UTILIZAÇÃO DE REFLEXÃO COMPUTACIONAL EM UM GERENCIADOR DE ARQUIVOS DISTRIBUÍDOS

Maurício Capobianco Lopes (FURB)
mclopes@furb.rct-sc.br

Fabrício Jailson Barth
fjbarth@inf.furb.rct-sc.br

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. A implementação é realizada utilizando as ferramentas OpenJava e Javassist.

Palavras-Chave: Orientação a Objetos, Reflexão Computacional.

1 INTRODUÇÃO

Sistemas modernos requerem um alto grau de confiabilidade, disponibilidade, performance e tolerância a falhas. A 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 meta-objetos (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.

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. Assim, pretende-se atribuir 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 são estudadas as ferramentas OpenJava [TAT2000] e Javassist [CHI2000], que permitem implementar conceitos de reflexão computacional sobre a linguagem Java em tempo de compilação e execução, respectivamente.

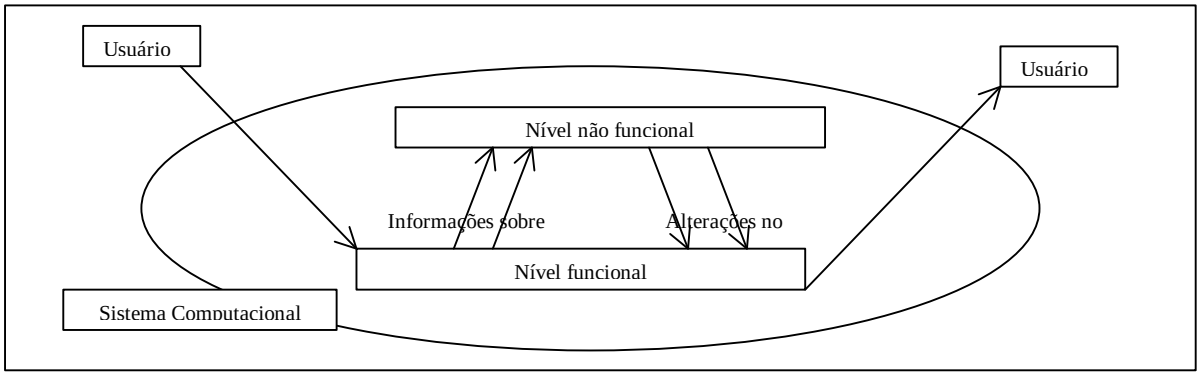
Mais detalhes sobre este trabalho podem ser obtidos em [BAR2000]

2 REFLEXÃO COMPUTACIONAL

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.

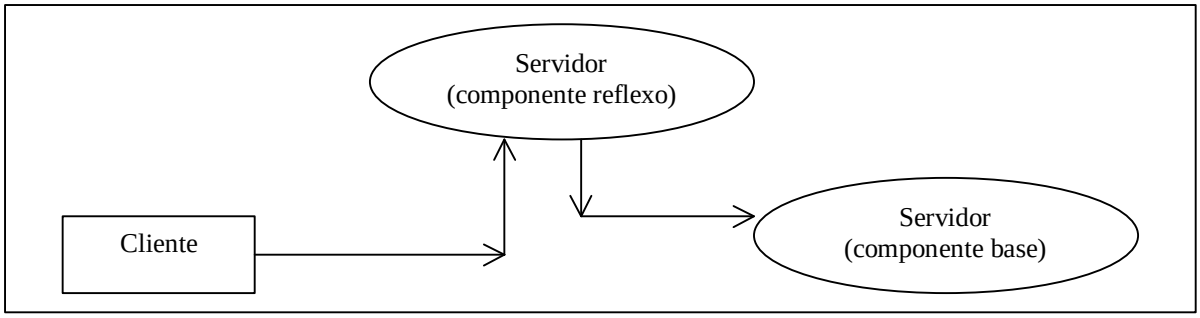
Na figura 1, pode-se visualizar, de forma genérica, o processo de reflexão em um sistema computacional. O sistema computacional é dividido em dois ou mais níveis. O ambiente externo (usuário) envia uma mensagem ao sistema computacional. Esta mensagem é tratada pelo nível funcional, que é responsável por executar corretamente a tarefa, e o nível não funcional realiza a tarefa de gerenciar o funcionamento do nível funcional.

