DESARROLLO DE SOFTWARE (E.INGENIERÍA SOFTWARE) - 1617 (COMÚN)

Información sobre la práctica 1

Práctica 1- Programación con 4 patrones en Java.

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 implementarán una *clases factoría* 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 dos clases factoría: `FactoriaCarreraMontaña` y `FactoriaCarreraCarretera`, que implementarán el método de fabricación anteriormente aludido que, a su vez, creará uno de los objetos: `ArrayList<Bicicleta>[N]` llenándolo de bicicletas de la clase compatible con la modalidad de carrera que nos interese.

La clase *Bicicleta* se debería definir como una clase abstracta y en su constructor aparecería un parámetro 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.

Por supuesto, tendremos que añadir métodos de fabricación adicionales para que las bicicletas sean unos objetos compuestos de *manillar*, 2 *ruedas y marco*. Una carrera se compondrá de **N** bicicletas. Por consiguiente, se han de programar métodos de fabricación específicos para dichos objetos "básicos"*: cuadroCarretera(), manillarCarretera(), ruedasCarretera(), cuadroMontaña(), manillarMontaña(), ruedasMontaña().* `CuadroMontaña`, `ManillarMontaña`, `RuedaMontaña` son clases específicas que "versionarán" las clases genéricas: `Cuadro`, `Manillar`, `Ruedas`.

2) Programar, utilizando el patrón de diseño más adecuado, la simulación de un programa simple de monitorización de datos meteorológicos. El programa `Simulador` se encarga 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 programa `Simulador`, ya que dicho rango de temperaturas dependerá de la época del año y de la región. El programa `Simulador` definirá un método ejecutable (el método público `run()`) si se decide su implementación como una hebra, tal como la siguiente:

```
Random r= new Random(t2);

int temperatura;

while (true){

  temperatura= r.next(Integer);

  try {sleep(60)}

  catch(java.lang.InterruptedException e){

    e.printStackTrace();

  }

  observablePantalla.notificarObservador();

}
```

Se supondrá que un programa de usuario crearía una única instancia de la clase `Pantalla`, que a su vez implementa la interfaz `Observador`. Esta clase define un método público y estático: `refrescarPantalla()`, que muestra el valor actual de la temperatura al usuario en un formato textual.

La clase `Pantalla` tiene, por tanto, un carácter de observador y ha de implementar la interfaz correspondiente.

La implementación de la simulación se ha de realizar utilizando creación de hebras de Java. Además, se deben añadir observadores adicionales, tales como: `botonCambio`, `graficaTemperatura`, `tiempoSatelital`.

`botonCambio` sería una entidad observadora a la que notificaría el programa `Simulacion`, pero también podría cambiar el estado de la simulación asignando directamente el valor de la temperatura actual.

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 .

El método `manejarEvento()` ha de contener la lógica necesaria para actualizar la presentación de la información al programa de usuario en grados Celsius o Fahrenheit y también ha de encargarse de "repintar" la pantalla.

Por 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. Ambas clases implementarán los métodos públicos: `incluirObservador(Observador o), notificarObservador()`.

A instancia del objeto Simulador se crea el `observablePantalla,` asignándole memoria. A su vez, el método `incluirObservador()` asignará memoria a un objeto "o" (puede ser una lista de objetos) que implementa la interfaz `Observador`.

El método `notificarObservador()` se puede implementar haciendo una llamada al método de actualización `manejarEvento()` de los observadores correspondientes.

3) Utilizando el patrón que se considere más adecuado para recorrer una estructura de objetos, desarrollar un programa para generar presupuestos de configuración de un computador simple, formado por los siguientes elementos: `Disco`, `Tarjeta`, `Bus`. El programa mostrará el precio de cada posible configuración de *`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 encarga 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 sólo un objeto `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` mostrará, además del coste total de un equipo, las marcas de sus componentes.

Utilizando creación de hebras en Java, lanzar 100 consultas de precios concurrentes para diferentes tipos de clientes de tal manera que si producen más de 25 peticiones por parte de un mismo tipo de clientes se les haría un 5% de descuento adicional en el precio final de los equipos.

Presentar los resultados del programa en forma de tabla, en filas tipo de cliente, indicar: "regular", "VIP", "mayorista"; en las columnas: el número de peticiones, precio unitario y descuento aplicado para cada encargo.

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 calcular la velocidad inicial del vehículo a partir de un dato de entrada, p.e.: revoluciones del eje, instalando posteriormente un manejador de eventos que "reaccione" 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".

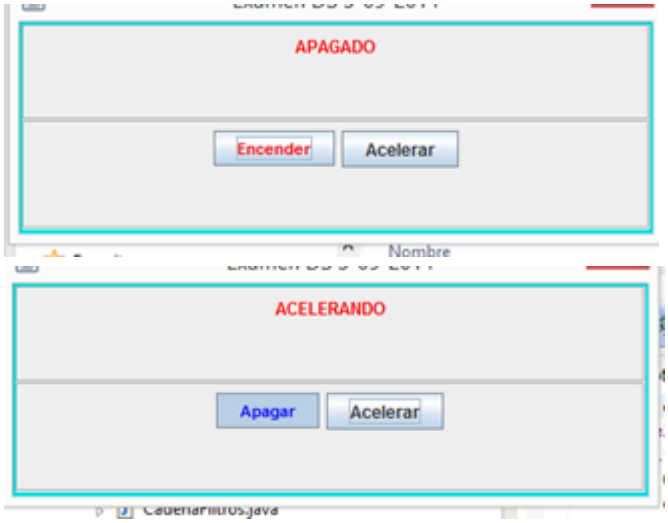

Figura (a) Figura (b)

Última modificación: jueves, 18 de febrero de 2016, 13:09