Práctica 1

Uso de 4 patrones de diseño en orientación a objetos

1.1. Programación y objetivos

Esta práctica constará de 4 partes, una por sesión. En cada sesión habrá al menos un ejercicio de aplicación de un patrón de diseño. Tendrá una puntuación en la nota final de prácticas de 3 puntos sobre 10. Se deben realizar diagramas de clase para cada ejercicio, que adapten el patrón a cada problema concreto.

1.1.1. Objetivos generales de la práctica 1

1. Familiarizarse con el uso de herramientas que integren las fases de diseño e implementación de código en un marco de Orientación a Objetos (OO)
2. Aprender a aplicar patrones de diseño de distintos tipos
3. Adquirir destreza en la práctica de diseño OO
4. Aprender a aplicar patrones de diseño en distintos lenguajes OO

1.1.2. Planificación y competencias específicas

Como en el resto de las prácticas, se espera del estudiante que en cada sesión de prácticas:

1. Entienda bien lo que se pide.
2. Realice el diseño que utilizará (diagrama de clases) junto con su compañero de prácticas y preguntando al profesor y a otros compañeros sobre posibles decisiones a tomar.

3. Empiece a implementar. La implementación deberá ser terminada en tiempo de trabajo fuera de la sesión y antes de la siguiente sesión de prácticas.

NOTA: En la primera sesión también se recomienda que instale las herramientas necesarias para la realización completa de la práctica.

Sesión/Parte	**Semana**	**Competencias**
S1	17-20 febrero	Aplicar tres patrones creacionales entendiendo las relaciones entre los mismos
S2	24-27 febrero	Aplicar un patrón conductual
S3	2-5 marzo	Usar una librería OO para GUI[1] con otro patrón conductual
S4	9-12 marzo	Usar patrones combinados de diseño

1.2. Criterios de evaluación

Para superar cada parte será necesario cumplir con todos y cada uno de los siguientes criterios:

- Capacidad demostrada de trabajo en equipo (reparto equitativo de tareas)
- Implementación completa y verificabilidad (sin errores de ejecución)
- Fidelidad de la implementación al patrón de diseño
- Reutilización de métodos (ausencia de código redundante)
- Validez (se cumplen los requisitos funcionales)

Se valorarán además otros criterios de evaluación:

- Capacidad de explicar a otro equipo una solución de diseño
- Realización de las partes opcionales de la práctica

1.3. Plazos de entrega y presentación de la práctica

Cada ejercicio será subido a PRADO en una tarea que terminará justo antes del inicio de la sesión siguiente de prácticas (a las 15:30 horas del día de la sesión siguiente). La práctica completa será presentada una vez realizadas todas las entregas, mediante una entrevista con el profesor de prácticas.

1.4. SESIÓN 1ª: Patrones "Factoría Abstracta", "Método Factoría" y "Prototipo" (1 punto)

Esta práctica se desarrollará en Java y en Ruby.

1.4.1. Ejercicio S1E1: Factoría Abstracta y Método Factoría en Java (0.6 puntos)

Programa 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 *FactoriaCarreraYBicicleta*, una interfaz de Java, declara los métodos de fabricación públicos:

- *crearCarrera* que devuelve un objeto de alguna subclase de la clase abstracta *Carrera* y
- *crearBicicleta* que devuelve un objeto de alguna subclase de la clase abstracta *Bicicleta*.

La clase *Carrera* tiene al menos un atributo $ArrayList < Bicicleta >$, con las bicicletas que participan en la carrera. La clase *Bicicleta* tiene al menos un identificador único de la bicicleta en una carrera. Las clases factoría específicas heredan de *FactoriaCarreraYBicicleta* y cada una de ellas se especializa en un tipo de carreras y bicicletas: las carreras y bicicletas de montaña y las carreras y bicicletas de carretera. Por consiguiente, asumimos que tenemos dos clases factoría específicas: *FactoriaMontana* y *FactoriaCarretera*, que implementarán cada una de ellas los métodos de fabricación *crearCarrera* y *crearBicicleta*.

