Elementos Básicos de um Proxy da API de Java™ Descritos através de um Exemplo Simples

Getúlio Moreira
DICC – IME - Universidade do Estado do Rio de Janeiro
getuliogsm@hotmail.com

Maria Alice Silveira de Brito
DICC – IME - Universidade do Estado do Rio de Janeiro
malice@ime.uerj.br

Abstract

In this paper we identify the difficulties associated with the correct understanding of the dynamic proxy Java API. The main obstacles are: the diversity of elements working together for the proxy to achieve its goal, and the dependencies that this functionality has on reflection mechanisms, giving the opportunity to hide code parts from the application on a meta level. We have created a very simple example, applying the proxy in its more primitive role as a message interceptor. This example is described with the purpose of make it clear to the programmer how to identify the four main elements that are the agents of each action regarding creation and invocation: the proxy class, the proxy instance, the invocation handler and the target object. Beyond that, we present the regions where code can and/or must be introduced identifying the base and meta levels of programming.

Resumo

Neste trabalho, identificamos as causas das dificuldades de entendimento do uso de dynamic proxy da API de Java, sendo as principais a diversidade de elementos que entram em cena, durante o funcionamento do proxy, e, os artifícios desse funcionamento, que dependem dos conceitos de reflexão, o qual dá oportunidades à programação de incluir código, em partes (meta-nível), que ficam escondidas do código da aplicação. Assim, criamos um exemplo bem simples, empregando o proxy no seu papel mais primitivo possível, quando implementa um interceptor de mensagens. Esse exemplo foi acompanhado de descrições, com o propósito de permitir a um programador identificar com clareza os quatro elementos principais, que são os agentes de cada ação, no curso da criação e da invocação: a classe proxy, a instância de proxy, o invocation handler e o objeto alvo, além das regiões em que deve/pode introduzir código, identificando os níveis de programação base e meta.

1. Introdução

No contexto de linguagens de programação, em particular, bibliotecas que implementam facilidades de sistema para a linguagem Java, apresentamos exemplos simples que empregam a facilidade *proxy*, oferecida desde a versão 1.3 da Jsdk.

O *proxy* é considerado um dos padrões de projeto, na Engenharia de Software, que tem como finalidade permitir que clientes de um serviço utilizem um representante do componente que oferece o serviço . Segundo os autores, em [1, 2], seu uso aumenta a eficiência, a segurança e facilita o acesso ao serviço. Um *proxy* pode substituir um servidor quando esse enfrenta problemas e permite criar uma independência de endereçamento e implementação do servidor. O uso mais comum de um *proxy* é em contexto de sistemas distribuídos, quando um cliente precisa acessar um serviço de um outro componente remoto. O acesso direto é tecnicamente possível, mas pode não ser a melhor opção, na maioria das vezes, em que um cliente precisa das independências apontadas acima.

A motivação para esse trabalho surgiu das dificuldades de entendimento, nas consultas a documentos disponíveis sobre *dynamic proxy* da API de Java, durante a implementação de um interceptor de mensagens. Essas dificuldades ficaram associadas à diversidade de elementos que entram em cena, durante o funcionamento do *proxy*, e, além disso, alguns truques desse funcionamento dependem da reflexão [3,4], um paradigma de programação que dá oportunidades à programação de incluir código, em partes (meta-nível), que ficam escondidas do código da aplicação. Quando o programador emprega o *proxy*, ele necessita saber de que lado ou em que elemento deve efetuar suas intervenções, para especificar ou configurar determinado aspecto do *proxy*. Os autores desses documentos consultados tiveram propósitos, que variaram desde o compromisso de oferecer uma especificação, passando por tutoriais, oferecendo exemplos que ajudariam o entendimento e até artigos que apresentam trabalhos que empregaram *dynamic proxy* da API de Java, mas não fizeram uma abordagem à luz do funcionamento, dos elementos envolvidos e das características de reflexão envolvidas.

Nosso objetivo não foi o de fazer uma abordagem completa e rigorosa sobre a facilidade *dynamic proxy* e sim o de examinar o seu mais simples emprego, quando é implementado um interceptor de mensagens primitivo, para facilitar o entendimento dos programadores que necessitem dessa facilidade. Assim, seria interessante a um programador contar com um documento que reunisse as contribuições didáticas dos documentos consultados, para facilitar o entendimento do emprego de *dynamic proxy* da API de Java. O resultado dessa reunião deveria ser acompanhado de uma enumeração dos elementos (classes e instâncias) envolvidos e da identificação das regiões e elementos que deveriam receber as intervenções do programador, a serem efetuadas em ambos os lados da programação, no nível básico (aplicação) e no nível meta (reflexão) [3].

