming wizard for Java.** In proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java. October, 1998.

[CHI2000] CHIBA, S. **Welcome to Javassist 0.6**. Abril, 2000. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/. Data de aquisição: abril, 2000.

[CYS1997] CYSNEIROS, L.; LEITE J. **Definindo requisitos não funcionais.** XI Simpósio Brasileiro de Engenharia de Software. Fortaleza, CE. Outubro, 1997.

[FER1989] FERBER, J. **Computational Reflection in Class Based Object-Oriented Languages.** SIGPLAN Notices, New York, v. 24, n. 10, p. 317-326, Oct. 1989. Trabalho apresentado no OOPSLA, 1989, New Orleans, Louisiana.

[GOL1998] GOLM, M.; KLEINÖDER, J. **MetaXa and the future of Reflection.** Endereço eletrônico: http://www.informatik.uni-erlangen.de/. Data de aquisição: março, 2000.

[KIC1991] KICZALES, G., RIVIERIS, J. and BODROW D. **The art of the metaobjects protocol.** Cambridge : MIT Press, 1991.

[KIC1996] KICZALES G., PAEPCKE, A. **Open implementations and metaobject protocols.** Relatório de pesquisa da Xerox Corporation. 1996.

[KLE1996] KLEINÖDER, J.; GOLM, Michael. **MetaJava: An Efficient Run-Time Meta Architecture for Java.** Adquirido no endereço eletrônico: http://www.informatik.uni-erlangen.de/. Data de aquisição: março, 2000. Published in the Proceedings of the International Workshop on Object Orientation in Operating Systems – IWOOOS, 1996, Seattle, Washington.

[LIS1997] LISBÔA, M. **Arquiteturas de meta-nível.** Tutorial XI Simpósio Brasileiro de Engenharia de Software. Fortaleza, CE. Outubro, 1997.

[MAE1987] MAES P. **Concepts and experiments in computacional reflection.** ACM SIGPLAN Notices, 22(12): 147 – 155, Dec. 1987.

[MAL1992] MALENFANT, J.; DONY, C.; COINTE, P. **Behavioral reflection in a prototype-based language.** In: International Workshop on New Models for Software Architecture / Reflection and Meta-Level Architectures, 1992. Tokyo, Japan. Proceedings....: [s.1.:s.n.], 1992, p. 143-153.

[NAK1992] NAKAJIMA, S. **What makes a language reflective and how ?** In: International Workshop on New Models for Software Architecture / Reflection and Meta-Level Architectures, 1992, Tokyo, Japan. 1992, p. 125-136.

[POS2000] POSSAMAI, C. **Protótipo de gerenciamento de arquivos para ambiente distribuído.** Blumenau, 2000. Trabalho de conclusão de curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

[RAT2000] **Rational.** Endereço eletrônico da empresa Rational. Endereço eletrônico: http://www.rational.com. Data de aquisição: Novembro, 1999.

[STE1994] STEEL, L. **Beyond objects.** European Conference on object-oriented programming. Bologna, Italy. 1994, p. 1 – 11. (Lecture notes in computer science n. 821).

[TAT1999] TATSUBORI, M. **An extension mechanism for the Java language.** A dissertation submited to the Graduate School of Engineering University of Tsukuba. 1999.

[TAT2000] TATSUBORI, M. **Welcome to OpenJava 1.0**. Janeiro, 2000. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/. Data de aquisição: fevereiro, 2000.

[WEL1998] WELCH, I.; STROUD, Robert. **Dalang – A reflective Java extension.** Endereço eletrônico: http://www.cs.ncl.ac.uk/people/i.s.welch. Data de aquisição: abril, 2000.

[WU1996] WU, S. **Reflective Java: The design, implementation and applications.** FTP: Architecture Projects Management Limited Cambridge. UK, 1996. Endereço eletrônico: http://www.ansa.co.uk/. Data de aquisição: fevereiro, 2000.

[WU1997] WU, S. **Reflective Java: making Java even more reflexible.** FTP: Architecture Projects Management Limited. Cambridge, UK, 1997. Endereço eletrônico: http://www.ansa.co.uk/. Data de aquisição: fevereiro, 2000.