Por otra parte, las clases *Bicicleta* y *Carrera* se debería definir como clases abstractas y especializarse en clases concretas para que la factoría de montaña pueda crear productos *BicicletaMontana* y *CarreraMontana* y la factoría de carretera pueda crear productos *BicicletaCarretera* y *CarreraCarretera*.

Para programar la simulación de cada bicicleta en cada carrera crearemos hebras en Java.

1.4.2. Ejercicio S1E2: Método Factoría en Ruby (opcional) (0.4 puntos)

Diseña e implementa una aplicación inspirada en el ejercicio anterior que cumpla los siguientes requisitos:

- Aplique el patrón "Prototipo" en vez del patrón "Método Factoría" con el patrón "Factoría Abstracta"
- No requiera programación concurrente
- Se implemente en Ruby

Ten en cuenta que en Ruby no puede declararse una clase como abstracta. La forma de impedir instanciar una clase es declarando privado el constructor o utilizando algún mecanismo para que se lance una excepción si el cliente de la clase intenta instanciarla.

1.4.3. Algunas cuestiones para esta sesión

¿Cómo crear un hilo en Java?

Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando el interfaz Runnable, la otra es extender la clase Thread. Si optamos por crear hilos mediante la creación de clases que hereden de Thread, la clase hija debe sobreescribir el método run():

```
class MiThread extends Thread {
 @Override
 public void run() {
        . . .
        }
```

Los métodos java más importantes sobre un hilo son:

- *start*(): arranque explícito de un hilo, que llamará al método *run*()
- *sleep*(*retardo*): espera a ejecutar el hilo retardo milisegundos
- *isAlive*(): devuelve si el hilo está aun vivo, es decir, *true* si no se ha parado con *stop*() ni ha terminado el método *run*()

Patrones "Factoría Abstracta", "Método Factoría" y "Prototipo"

Debe tenerse en cuenta que los patrones "Método Factoría" y "Prototipo" son dos formas concretas de crear objetos por las clases factorías en el patrón "Factoría Abstracta".

1.5. SESIÓN 2ª: Patrón Visitante (0,5 puntos)

Los ejercicios de esta sesión se desarrollarán en C++.

1.5.1. Ejercicio S2E1: Visitante básico (0.4 puntos)

Utilizando este patrón (ver Figura 1.1) se pretende recorrer una estructura de componentes que forman un equipo de cómputo (clase *Equipo*), 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. Cada componente es una subclase de *ComponenteEquipo*. Las clases *Disco*, *Tarjeta*, *Bus* extienden a la clase abstracta *ComponenteEquipo* e implementan todos sus métodos abstractos. La programación del método *aceptar*(*VisitanteEquipove*) en cada una de las clases anteriores consistirá en una llamada al método correspondiente de la clase abstracta *VisitanteEquipo*. Si la implementación fuera en Java, el código siguiente sería correcto:

```
public abstract class VisitanteEquipo{

  public abstract void visitarDisco(Disco d);

  public abstract void visitarTarjeta(Tarjeta t);

  public abstract void visitarBus(Bus b);

}
```

Debe adaptarse este código a C++, teniendo en cuenta que en C++ no existe una palabra reservada para declarar una clase como abstracta. En C++ una clase es abstracta cuando tiene declarado un método de ligadura dinámica (virtual) y no se implementa o hereda un método de este tipo y no lo implementa.

Las subclases de *VisitanteEquipo* definirán algoritmos concretos que se aplican sobre la estructura de objetos que se obtiene de instanciar las subclases de *Equipo*. Así, se definirán las siguientes subclases de *VisitanteEquipo*:

- *VisitantePrecio*: calcula 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.

- *VisitantePrecioDetalle*: mostrará los nombres de las partes que componen un equipo y sus precios.

El programa principal (*main*) se encargará de crear varios equipos y calcular el precio total de cada uno y los precios detallados y nombres de cada componente usando los visitantes adecuados.