Figura 1: Visualização genérica de um sistema computacional reflexivo



Outra forma de caracterizar reflexão computacional é demonstrado na figura 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.

Figura 2: Exemplo de modelo reflexivo



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

Na reflexão computacional a representação de classes, métodos, atributos e objetos são redefinidas por meio de metaclasses e metaobjetos. O nível no qual as metaclasses e metaobjetos estão dispostos é chamado de meta-nível.

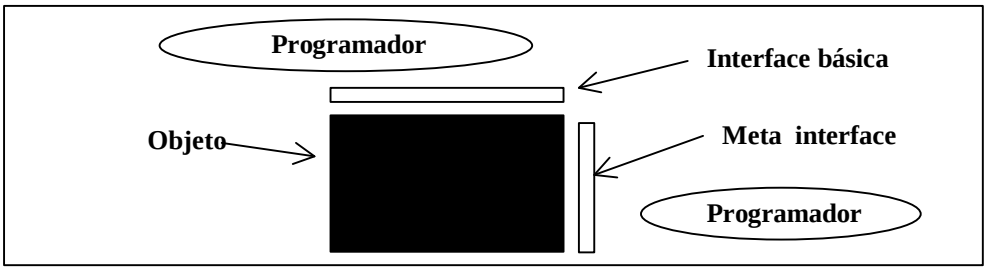
Um sistema reflexivo consolida-se depois de completado três estágios: (a) obter uma descrição abstrata do sistema tornando-a suficientemente concreta para permitir operações sobre ela, (b) utilizar esta descrição concreta para realizar alguma manipulação e (c) modificar a descrição obtida com os resultados da reflexão computacional, retornando a descrição modificada ao sistema. Para realizar tal serviço é necessário dispor de algum mecanismo que permita 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 protocolo de metaobjetos (*Metaobject Protocol* - MOP).

2.1 PROTOCOLO DE METAOBJETOS

Em sistemas orientados a objetos convencionais o analista faz a modelagem dos objetos de forma fechada. A medida que os programadores vão utilizando o objeto, encontram maiores dificuldades para adaptar o objeto às suas necessidades, e em reutilizar este objeto para aplicações distintas. Se ao objeto 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.

Segundo [KIC1996], implementações abertas possibilitam ao programador ter acesso a implementação, poder inspecionar, entender e modificar a implementação de acordo com as características que deseja implementar, obtendo maior funcionalidade e desempenho do componente. Quando um objeto contêm, além de sua interface básica, uma meta interface que dá acesso a sua implementação, este objeto é chamado de objeto aberto. O modelo de um objeto aberto é apresentado na figura 3.

Figura 3: Objeto aberto



Fonte:[KIC1996]

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 meta-objeto, formando o protocolo de meta-objetos.

Segundo [KIC1991], o conjunto de métodos da meta-interface formam o protocolo de meta-objetos, que pode ser dividido em: (a) *protocolo de introspecção* que consiste de

mecanismos que permitem obter informações sobre as classes, suas herdeiras, seus nomes e dados sobre a estrutura, (b) *protocolo de invocação* que é a capacidade da metaclasse em invocar um método da sua classe de nível base de modo direto e (c) *protocolo de intercessão* que possibilita realizar mudanças de comportamento ou de estrutura do objeto base.

