FIUBA

75-62 Técnicas de Programación Concurrente II
2006

Trabajo Práctico 1
Repaso General de Java

Ing. Osvaldo Clúa

Los siguientes proyectos están dirigidos a un repaso de Java básico. En los TP posteriores se irán combinando. Es obligatorio respetar los nombres del eneunciado.

Además de lo pedido, una clase de prueba deberá explorar todas las opciones. Los tres puntos del Ejercicio 3 engloban a los anteriores. El 4 es independiente.

1 Fracciones

1.1 Programar una clase **Fraccion** con campos privados **num** y **den** para numerador y denominador. Programar los métodos de get y set correspondientes.

1.2 Agregarle un método **toString**();

1.3 Hacer un método privado estático **mcd** que encuentre el mcd entre el **num** y el **den** . Usarlo para un método privado **normalizar.**

1.4 Crear una ***Exception*** **CeroDenExc** que tenga como dato el numerador. Esta excepcion se lanzará desde un constructor cada vez que se quiera crear una fracción con denominador cero.

1.5 Agregar además un constructor sin parámetros (crea una fracción 1/1) y un constructor de copia.

1.6 Modificar los constructores necesarios para que toda fracción este siempre normalizada (simplificada).

1.7 Crear una ***interface*** **OperFrac** con los métodos **mas** y **menos** que reciben una fracción y devuelven otra fracción con la suma o la resta de las dos fracciones respectivamente. Modificar **Fraccion** para que implemente esta clase.

1.8 Crear una ***interface*** **ComparaFrac** con un método **compareTo** que recibe una **Fraccion** devuelve un entero positivo, negativo o nulo si la **Fraccion *this*** es mayor, menor o igual al parámetro.

2 Angulos

2.1 Repetir el ejercicio anterior para una clase **Angulo** capaz de almacenar grados, minuto y segundos sexagesimales. El constructor sin parámetros construirá un ángulo de (0,0,0) .No tendrá ***Exception*** y la normalización asegurará que el ángulo esté entre]-360 y 360[(intervalo abierto en ambos extremos), los minutos y los segundos menores que 60. Puede haber ángulos negativos. Las operaciones **mas** y **menos** son de la ***interface*** **OperAng** y la **compareTo** de la interface **ComparaAng.**

3 Interfaces gráficas.

3.1 Construir una interface gráfica para visualizar un objeto del tipo ***Fracción*** llamada **FrGraf**. Desciende de ***Frame,*** como título lleva el identificador y como contenido el valor. **FrGraf** tiene una copia de la fracción que representa.

3.2 Agregarle un botón "mitad" que cambie el valor de la fracción a la mitad.

3.3 Repetir para ***Angulo*** pero con el valor del ángulo y un dibujo del mismo. El botón convierte el ángulo a su mitad.

4 Eventos

4.1 Siguiendo el modelo de la explicación que sigue a los ejercicios, programar con eventos un juego de sorteos. Una clase *Sorteo* con un método *go()* va sacando números al azar. Cuando uno de los números está entre [40,49] se notifica a los objetos *Cantadores* que festejan con un mensaje y termina el juego.

4.2 Modifique el anterior para que haya al menos 3 *Cantadores* distintos y que cada uno verifique un intervalo distinto.

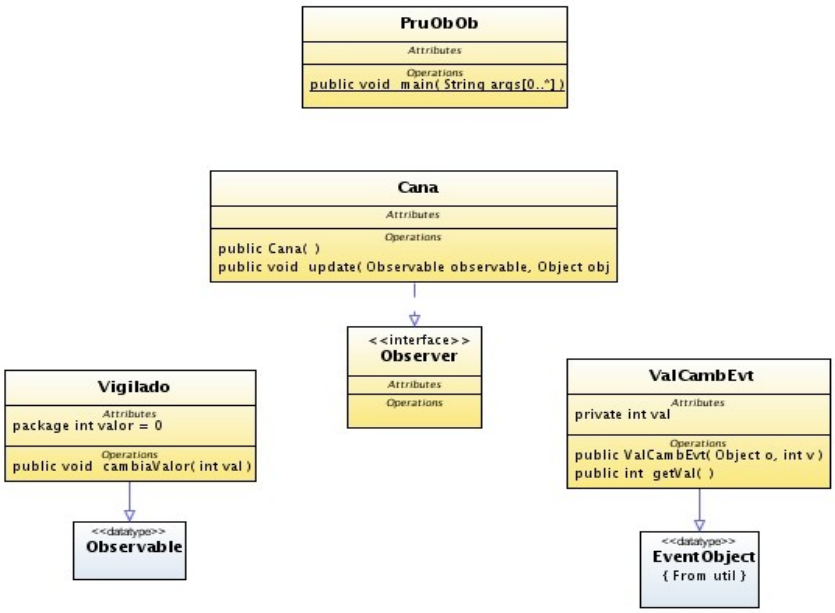
4.3 Haga una interface gráfica para 4.2 usando MVC.