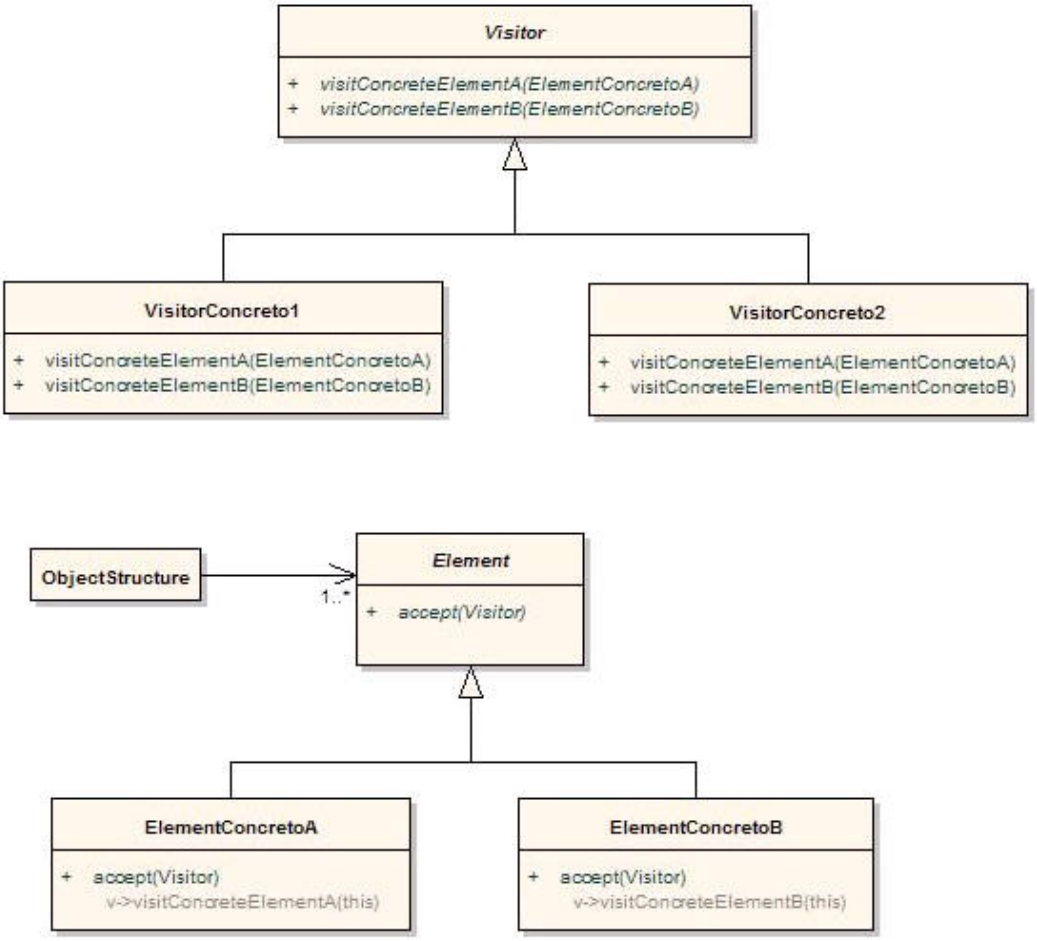

Figura 1.1: Diagrama de clases del patrón Visitante. Este patrón busca separar un algoritmo de la estructura de un objeto. La operación se implementa de forma que no se modifique el código de las clases que forman la estructura del objeto. Este patrón debe utilizarse cuando se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases. Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general. Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere "contaminar" a dichas clases.

1.5.2. Ejercicio S2E2: Categorías de visitantes (opcional) (0.1 puntos)

Se desea tener en cuenta a la hora de calcular los precios que hay tres tipos distintos de personas: cliente sin-descuento, estudiante (10 % descuento) y cliente mayorista (15 % descuento). El programa principal deberá ahora calcular los precios de cada equipo y sus partes teniendo en cuenta de qué tipo de cliente se trata. Se deben realizar los cambios necesarios en el diseño para incluir este nuevo requisito. Señala además qué cambios en el patrón has tenido que realizar para añadir este nuevo requisito.

1.6. SESIÓN 3ª: Patrón Observador (0,75 puntos)

Los ejercicios de esta sesión se desarrollarán en Java. Se puede utilizar un asistente para desarrollar la GUI, como el que incluye el IDE NetBeans.

