Práctica 1- Programación con 4 patrones en Java

Patrón *Factoría Abstracta*

1) Programar utilizando hebras la simulación de 2 carreras simultáneas con el mismo número inicial (N) de bicicletas. N no se conoce hasta que comienza la carrera. De las carreras de montaña y carretera se retirarán el 20% y el 10% de las bicicletas, respectivamente, antes de terminar. Ambas carreras duran exactamente 60 s. y todas las bicicletas se retiran a la misma vez.

Supondremos que Carrera, una clase interfaz de Java, declara el método de fabricación: `crearCarrera()`, que implementará una clase factoría (`FactoriaCarrera`) para la creación de 2 objetos `ArrayList` de Java que van a incluir las bicicletas de cada tipo (`CARRETERA`, `MONTANA`), que participan en cada una de las dos modalidades de carrera.

Asumimos que tenemos dos clases factoría específicas: `FactoriaCarreraMontaña` y `FactoriaCarreraCarretera`, que implementarán cada una de ellas el método de fabricación crearCarrera()anterior que, a su vez, creará uno de los objetos: `ArrayList<Bicicleta>[N]` llenándolo de la clase de bicicletas compatible con la modalidad de carrera que nos interese.

Por otra parte, la clase Bicicleta se debería definir como una clase abstracta y en su constructor se pasaría como argumento un valor del tipo:

```
public  enum TC{
   MONTANA, CARRETERA
  }
```

y el resto de sus métodos podrían ser: `TC getTipo()`, `void setTipo()`, redefiniendo además el método `toString()` para que devuelva la cadena: "*bicicleta* - " + *tipo*.

En Java no existe una forma estándar de liberar memoria (como `delete()` de C++), pero necesitaremos eliminar bicicletas durante la ejecución del programa. Supondremos que el asignar explícitamente referencias a `null` se puede considerar una técnica legítima de liberar memoria en Java y dejaremos que su recolector de basura haga el trabajo automáticamente.
Para programar la simulación de las 2 carreras simultáneas de bicicletas utilizaremos creación de hebras de Java.

Patrón Observador

2) Programar, utilizando este patrón de diseño, un programa simple de simulación de la monitorización de datos meteorológicos. El programa `Simulador` se ha de encargar de generar aleatoriamente una temperatura dentro de un rango pre-definido: `[t_1,…,t_2]` de temperaturas, cada `60 s.` aproximadamente. `t_1` y `t_2` no se conocen hasta que comienza la ejecución del `Simulador`, ya que dicho rango de temperaturas dependerá de la época del año y de la región. El programa definirá un método ejecutable (método público `run()`) para su implementación como una hebra de Java, tal como el siguiente:

```
Random r= new Random(t2); int temperatura;
while (true){
  temperatura= r.next(Integer);
  try {sleep(60)}
  catch(java.lang.InterruptedException e){
    e.printStackTrace();
  }
  observableTemperatura.notificarObservador();
}
```

Se supondrá que el programa de usuario ha de crear una única instancia de la clase `Pantalla`, la cual también implementará la interfaz `Observador`. Esta clase define un método público y estático: `refrescarPantalla()`, que muestra el valor actual de la temperatura al programa de demostración del usuario en un formato textual, pero dentro de una entidad *frame* de Java/Swing. `public static void refrescarPantalla()` se puede programar como sigue:

```
… private static JLabel
…
labelTemperatura.setText(String.valueOf(observableTemperatura.getTemp()));
```

La clase `Pantalla` tiene, por tanto, un carácter de *observador* y también ha de heredar de `javax.swing.JFrame` e implementar la interfaz `Runnable`, ya que la pantalla ejecutará un bucle que sólo llama a `refrescarPantalla()`, independientemente del simulador.

Suponemos la existencia de una clase interfaz `Observador` de Java, que declara el método de actualización: `manejarEvento()`, para que lo implementen las clases observadoras concretas . Este método se puede programar, por ejemplo, en una clase observadora específica como sigue:

```
temp = ObservableTemperatura.getInstancia().getTemp();

System.out.println("Estoy con evento de PantallaTemperatura: " + temp + " °C.");
```

Por otra parte, el método `manejarEvento()` de una clase *`observadora`* ha de contener la lógica necesaria para actualizar la presentación de la información al usuario tanto en grados Celsius como Fahrenheit y también ha de encargarse de "repintar" la pantalla.

Además del anterior, se pueden añadir a la aplicación observadores adicionales, tales como: `botonCambio`, `graficaTemperatura`, `tiempoSatelital`, para mejorar el aspecto y funcionalidad. En este caso, `botonCambio` sería una entidad observadora a la que notificaría el programa `Simulacion`, pero también podría servir para cambiar el estado de la simulación asignando directamente el valor de la temperatura actual.

Por otra parte, debido a motivos de reusabilidad del código, podemos asumir dos clases observables: `ObservablePantalla (subclase)` y `Observable` de la que hereda, consiguiéndose, de esta forma, un código más claro y transportable. Una clase *observable* ha de implementar los métodos públicos: `incluirObservador(Observador o)`, `notificarObservador()`, de acuerdo con el patrón que estamos utilizando.

A instancia del objeto Simulador se crea el `observablePantalla`, asignándole memoria. A su vez, esta clase incluye el método `incluirObservador()` que asignará memoria a una lista de objetos, que implementan la interfaz `Observador`:

```
private ArrayList<Observador> observadores = new ArrayList<Observador>();
…
  public void incluirObservador(Observador o) {
       observadores.add(o);
    }
```

El método `notificarObservador()` se puede implementar haciendo una llamada cada uno de los métodos de actualización `manejarEvento()` incluidos en el array-list `observadores` correspondiente.