A introspecção e a invocação fornecem o suporte para que aconteça a reificação ou materialização, que segundo [LIS1997], é "*a transformação de informações sobre a execução de um programa orientado a objetos em dados disponíveis ao próprio programa.*"

Segundo [KIC1996] os protocolos de metaobjetos podem ocorrer em tempo de compilação (*Compile Time MOP)*, onde a estrutura e comportamento das classes de nível inferior são modificados no momento da compilação - as ferramentas que implementam este protocolo possuem um compilador próprio - ou em tempo de execução (*Run Time MOP)* não necessitando de compiladores, mas apenas do próprio protocolo que fornece a possibilidade de reflexão, ocorrendo a mudança de comportamento e estrutura do objeto sem alterar o seu código fonte.

2.2 MODELOS DE REFLEXÃO

Segundo[BUZ1998] os modelos de reflexão computacional podem ser divididos em (a) modelo de metaclasses, onde todo o objeto instanciado pela mesma classe terá o mesmo comportamento reflexivo de acordo com o que está modelado em sua metaclasse - neste modelo pode ocorrer tanto reflexão estrutural como comportamental – (b) modelo de metaobjetos onde a associação é por objetos e não por classes, isto é, qualquer objeto pode ser associado a um ou mais metaobjetos, que definem, implementam ou participam de diferentes formas da execução dos objetos de nível-base e (c) modelo de metacomunicações quando cada mensagem é materializada como um objeto, ou seja, existe uma classe Mensagem que representa as mensagens.

2.3 MECANISMOS DE EXTENSÃO REFLEXIVOS PARA A LINGUAGEM JAVA

Neste trabalho são apresentadas duas extensões da linguagem Java que possibilitam sua utilização em sistemas reflexivos, proporcionando reflexão estrutural e comportamental e possuindo um metaprotocolo. A diferença principal entre as duas ferramentas, é que uma proporciona reflexão em tempo de compilação e a outra em tempo de execução:

2.3.1 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 diferença do compilador regular Java para o compilador do OpenJava, é que o compilador do OpenJava recorre a bibliotecas de meta nível além de bibliotecas regulares. 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].

2.3.2 JAVASSIST

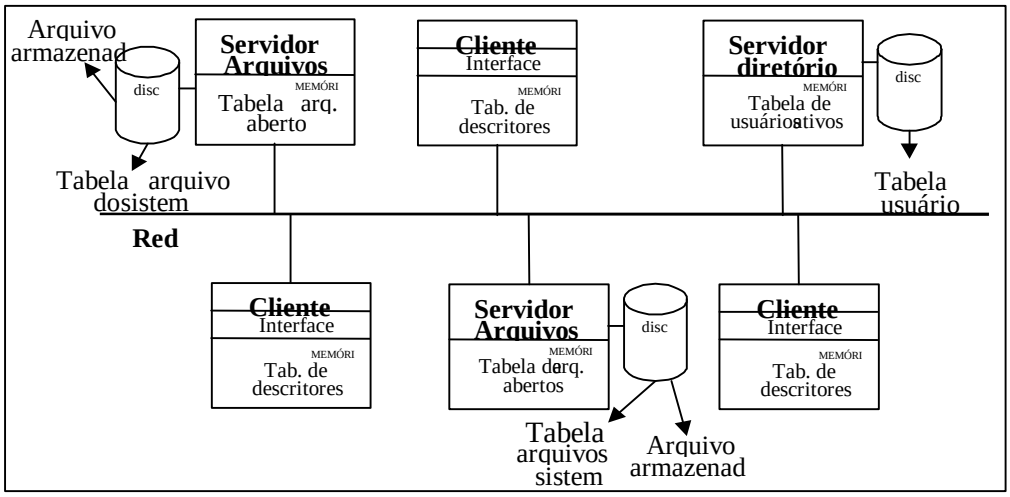
Segundo [CHI1998], Javassist é uma ferramenta que visa facilitar o desenvolvimento de aplicações que utilizam a linguagem Java. Permite que programadores possam automatizar alguns tipos de definições de classe e permite a realização de reflexão computacional em tempo de execução. Javassist possui uma API que possibilita ao programador uma abstração maior no desenvolvimento de suas aplicações e elaboração de novas estruturas genéricas. Segundo [CHI1998], Javassist não é uma ferramenta reflexiva tão completa quanto as ferramentas MetaXa [GOL1998] e Dalang [WEL1998], mas devido a sua API simples, ao modelo que propõe e a não utilização de qualquer compilador ou ferramenta para suporte em tempo de execução, ela é considerada 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].