Neste trabalho, descrevemos os elementos envolvidos na facilidade *dynamic proxy* da API de Java. Além disso, chamamos a atenção para a visibilidade nos níveis meta, base e os que ficariam invisíveis, como é o caso do *class loader* e da classe *proxy*, associada às interfaces. Um outro destaque foi o de fazer o acompanhamento da seqüência de passos dados pela cadeia de elementos que tomam parte em uma invocação e no retorno.

Esse trabalho é apresentado nas três seções, a seguir: 2. Trabalhos Relacionados; 3. Exemplo simples do uso de um *proxy*; e 4. Conclusões.

2. Trabalhos Relacionados

Em setembro de 2005, quando o nosso trabalho já tinha sido desenvolvido, a Sun disponibilizou o capítulo 4 *Using Java's Dynamic Proxy*, do livro *Java Reflection in Action* [5], em que uma das intenções dos autores foi a de popularizar o paradigma da reflexão entre os programadores.

Durante nossa experiência, constatamos que os autores das publicações consultadas tiveram propósitos, que variaram desde o compromisso de oferecer uma especificação, passando por tutoriais, oferecendo exemplos que ajudassem o entendimento e até artigos que apresentam trabalhos que empregaram *dynamic proxy* da API de Java.

Consideramos que seria interessante a um programador contar com um documento que organizasse essas contribuições existentes, com a intenção de facilitar o seu entendimento do emprego de *dynamic proxy* da API de Java. O resultado dessa reunião deveria ser acompanhado de uma enumeração dos elementos (classes e instâncias) envolvidos e das regiões e elementos que deveriam receber as intervenções do programador, a serem efetuadas em ambos os lados da programação, no nível básico (aplicação) e no nível meta (reflexão).

As primeiras documentações consultadas foram as especificações correspondentes a **`Class Proxy`** [6] e **`Interface InvocationHandler`** [7] . Nesses textos, devido ao compromisso de ser apenas uma especificação, as classes e suas instâncias são, brevemente, descritas, não sendo esclarecidas as intervenções necessárias por parte do programador, quando a facilidade *proxy* está sendo empregada.

Em busca de uma descrição mais estendida de como seria possível ao programador aproveitar a interceptação, para incluir algum código adicional, antes

da mensagem chegar ao objeto alvo, foi consultado o documento *Dynamic Proxy Classes* [8], cujo propósito é o de servir como um tutorial ao programador. Ele descreve o que é uma *dynamic proxy class* e faz o elo com os outros elementos: as interfaces que ela implementa e a sua instância; o *class loader;* a classe do *invocation handler* e a sua instância; e o objeto alvo. Nessa descrição, em especial, a do funcionamento desses elementos, durante uma invocação de um método sobre uma instância de uma classe *dynamic proxy*, duas ações aparecem na ordem inversa, de forma consecutiva, como a seguir. A descrição precedente refere-se à invocação intermediária do método exclusivo **`invoke`** da instância do *invocation handler* e a seguinte refere-se a recodificação do método em um objeto da classe **`java.lang.reflect.Method`**. Essa inversão na ordem das descrições pode confundir o programador, que neste ponto ainda não está muito seguro dos papéis assumidos por cada elemento desse conjunto.

A ordem de ações entre a invocação do emissor e a invocação efetiva ao objeto alvo é a seguinte: primeiro a instância do *proxy* recodifica a invocação em um objeto da classe **`java.lang.reflect.Method`** e em uma lista de argumentos, depois passa esses objetos como argumentos ao método **`invoke`** do *invocation handler*, que, então, fará a invocação efetiva ao objeto alvo.

Nas seções posteriores desse segundo documento consultado, é descrito como se cria uma classe *proxy* e sua instância, acompanhadas de suas propriedades. Na descrição das propriedades da instância da classe *proxy*, embora sua maior parte seja dedicada ao *invocation handler*, particularmente, ao seu método **`invoke`**, não se chega ao fim dessa leitura, com clareza sobre o que e aonde o programador deverá intervir, para incluir o código pretendido. A condição de que a classe *proxy* será obtida automáticamente não vai sendo alertada ao longo da descrição.

Acompanhando-se o primeiro exemplo, que é bem simples, a classe do *invocation handler* é chamada **`DebugProxy`**, suscitando, ao leitor, dúvidas como as seguintes: 1) se ela é uma classe que vai ajudar a construir as classes *dynamic proxy*, em tempo de execução, e o *invocation handler* associado, aproveitando os métodos; ou 2) se é a própria classe *proxy*, que incorpora os mecanismos de *invocation handler* e, em tempo de execução, para cada lista de interfaces, é gerada uma ascendente dessas.

