Persistência Rasa pela Extensão de Proxy e Serializable da API de Java

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

Paulo Rogério Motta Júnior
IC – Universidade Federal Fluminense
prmottajr@uol.com.br

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

Abstract

In this paper we introduce an implementation of a simple library as a Java package that allows shallow, transparent and orthogonal persistence regarding data types. This capacity was achieved by the extension of the Proxy and Serialization features of the Java API combined with a simple broker that is also responsible for object storage.

Resumo

Neste artigo, apresentamos a implementação de uma biblioteca muito simples, como um package de Java, que efetua a persistência rasa, transparente e ortogonal a tipo.Essa capacidade foi alcançada pela extensão dos recursos de Proxy e de Serialization da API de Java, combinada a um simples broker, que cuida também do armazenamento dos objetos.

1. Introdução

A persistência de um objeto significa a sua sobrevivência entre execuções de programas que o referenciam. Em Java, a persistência vem sendo tratada por diversas abordagens [1], sendo a sua forma mais simples, a Serialização de Objeto [2], considerada a implementação básica de persistência na linguagem. A idéia de seu mecanismo é em resumo a cópia do estado de um objeto em uma seqüência de bytes e a reconstrução dessa seqüência em um objeto, posteriormente. A segunda abordagem resolve a persistência do objeto, pela armazenagem do objeto em bancos de dados relacional ou orientado a objeto. Uma terceira abordagem, utilizada em JavaData Objects [3] e Hibernate [4], faz uso de um arcabouço que realiza o mapeamento dos objetos para tabelas relacionais, tornando a persistência transparente e permitindo que a programação em Java fique completamente orientada a objeto enquanto os dados são persistidos por um banco de dados relacional. Uma abordagem mais recente é chamada de prevalência, sendo baseada na serialização dos comandos que causam efeito nos dados de um objeto, ficando esses comandos armazenados em um log.

O mecanismo padrão de persistência, em Java, também tem sido usado como um protocolo de *marshalling* para o arcabouço de Java Remote Method Invocation (RMI). Com essas facilidades combinadas, consegue-se a persistência de objetos em um arquivo local ou, alternativamente, em um sistema de arquivos remotos, via socket based streams, ou via Remote Method Invocation (RMI), como pode ser visto, em [5].

A nossa motivação surgiu quando desenvolvíamos um mecanismo de controle de concorrência, baseado na recuperação individual de um objeto. Como a serialização em Java alcança toda a árvore de composição, procuramos desenvolver uma biblioteca que implementasse a persistência rasa, também chamada

de *shallow copy/write*, ou *shallow load/unload*, ou *lazy load/unload*.

Assim, desenvolvemos uma biblioteca simples (package) com os recursos de Serialization [2] e Proxy [6, 7] de Java. Combinado a esses recursos introduzimos um broker [8] muito simples, implementado por uma HashTable, o qual controla o armazenamento e carga dos objetos. Estamos no momento, implementando a parte remota. Além do aspecto raso da persistência, atendemos aos critérios de ortogonalidade [9] e de transparência na programação, a qual toma conhecimento do aspecto de persistência, apenas durante a construção do objeto.

Os objetos de um programa que empregue esse package tornam-se persistentes, bastando para isso que a construção do objeto seja providenciada por intermédio do Proxy, como podemos ver, na seção de exemplo.

Caso um programador de software básico queira implementar um middleware para transação, essa biblioteca poderá ser estendida.

O restante desse artigo é organizado, como a seguir: Seção 2 apresenta o Modelo de Computação; a Seção 3 um exemplo de emprego desse pacote; e a seção 4 as conclusões.

2. Modelo de Computação

A persistência rasa significa que somente será gravado/restaurado o estado do objeto, não será gravado/restaurado o estado de qualquer um dos objetos que esse objeto em foco esteja referenciando, bem como o daqueles da cadeia sucessiva da composição. Em resumo, os objetos que pertencem à composição desse objeto não serão gravados/restaurados.

Uma das utilidades da persistência rasa é a economia de memória durante a execução de uma aplicação que contém uma grande quantidade de objetos envolvidos. Geralmente os objetos não são usados ao mesmo tempo, assim, quando o objeto principal da aplicação é alocado, os objetos referenciados por ele ainda não precisam ser alocados. A alocação de um objeto somente é efetuada, quando surge a execução de alguma instrução que referencia esse objeto. Essa medida adotada para alocação por demanda dos objetos da aplicação faz com que a memória seja economizada.

Uma outra utilidade de persistência rasa é em compartilhamento de objetos por transações distintas, que contam com mecanismo de controle de concorrência localizado em cada objeto, como por exemplo, a atomicidade local [10]. No momento do *commit* de alguma das transações, a parte de recuperação da atomicidade local só poderá gravar o estado do objeto que está sob o seu controle. Essa gravação, então, tem que ser realizada com persistência rasa, para evitar que a composição do objeto seja gravada.

Na persistência rasa, uma restauração vai ocorrer, automaticamente, somente nas duas condições seguintes: uma execução de uma instrução de envio de mensagem for executada e o objeto alvo desse envio ainda não estiver alocado. Uma vez restaurado, o objeto fica alocado na memória principal (*heap*) até que a execução termine ou uma ação de desalocação proposital surja a pedido da programação. As referências que surgirem depois no curso da computação a este objeto alvo não causarão mais as ações de restauração.