1.6.1. Ejercicio S3E1: Monitorización de datos meteorológicos (GUI y patrón observador) (0,6 puntos)

Programa, utilizando el patrón de diseño Observador (ver Figura 1.2), un programa *Simulador* simple de simulación de la monitorización de datos meteorológicos. El programa debe crear un sujeto-observable con una *temperatura* y tres observadores: *pantallaTemperatura*, *botonCambio* y *graficaTemperatura*. Cada vez que el sujeto actualiza su *temperatura*, lo que hace de forma regular (mediante una hebra), deberá notificar el cambio a los observadores que tenga suscritos (*graficaTemperatura*, *botonCambio*) (comunicación push). El observador *botonCambio* no es un mero observador, sino que también podrá cambiar la temperatura del sujeto a petición del usuario. El observador *pantallaTemperatura* no está suscrito, sino que se comunicará con el sujeto observable de forma asíncrona (comunicación pull).

1.6.2. Ejercicio S3E2: Ampliación con información geográfica (opcional) (0,15 puntos)

De forma opcional se puede añadir un cuarto observador: *TiempoSatelital*, que muestre la temperatura sobre una posición de un mapa y acceda por suscripción a otros sujetos observables para mostrar las temperaturas en otras posiciones del mapa.

1.7. SESIÓN 4ª: Patrón Filtros de intercepción (0,75 puntos)

Esta sesión consta de un sólo ejercicio, que se desarrollará en Java. Se puede utilizar un asistente para desarrollar la GUI, como el que incluye el IDE NetBeans.

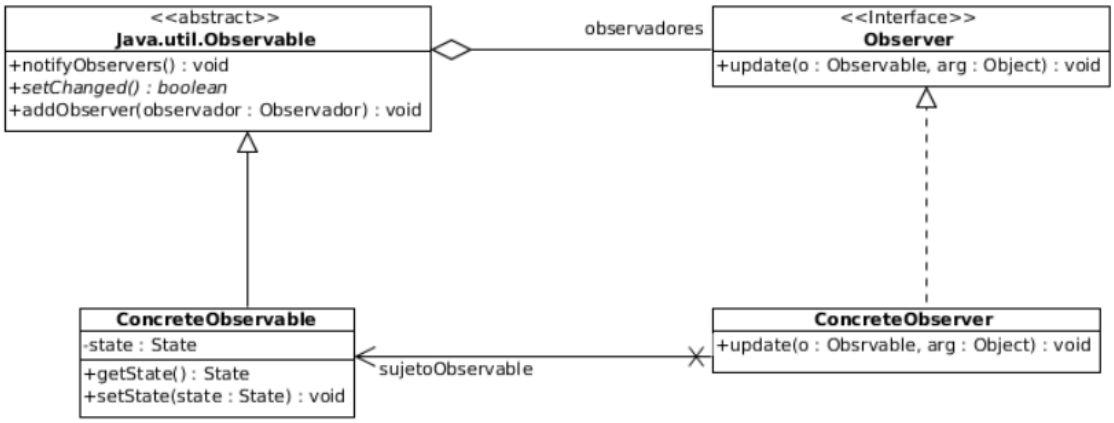

Figura 1.2: Diagrama de clases del patrón Java Observable-Observer.

1.7.1. Ejercicio S4E1: Simulador del movimiento de un vehículo con cambio automático

Queremos representar en el salpicadero de un vehículo (una hebra) los parámetros del movimiento del mismo (velocidad lineal en km/h, distancia recorrida en km y velocidad angular -"revoluciones"- en RPM), calculados a partir de las revoluciones del motor. Queremos además que estas revoluciones sean primero modificadas (filtradas) mediante software independiente a nuestro sistema, capaz de calcular el cambio en las revoluciones como consecuencia (1) del estado del motor (acelerando, frenando, apagando el motor . . .) y (2) del rozamiento.

Por tratarse de una mera simulación, no vamos a programar los servicios de filtrado como verdaderos componentes independientes, sino como objetos de clases no emparentadas que correrán todos bajo una única aplicación java (proyecto en el IDE).

Usaremos para este ejercicio el patrón arquitectónico "Filtros de intercepción" (ver Figura 1.3).

En nuestro ejercicio, se crearán dos filtros (clases *CalcularVelocidad* y *RepercutirRozamiento*, que implementan la interfaz *Filtro*) para calcular la velocidad angular ("revoluciones") y modificar la misma en base al rozamiento.