Com essas dúvidas em mente, o leitor tenta acompanhar o segundo exemplo, que explora mais poderes da facilidade de *dynamic proxy*, não conseguindo ainda esclarecer as dúvidas acima. Em primeiro lugar, a finalidade desse exemplo parece ser a de gerar subclasses de **`Delegator`**, dependendo da interface do método invocado. Em segundo lugar, o próprio conceito de linguagem de programação, chamado delegação, leva o leitor a desconfiar se ele é um elemento interno do pacote oferecido por Java ou se é um elemento semântico da programação.

O terceiro documento consultado [9] é um artigo que trata de interposição de método, sendo comparadas as soluções desenvolvidas em C e Java. Nesse artigo é dito que pela facilidade de *dynamic proxy*, oferecida pela API de Java, o programador pode criar a sua própria classe *proxy*, que intercepta a chamada do método, faz algum pré-processamento, invoca o método original e, em seguida, faz o pós-processamento. Após essa observação, apresenta a implementação de um método **`invoke`**, para mostrar essas três fases, incluídas no código desse método. Como a classe *Proxy* é criada automaticamente, neste ponto, o programador permanece com dúvidas sobre à qual instância ou classe o método **`invoke`** pertence, se ao *proxy* ou ao *invocacation handler*. Em seguida, um exemplo, incluindo esse método **`invoke`**, é apresentado por completo e essas dúvidas permanecem sem respostas.

No quarto documento, intitulado Using Dynamic Proxies to Layer New Functionality Over Existing Code [10], é apresentado um exemplo, dando orientação ao programador sobre cada passo que deve ser efetuado, nomeando os elementos nesses passos e quem é o responsável pelo que vai acontecer na criação e nas invocações, como podemos ver, a seguir.

No texto, é explicado que um *dynamic proxy* é uma classe especial criada, em tempo de execução, pela Java Virtual Machine (JVM). O programador pode pedir uma classe *proxy* que implementa qualquer interface ou mesmo um grupo de interfaces, pela seguinte chamada:

```
Proxy.newProxyInstance(
```

```
ClassLoader classLoaderToUse,
Class[] interfacesToImplement,
InvocationHandler
objToDelegateTo)
```

fazendo com que a JVM gere, então, uma nova classe que implementa as interfaces dadas no segundo parâmetro acima, direcionando todas as chamadas ao método exclusivo do **InvocationHandler**:

```
public Object invoke(
Object proxy,
Method meth, Object[] args)
throws Throwable;
```

tudo que o programador tem, então, a fazer é implementar o método **invoke** numa classe que implementa a interface **InvocationHandler**. A classe *proxy* providencia o direcionamento então de todas as chamadas, dispensando o programador dessa responsabilidade.

Com essas indicações, um exemplo simples é apresentado e reproduzido, a seguir:

```
class Logger implements
InvocationHandler {
  private Object delegate;
  public Logger(Object o) {
  delegate = o;
  }

public Object invoke(
Object proxy,
Method meth, Object[] args)
    throws Throwable {
System.out.println(
meth.getName());
try {
return meth.invoke(delegate,
rgs);
  }
catch(
InvocationTargetException e )
{
throw e.getTargetException ();
}
}
}
```

e o método **main** utiliza esse mecanismo como, a seguir:

```
 public static void
main(String[] args) {
  Explorer real = new
ExplorerImpl();
  Explorer wrapper = (Explorer)
      Proxy.newProxyInstance(
        Thread.currentThread().

  getContextClassLoader(),
        new Class[]
{Explorer.class},
        new Logger(real));
  test(wrapper);
 }
```

Podemos notar que, nesse exemplo, o identificador da classe **Logger**, que implementa o **InvocationHandler**, não inclui uma subcadeia **Proxy**, evitando que o leitor seja induzido a pensar que é uma classe estendida de **Proxy**. A construção foi incluída na aplicação da seguinte forma:

1) na criação do *proxy*, foi chamado o método estático:

Proxy.newProxyInstance; e

2) na classe que implementa o **InvocationHandler**, foram implementados o construtor, que apenas preenche a referência efetiva ao objeto alvo das chamadas e o método **invoke**. A programação nesse exemplo deixa a aplicação com um pouco de visibilidade dos aspectos envolvidos no mecanismo de *dynamic proxy*, no código que pede a criação do *dynamic proxy*.

O quinto documento consultado [11] aborda o *dynamic proxy*, fazendo uma analogia com meta-objeto [3], em que a interface pode substituir o protocolo de meta-objeto [4]. Dessa forma, o *dynamic proxy* reside no nível meta e os objetos da aplicação no nível base. O exemplo apresentado nesse documento emprega *dynamic proxy* para implementar um meta-objeto, cuja funcionalidade embutida é a de perceber se está sendo invocado um método que seja capaz de produzir mudança de estado no objeto alvo. Caso uma mudança assim ocorra, ela é tomada como uma notificação de evento. O meta-objeto, então, salva o valor do atributo anterior, invoca o método original, obtém o novo valor do atributo e dispara um evento ao **listener**, que é o

segundo atributo *do meta-objeto*. Os autores referiram-se ao *Invocation Handler* como *proxy* e meta-objeto.

