Ficha de clase número: 03
Fecha: ***Turno mañana:*** 25 de Marzo de 2008
Turno tarde: 26 de marzo de 2008

Tema Principal: Un Framework para Archivos de Registros.

Temas particulares: Planteo de un marco de trabajo (framework) para el desarrollo de clases que gestionen archivos registros de longitud fija. Uso de RandomAccessFile en ese framework. Clases Generalizadas. ABM en un archivo de registros. Bajas físicas y lógicas.

1.) Serialización (Grabación de objetos). (Referencia: DLC06-RegisterFile)

Hemos visto que la clase **RandomAccessFile** permite manejar un archivo de disco como si fuera un gran arreglo de bytes en memoria externa, brindando acceso directo a cada byte mediante el método *seek()*. Esto es muy conveniente, pero nos proponemos ahora un salto hacia arriba en cuanto a la abstracción de datos: pretendemos el desarrollo de un marco de trabajo o *framework* en el cual planteemos clases diseñadas para manejar archivos de acceso directo, pero que sean capaces de almacenar y recuperar información de los *objetos* de una aplicación.

Podría inmediatamente objetarse esta idea, alegando que Java ya permite tal cosa mediante la Serialización. Sin embargo, note el lector que nuestra propuesta es el planteo de clases para gestión de archivos de *acceso directo* que almacenen información de objetos. La *serialización* permite almacenar objetos en disco, a través de archivos que luego se manejan secuencialmente. Pero nuestro planteo sugiere la posibilidad de almacenar información de objetos, y luego saltar a la posición de cualquiera de ellos en el archivo y grabar o recuperar en esa posición, sin el recorrido secuencial.

Por otra parte, digamos que la serialización es un mecanismo sutilmente más complicado que lo que nosotros mostraremos aquí. La serialización almacena información acerca de la clase del objeto y también sobre los objetos relacionados con el que se graba, de forma de poder reconstruir integramente un objeto desde el disco, cualquiera sea su complejidad. En cambio nuestro framework buscará el tratamiento de objetos como si fueran registros de datos que deben almacenarse en disco. En la serialización es la propia máquina virtual la que determina la forma en que debe grabarse y leerse un objeto. En nuestro modelo, es el programador de la clase cuyos objetos serán grabados o leídos quien debe indicar de qué forma se graba o lee cada objeto.

Para el diseño de las clases necesarias, debemos enfrentar algunos problemas básicos, y las pautas generales que guíen el diseño pueden resumirse como sigue:

- La idea es que podamos agregar datos de objetos al archivo, modificar esos datos, y borrar datos de un objeto en el archivo. Es decir, nuestro *framework* debe permitir la implementación de la conocida técnica *ABM* (altas, bajas y modificaciones) sobre un archivo. Pero para posibilitar el *seeking* (o sea, el acceso directo al registro del objeto que queremos acceder) será necesario que los registros en disco sean todos del mismo tamaño en un archivo dado. Esto trae un potencial problema con aquellos objetos que tengan atributos de tipo *String*,

pues los valores de estos podrían diferir en tamaño entre objetos diferentes de la misma clase. Así, si una clase *Estudiante* tiene un atributo *String nombre*, distintos objetos de la clase *Estudiante* tendrán valores como *"Ana"* o *"Carlos"* cuyas longitudes en memoria son diferentes. Pero al llevarlos a disco, debemos asegurar que el valor de atributos como estos se graben en forma de cadenas del mismo largo. La solución elegida en nuestro modelo consiste en grabar las cadenas agregándoles tantos blancos como sean necesarios al final, para ajustar sus tamaños a un tamaño prefijado. Y luego, al leer esos datos, eliminar los blancos antes de usar las cadenas en memoria. En nuestro *framework* hemos desarrollado y provisto dos métodos para hacer esto: *writeString()* y *readString()* (como métodos *static* dentro de la clase *RegisterFile*).

- Como consecuencia de lo anterior, está claro que un objeto en memoria de una clase cualquiera, no será necesariamente representado en disco de la misma forma que como está en memoria... Las cadenas de caracteres al grabarse serán forzadas a tener cierta longitud fija, y tampoco nos importa en este modelo guardar información sobre los métodos que cada objeto disponía antes de ser grabado (no estamos serializando... sólo estamos guardando los datos que cada objeto tenía). La idea final es que sin importar de qué clase era un objeto, será representado dentro del archivo como un registro de longitud fija, y estos registros serán modelados como objetos de una clase llamada *Register*, diseñada a tal fin dentro del framework.