Patrón Visitante

3) Utilizando este patrón se pretende recorrer una estructura de objetos, y desarrollar un programa para generar presupuestos de configuración de un computador simple, que está conformado con los siguientes elementos: Disco, Tarjeta, Bus. El programa mostrará el precio de cada posible configuración de un equipo:

```
public abstract class Equipo{
  private String nombre;
  public Equipo(String nombre){

    this.nombre= nombre;
  }
  public String nombre(){
    return nombre;

  }
  public abstract double potencia();
  public abstract double precioNeto();
  public abstract double precioConDescuento();
  public abstract void aceptar(VisitanteEquipo ve);
}
```

Las clases `Disco`, `Tarjeta`, `Bus` extienden a la clase abstracta `Equipo` e implementan todos sus métodos.

La programación del método `aceptar(VisitanteEquipo ve)` en cada una de las clases anteriores consistirá en una llamada al método correspondiente de la clase abstracta `VisitanteEquipo`:

```
public abstract class VisitanteEquipo{

  public abstract void VisitarDisco(Disco d);

  public abstract void VisitarTarjeta(Tarjeta t);

  public abstract void VisitarBus(Bus b);

 }
```

Las subclases de `VisitanteEquipo` definirán algoritmos concretos que se aplican sobre la estructura de objetos que se obtiene de instanciar las subclases de Equipo. Por ejemplo, la subclase visitante: `VisitantePrecio` puede servir para calcular el coste neto de todas las partes que conforman un determinado equipo (disco+tarjeta+bus), acumulando internamente el costo de cada parte después de visitarla.

Además utilizando el patrón Visitante podemos adaptar la tabla de precios, que incluye a todos los componentes de un equipo, a diferentes tipos de clientes (clientes “VIP”, con descuento especial, etc.), simplemente cambiando la clase `VisitantePrecio`.

El programa `Cliente` se encargará de generar aleatoriamente el tipo de cliente, es decir: cliente sin-descuento, VIP (10% descuento), mayorista(15% descuento) y obtener el coste total de una configuración de equipo utilizando para ello una instancia de subclase de `VisitanteEquipo`.

Programar más subclases del tipo `Visitante` para mostrar los nombres de las partes que componen un equipo y sus precios. Ahora el programa `Cliente` deberá mostrar, además del coste total de un equipo, las marcas de sus componentes.

Patrón Interceptor

4) Utilizando el patrón arquitectónico “Interceptor”, aplicado a una parte del problema de control SCACV ("sistema de control automático para la conducción de un vehículo"), desarrollar un diagrama de clases y un proyecto de Eclipse/Java para simular el cálculo de la velocidad inicial del vehículo a partir de un dato de entrada, p.e.: revoluciones del eje. Después, la aplicación instala un manejador de eventos que *reacciona* cuando se pulsen cualquiera de los 2 botones: ``*Encender*" (el motor del vehículo) y "*Acelerar*" la velocidad de crucero.

Para que el ejercicio sea considerado correcto hay que realizarlo de acuerdo con los siguientes requerimientos:

-Programar una clase anónima `(WindowAdapter())` con Swing (Java) para terminar bien la ejecución de la clase Interfaz correspondiente:

```
this.addWindowListener (new WindowAdapter(){
   public void windowClosing(WindowEvent e){
     System.exit(0);
   }
});
```

Hay que programar los botones “Encender”, “Acelerar” y la etiqueta “APAGADO” / “ACELERANDO” dentro de un objeto panel de botones, cuyo esqueleto en Swing sería algo como lo siguiente:

```
import java.awt.*;import javax.swing.BoxLayout;
import javax.swing.JPanel;import javax.swing.border.*;
public class PanelBotones extends JPanel {
  private javax.swing.JButton BotonAcelerar, BotonEncender;
  private javax.swing.JLabel EtiqMostrarEstado;
  public PanelBotones(){ ...};//constructor
  synchronized private void
      BotonAcelerarActionPerformed(java.awt.event.ActionEvent evt){...};
  synchronized private void
     BotonEncenderActionPerformed(java.awt.event.ActionEvent evt){...};
}
```

Funcionamiento de los botones:

-Inicialmente la etiqueta del panel principal mostrará el texto “APAGADO” (ver figura a) y las etiquetas de los botones, el nombre de cada uno.

-El botón “Encender” será de selección de tipo conmutador `JToggleButton`, cambiando de color y de texto (“Encender”/”Apagar”) cuando se pulsa.

-La pulsación del botón “Acelerando” cambia el texto de la etiqueta del panel principal a “ACELERANDO” (ver figura b), pero sólo si el motor está encendido; si no, no hace caso a la pulsación del usuario.

-Si ahora se pulsa el botón que muestra ahora la etiqueta “Apagar”, la etiqueta del panel principal volverá a mostrar el texto `inicial` “APAGADO”.

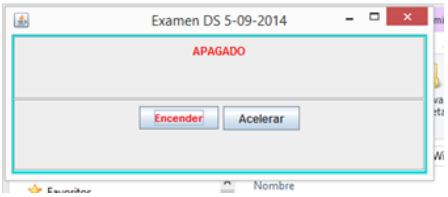

Figura (a)

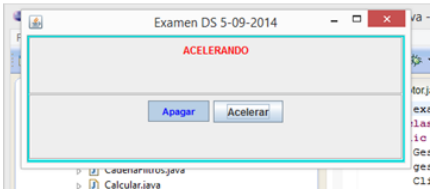

Figura (b)