Em relação à analogia entre um *proxy* e um meta-objeto, à primeira vista parece ser procedente, mas, examinando melhor, vamos observar o seguinte: 1) cada meta-objeto deveria ser um elemento único e exclusivo de um objeto alvo, o qual pode ter mais de um meta-objeto, para implementar diversas facilidades de sistema e de interface com usuário; 2) um *proxy* é uma referência a um objeto alvo, e um objeto pode ter várias referências espalhadas em tempo de execução ou até mesmo residindo em atributos de objetos que não se encontrem alocados. Assim, o meta-objeto está associado ao objeto alvo e no seu espaço de endereçamento e o *proxy*, que referencia o objeto alvo, é o valor de um atributo, variável local ou parâmetro, que pode estar do lado do objeto emissor. A analogia mais apropriada é com um protocolo de meta-objeto [4], porque os dois assumem o papel de interceptor.

Notamos pelos documentos que consultamos, acima, que nenhum deles teve o compromisso de apresentar uma abordagem extensa da facilidade de *proxy* com todos os seus elementos. Tal extensão deveria ser resultado de um trabalho bem mais rigoroso. Nossa intenção também não foi a de trazer esse rigor, como já anunciamos, no início dessa seção. Como nos trabalhos anteriores, também apresentamos um exemplo de emprego do *proxy*. A diferença, em relação aos acima comentados, foi a de tentar descrever os elementos envolvidos no seu funcionamento, suas responsabilidades, sua visibilidade, e aonde o programador deve incluir código, relacionado ao *dynamic proxy*.

No exemplo que apresentamos, foi redefinido, na classe que implementa a interface **`InvocationHandler`**, um terceiro método, que é estático, chamado **`newInstance`**, o qual chama o método **`Proxy.newInstanceProxy`**, escondendo um pouco mais da aplicação, os aspectos de *dynamic proxy*.

Para esse exemplo, na seção seguinte, descrevemos os elementos envolvidos na facilidade *dynamic proxy* da API de Java. Além disso, chamamos a atenção para a visibilidade nos níveis meta, base e os que ficariam invisíveis, como é o caso do *class loader* e da classe *proxy*, associada às interfaces. Um outro destaque foi dado ao acompanhamento da seqüência de passos dados pela cadeia de elementos que tomam parte em uma invocação e no retorno.

3. Exemplo simples do uso de um *proxy*

O *proxy* é considerado um dos padrões de projeto, na Engenharia de Software. Padrões são soluções (abstrações de software) para problemas específicos, que ocorrem de forma recorrente em um determinado contexto, que foram identificados a partir da experiência coletiva de desenvolvedores de software [1].

A idéia do padrão *proxy* é permitir que clientes de um serviço utilizem um representante do componente que oferece o serviço. Seu uso aumenta a eficiência, a segurança e facilita o acesso ao serviço. O *proxy* pode substituir o servidor quando ocorrem problemas com o servidor. O *proxy* permite criar uma independência de endereçamento e implementação do servidor [2].

Seu uso mais comum é no contexto de sistemas distribuídos, em que um cliente precisa acessar um serviço de um outro componente remoto. O acesso direto é tecnicamente possível, mas pode não ser a melhor opção.

Os problemas com o acesso direto são perda de eficiência em tempo de execução, custo alto, porque para cada acesso é necessária uma especificação da parte distribuída, além de não ser seguro. Mas o pior dos problemas é a dependência, tornando o código cliente particular a um endereço de rede e com poluições nas invocações dos métodos do servidor.

Consideramos que esse comportamento pode ser generalizado para qualquer tipo de funcionalidade paralela e distinta da semântica da aplicação, como, por exemplo, em necessidades de monitoração, configuração, depuração de programas, entre outras. Uma funcionalidade assim pode ser agregada a um objeto da aplicação, dando a oportunidade de introduzir pré e pós-processamento durante a invocação de um método.

Para o alcance dessa interceptação, em Java, o funcionamento do *proxy* entra em cena. Os pontos de

introdução desses tipos de código são no pré-processamento e pós-processamento da invocação, que se situam no método **invoke** do *invocation handler*, respectivamente, antes e após a invocação efetiva do método ao objeto alvo.

Na versão de Java J2SE v1.3, foi introduzida a *dynamic proxy class* no pacote de Java *Reflection*. Uma *dynamic proxy class* implementa, em tempo de execução, uma lista de interfaces especificadas. Quando um método de uma dessas interfaces é invocado sobre uma instância dessa *dynamic proxy class*, ele é recodificado (transformado em um objeto da classe **Method**) e despachado para o objeto alvo, por uma interface uniforme.