3 GERENCIADOR DE ARQUIVOS DISTRIBUÍDOS

O gerenciador de arquivos distribuídos [POS2000] possibilita que o usuário acesse localmente ou remotamente os arquivos de forma transparente, sendo que esta transparência 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 (figura 4). Ele 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.

Figura 4: Estrutura geral do gerenciador de arquivos distribuídos



Fonte : [POS2000]

Com o objetivo de validar os conceitos vistos anteriormente, será atribuído a este gerenciador novas tarefas para adaptar funcionalidades já existentes e adicionar características não funcionais ao sistema.

4 DESENVOLVIMENTO

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

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

4.1 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 1.

Quadro 1: Comportamento atual dos métodos *retornaServidor* e *retornaDominio*

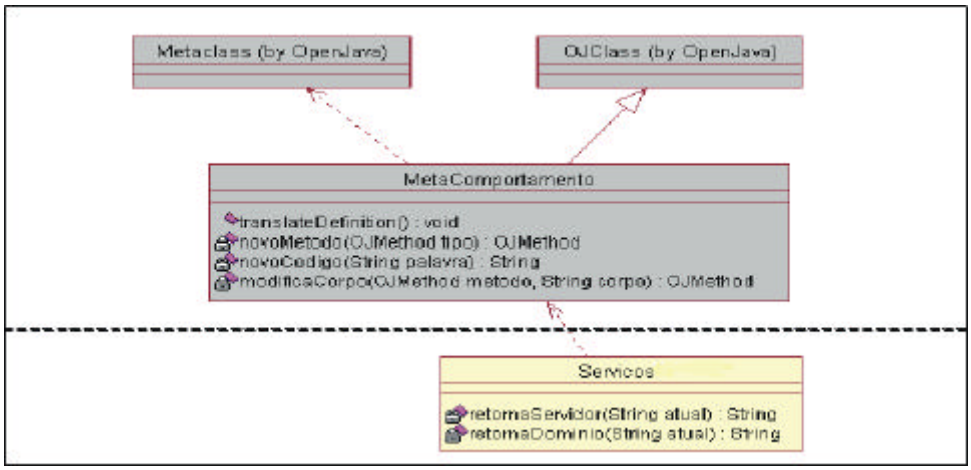
```
private String retornaServidor (String atual) {
    if (Servidor1 == atual)
          return Servidor2;
    else
          return Servidor1;
}
private String retornaDominio (String atual) {
    if (Servidor1 == atual)
          return Dominio2;
    else
          return Dominio1;
}
```

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 1) se obtêm qual servidor foi utilizado por último para armazenar determinado arquivo.

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.

Na figura 5 pode ser vista que a classe 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: Diagrama de classes da tarefa de alteração do comportamento



A metaclasse *MetaComportamento* possui os seguintes métodos:

a) *novoMetodo(OJMethod)*: retorna o método *espacoDisco*, que será um novo método pertencente a classe base *Servicos*, que retorna a máquina com o maior espaço livre em disco. O parâmetro *OJMethod* permite utilizar as características do método passado para criar o método novo;

b) *novoCodigo(String)*: retorna o novo código fonte que irá sobrepor os já existentes nos métodos *retornaServidor* e *retornaDominio*. O parâmetro é apenas para diferenciar o método *retornaServidor* do método *retornaDominio*;

c) *modificaCorpo(OJMethod,String)*: adiciona à classe base o método alterado. Ele recebe as características (*OJMethod*) e o corpo do novo método (*String*);

d) t*ranslateDefinition()*:realiza a reflexão sobre os métodos reflexivos.