En la explicación que sigue hay entremezclado algunos ejercicios de práctica. No son obligatorios para entregar este Trabajo Práctico pero su solución debe conocerse (y ayudan ...)

Eventos

Java ofrece una implementación del *pattern* conocido como *Observer.* En este *pattern* los objetos definidos como Observadores son notificados cuando algo ocurre en el o los elementos definidos como Observables. Es uno de los componentes primitivos del viejo paradigma llamado *Model View Controller.*

Su UML es:



La clase *Observable* es **Vigilado** que extiende la clase ***Observable*** de *Java.util*

```
1. import java.util.*;
2. /*
3.  * Vigilado.java
4.  *
5.  */
6. public class Vigilado extends Observable{
7.
8.     int valor=0;
9.     public void cambiaValor(int val) {
10.        valor=val;
11.        setChanged();
12.        notifyObservers(new ValCambEvt (this,val));
13.        clearChanged();
14.    }
15.
16.}
```

Cuando se cambia su valor, coloca su estado de observación en *cambiado* y notifica a los *Observadores* interesados. Luego borra el estado de *cambiado*. Es interesante ver que pasa si se comentan las líneas que manejan el estado.

La notificación lleva un evento del tipo **ValCamEvt** que extiende ***EventObject***

```
1. import java.util.EventObject;
2. /*
3.  * ValCambEvt.java
4.  */
5. public class ValCambEvt extends EventObject{
6.
7.     private int val;
8.     public ValCambEvt(Object o, int v){
9.         super(o);
10.        val=v;
11.    }
12.
13.    public int getVal() {
14.        return val;
15.    }
16.
17.}
```

Este evento lleva como dato el objeto que produjo el evento y el nuevo valor. Al descender de ***EventObject*** está obligada a recibir en su constructor el ***Object*** o que es quien produjo el evento. No hay obligación de usar eventos, ***notifyObservers(...)*** acepta cualquier objeto como parámetro.

La clase vigilante es Cana que implementa ***Observer***:

```
1. /*
2.  * Cana.java
3.  */
4. import java.util.*;
5. public class Cana implements Observer {
6.
7.     public Cana() {
8.     }
9.
10.    public void update(Observable observable, Object obj) {
11.        System.out.println(" Se ha cambiado el Objeto "+observable+" a
   "+((ValCambEvt) obj).getVal());
12.    }
13.
14.}
```

Debe programar entonces *update* con dos parámetros: El primero es el ***Object*** que cambió y el segundo un argumento cualquiera (en nuestro caso un evento). Al hacerse el ***notifyObservers(...)*** se llama a los observadores en cualquier orden en el método ***update***. En este caso se imprime un mensaje después de hacer el *cast* correspondiente.

El *main* es:

```
1. /*
2.  * PruObOb.java
3.  */
4.
5. public class PruObOb {
6.
```

```
7.      public static void main(String[] args) {
8.          Vigilado v1=new Vigilado();
9.          Vigilado v2=new Vigilado();
10.         System.out.println("v1="+v1);
11.         System.out.println("v2="+v2);
12.         Cana c=new Cana();
13.         v1.addObserver(c);
14.         v2.addObserver(c);
15.         v1.cambiaValor(34);
16.         v2.cambiaValor(55);
17.
18.     }
19.
20.}
```

Como práctica juegue con el código:

1. Cambie el evento por un objeto cualquiera.
2. Coloque dos observadores al mismo observable.
3. ¿Qué otros métodos tiene *Observable*?
4. ¿Qué pasa si no se usan las llamadas a *stateChanged( )?*
5. ¿Hay algo de concurrencia en este *pattern*? y ¿en el manejo de las GUI?