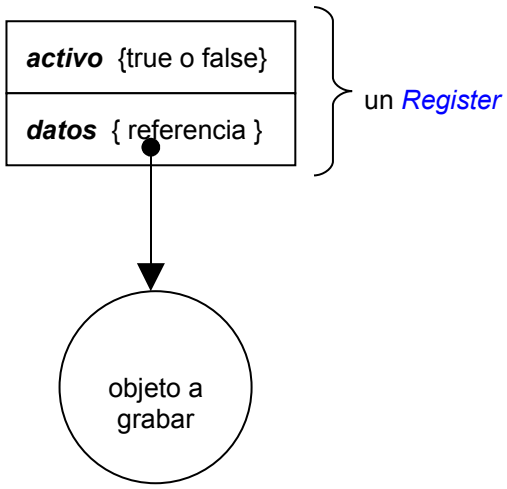
- Un problema no menor es el de realizar bajas (eliminaciones) de registros dentro del archivo. Una vez que un regsistro fue grabado, ocupa lugar en el disco dentro del archivo. ¿Cómo se borra un registro contenido dentro de un archivo? Hay dos vías clásicas:

  - ✓ La primera es el *borrado físico*: se abre un segundo archivo temporal, luego se recorre el archivo original y cada registro del mismo se va grabando a su vez en el temporal. Se hace esto con todos los registros del original, salvo aquel o aquellos que se quieren eliminar. Al finalizar, el archivo original se borra, y el temporal se renombra con el nombre del original. De esta forma, los registros se borran realmente del archivo, aunque el proceso es lento, pues deben trasladarse todos los registros al archivo temporal, incluso los que no se querían borrar.

  - ✓ La segunda es el *borrado lógico*, o *marcado lógico*, que consiste en simplemente marcar el registro como "*borrado*" o "*no borrado*". La eliminación de un registro consiste en acceder a él, leerlo, cambiar su estado de "no borrado" a "borrado", y volver a grabarlo en el mismo lugar desde donde se leyó. De esta forma el registro sigue ocupando lugar físico, pero el trabajo de hacer el marcado es mucho más rápido que el trabajo de la eliminación física (sobre todo si el acceso al registro puede hacerse en forma directa, sin recorrido secuencial).

- En la práctica se suele usar un *esquema combinado*: cuando se solicita una baja, se usa marcado lógico para reducir los tiempos de respuesta. El archivo tendrá registros activos y registros no activos. Todos los procesos deben hacerse sólo sobre los activos. Con el tiempo, la cantidad de registros no activos será molesta, pues las aplicaciones deben filtrar esos registros para no procesarlos y ese filtrado llevará cierto tiempo cada vez mayor... Entonces la idea es aplicar un borrado lógico en algún momento no crítico (por ejemplo, cada vez que el sistema arranque o cada vez que se cierre) y en ese momento eliminar de una sóla vez todos los registros marcados como borrados (o no activos).