O objetivo, nesta seção, é o de empregar a API de Java para *Dynamic Proxy*, mostrando apenas os aspectos de *proxy* em uma simples interação com um objeto referenciado por um *proxy*. Pretendemos abrir as perspectivas de seu uso como um representante para qualquer objeto servidor, considerado o objeto alvo de uma instrução comum de envio de mensagem. Nossa intenção é a de usá-lo como se fosse uma referência inteligente a esse objeto alvo, introduzindo código que habilite o pré-processamento da mensagem enviada e o pós-processamento do resultado retornado.

As classes **Proxy** e suas instâncias são criadas com o uso de métodos estáticos da classe **java.lang.reflect.Proxy**.

O método **getProxyClass** da classe **Proxy** retorna o objeto **java.lang.Class** para uma classe *proxy*, dados o *class loader* e a lista de interfaces. A classe *proxy* será definida no *class loader* especificado e implementará todas as interfaces supridas. Cada classe *proxy* tem um construtor público com um argumento que é a implementação da interface **InvocationHandler** especificada em **java.lang.reflect**.

Uma alternativa mais simples ao uso de reflection API para o acesso ao construtor público, que cria uma instância de *proxy*, pode ser através da chamada a **newProxyInstance** de **Proxy**. Esse método estático combina as ações de chamada ao método **Proxy.getProxyClass** e de invocação do construtor com um *invocation handler*, como podemos ver, no exemplo [9] da chamada ao método estático (método de classe) **newInstance** da classe **TraceProxy** e sua implementação, nos respectivos trechos de código, abaixo.

```
...
Bar b =
(Bar)TraceProxy.newInstance(new
BarImpl());
...
public class TraceProxy
implements
java.lang.reflect.InvocationHand
ler {
public static Object
newInstance(Object obj) {
return
java.lang.reflect.Proxy.newProxy
Instance(
obj.getClass().getClassLoader(),
obj.getClass().getInterfaces(),
new TraceProxy(obj));
}
...
}
```

Um outro aspecto importante a ser notado é o seguinte: quando uma instância de **proxy** é criada, deve lhe ser aplicada coerção (*cast*) com uma das interfaces que ela implementa, para que depois seja possível invocar qualquer método pertencente a essa interface. No exemplo acima, vemos uma coerção com a *interface* **Bar** aplicada à instância de *proxy* retornada da criação.

Ainda sobre a criação, é apropriado recapitularmos os quatro mais importantes objetos e classes criados ou envolvidos durante a execução da chamada ao método estático **Proxy.newProxyInstance**. Eles são os seguintes:

1. a criação implícita da classe *proxy*, caso ainda não exista;
2. a criação explícita da instância da classe *proxy*, sendo o resultado retornado do método;
3. a criação explícita da instância do *invocation handler*; e
4. o envolvimento do objeto alvo a ser referenciado pelo *proxy* no envio futuro das mensagens, sendo o argumento de **newInstance** e podendo ser criado

no próprio parâmetro, como no exemplo acima, ou referenciando um já existente.

A classe **BarImpl** implementa a interface **Bar**, assim, qualquer invocação de método sobre **b** será interceptada pelo *proxy* referenciado por **b**, que, por sua vez, invocará o método do objeto alvo que será instância de **BarImpl**.

Quando um método é invocado sobre uma instância de uma *dynamic proxy class*, as duas principais ações são tomadas, como podemos ver a seguir:

1. o método é recodificado para as seguintes instâncias: uma instância da classe **java.lang.reflect.Method** e a lista de argumentos do tipo **Object**. Os argumentos de tipo primitivo são embalados nas classes apropriadas, como por exemplo, **java.lang.Integer** ou **java.lang.Boolean**. O *proxy* utilizará essas instâncias como argumentos, na invocação ao método **invoke** do seu *invocation handler* associado.

2. no método **invoke** da instância do *invocation handler*, o programador tem a liberdade de modificar a sua implementação. Ele pode, como já anunciamos acima, incluir algum código antes e depois de realizar a invocação. Em outras palavras, no meta-nível, é dada a chance ao programador de introduzir código para o pré e pós-processamento de uma invocação.