Uma gravação de um objeto vai ocorrer somente a pedido da programação. Ela não ocorre automaticamente como conseqüência de uma gravação de um objeto de classe que implementa a interface *serializable* e que aninha este objeto.

No caso da persistência rasa, nós estamos fazendo com que o estado de cada objeto seja salvo da seguinte forma: 1) se o tipo, que aparece na declaração de um atributo, é um tipo primitivo, como, por exemplo, inteiro, real, caracter ou booleano, então o seu valor deve ser salvo, sendo o mesmo tratamento adotado para a persistência padrão de Java; e 2) se o tipo que aparece na declaração de um atributo é uma interface (tipo), então o valor associado ao atributo é uma referência e deve ser salva. Essa referência é implementada por um *proxy*, devendo sempre ser uma instância de uma classe *proxy* que implementa a mesma interface que aparece na declaração.

Nesta solução apresentada aqui, as duas facilidades oferecidas pela API de Java: *dynamic proxy* e *serializable classes* aparecem combinadas com o

propósito de adicionar a capacidade de persistência rasa a um objeto. Para isso foi necessário ainda introduzir um terceiro elemento, que é enquadrado no padrão de *design* chamado *broker*.

O padrão *Broker* está classificado em [11] como um padrão de projeto. Ele pode ser pensado como um corretor (de imóveis, planos de saúde, seguros, automóveis, etc...) e tem sido utilizado para estruturar sistemas distribuídos, separando componentes que interagem através de chamadas remotas de serviços. Sua responsabilidade é a de coordenar a comunicação, encaminhando as solicitações e transmitindo os resultados ou exceções.

Seu uso mais comum tem sido em ambiente distribuído e heterogêneo com componentes cooperativos independentes. Como os sistemas acoplados não oferecem flexibilidade, a manutenção encontra dificuldades. Assim, introduzindo o componente *broker*, alcança-se mais desacoplamento entre clientes e servidores, permitindo independência de plataforma (ambientes heterogêneos) e de endereçamento [11].

Pela nossa solução, o *proxy* intercepta o envio da mensagem ao objeto alvo e, na seqüência da interceptação, incluímos a participação do *broker*. Este último providencia a entrega da mensagem ao objeto alvo, assumindo o papel de mensageiro. Assim, no caminho de ida, providencia a entrega da mensagem ao objeto alvo e, na volta, o resultado ao objeto emissor.

Esse *broker* conta com a ajuda de uma tabela de objetos da aplicação, cujas entradas são associadas a esses objetos. Cada uma dessas entradas guarda efetivamente as informações do objeto e tem a capacidade de efetuar a entrega definitiva ao objeto alvo, auxiliando a funcionalidade de mensageiro do *broker*.

Assim, em relação a interceptação, provida pelo *Proxy*, acrescentamos mais dois segmentos na cadeia de interceptação da mensagem, o *broker* e a entrada na tabela correspondente ao objeto alvo.

Além da responsabilidade pela entrega efetiva da mensagem ao objeto alvo, a entrada na tabela de objetos providencia o armazenamento e restauração do objeto. Acrescentamos, assim, mais uma propriedade ao *broker* que construímos, a de armazém dos objetos persistentes.

Nas subseções a seguir, descrevemos como o *proxy* e o *broker* foram construídos, para que cooperassem, no sentido de fazer com que o objeto da aplicação adquirisse a capacidade de persistência rasa.

2.1 Proxy e Invocation Handler

No código da aplicação, os objetos serão referenciados por *proxies*. Para isso, cada declaração de variável ou atributo deve ser feita com Interface e cada objeto da aplicação deve ser criado juntamente com a instância de uma classe de *proxy*, a qual possui um método estático chamado `newInstance`, que retorna o *proxy* (referência).

O principal papel do *proxy* é interceptar a mensagem enviada, para que a mesma, antes de ser entregue ao objeto alvo, passe pelo *invocation handler*.

No *invocation handler* que estamos implementando aqui, a mensagem não será entregue ao objeto alvo por ações do seu método **`invoke`**, porque foi acrescentada à seqüência de interceptação outros dois segmentos, o *broker* e a entrada na tabela de objetos da aplicação, como já anunciamos acima. Assim, algumas diferenças serão notadas em relação ao uso mais comum de *proxy*.

Logo abaixo se encontra o código de implementação do *Invocation Handler* associado ao *proxy*, e, em seguida, comentamos essa construção, ressaltando seus aspectos mais importantes.

```
/* PrPeProxy Class  */

import
java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import
java.lang.reflect.InvocationTargetExcep
tion;
import java.lang.reflect.Method;
import java.io.Serializable;

public class PrPeProxy implements
InvocationHandler, Serializable{
   private static final long
serialVersionUID = 42L;
   public static Broker  b;
   private Integer id;


public PrPeProxy(Object o) {
```

```
      super();
      this.id = b.put(o);
   }
public static setBroker (Broker br){
   b = br;
   }
public    static    Object    newInstance
(Object o){
   return Proxy.newProxyInstance (
   o.getClass().getClassLoader(),
   o.getClass().getInterfaces(),
   new PrPeProxy(o));
   }

      /* (non-Javadoc)
                        *                @see
java.lang.reflect.InvocationHandler#inv
oke(java.lang.Object,
java.lang.reflect.Method,
java.lang.Object[])
       */
   public  Object  invoke(Object  proxy,
Method me, Object[] args)
      throws Throwable {
      Object result;
      result = null;
      try {
      result   =   b.runMethod(proxy,   me,
args, id);
      }
      catch    (InvocationTargetException
e) {

   javax.swing.JOptionPane.showMessageD
ialog(null,"Invocation           Target
Exception: "+e);
      };
      return result;
   }
}
```