A metaclasse *MetaComportamento* é codificada utilizando-se pacotes da linguagem Java e do pré-processador OpenJava. Na figura 5 observa-se que a metaclasse *MetaComportamento* herda características da classe *OJClass* (como toda metaclasse em OpenJava) e é instanciada a partir da classe *Metaclass*. Já na classe base *Servicos*, é necessário alterar a assinatura da classe, de *class Servicos extends Thread*, para *class Servicos extends Thread instantiates MetaComportamento*.

Na compilação 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 (quadro 2).

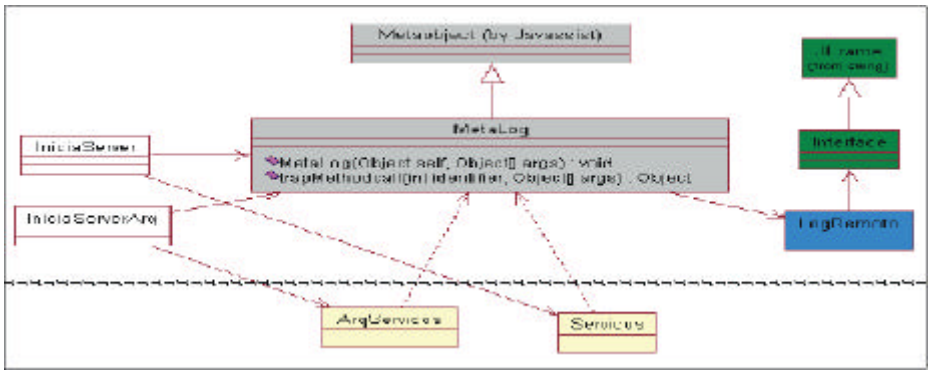
Quadro 2: Novo comportamento dos métodos transformados

```
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;}
}
```

4.2 *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* é implementado através de um metaprotocolo em tempo de execução. Na Figura 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 6: Diagrama de classes da tarefa para geração de *log*



A metaclasse *MetaLog*, possui os métodos:

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

b) *trapMethodcall (int identifier, Object[] args)*: intercepta as mensagens direcionadas a uma classe base. Os parâmetros são 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. Assim, ela obtêm informações sobre as classes de nível base e envia estas informações para a classe *LogRemoto*. A classe *LogRemoto* redireciona a mesma informação para a classe *Interface*, que torna público os resultados. A metaclasse *MetaLog* é herdeira da classe *Metaobject*, característica comum a todas as metaclasses que utilizam o metaprotocolo da ferramenta Javassist.

No quadro 3 pode-se visualizar o código fonte descrito no método construtor da metaclasse *MetaLog*, que é executado toda vez que uma classe do nível base é instanciada.

Quadro 3: 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); }
}
```

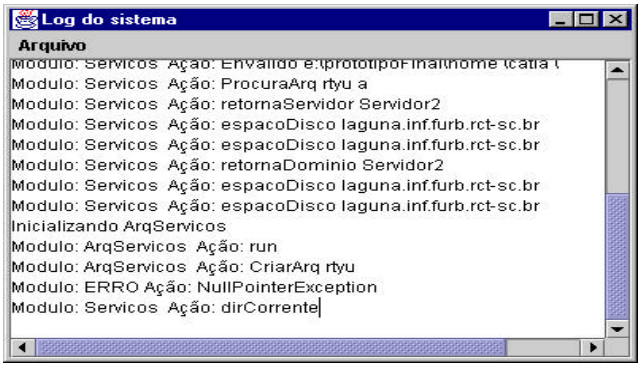
No código apresentado no quadro 4, deve-se verificar que a classe *IniciaServer* associa a classe *Servicos* com a metaclasse *MetaLogServerArq*, e logo após determina que a classe *Server* inicie a sua execução.

Quadro 4: Código fonte da classe *IniciaServer*