No exemplo abaixo, o código no pré e pós-processamento providencia o rastreamento de cada invocação, assim, são exibidos a invocação, antes de ser efetivamente executada, e o resultado, antes de retornar, efetivamente.

```
public Object invoke(Object proxy,
Method m, Object[] args) throws
Throwable
{
Object result;
try {
/* código introduzido no pré */
System.out.print("begin method " +
m.getName() + "(");
for(int i=0; i<args.length; i++) {
if(i>0) System.out.print(",");
                  System.out.print("
" +
args[i].toString());
            }
            System.out.println(" )");
/* fim do pré-processamento */

result = m.invoke(obj, args);
}
 catch (InvocationTargetException e)
{
            throw
e.getTargetException();
        }
 catch (Exception e) {
           throw new RuntimeException
           ("unexpected invocation
exception: " +

e.getMessage());
        }
/* código introduzido no pós */
finally {

System.out.println("end method " +
m.getName());
}
/* fim do pós-processamento */
return result;
}
```

Chamamos a atenção neste ponto para a extensão da oportunidade de introduzir código no pré e pós-processamento da invocação, como por exemplo, incluir uma cadeia de invocações a objetos intermediários, até que seja efetivamente atingido o objeto alvo. Nesse caso, quando o objeto alvo retorna, no pós-processamento, essa cadeia é percorrida de volta. Com essas possibilidades de interposições intermediárias, vemos uma chance para a implementação de protocolos de meta-objeto com *proxy*.

3.1 Exemplo com o uso de Dynamic Proxy da API de Java

O nosso propósito neste exemplo é o mesmo adotado pelos autores em [9], o qual reproduzimos acima. Ressaltamos a facilidade *dynamic proxy* oferecida pela API de Java, através de uma semântica bem simples, em que trabalhamos com um único objeto alvo que é referenciado por um *proxy*.

O objeto alvo é referenciado diretamente pela variável local **obj1**, que foi declarada com a classe **C1**, no método **main**. Nossa intenção, no entanto, é produzir uma interação, em que o objeto alvo seja alcançado indiretamente pelo *proxy*. Esse *proxy* é referenciado pela variável local **o1** declarada com a interface **I1**.

A semântica desse exemplo foi elaborada, tendo em vista a simplicidade, como podemos ver, a seguir:

1. No método **main**, é criado o objeto alvo da classe **C1**, referenciado por **obj1**;

2. No método **main**, é criado o *proxy*, pela invocação do método **newInstance**, com o argumento **obj1**, cujo retorno, um *proxy*, é atribuído à variável local **o1**, declarada com a interface **I1**. No retorno da criação, é efetuada uma coerção sobre o *proxy* retornado com a interface **I1**.

3. No método estático newInstance da classe **IHandlerCalledByProxy** do *invocation handler*, optamos pelo caminho mais simples, em que incluimos em seu código a invocação do método **newProxyInstance** da classe *Proxy*, com a finalidade de criar uma classe *proxy*, uma instância *proxy* dessa classe *proxy* que é retornada e, além disso, para o terceiro argumento, foi invocado **new IHandlerCalledByProxy** para criar o *invocation handler* a ser associado a instância desse *proxy*. Assim, nessa chamada, foram passados os três argumentos seguintes:

a. primeiro argumento é o *class loader* usado para definir a classe *proxy*;

b. segundo argumento é o conjunto de interfaces que a classe *proxy* deve implementar;

c. terceiro argumento é a instância do *invocation handler* a ser associada à instância de *proxy* e obtida no retorno da invocação do construtor **IHandlerCalledByProxy**. Para esse construtor foi enviada a referência ao objeto alvo, que será guardada no atributo do *invocation handler*. Esse atributo dará chances ao *invocation handler*, durante a interceptação, de referenciá-lo para efetuar a invocação definitiva.

4. Voltando ao método **main**, vamos nos concentrar no corpo do loop de três passos, que possui o único comando:

```
resp = o1.m()
```

a invocação do método **m** é interceptada pelo *proxy*, que prepara os argumentos, criando um objeto da classe **Method** e uma lista dos argumentos propriamente ditos, passando tudo isso, como argumentos, ao método **invoke** do *invocation handler* associado a esse *proxy*.

5. No método **invoke** do *invocation handler*, o método **m** é invocado sobre o objeto alvo **obj**, com uma lista dos argumentos originais passados, através da seguinte construção:

```
result = me.invoke(obj,args);
```

e, no retorno, seu resultado é guardado na variável local **result**, que, por sua vez, retorna ao emissor, que foi o código do método **main**.

6. O método **m** (no objeto alvo da classe **C1**) efetua as seguintes ações:

a. incrementa o atributo **v**,

b. exibe o valor de **v**,

c. e retorna esse valor de **v** ao chamador, que é o método invoke do *invocation handler*, que, por sua vez, retornará este resultado ao chamador **main** do objeto emissor da mensagem original **m**.