Como já observamos acima, a classe `PrPeProxy` que implementa o *invocation handler* e o associa ao *proxy*, que especificamos, é um pouco diferente das encontradas, no uso mais simples. Ela não possui o objeto alvo declarado e sim os dois atributos:

a) de instância, a chave de identificação `id` do tipo `Integer`, a qual permite ao *broker* buscar o objeto na tabela de objetos; e

b) de classe, a referência ao *broker* `b` do tipo `Broker` , que deve ser preenchida antes que um construtor `PrPeProxy` seja invocado, porque o construtor `PrPeProxy` pede através da chamada `b.put(o)`, que o *broker* inclua a referência `o` em sua tabela de objetos, retornando a chave de busca correspondente a entrada aonde a referência ao objeto fica guardada.

O papel do atributo **`Broker b`** na interceptação será o de possibilitar que o *invocation handler* repasse a interceptação ao *broker*. Para isso, o método **`invoke(Object proxy, Method me, Object[] args)`** não deve mais invocar diretamente o objeto alvo, repassando ao *broker*, o qual se encarregará dessas ações.

Como o *broker* precisa saber qual é o objeto da tabela de objetos que deve receber essa mensagem, é preciso também passar a sua chave de identificação. Para isso, o atributo **`id`** do *invocation handler* é passado como parâmetro na invocação do método **`runMethod`** sobre o *broker*.

O atributo **`id`** representa a chave da entrada na tabela que armazena os objetos da aplicação. Como já dissemos acima, as entradas dessa tabela são úteis ao *broker* para que ele desempenhe o seu papel de mensageiro. Em cada uma dessas entradas, há uma referência efetiva ao objeto da aplicação associado ao *invocation handler* e, consequentemente, ao *proxy*.

O método **`invoke`**, que entra em cena quando a mensagem é interceptada pelo *proxy*, executa na instância do *invocation handler*, assim, ele tem acesso aos atributos **`b`** e **`id`**, que são usados no comando

"result = b.runMethod(proxy, me, args, id)" ,

repassando, assim, os argumentos para o *broker*, que, por sua vez, os repassa para a entrada, que, finalmente, faz a invocação ao método efetivo do objeto alvo.

No uso mais comum de *proxy*, como o visto na seção anterior, é feita a invocação efetiva do método, no código de **`invoke`**, mas nós precisamos adiá-la, porque estamos usando o modo raso de persistência. Neste modo, o objeto alvo pode não se encontrar alocado ainda. Assim, o *broker* deverá tomar providências, auxiliado pela entrada da tabela correspondente ao **`id`** .

Ainda, no método **`invoke`**, no caminho de volta, a variável local **`result`** receberá o resultado da execução do método efetivo, que terá sido repassado desde o retorno do método efetivo do objeto alvo, passando no caminho de retorno pela entrada, *broker*, *invocation handler*, e, finalmente, devendo retornar à chamada original, pela execução do comando **`return results`**.

No momento estamos fazendo um exercício simples de persistência rasa, ainda sem acesso remoto, mas é nossa intenção incluir mais dois atributos, na implementação do *invocation handler*, o *path* do *broker* local e o *path* do *broker* aonde o objeto da aplicação associado a este *proxy* reside, caso esse objeto seja remoto. A adição desses dois atributos dispensa o programador da tarefa de interação com a API para acesso remoto oferecida nos *packages* com o prefixo **`java.rmi`**.

2.2 Broker

Como já comentamos acima, pela nossa solução, o *proxy* intercepta o envio da mensagem ao objeto alvo e na seqüência da interceptação, da qual o *broker* faz parte, é providenciada a entrega ao objeto alvo. Assim, o *broker* assume o papel de mensageiro, entregando a mensagem ao objeto alvo e o resultado ao objeto emissor.

Este *broker* conta com uma tabela implementada por uma classe *Hashtable*, referenciada pelo atributo **`apObjs`**. As entradas dessa tabela são da classe implementada também por nós, chamada *TableEntry*, sendo cada instância exclusiva a cada objeto da aplicação. Essa entrada é responsável pelo armazenamento do seu objeto, no caso de ser local, ou pelo acesso ao objeto, no caso de ser remoto (essa parte ainda não se encontra implementada).

A entrada da tabela guarda as seguintes informações: a identificação do objeto, que é, ao mesmo tempo, a chave de busca na *Hashtable*; a classe do objeto; o objeto declarado com a classe **`Object`**; e o nome do arquivo aonde o objeto fica armazenado. Devemos observar que a maioria dos métodos dessa classe *TableEntry* tem os mesmos nomes dos métodos do *broker*. Essa repetição significa um repasse de parâmetros à entrada da tabela, com a finalidade de evitar a quebra de encapsulamento entre o *broker* e cada entrada.