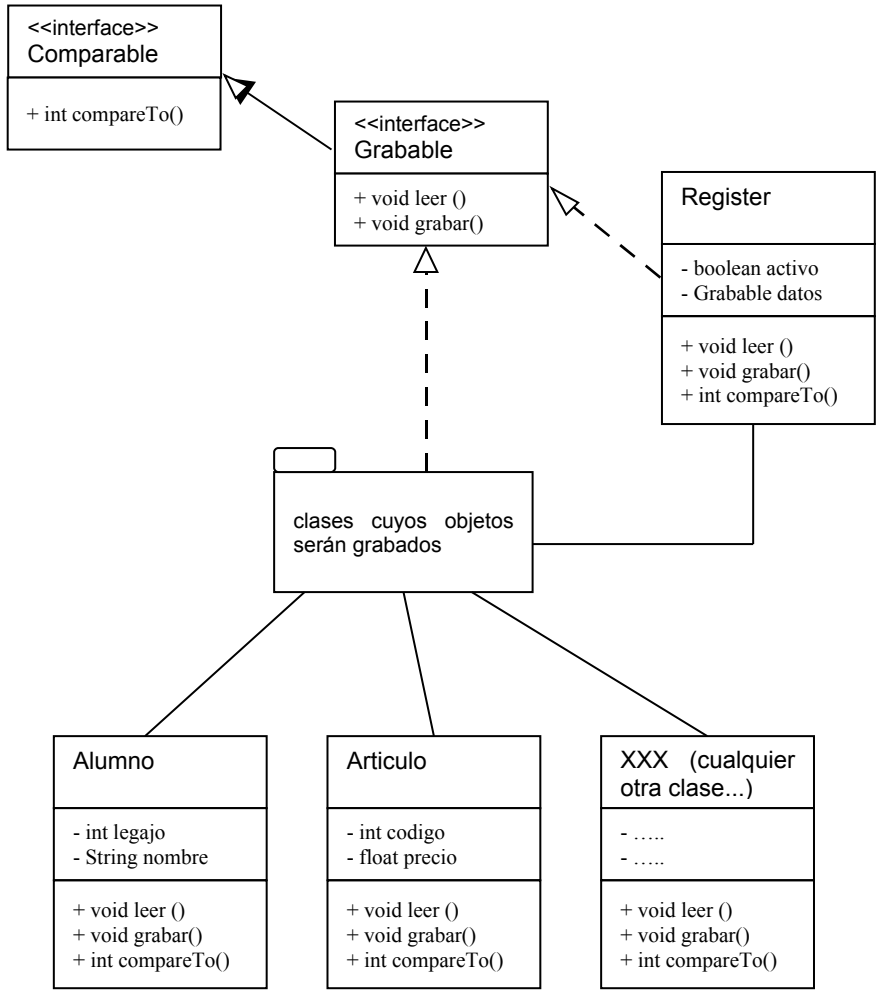
- Por lo tanto, todo registro que se grabe en un archivo de nuestro framework, deberá tener un atributo especial *boolean*, que llamaremos "***activo***", el cual indicará si ese registro está activo (no borrado) con el valor *true*, o si está no activo (borrado) con el valor *false*. Como ese atributo debe estar en todos los objetos que representen un registro, la idea en cuanto al diseño es simple: la clase *Register* tendrá ese atributo, más una referencia al objeto cuyos datos serán grabados en ese registro:



- Los objetos a grabar deberían poder ser de cualquier clase. Esto es, la referencia *datos* contenida dentro de todo *Register* debe ser polimórfica. Si bien la primera idea es que entonces *datos* sea de tipo *Object*, eso sería demasiado general, pues es de esperar que los objetos que se graben tengan alguna característica y/o comportamiento que *Object* no puede describir. Y de hecho hay una: esos objetos *deben*

saber como grabarse ellos mismos dentro de un *RandomAccessFile*. Sin importar de qué clase sean esos objetos, deberían contar al menos con un par de métodos como *leer()* y *grabar()* tales que al invocarlos puedan leer y grabar al objeto en un archivo de acceso directo, haciendo que los detalles de cómo hacer eso queden como responsabilidad de la clase (y no del programador que quiere leer o grabar). La forma obvia de lograr todo es definir una clase de interface que llamaremos *Grabable*, la cual incluirá esos métodos y algún otro que luego veremos. Entonces la referencia *datos* deberá ser de tipo *Grabable*. Como añadido, haremos que la interface *Grabable* derive de la interface *Comparable* y de esta forma nos aseguraremos que todo objeto *Grabable* también incluirá una implementación de *compareTo()*.

- Notemos que los objetos *Register* tendrán entonces una referencia a un objeto *Grabable* que contendrá los datos a grabar. Pero el propio *Register* (por incluir el atributo *activo* que debe grabarse junto a los datos) debe saber grabarse él mismo en un *RandomAccessFile*... Por ese motivo, la clase *Register* también implementará *Grabable*. Una primera idea de un diagrama de clases sería la que se ve a continuación:



- La clase encargada de representar a los archivos será *RegisterFile*. Un objeto *RegisterFile* contendrá un atributo *File fd* para manejar el pathname abstracto del archivo (o file descriptor) y un atributo *RandomAccessFile maestro* para manejar los accesos al contenido del archivo. Esta clase *RegisterFile* hará la abstracción final: será capaz de abrir y cerrar un archivo para grabar objetos *Register* en él a través de métodos como *add()* (para grabar evitando repeticiones de registros) o *append()* (para grabar sin cuidar la repetición de registros), o bien modificar registros (método *update()*) o borrar registros en forma lógica (método *remove()*). También incluirá un método para limpiar los registros borrados del archivo (método *clean())* y uno más para buscar un objeto dentro del archivo (*search()*). La clase incluirá otros métodos de soporte de operaciones básicas como *close()*, *seekByte()*, *eof()*, *length()*, *goFinal()*, *rewind()*, etc., que simplemente servirán para enmascarar llamadas a servicios de la clase *RandomAccessFile* sin tener que lidiar con la captura de excepciones de IO. La clase incluirá además un par de métodos *static writeString()* y *readString()* para grabar y leer cadenas de longitud fija.

- El servicio básico de tomar un *Register* y grabarlo en el archivo, o leer un *Register* desde el mismo, será prestado por dos métodos de *RegisterFile* llamados respectivamente ***void write(Register reg)*** y ***void read(Register reg)***. Estos dos métodos serán invocados por *add()*, *append()*, *remove()*, *update()*, *clean()* y *search()* para llevar a cabo sus operaciones. Y aquí entra el último gran desafío: en la práctica, un *Register* tiene un atributo que es una referencia a un objeto *Grabable*. Si se abre un archivo para grabar registros, se esperaría que en un *mismo archivo* sólo se puedan grabar objetos de la *misma clase* (pues de otro modo fallaría el principio básico del tamaño uniforme). Pero si invocamos a *write()* enviándole primero un *Register* que contenga un *Alumno*, y luego lo volvemos a invocar enviándole un *Register* que contenga un *Articulo*, el método *write()* haría la grabación sin problemas *ambas* veces, aún en el mismo archivo (esto es justamente lo que se esperaría del polimorfismo... no?)

- Para que un archivo manejado como *RegisterFile* tenga *contenido homogéneo* o *contenido uniforme* (es decir, que no permita que se graben datos de objetos de clases distintas) usaremos la herramienta conocida como *parametrización de clases*, la cual es soportada desde Java 5 y ampliamente usada sobre todo en el package java.util para representar estructuras de datos de contenido homogéneo. Comencemos con la clase *Register*: Queremos que al crear un *Register*, el compilador rechace cualquier intento de asociar con él a un objeto de una clase que no haya implementado *Grabable*... La siguiente sería la forma de hacerlo:

```
public class Register < E extends Grabable > implements Grabable
{
    private  boolean  activo;
    private  E  datos;

    public Register (E  d)  {
       activo = true;
       datos = d;
    }

    public void setData(E d)  {
      datos = d;
    }

    public E getData()  {
       return datos;
    }

   // el resto de los métodos aquí...
}
```

- Como se ve, al declarar la clase se usa un especificador de tipo que responde a la forma **< E extends Grabable >**. Esto indica que cuando se declare una referencia de la clase *Register*, la misma aceptará que se le imponga una clase como parámetro, de forma que si luego se intenta crear un objeto *Register* asociado con otro objeto que no sea de la clase indicada, no compilará. En este caso, el parámetro **E** es el que representa a la clase impuesta, y aquí estamos diciendo que esa clase *debe* derivar de (o implementar) *Grabable*. El atributo *datos* se declara de tipo **E**, o sea, *datos* será de la clase que **E** represente cuando el *Register* se instancie. Estas serían formas válidas de usar la clase así declarada, suponiendo que *Alumno* y *Articulo* son clases que implementaron *Grabable*:

```
Register <Alumno>  r1;
Register <Articulo>  r2;

Alumno alu = new Alumno(23, “Juan”, 8);
Articulo art = new Articulo(34, “silla”);

r1 = new Register <Alumno> (alu);
r2 = new Register <Articulo> (art);

// lo siguiente no compilará
r1 = new Register ( new Articulo() );
r2 = new Register ( new Alumno() );
```