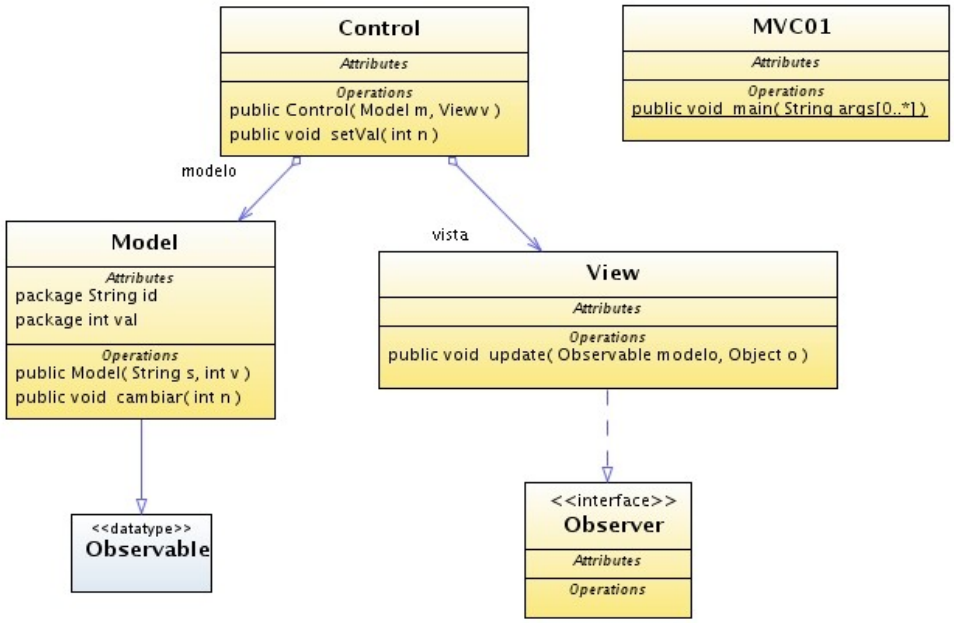
Los otros *patterns* que componen el MVC son *Composite* y *Strategy. Composite* es un *pattern* que implementa la relación *has_a,* factoreando las operaciones comunes de un grupo a una *interface.*

Un C*omposite* se encarga de aplicarlas a cada elemento del grupo (ademas hacen falta los métodos para agregar y quitar componentes y para acceder a ellos.

Strategy permite separar los algoritmos de su uso, generalmente haciendo que los algoritmos implementen una *interface* común (por ejemplo *execute( )*)

Ambos *pattern* aparecerán mas adelante en la materia.

Un ejemplo de MVC simplificado:



En esta caso v**ista** que es el ***viewer*** solo refleja los cambios del ***Model*** **modelo.** En el caso general, el ***viewer*** debería además manejar la interacción con el operador. La clase ***controller*** es **Control** que es quien maneja el modelo. el debería recibir además las entradas del operador desde **vista**. El *pattern Observer* es aparente (***Model -> View***). Si hubiera mas de una interface aparecería mas claro el *Composite* (***Control -> View)***. Si el "algoritmos" del modelo cambiara, aparecería el *pattern* de *Strategy* ***(Control->Model*** por medio de ***setVal(...)).*** **MVC01** es la clase principal.

```
1. /*
2.  * Model.java
3.  */
4. import java.util.*;
5. public class Model extends Observable {
6.     String id;
7.      int val;
8.     public Model(String s,int v) {
9.         id=s;
10.        val=v;
11.    }
12.    public void cambiar(int n){
13.        val=n;
14.        setChanged();
15.        notifyObservers(n);
16.        clearChanged();
17.    }
18.
19.}
20./*
21. * View.java
22. */
```

```
23.import java.util.*;
24.public class View implements Observer{
25.    public void update(Observable modelo, Object o){
26.        String id=((Model)modelo).id;
27.        Integer n=(Integer)o;
28.        System.out.println ("(View) El objeto "+id+" cambio al valor "+n);
29.    }
30.
31.}
32./*
33. * Control.java
34. */
35.
36.public class Control {
37.    Model modelo;
38.    View vista;
39.    public Control(Model m, View v) {
40.        modelo=m;
41.        vista=v;
42.    }
43.    public void setVal(int n){
44.        modelo.cambiar(n);
45.    }
46.}
47./*
48. * MVC01.java
49. */
50.
51.public class MVC01 {
52.
53.    public static void main(String[] args) {
54.        Model m= new Model ("Uno",45);
55.        View v= new View();
56.        Control c=new Control(m, v);
57.        m.addObserver(v);
58.        c.setVal(7);
59.        c.setVal(8);
60.    }
61.
62.}
```

Para practicar:

1. Agregue una interface gráfica al código anterior.
2. Permita modificar el valor desde la interface gráfica.