A interceptação que comentamos acima faz a execução alcançar o método **`invoke`** do *invocation handler*, que, por sua vez, repassa os parâmetros ao *broker*, pela invocação de seu método **`runMehod`**. Nesse método, no *broker*, então, os parâmetros são repassados ao método **`runMehod`** da entrada da tabela, associada ao objeto alvo. Nesse método da entrada, em primeiro lugar, são tomadas as ações para assegurar a alocação do objeto alvo, verificando se o valor da referência ao objeto ainda se encontra **`null`**, sendo, nessa condição, providenciada a restauração do objeto alvo da memória secundária para a memória principal (*heap*). Após essa garantia de alocação do objeto alvo, é comandada a invocação efetiva do seu método, via método **`invoke`**.

Chamamos a atenção para a providência de alocação do objeto alvo acima. É assim que, pela nossa solução, qualquer objeto da aplicação é restaurado da memória secundária para a memória principal. Em outras palavras, somente é restaurado se for o alvo de uma mensagem, caso contrário fica armazenado na memória secundária. Só existe uma outra alternativa de alocação de um objeto na memória principal, que é alcançada como uma conseqüência natural das ações de sua própria criação.

Logo abaixo, apresentamos o código das classes *Broker* e *TableEntry*. A classe **`HashTable`** já é oferecida por Java, e o atributo para essa tabela foi declarado como **`private Hashtable <Integer, TableEntry> apObjs`**, na classe *Broker*. Notamos que essa declaração utilizou a facilidade *Generic*, oferecida na versão Jsdk 1.5.

```
/* Broker Class */

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.lang.reflect.Method;
import java.util.Enumeration;
import java.lang.Integer;
import java.lang.Exception;
import
java.lang.reflect.InvocationTargetException
;
import java.util.Hashtable;

public class Broker implements Serializable
{
   private      static      final      long
serialVersionUID = 42L;

   private String  bFileName;
   private     Hashtable<Integer,TableEntry>
apObjs;
   private Integer next;

   public Broker() {
      super();
      this.bFileName = null;
      this.apObjs          =          new
Hashtable<Integer,TableEntry>();
```

```
      next = new Integer (0);
      }
   public Broker(String bFName) {
      super();
      this.bFileName = bFName;
      this.apObjs           =           new
Hashtable<Integer,TableEntry>();
      next = new Integer (0);
      }

   public TableEntry getEntry (Integer id) {
      return ((TableEntry)apObjs.get(id));
      }
   public Object getObj (Integer id) {
      TableEntry entr;
      entr = this.getEntry(id);
      return (entr.getObj(id));
      }

   public Integer put (Object o) {

      int intAux;
      TableEntry entr;

      entr = new TableEntry (next, o);
      apObjs.put(next, entr);
      intAux = next.intValue();
      next = new Integer (intAux + 1);
      return (new Integer (intAux));
      }
   public void write (Integer id){
      TableEntry entr;
      entr = (TableEntry)apObjs.get(id);
      entr.write();
      }
   public void close (Integer id) {
      TableEntry entr;
      entr = (TableEntry)apObjs.get(id);
      entr.close();
      }
   public void close () {
      for       (Enumeration       e       =
apObjs.keys();e.hasMoreElements();){
      Integer              key             =
(Integer)e.nextElement();
      write (key);
      close (key);
      }
      if (bFileName != null){
      try {
      FileOutputStream    fileOut    =    new
FileOutputStream (bFileName);
      ObjectOutputStream    out    =    new
ObjectOutputStream (fileOut);
      out.writeObject(this);
      }
      catch (Exception e){

   javax.swing.JOptionPane.showMessageDialog
(null,"writing file Exception: "+e);
      };
      }
   }
   public void display (Integer id) {
      TableEntry entr;
      entr = (TableEntry)apObjs.get(id);
      entr.display();
      }
   public void display () {
      for       (Enumeration       e       =
apObjs.keys();e.hasMoreElements();){
      Integer              key             =
(Integer)e.nextElement();
      display (key);
      }
   }
   public  Object  runMethod(Object  proxy,
Method me, Object[] args, Integer id)
   throws Throwable {
      Object result;
      TableEntry entr;
      entr = getEntry (id);
      try {
      result                                =
entr.runMethod(proxy,me,args);
      }
      catch (InvocationTargetException e) {
      throw e.getTargetException();
      }
      return result;
   }
}
```

/* TableEntry Class */

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import
java.lang.reflect.InvocationTargetException
;
import java.lang.reflect.Method;
import java.lang.Class;
import javax.swing.JOptionPane;

public    class    TableEntry    implements
Serializable {
   private      static      final      long
serialVersionUID = 42L;

   private Integer id;
   private Class c;
   private Object o;
   private String objFName;
   /**
    *
    */
   public  TableEntry(Integer  idObj,  Object
ob) {
      super();
      this.id = idObj;
      this.c = ob.getClass();
      this.o = ob;
      this.objFName      =      "fObj"      +
idObj.toString();
   }
   public Object read () {
      try {
      FileInputStream    fileIn    =    new
FileInputStream(objFName);
      ObjectInputStream in =   new
ObjectInputStream(fileIn);
      o = c.cast(in.readObject());
      }
      catch (Exception e){

   javax.swing.JOptionPane.showMessageDialog
(null,"reading file Exception: "+e);
      }
      return (c.cast(o));
      }
   public Object getObj (Integer idObj){
      if (id.equals (idObj))
      if (o == null)
         return (c.cast(read()));
      else return (c.cast(o));
      else return (c.cast(null));
   }
   public Integer getId (){
      return (id);
   }
   public void write (){
      if (o != null){
      try {
      FileOutputStream    fileOut    =    new
FileOutputStream (objFName);
      ObjectOutputStream    out    =    new
ObjectOutputStream (fileOut);
      out.writeObject(c.cast(o));
      }
```