- La propia clase *RegisterFile* también se parametriza, de tal forma de tomar un parámetro **E** que representa la clase de objetos que podrán grabarse en el archivo que se cree:

```
public class RegisterFile < E extends Grabable >
{
        private File fd;
        private RandomAccessFile maestro;
        .....

        public boolean append (E obj)
        {
            boolean resp = false;
            if( obj != null )
            {
               try
               {
                  goFinal();
                  write ( new Register (obj) );
                  resp = true;
               }
               catch(Exception e)
               {
                  System.exit(1);
               }
            }
            return resp;
         }

        // resto de los métodos aquí...
}
```

Finalmente, la interface *Grabable* queda así:

```
import java.io.*;
public interface Grabable extends Comparable
{
   int sizeOf();
   void grabar (RandomAccessFile a);
   void leer (RandomAccessFile a);
}
```

El método *sizeOf()* simplemente debe devolver el tamaño en bytes que tendría el objeto si se graba en disco, asignando a cada atributo de tipo *String* un tamaño fijo en bytes. Los métodos *grabar()* y *leer()* deben indicar cómo se graba un objeto en un RandomAccessFile, considerando que para grabar y leer cadenas deben usarse los métodos *writeString()* y *readString()* de la clase *RegisterFile*. Los tres métodos son simples de implementar, como podemos ver en el siguiente ejemplo de la clase *Alumno*, que implementa la interface (el estudiante puede tomar esta clase a modo de plantilla para programar otras clases en situaciones similares). Recuerde que *Grabable* deriva de *Comparable*, así que cualquier clase que implemente *Grabable* deberá implementar el método *compareTo()*.

```
import java.io.*;
public class Alumno implements Grabable
{
     private int    legajo;        // 4 bytes en disco
     private String  nombre;    // fijamos 30 caracteres máximo = 60 bytes en disco
     private float   promedio;   // 4 bytes en disco

     public Alumno () {
```

```
    }

    public Alumno (int leg, String nom, float prom) {
       legajo   = leg;
       nombre   = nom;
       promedio = prom;
    }

    public int sizeOf() {
         int tam = 68;       // 4 + 60 + 4. ¿Alguna duda?
         return tam;
    }

    public void grabar (RandomAccessFile a) {
       try
       {
           a.writeInt(legajo);
           RegisterFile.writeString (a, nombre, 30);
           a.writeFloat(promedio);
       }
       catch(IOException e)
       {
           System.exit(1);
       }
    }



    public void leer (RandomAccessFile a) {
       try
       {
           legajo   = a.readInt();
           nombre   = RegisterFile.readString(a, 30).trim();
           promedio = a.readFloat();
       }
       catch(IOException e)
       {
           System.exit(1);
       }
    }

    public int compareTo (Object x)
    {
      Alumno a = (Alumno) x;
      return this.legajo - a.legajo;
    }

    // resto de los métodos aquí...
}
```

Como se ve, *sizeOf()* sólo retorna la suma de los *bytes* que el objeto ocupará en disco, tomando a cada cadena con una longitud en caracteres asignada de hecho y un número de bytes igual a esa cantidad de caracteres, por dos. La longitud supuesta para cada cadena es elegida por el programador, a su arbitrio. Y los métodos *read()* y *write()* simplemente indican en qué orden se leen o se graban los atributos en el *RandomAccessFile* que viene como parámetro. El archivo representado por el RandomAccessFile debe venir abierto y ya

posicionado en el lugar correcto. Note que ambos métodos capturan la posible excepción de IO que se produciría en la operación, facilitando su uso posterior.

La clase *Register* es también simple. Recuerde que no sólo impone que el atributo datos sea de una clase **E** que derive o implemente *Grabable*, sino que la propia *Register* implementa *Grabable*. Note también que tiene un solo constructor, que exige un atributo de la clase parametrizada **E**:

```
import java.io.*;
public class Register < E extends Grabable > implements Grabable
{
    private boolean activo;    // marca de registro activo. Ocupa 1 byte en disco
    private E datos;           // los datos "puros" que serán grabados

    public Register (E  d) {
       activo = true;
       datos = d;
    }

    public boolean isActive () {
       return activo;
    }

    public void setActive(boolean x) {
      activo = x;
    }

    public void setData(E d) {
      datos = d;
    }

    public E getData() {
       return datos;
    }

    public int sizeOf() {
      return datos.sizeOf() + 1;   // suma 1 por el atributo "activo", que es boolean
    }

    public void grabar (RandomAccessFile a) {
       try
       {
          a.writeBoolean(activo);
          datos.grabar(a);
       }
       catch(IOException e)
       {
          System.exit(1);
       }
    }

    public void leer (RandomAccessFile a) {
       try
       {
          activo = a.readBoolean();
          datos.leer(a);
       }
       catch(IOException e)
```

```
        {
            System.exit(1);
        }
    }

    public String toString() {
      return getData().toString() + "\tActivo?: " + isActive();
    }

    public int compareTo (Object x) {
        Register r = (Register) x;
        return datos.compareTo(r.datos);
    }
}
```