```
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);
   }}
```

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 o resultado do *log*. No momento em que for iniciada a execução do LogRemoto, a interface mostrada na figura 7 é instanciada, e a medida do tempo vai recebendo novas mensagens, que visam demonstrar a situação do sistema gerenciador de arquivos.

Figura 7: Interface do *log* do sistema

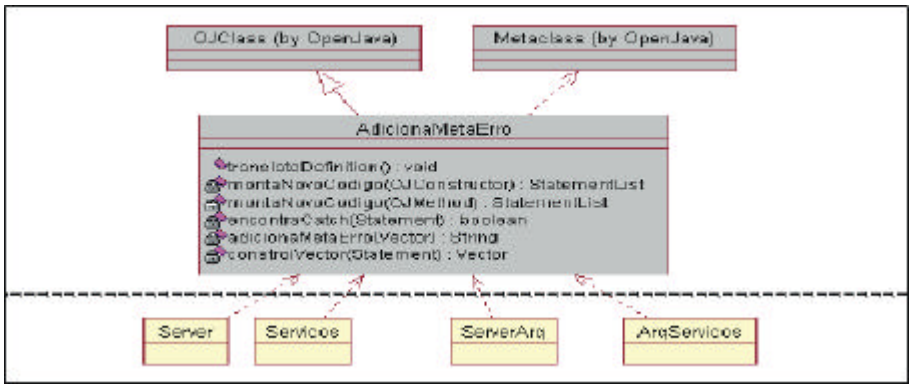


4.3 MÓDULO PARA TRATAMENTO DE ERROS

Esta tarefa tem como objetivo a construção de um módulo que trate os erros ocorridos durante a execução do sistema. O módulo, além de retornar o erro ocorrido, retorna o estado do objeto onde o erro ocorreu e, devido a 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 8 pode-se visualizar o diagrama de classes, para alterar as classes de nível base em tempo de compilação.

Figura 8: Diagrama de classes do módulo de erro em tempo de compilação



Nesta figura pode-se visualizar a metaclasse *AdicionaMetaErro*, que deve adicionar, a todas as classes de nível base, a capacidade de, quando ocorrer um erro, poder instanciar um objeto da classe *MetaErro.* Esta metaclasse possui os métodos:

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

b) *montaNovoCodigo (OJConstructor atual)*: 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)*: tem 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)*: caso encontre um bloco de tratamento de exceções inserido no *Statement* passado como parâmetro, retorna *true*, senão retorna *false*;

e) *constroiVector (Statement atual)*: retorna um Vetor, onde cada elemento é uma palavra que pertence ao *Statement* passado pelo parâmetro;

f) *adicionaMetaErro (Vector atual)*: recebe o Vetor resultado do método anterior e adiciona, na área de tratamento de exceções, a chamada para a classe *MetaErro*.

Quando é encontrado um bloco de tratamento de exceção, é posto em seu bloco uma chamada para a classe *MetaErro*, que em tempo de execução, quando houver um erro, poderá ser instanciada. As chamadas são embutidas dentro dos blocos de levantamento de exceções da

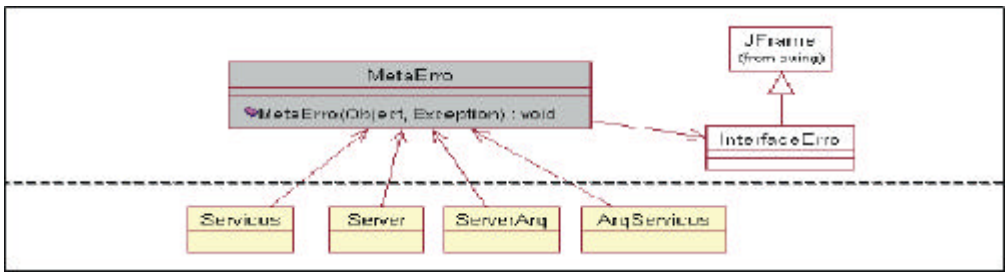
linguagem Java, como pode ser visto no quadro 5. Durante a compilação do sistema é utilizado o metaprotocolo do pré-processador OpenJava.

Quadro 5: Chamada da classe *MetaErro*