```
      catch (Exception e){

   javax.swing.JOptionPane.showMessageDialog
(null,"The write ( ) in object id
"+id.intValue()+" caused an Writing File
Exception!"+e);
      };
      }
   }
   public void put (Object ob){
      o = c.cast(ob);
      write ();
   }
   public void close (){
      o = (Object)null;
   }
   public void display (){

   JOptionPane.showMessageDialog(null,"Broke
r/entry - Id: "+id.intValue()+" Class:
"+c.getName()+" File name: "+objFName);

   }
   public Object runMethod (Object proxy,
Method met, Object[] args)
   throws Throwable {
      Object r;
      if (o == null) read ();
      try {
      r = met.invoke(c.cast(o), args);
      }
      catch (InvocationTargetException e) {
      throw e.getTargetException();
      };
      return r;
   }
}
```

2.3 Considerações

Reunimos as quatro classes correspondentes a esses objetos *proxy*, *broker*, tabela de entradas e entrada em um pacote de Java que denominamos **`firstShallowPersistencePackage`**. o qual dá suporte à persistência rasa.

Embora tenhamos implementado neste pacote os mecanismos que tornam os objetos da aplicação capazes de ser restaurados/gravados no modo raso, foi necessário contar com a ajuda dos mecanismos da API de Java (modo profundo) para a gravação/restauração dos *proxies*, *broker*, tabela de objetos e entradas dessa tabela, devendo então suas classes implementar a interface **`Serializable`**. As classes dos objetos da aplicação também precisam implementar essa interface, mas por artifícios, que destacamos, a seguir, conseguimos quebrar a persistência profunda para os objetos da aplicação:

1) Para a restauração de um objeto em modo raso, quando a execução é encerrada, o *broker* e toda a sua árvore de composição deve ser fechada, sendo providenciada nessa operação, a gravação de todos os seus elementos. Nesse caminho, o broker faz uma iteração por todas as entradas de sua tabela, solicitando a cada uma a gravação de seu objeto e depois o seu fechamento. No fechamento da entrada, o atributo que referencia o objeto da aplicação é preenchido com o valor **`null`**. Essa medida causará a restauração em modo raso de qualquer objeto de aplicação referenciado por alguma aplicação, em uma próxima execução que contar com esse *broker*.

2) Para a gravação em modo raso, os atributos dos objetos da aplicação foram forçados a ser referências a *proxies*, os quais só tem a chave de acesso (**`id`**) da entrada do objeto referenciado na tabela. Então, quando a persistência de Java está tentando descer pela composição de um objeto da aplicação, pelos seus atributos, ela só consegue chegar aos *proxies*. Cada *proxy* contém o atributo de instância **`id`** do tipo **`Integer`**, que é considerado um tipo primitivo pela persistência de Java, interrompendo, assim, a descida pela composição. Em outras palavras, a persistência profunda é quebrada, porque o seu mergulho na composição é impedido por essa chave, que é um inteiro.

3. Exemplo

O exemplo que apresentamos é bem simples com dois objetos a serem persistidos no modo raso, um associado ao *proxy* referenciado por `o1`, que é atributo do objeto referenciado por `t` da classe `TopLevel`, e o outro associado ao *proxy* referenciado por `o2`, na instância de `C1`.

O cliente principal neste exemplo é a classe **`PrPeClient`**, que possui um método `main` estático, quer dizer um método de classe. O método `main` possui uma variável local `t` declarada como sendo da classe `TopLevel`[1]. Esse método tem apenas a

1 A princípio, essa classe `TopLevel` não existiria, mas é o método `main` da classe **`PrPeClient`** que será executado, e na execução não é instanciado um objeto dessa classe, é a própria classe que está funcionando como objeto. Precisávamos de um objeto *top-level* para ser gravado no modo de presistência

finalidade de comandar a execução da aplicação, em três etapas: 1) criar ou restaurar da memória secundária a raiz da composição, implementada pela classe `TopLevel`, que fica guardada na variável local `t` do método `main`; 2) iniciar a execução, invocando o método `topLevelMain()` do objeto raiz, referenciado por `t`; e 3) salvar na memória secundária a árvore de composição da aplicação, referenciada por `t`. Cabe notar que, neste parágrafo, não estamos falando de persistência rasa e sim de persistência comum de Java, aquela que tenta restaurar/gravar tudo que encontra pela árvore de composição.

O objeto referenciado por `t` é o que realmente realiza a semântica da aplicação. Podemos observar que o método `main` é responsável apenas pelas três ações enumeradas acima. Na gravação/restauração de t, os outros objetos que também são gravados/restaurados são o *proxy* referenciado por `o1`, do *broker* referenciado por `b`, da tabela de entradas e as próprias entradas, que deverão estar com o valor `null`, no atributo `o`. Assim, mesmo que `t` seja gravado em persistência profunda, os objetos associados a `o1`, em `t`, e a `o2`, na instância de `C1`, não serão gravados, nesse caminho. Veremos mais à frente, como eles serão gravados.