Observe la sencillez del método *sizeOf()*: llama al método *sizeOf()* implementado en la clase del atributo *datos*, y suma 1 al valor retornado. El uno sumado, es el tamaño en bytes del atributo *activo* (un *boolean* se graba como un byte en disco). Los métodos *read()* y *write()* aplican una táctica parecida: llaman al método de la clase del atributo datos, y completan la operación leyendo o grabando un byte para el atributo *activo*.

Finalmente, la clase *RegisterFile* usa todas estas piezas en la forma que se describió. Listamos una parte de la clase para tomar una idea básica de algún detalle adicional:

```
import java.io.*;
public class RegisterFile < E extends Grabable >
{
   private File fd;                              // descriptor del archivo
   private RandomAccessFile maestro;     // manejador del archivo

   private Grabable testigo = null;     // registra la clase de objetos que se graban en el archivo

   public RegisterFile ( E obj )
   {
      this ("newfile.dat", "r", obj);
   }

   public RegisterFile (String nombre, String modo, E obj)
   {
      this (new File(nombre), modo, obj);
   }

   public RegisterFile (File file, String modo, E obj)
   {
      if (obj == null) { throw new NullPointerException ("No se indicó la clase a grabar..."); }
      testigo = obj;
      fd = file;
      if ( modo == null || ( !modo.equals("r") && !modo.equals("rw") ) )
      {
         modo = "r";
      }

      try
```

```
    {
       maestro = new RandomAccessFile(fd, modo);
    }
    catch(IOException e)
    {
       System.exit(1);
    }
}

public boolean delete()
{
    return fd.delete();
}

public boolean rename(RegisterFile nuevo)
{
    return fd.renameTo(nuevo.fd);
}

public void close()
{
    try
    {
       maestro.close();
    }
    catch(IOException e)
    {
          System.exit(1);
    }
}


public void seekByte (long b)
{
    try
    {
       maestro.seek(b);
    }
    catch(IOException e)
    {
          System.exit(1);
    }
}

public void rewind()
{
    try
    {
       maestro.seek(0);
    }
    catch(IOException e)
    {
          System.exit(1);
    }
}

public long bytePos ()
{
    try
```

```
    {
       return maestro.getFilePointer();
    }
    catch(IOException e)
    {
          System.exit(1);
    }
    return -1;
}

public void goFinal ()
{
    try
    {
       maestro.seek(maestro.length());
    }
    catch(IOException e)
    {
        System.exit(1);
    }
}

public long length()
{
    try
    {
       return maestro.length();
    }
    catch(IOException e)
    {
          System.exit(1);
    }
    return 0;
}

public boolean eof ()
{
    try
    {
       if (maestro.getFilePointer() < maestro.length()) return false;
       else return true;
    }
    catch(IOException e)
    {
        System.exit(1);
    }
    return true;
}

public boolean write (Register r)
{
    if(r != null)
    {
       try
       {
          r.grabar(maestro);
       }
       catch(Exception e)
       {
```

```
            System.exit(1);
        }
        return true;
    }
    return false;
}

public boolean read (Register r)
{
    if(r != null)
    {
        try
        {
            r.leer(maestro);
        }
        catch(Exception e)
        {
            System.exit(1);
        }
        return true;
    }
    return false;
}

public long search (E obj)
{
    if( obj == null ) return -1;
    long pos = -1, actual;
    try
    {
        Grabable r2 = obj.getClass().newInstance();
        Register reg = new Register(r2);
        actual = bytePos();

        rewind();
        while (!eof())
        {
            read(reg);
            if (reg.getData().equals(obj) && reg.isActive())
            {
                pos = bytePos() - reg.sizeOf();
                break;
            }
        }
        seekByte(actual);
    }
    catch(Exception e)
    {
            System.exit(1);
    }
    return pos;
}


public boolean add (E obj)
{
    boolean resp = false;
    long pos;
```

```
    if( obj != null )
    {
       try
       {
          pos = search(obj);
          if (pos == -1)
          {
             goFinal();
             write(new Register <E> (obj));
             resp = true;
          }
       }
       catch(Exception e)
       {
          System.exit(1);
       }
    }
    return resp;
}

public boolean append (E obj)
{
    boolean resp = false;
    if( obj != null )
    {
       try
       {
          goFinal();
          write(new Register <E> (obj));
          resp = true;
       }
       catch(Exception e)
       {
          System.exit(1);
       }
    }
    return resp;
}

public boolean remove (E obj)
{
    boolean resp = false;
    long pos;

    if( obj != null )
    {
       try
       {
          Grabable r2 = obj.getClass().newInstance();
          Register reg = new Register(r2);

          pos = search(obj);
          if (pos != -1)
          {
              seekByte(pos);
              read(reg);
              reg.setActive(false);

              seekByte(pos);
```

```
                write(reg);
                resp = true;
            }
        }
        catch(Exception e)
        {
            System.exit(1);
        }
    }
    return resp;
}

public void clean()
{
    try
    {
        Grabable r2 = testigo.getClass().newInstance();
        Register reg = new Register(r2);

        RegisterFile temp = new RegisterFile ("temporal.dat", "rw", testigo);
        temp.maestro.setLength(0);
        this.rewind();
        while (!this.eof())
        {
            this.read( reg );
            if ( reg.isActive() )
            {
                temp.write(reg);
            }
        }

        this.close();
        temp.close();

        this.delete();
        temp.rename(this);
    }
    catch(Exception e)
    {
        System.exit(1);
    }
}

 public static final String readString (RandomAccessFile arch, int tam)
 {
     String cad = "";

     try
     {
        char vector[] = new char[tam];
        for(int i = 0; i<tam; i++)
        {
           vector[i] = arch.readChar();
        }
        cad = new String(vector,0,tam);
     }
     catch(IOException e)
     {
          System.exit(1);
```

```
        }

        return cad;
    }

    public static final void writeString (RandomAccessFile arch, String cad, int tam)
    {
        try
        {
            int i;
            char vector[] = new char[tam];
            for(i=0; i<tam; i++)
            {
              vector[i]= ' ';
            }
            cad.getChars(0, cad.length(), vector, 0);
            for (i=0; i<tam; i++)
            {
              arch.writeChar(vector[i]);
            }
        }
        catch(IOException e)
        {
             System.exit(1);
        }
    }
}
```