```
try{
   //codigo com possibilidades de erro
}catch (Exception e){
   new MetaErro(this,e);
}
```

O diagrama de classes, em tempo de execução, do módulo de tratamento de erros, (figura 9) 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.

Figura 9: Diagrama de classes do módulo de erros em tempo de execução



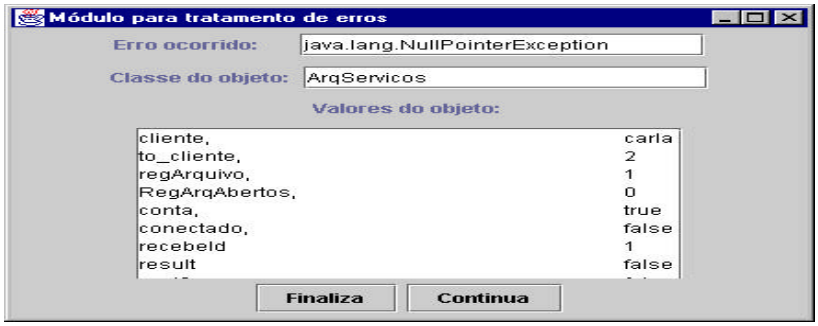
A classe *MetaErro* possui apenas um método construtor, pois quando ocorre um erro a classe *MetaErro* é instanciada e é no método construtor que ocorre todos os controles para futura visualização dos dados na classe *InterfaceErro*. Após transformado o sistema, quando ocorrer qualquer espécie de erro, 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 para realizar introspecção sobre a classe onde ocorreu o erro e sobre a exceção (quadro 6). Depois de compiladas as classes, pode-se executá-las normalmente utilizando a máquina virtual Java padrão.

Quadro 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);
      }
}
```

Na figura 10 pode-se visualizar o funcionamento da classe *InterfaceErro*.

Figura 10: 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 o erro que ocorreu, em que módulo e o valor dos atributos do objeto. Este módulo apenas se utiliza da capacidade introspectiva da linguagem Java, ou seja, apenas obtêm valores, não os modificando.

5 CONCLUSÕES

Este trabalho adicionou novas funcionalidades ao gerenciador de arquivos, proporcionando uma melhor performance e alguns serviços extras para o sistema.

Através do protótipo implementado e dos exemplos listados, pode-se verdadeiramente perceber as vantagens da utilização de reflexão computacional em sistemas orientados a objetos. Em teoria, o modelo reflexivo, aliado a arquitetura de meta-níveis, visa facilitar o trabalho do desenvolvedor, porém o conjunto de modelos teóricos, ainda não é totalmente correspondido na prática. Pôde-se perceber, durante o trabalho, que as ferramentas disponíveis ainda não satisfazem, de forma completa, a teoria descrita.

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. Pôde-se verificar, neste trabalho, 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, por exemplo, tolerância a falhas.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[BAR2000] BARTH, Fabricio J. **Utilização de reflexão computacional para implementação de aspectos não funcionais em um gerenciador de arquivos distribuídos**. Blumenau, 2000. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Univ. Regional de Blumenau.

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

[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**. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/. Aquisição: abril, 2000.

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

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

[POS2000] POSSAMAI, C. **Protótipo de gerenciamento de arquivos para ambiente distribuído.** Blumenau, 2000. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Univ. Regional de Blumenau.

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

[TAT2000] TATSUBORI, M. **Welcome to OpenJava 1.0**. Endereço eletrônico: http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/. Aquisição: fev., 2000.

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