```
/*  PrPeClient  */

package firstShallowPersistencePackage;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import javax.swing.JOptionPane;
/**
 * @author malice
 *
 */
public class PrPeClient {

   /**
    *
    */
   public PrPeClient() {
      super();
   }

public static void main(String[] args) {
   TopLevel t = null;
   String tFileName = "topLevelFile";
   String answerTime;
   int executionTime = 0;
/*
   (creat/restor)ing the top-level
*/
   answerTime =
   JoptionPane.showInputDialog(
   "Answer new or old execution ? ");
   if (answerTime.equals("new"))
      t = new TopLevel ();
   else {
      executionTime = 1;
      JoptionPane.showMessageDialog
      (null,
      "The Composition Tree Restoring
      Operation begins !");
      try {
      FileInputStream fileIn = new
      FileInputStream(tFileName);
      ObjectInputStream in = new
      ObjectInputStream(fileIn);
      t = (TopLevel)in.readObject();
      }
      catch (Exception e){
      JoptionPane.showMessageDialog(
      null,"reading file Exception: "
      +e);
      }
      };
/*
   the top-level gets the control
*/
   JoptionPane.showMessageDialog(
   null,"The TopLevel Execution begins!
   ");

   t.topLevelMain (executionTime);

   JoptionPane.showMessageDialog(
   null,"The TopLevel Execution already
   is finished");

/*
   saving the top-level
*/

   JoptionPane.showMessageDialog(
   null,"Composition Tree Saving
   Operation begins !");
   try {
      FileOutputStream fileOut = new
      FileOutputStream (tFileName);
      ObjectOutputStream out = new
      ObjectOutputStream (fileOut);
      out.writeObject(t);
      }
   catch (Exception e){
      JoptionPane.showMessageDialog(
      null,"writing file Exception: "
      +e);
      }
   }
}
```

profunda, assim a classe `TopLevel` foi implementada. Esta questão de início dos tempos ainda não foi bem tratada por nós, talvez tudo deva ser gravado através do *broker*, que então ficaria visível, como um gerente de aplicações e deixaria visível os ícones de objetos *top-level* das aplicações.

A classe TopLevel também possui apenas um método de instância, chamado de `topLevelMain()`.

Nós declaramos dois atributos de instância para essa classe, **`o1`** como sendo do tipo (interface) `I1`, e `b` como sendo da classe `Broker`. Cabe notar neste ponto que não estamos satisfeitos com essas referências ao

broker e tentaremos em versões futuras deixá-las com maior transparência.

A primeira providência tomada no método `topLevelMain()` é verificar se é a primeira execução, para, então, providenciar as criações do *broker* em `b` e do *proxy* em **`o1`** que vai ser associado ao objeto da classe **`C1.`**

Quando o *proxy* é criado, pela chamada ao método de construção **`newInstance`**, sua referência é guardada em **`o1`**, ele é associado ao objeto semântico (da aplicação) da classe **`C1`** e ao *invocation handler* da classe **`PrPeProxy`**. Esse método **`newInstance`** recebe, como parâmetro, o objeto semântico a ser associado. O retorno desse método é o *proxy*, que possui a mesma interface que o objeto da classe **`C1`**, no caso então a interface **`I1`**.

Caso não seja a primeira execução, os objetos referenciados por `b` e por `o1` são alocados no momento da restauração do objeto *top-level*.

A partir dessa condição de alocação, o método `topLevelMain()` passa a efetivamente executar sua simples semântica, que é um laço de três iterações invocando o método `m()` de `o1`.

Observar que o *proxy*, referenciado por `o1`, intercepta o método **`m`**`()` que foi invocado, passando o controle ao método `invoke ()`, do *invocation handler*, que repassará os parâmetros a `runMethod` do *broker*, que, por sua vez, repassará ao `runMethod` da entrada, correspondente ao objeto alvo, associado ao *proxy* de `o1`, que, definitivamente, então, invocará o método `m()` da instância de `C1`.

Ainda no `topLevelMain()`, quando essas três iterações forem concluídas, é pedido ao *broker*: a exibição de todas as entradas pela instrução **`b.display()`** e o fechamento de todas as entradas pela instrução **`b.close()`**. Esse método **`close()`** do broker invoca os seus métodos **`write(Integer id)`** e **`close(Integer id)`** para todas as entradas de sua tabela de objetos.

No método **`write(Integer id)`**, ainda no *broker*, é obtida a entrada associada ao **`id`** e invocado então o seu método **`write()`**. Nesse método **`write()`** da entrada, é providenciada a gravação do seu objeto no arquivo cujo nome está guardado no atributo **`objFName`**.

No método **`close(Integer id)`**, ainda no *broker*, é obtida a entrada associada ao **`id`** e invocado então o seu método **`close()`**. Nesse método **`close()`** da entrada, é providenciado o preenchimento do valor **`null`** no atributo que referencia o objeto.

A medida tomada acima de preencher com o valor **`null`** a referência ao objeto nas entradas tem influência durante a gravação/restauração do objeto *top-level*, que salva/restaura, automáticamente, toda a composição, efetuada pelos métodos de persistência da API de Java. Na gravação de cada entrada da tabela do *broker*, como a referência ao objeto estará com o valor **`null`**, o objeto não será regravado. Na restauração do *broker*, como a referência ao objeto estará com o valor **`null`**, o objeto não será restaurado.