Al final de la clase se pueden ver los métodos static *writeString()* y *readString()*. El primero de ellos toma como parámetro un *RandomAccessFile* ya abierto y posicionado, más un String para grabar, más un valor tam que es la cantidad de caracteres a la cual se quiere ajustar el *String* cuando se grabe. El método arma un arreglo de caracteres inicialmente lleno de blancos, de tamaño *tam*, y vuelca el contenido del *String* en ese arreglo. Luego lo graba caracter a caracter. Y el método *readString()* toma como parámetro un *RandomAccessFile*, más la cantidad tam de caracteres a leer. Crea un arreglo de tam caracteres, y luego lo va llenando con caracteres que lee desde el disco. Cuando termina, convierte el arreglo en un *String* y lo retorna. Notar que los blancos que pudieran haber quedado al fondo del *String* no se remueven.

La clase tiene un atributo *Grabable testigo* el cual se usa para mantener una referencia a un objeto cualquiera de la clase cuyos objetos se graban en el archivo. Los constructores de la clase *RegisterFile* piden como parámetro una referencia a un objeto de la clase a grabar, y esa referencia se asigna en el atributo *testigo*. Esto se hace pues algunos métodos de la clase *RegisterFile* necesitan en algún momento crear un objeto de la clase que se graba, para poder a su vez crear un *Register* y comenzar a leer o grabar en el archivo. Un ejemplo es el método *clean()* que elimina los registros marcados como borrados. Como el método *clean()* no sabe de qué tipo son los objetos contenidos en el *Register*, no puede crear un *Register* que le permita leer del archivo original o grabar en el temporal. El atributo testigo siempre apunta a un objeto de la clase que se graba, por lo cual la expresión:

```
Grabable r2 = testigo.getClass().newInstance();
```

permite crear un objeto de la clase correcta, que luego se usa para crear un *Register*:

```
Register reg = new Register (r2);
```