7. Voltando ao método **main**, o resultado é atribuído a variável local **resp** e exibido, concluindo um passo de repetição.

```
import javax.swing.JOptionPane;
import java.lang.reflect.*;

public interface I1 {
  public int m ();
}

public class C1 implements I1{
      int valor;

  public C1 () {
    super();
    this.valor = 0;
  }
```

```
   public int m (){
     ++ valor;
       JOptionPane.showMessageDialog(
      null,"In instance of C1/m(),
      valor is "+valor);
     return (valor);
   }
}


public class IHandlerCalledByProxy
implements InvocationHandler {

      private Object obj;

   public static Object newInstance
(Object o){
     return Proxy.newProxyInstance (

   o.getClass().getClassLoader(),

   o.getClass().getInterfaces(),
          new IHandlerCalledByProxy
(o));
   }
   private
IHandlerCalledByProxy(Object ob) {
     super();
     this.obj = ob;
   }

      public Object invoke(Object
proxy, Method me, Object[] args)
            throws Throwable {
     Object result;
     try {
       result = me.invoke(obj, args);
     }
     catch (InvocationTargetException
e) {
       throw e.getTargetException();
     };
     return result;
}
}

public class ProxyClient {

      public ProxyClient() {
     super();
      }

      public static void main(String[]
args) {
      int i, resp;

   C1 obj1 = new C1 ();

   I1 o1 =
(I1)IHandlerCalledByProxy.newInstance
(obj1);
   JOptionPane.showMessageDialog(null,"O1
and proxy were instanced");
   JOptionPane.showMessageDialog(null,
"ProxyClient begins");
   for (i=0;i <= 2; i++) {
     resp = o1.m();
     JOptionPane.showMessageDialog(
                  null,"In Client/main:
            result of m () inv "+i+" is
            "+resp);
   }
      }
}
```

Sobre os passos descritos acima, cabe destacar a transparência alcançada no método **main**, na invocação do método **m** até o seu retorno, pela simples construção, a seguir:

```
resp = o1.m()
```

a interceptação com várias ações não aparece no código do cliente, ficando resumida a um envio de mensagem comum, da programação em Java. Na interceptação desse envio de mensagem, o *proxy* toma o cuidado de transformar todas as informações necessárias à reconstituição da invocação original desse método **m** nos parâmetros, os quais são, então, passados ao método **invoke** do *invocation handler* associado ao *proxy*. Esse método **invoke**, por sua vez, invoca o método **invoke** do objeto (da classe **Method**) referenciado pelo parâmetro **me**, passando como parâmetros o objeto alvo e os parâmetros originais. Neste ponto então é efetuada a invocação efetiva ao método **m** do objeto alvo referenciado por **obj**, que após ser executado, retorna o resultado ao *invocation handler*, sendo então atribuído à variável local **result**. O valor de **result**, por sua vez, é retornado ao método **main**, e atribuído à sua variável local **resp**, sendo o método **main**, o responsável original por essa invocação.

3.2 Considerações

Uma das nossas idéias foi a de rastrear os passos dos dois momentos da computação, em que o *proxy* entra em cena: 1) na criação da referência (*proxy*) a um objeto

alvo e 2) na invocação de um método de um objeto alvo referenciado pelo *proxy*.

Complementando esse rastreamento, ressaltamos os quatro elementos principais, que são os agentes de cada ação, no curso da criação e da invocação: a classe *proxy*, a instância de *proxy*, o *invocation handler* e o objeto alvo, além dos espaços de intervenção do programador, identificando os níveis de programação base e meta.

O leitor vai encontrar, na literatura relacionada ao uso de *proxy*, diversas descrições e exemplos e, naturalmente, detectará semelhanças com este texto, por terem todos a mesma intenção, que é a de esclarecer como deve ser empregado o *proxy*. Nossa expectativa é a de que, durante a leitura deste texto, um programador consiga identificar com clareza os elementos do *proxy*, seus papéis, o espaço aonde pode fazer intervenções de código e o seu funcionamento.

Uma das características deste nosso trabalho, como já anunciamos antes, é a simplicidade proposital do exemplo apresentado, possibilitando o foco nos elementos e nos espaços de intervenção do programador, além de ter sido definido para que o *proxy* ficasse enquadrado no papel mais geral possível, que é o de interceptor.

4. Conclusões

Para gerar uma documentação que simplificasse o entendimento do emprego de *dynamic proxy* da API de Java, não inovamos o formato dos textos que apresentavam um exemplo do emprego de *proxy*, apenas ressaltamos os aspectos que causaram as maiores dificuldades de entendimento, acompanhados de mais detalhes, para ajudar nesses esclarecimentos. Essas dificuldades foram a diversidade de elementos que entram em cena, durante o funcionamento do *proxy*, os truques desse funcionamento resolvidos com reflexão, e a localização precisa das intervenções do programador, explicitando se no nível meta ou no nível base de programação.