```
/* TopLevel Class*/
import javax.swing.JOptionPane;
import java.io.Serializable;

public class TopLevel implements
Serializable {
   private static final long
serialVersionUID = 42L;
   private Broker b;
   private I1 o1;

   public TopLevel() {
      super();
      }

public void topLevelMain(int
executionTime) {
int i, resp=0;

if (executionTime == 0){
b = new Broker();
PrPeProxy.setBroker (b);
o1 = (I1)PrPeProxy.newInstance(new
C1(1));
JOptionPane.showMessageDialog(null,"O1,
id and proxy were instanced");
}
JOptionPane.showMessageDialog(null,"Top
Level begins");
   for (i=0;i <= 2; i++) {

   JOptionPane.showMessageDialog(null,"
In topLevelMain - TIME "+i+": the m( )
method of o1 proxy will be invoked
before the target object");
      if (o1 != null) resp = o1.m();

   JOptionPane.showMessageDialog(null,"
In TopLevelMain - TIME "+i+": the m( )
method returns the integer value:
"+resp);
   }
```

```
   if (b != null) {
      b.display();
      b.close();
      }
   }
}
```

A classe **C1** possui dois atributos: **v** do tipo **int** e **o2** do tipo **I2** (interface) e o método **m()**. O objeto referenciado por **o2** é criado no construtor de **C1**.

No objeto semântico alvo, quer dizer, no objeto da classe **C1**, o método **m()** invoca o método **n(v)** do objeto associado ao *proxy*, referenciado pelo atributo **o2**. Após o retorno de **n(v)**, o resultado retornado é guardado no atributo **v** e exibido. Em seguida, o atributo **v** é incrementado e seu valor retornado. Esse retorno fará o caminho de volta, como já anunciamos acima, retornando à sua entrada na tabela, em seguida ao broker, depois ao *invocation handler* e, finalmente ao método **topLevelMain()**, que está no objeto da aplicação emissor.

```
package firstShallowPersistencePackage;

public interface I1 {
   public int m ();
}
```

```
package firstShallowPersistencePackage;
import java.io.Serializable;
import javax.swing.JOptionPane;

public class C1 implements I1,
Serializable {
   private static final long
serialVersionUID = 42L;
   private int v;
   private I2 o2;

   public C1(int val) {
      super();
      // TODO Auto-generated constructor
stub
      this.v= val;
      this.o2                          =
(I2)PrPeProxy.newInstance(new C2(10));

   JOptionPane.showMessageDialog(null,"
O2, id and proxy were instanced");

   JOptionPane.showMessageDialog(null,"
constructor of C1 is working");
   }
   public int m (){
      v = o2.n(v);

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

Para a instanciação do objeto referenciado por **o2**, repetimos o mesmo artifício adotado para o objeto referenciado por **o1**, acima. Assim, o objeto referenciado por **o2** também é um *proxy* da classe **P**, que, neste caso agora, é associado ao objeto semântico da classe **C2**. Como já vimos, no caso de **o1**, essa associação é efetuada durante a criação desses dois objetos, realizada pelos seguintes códigos: no método **m()**, no método estático auxiliar de construção **newInstance()** e no construtor da classe **P**. O método **newInstance()** continua recebendo dois parâmetros, o objeto semântico a ser associado e o *broker*, que é o mesmo informado no início da computação, que vai administrar a persistência do objeto semântico da classe **C2**. A classe **C2** possui um atributo chamado **num** do tipo **int** e um método **n()**.

```
package firstShallowPersistencePackage;

public interface I2 {
   public int n (int val);
}
```

```
package firstShallowPersistencePackage;

import java.io.Serializable;
import javax.swing.JOptionPane;

public class C2 implements I2, Serializable
{
   private static final long
serialVersionUID = 42L;
   private int num;
   public C2() {
      super();
      // TODO Auto-generated constructor
stub
   }
   public C2 (int number) {
      super();
      this.num = number;
   };
   public int n(int val){
      num +=10;
      JOptionPane.showMessageDialog(null,"in
method n, num attribute of o2 reference
object: "+num);
      return (num*val);
   };
}
```

Inicialmente, o método **main** interage com o usuário, afim de saber se a variável **o1** vai referenciar um novo par de objetos ou um par de objetos já existente e administrado pelo *broker*, que foi passado como parâmetro. Após estar resolvida a alocação referenciada por **o1**, o método **main()** invoca o método **m()** do objeto referenciado por **o1**. Quando essa invocação é retornada, seu resultado é exibido.

Essa invocação de **m()** passa por algumas interceptações até sua chegada ao objeto alvo semântico da classe **C1**. A primeira interceptação, efetuada pelo *proxy* (**InvocationHandler**), causa a invocação do seu método **invoke**. O procedimento comum adotado, neste método do *proxy*, é o de efetuar a invocação ao objeto alvo, como pode ser visto no exemplo anterior, que ilustra o uso do *proxy*, isoladamente. No caso que estamos apresentando aqui, em que pretendemos prover a persistência rasa, precisamos de mais uma interceptação, efetuada pelo *broker*, antes de ser invocado o método **m()** do objeto alvo semântico. Assim, durante a execução do método **invoke** no *proxy*, ainda é pedido ao *broker*, através da invocação **runThisMethod (objId, m)**, para que sejam efetuadas as últimas tarefas antes de ser efetuada a verdadeira invocação ao objeto semântico alvo. No *broker*, então, essas tarefas devem ser efetuadas na seguinte ordem: 1) acessar a entrada associada ao objeto semântico alvo da tabela de objetos administrados pelo *broker*; 2) verificar se o objeto alvo se encontra selecionado na memória heap e, caso essa condição não esteja satisfeita, restaurar esse objeto da memória secundária; e 3) efetuar a verdadeira invocação ao objeto semântico alvo.