Otros métodos de la clase enfrentan un problema parecido, pero la diferencia es que estos otros toman un parámetro *E obj*, que puede usarse para crear una instancia de *Register* sin recurrir al testigo. Es el caso del método *add()*, entre otros:

```
public boolean add (E obj)
{
   boolean resp = false;
   long pos;

   if( obj != null )
   {
      try
      {
         pos = search(obj);
         if (pos == -1)
         {
            goFinal();
            write(new Register <E> (obj));
            resp = true;
         }
      }
      catch(Exception e)
      {
         System.exit(1);
      }
   }
   return resp;
}
```

Para finalizar, hagamos una breve exposición de algunas ventajas y desventajas de nuestro framework:

☺ ***Lo bueno:*** Este *framework* permite crear archivos de objetos, ajustando su contenido a una clase particular de objetos. Permite hacer altas, bajas y modificaciones, así como bajas físicas si fuera necesario. La clase *RegisterFile* puede modificarse para seguir añadiendo funcionalidad: podría pensarse en métodos para ordenar el archivo (al fin y al cabo, el ordenamiento requiere acceso directo y un *RegisterFile* lo tiene... el método *seekByte()* brinda esa característica). Además, es sencillo crear clases que implementen Grabable para "subirse" a un *RegisterFile*.

😐 ***Lo malo:*** La clase *RegisterFile* pide en sus constructores un parámetro que luego se asigna como *testigo* de la clase que se graba... Eso no parece necesario pues la clase *RegisterFile* está parametrizada con la clase **E** (que representa a la clase testigo que estamos necesitando) Sin embargo, el parámetro **E** no brinda ninguna información acerca de la clase real en tiempo de compilación, por lo que el testigo que apunte a un objeto creado en tiempo de ejecución parece ineludible. Además, la clase *RegisterFile* todavía está en etapa de testing... y es posible que algún problema de parametrización de clases aún subsista en métodos como *search()*, *remove()* o *clean()*, en los cuales se crean objetos *Register* con la siguiente secuencia:

```
Grabable r2 = obj.getClass().newInstance();
Register reg = new Register(r2);
```

La creación del *Register* en la segunda línea no está parametrizada en cuanto a la clase que se acepta como válida para el *Register*, por lo que en principio cualquier clase sería aceptada para r2. Si bien los métodos están planteados de forma que no parece que pueda llegar un objeto incorrecto hasta esas líneas, el tema es que el posible *bug* existe (aún).

😐 ***Lo feo:*** El parámetro **E** con el que se parametrizan las clases *Register* y *RegisterFile* se declara de forma que se acepta que derive de *Grabable*: <E extends Grabable >. Esto implica que cualquier objeto de cualquier clase que derive de o implemente *Grabable* podrá almacenarse en nuestros archivos. Hasta aquí todo bien. Pero en un mismo archivo podrían entonces grabarse objetos de clases distintas: Si **A** implementa *Grabable*, y **B** deriva de **A**, entonces ambas clases encajan con **E** y un mismo archivo dejaría que se graben objetos **A** y objetos **B**... ¿Porqué decimos que esto es *feo* si en términos de POO es perfectamente natural? Por que tendríamos registros de distinto tamaño en el mismo archivo, y adiós *seeking*...

➢ ***Ejercicios para la semana 3:*** Ninguno de los dos ejercicios deben ser entregados por aula virtual. Uno es un foro abierto, y el otro un desarrollo de aula.

a.) Se agregará en el aula virtual un foro interno para discutir los temas planteados como bueno, malo y feo. Se invita a todos a expresar sus opiniones y plantear constructivamente soluciones a los problemas planteados. En ese mismo foro, se lanzarán temas técnicos relativos a la materia y al lenguaje Java.

b.) Desarrollar en aula una clase que implemente Grabable, y luego un main que cree un RegisterFile para grabar objetos de esa clase en el archivo. Siga el modelo del Ejemplo DLC06-RegisterFile.