Os quatro mais importantes elementos (objetos e classes) envolvidos no funcionamento do *proxy* e criados a partir da invocação do método estático **`newInstance`** da classe que implementa um *invocation handler*, que, por sua vez, invoca o método estático **`Proxy.newProxyInstance (Object obj)`**, são os seguintes:

a classe *proxy*, que é criada, automáticamente, caso ainda não exista;

a instância da classe *proxy* (1), criada, automáticamente, nesta chamada, como o próprio nome do método diz;

a instância da própria classe *invocation handler*, criada a pedido da programação na execução desse método **`newInstance`**, e que serve como argumento para a criação da instância do *proxy*, ficando assim amarrado a esse *invocation handler*;

o objeto alvo das mensagens, que é o argumento de **`newInstance`**, sendo criado no próprio parâmetro, como no exemplo acima, ou referenciado a um já existente.

Quando um método de um objeto alvo é invocado sobre uma instância de uma *dynamic proxy class*, é efetuada uma interceptação, que pode ser acompanhada pelas seguintes ações:

1. A invocação do método é recodificada para as seguintes instâncias: uma instância da classe **`java.lang.reflect.Method`** e a lista de argumentos do tipo **`Object`**. Se algum argumento é de um tipo primitivo, ele é embalado na classe apropriada, como, por exemplo, **`java.lang.Integer`** ou **`java.lang.Boolean`**. O *proxy* utilizará essas instâncias como argumentos na invocação do método **`invoke`**, do *invocation handler*.

2. No método **`invoke`** da instância do *invocation handler*, o programador tem a liberdade de modificar a sua implementação. Ele pode, por exemplo, incluir algum código antes e depois de realizar a invocação. Em outras palavras, no nível meta, é possível fazer pré e pós-processamento sobre a invocação.

Tendo maior clareza sobre esses aspectos apresentados acima, desenvolvemos um exemplo que emprega, isoladamente, o *proxy* da API de Java para *Dynamic Proxy*. Escolhemos uma simples interação entre dois objetos, para que pudéssemos dar enfoque aos aspectos do *proxy*, o qual referencia o objeto alvo desta

interação. Com esse exemplo, nossas pretensões, em primeiro lugar, eram as de esclarecer os papéis desses aspectos, no funcionamento do *proxy*, e, em segundo lugar, abrir as perspectivas de seu uso, como um representante para qualquer objeto servidor, considerado o objeto alvo de uma instrução comum de envio de mensagem. Por um outro ângulo, mostramos também como o *proxy* pode ser elevado à condição de uma referência inteligente a um objeto alvo. Assim, explicamos como essa inteligência pode ser adicionada pelo programador, introduzindo código que habilite o pré-processamento da mensagem enviada e o pós-processamento do resultado retornado.

Com esse exemplo, evidenciamos os elementos envolvidos e chamamos a atenção para a visibilidade nos níveis meta, base e os que ficariam invisíveis, como é o caso do *class loader* e da classe *proxy*, associada às interfaces. Na descrição da seqüência de passos dados, durante uma invocação, esclarecemos o funcionamento de um *proxy*. Assim, pela leitura deste documento, um programador tem a chance de conhecer os elementos, suas responsabilidades, sua visibilidade, e aonde ele deve incluir código, relacionado ao *dynamic proxy*.

Referências

[1] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series Professional, 1995.

[2] J. C. Leite, Notas de Aula de Engenharia de Software - Catálogo de Padrões – GoF. www.dimap.ufrn.br/~jair/mes/slides/Padroes.pdf. 2004.

[3] P. Maes, "Concepts and experiments in computational reflection", *OOPSLA'87 – Conference on Object Oriented Programming Systems Languages and Applications*, 1987.

[4] Kiczales, G., *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[5] Forman, I.R., N. Forman, *Java Reflection in Action*, Manning Publications, 2004.

[6] "Class Proxy of java.lang.reflect of Java™ 2 Platform Standard Edition 5.0. API Specification", http://java.sun.com/j2se/1.5.0/docs/api/

[7] "Interface InvocationHandler of java.lang.reflect of Java™ 2 Platform Standard Edition 5.0. API Specification", http://java.sun.com/j2se/1.5.0/docs/api/

[8] "Dynamic Proxy Classes" http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[9] T. Harpin, "Using java.lang.reflect.Proxy to Interpose on Java Class Method", http://java.sun.com/developer/technicalArticles/JavaLP/Interposing/

[10] S. Halloway "Using Dynamic Proxies to Layer New Functionality Over Existing Code", java.sun.com/developer/TechTips/2000/tt0530.html. 2000.

[11] Y. Hassoun, R. Johnson and S. Counsell, "Reusability, Open Implementation and Java's Dynamic Proxies", PPPJ03 – Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java™,Kilkenny City, Ireland, 2003.