No objeto semântico alvo, quer dizer, no objeto da classe **C1**, o método **m()** incrementa o atributo de seu objeto, chamado **v**, e invoca o método **n()** do objeto associado ao *proxy*, referenciado pelo atributo **o2**. Após o retorno de **n()**, o resultado retornado é exibido e o método **m()** retorna, como resultado, o atributo **v**, ao método **main()**, que o chamou. Nesse caminho de volta, os retornos vão sendo efetuados nos métodos de interceptação, repassando os resultados, até o chamador inicial, que é o método **main**.

As interceptações da invocação de **n()**, ocorrem da mesma forma, como as descritas acima para o método **m()**. No método **n()** do objeto semântico alvo da classe **C2**, o atributo chamado **num** é multiplicado por **10** e é retornado ao método **m()**, que foi o chamador deste método **n()**. Da mesma forma que no retorno de **m()**, os resultados vão sendo repassados de retorno em retorno, pelas interceptações, até o chamador original, que neste caso foi o método **m()**, do objeto semântico remetente, referenciado pela variável local **o1** do método **main**.

4. Conclusões

Em relação à transparência dos recursos pela aplicação, observamos que a sintaxe dos envios de mensagens aos objetos associados aos proxies o1 e o2, no exemplo apresentado, na seção 3, foi a mesma especificada para a programação em Java, sem qualquer ruído relacionado a persistência, mesmo quando esses objetos ainda não se encontravam alocados. Conseguimos reduzir esses ruídos e destacamos os seguintes aspectos de visibilidade, a seguir:

1) a tabela de entradas bem como suas entradas não são visíveis à aplicação;

2) definimos o *broker* como um atributo da classe *top-level*, mas uma outra alternativa seria defini-lo como uma variável local do método **main**, a ser preenchida pela leitura de um objeto persistente, cujo *pathname* do arquivo poderia estar em um argumento do método **main**. Devemos observar que nenhuma das alternativas nos agrada, considerando que a melhor seria que o usuário pudesse interagir com um gerente de objetos, que administra as aplicações e os *broker*s, o qual, por meio de opções ao usuário, alocaria a aplicação, o broker e preencheria, automáticamente, a referência ao *broker* no atributo de classe de **PrPeProxy**, antes de passar o controle à aplicação;

3) pela forma escolhida, inicialmente, para a definição do *broker*, em (2) , sua referência deve ser preenchida no atributo de classe **b** de **PrPeProxy** a pedido do objeto *top-level* da aplicação, no início dos tempos da execução e, no fim, deve ser pedido ao *broker* o seu fechamento, invocando o seu método **close ()**;

4) a classe **PrPeProxy**, além de entrar em cena no início dos tempos, é referenciada durante a criação do *proxy* associado a cada objeto da aplicação.

Examinando esses aspectos apontados acima, constatamos que a nossa expectativa de tornar transparentes os recursos alheios à aplicação, como por exemplo a persistência de seus objetos, somente não foi atingida, nas poucas situações a seguir. No início e fim dos tempos com o *broker*, e, em cada criação do objeto

da aplicação. Embora a qualquer momento um objeto possa ser associado a um proxy, a disciplina recomendada é fazer essa associação durante a criação do objeto. Isso é possível pela invocação ao método estático **`newInstance (Object o)`** da classe **`PrPeProxy`**, o qual devolve um *proxy*.

Como foi observado em (2) acima, poderia ser desenvolvido um gerente de objetos que administrasse o *broker* e as aplicações, tornando o *broker* transparente à aplicação, restando então apenas a referência à classe **`PrPeProxy`**, na criação de cada objeto da aplicação.

A persistência em arquivo remoto está em fase de implementação, para juntarmos a essa biblioteca, aproximando nosso pacote do padrão de java para persistência profunda.

Referências

[1] J. H. Paterson, J. Haddow, "Approaches to Object Persistence in Java Projects", *ITICSE'04*, Leeds, United Kingdom, 2004.

[2] Object Serialization http://java.sun.com/j2se/1.4.2/docs/guide/serialization/index.html

[3] J. Hrivnác, "Transparent Persistence with Java Data Objects", *CHEP'03*, La Jolla, 2003.

[4] "Hibernate – Object/Relational Mapping and Transparent Object Persistence for Java and SQL Databases" http://www.hibernate.org , 2004

[5] T. Lunney, A. McCaughey, "Object Persistence in Java", *PPPJ 2003*, Kilkenny City, Ireland, 2003.

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

[9] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, S. Spence. "Orthogonally Persistent Java", *ACM SIGMOD Record*, 1996.

[10]W. Weihl. "Local Atomicity Properties: Modular Concurrency Control for abstract Data Types". *ACM Transactions on Programming Languages and Systems*. 1989.

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