A continuación se explican las entidades de modelado necesarias para programar el patrón "Filtros de intercepción" para este ejercicio.

Objetivo (target): Representa el salpicadero o dispositivo de monitorización del movimiento de un coche.

Filtro: Esta interfaz implementada por las clases *RepercutirRozamiento* y *CalcularVelocidad* arriba comentadas. Estos filtros se aplicarán antes de que el *salpicadero* (objeto de la clase *Objetivo*) ejecute sus tareas propias (método *ejecutar*) de mostrar las revoluciones, velocidad lineal a las que se mueve y la distancia recorrida.

Cliente: Es el objeto que envía la petición a la instancia de *Objetivo*. Como estamos

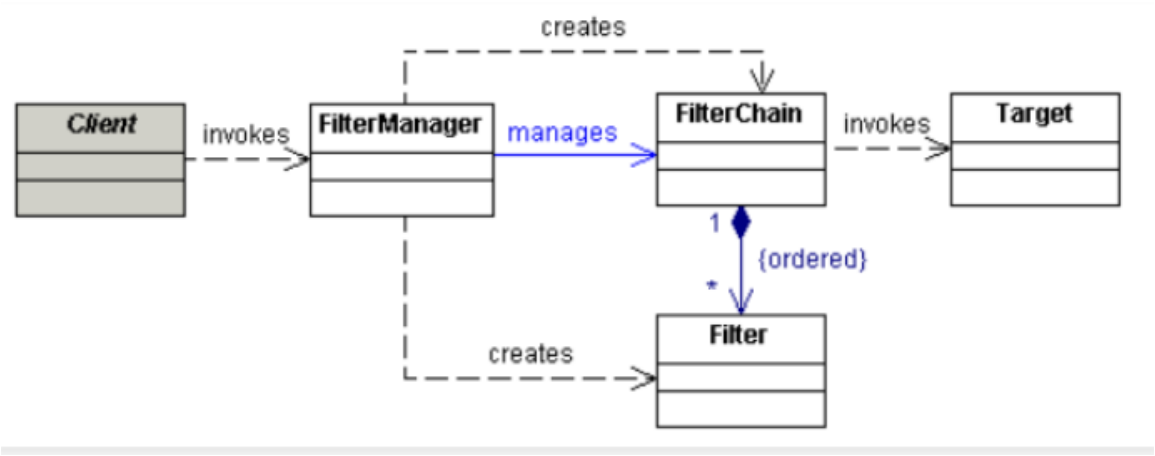

Figura 1.3: Diagrama de clases correspondiente al patrón interceptor de filtros. [Fuente: Patrón Filtros de intercepción.]

usando el patrón Filtros de intercepción, la petición no se hace directamente, sino a través de un gestor de filtros (*GestorFiltros*) que enviará a su vez la petición a un objeto de la clase *CadenaFiltros*.

CadenaFiltros:Tendrá una lista con los filtros que se aplicarán y los ejecutará en el orden en que fueron introducidos en la aplicación. Tras ejecutar esos filtros, se ejecutará la tarea propia del motor del coche (método *ejecutar* de la clase *Objetivo*), todo dentro del método *ejecutar* de *CadenaFiltros*.

GestorFiltros: Se encarga de gestionar los filtros: crea la cadena de filtros y tiene métodos para añadir filtros concretos y que se ejecute la petición por los filtros y el "objetivo" (método *peticionFiltros*).

En nuestro caso, el filtro que calcula el rozamiento, considera una disminución constante del mismo, p.e. -1. Así, la cadena de filtros deberá enviar al objetivo (el *salpicadero*) el mensaje ejecutar para calcular las velocidades y la distancia recorrida a partir del cálculo actualizado de las revoluciones del eje, hecho por los filtros. Para convertir las revoluciones por minuto (RPM) en velocidad lineal (v, en km/h) podemos aplicar la siguiente fórmula:

$$v = 2\pi r \times RPM \times (60/1000) km/h,$$

siendo r el radio del eje en metros, que puede considerarse igual a 0,15.
Los métodos y secuencia de ejecución que representan dichos servicios serán:

1. `double ejecutar(double revoluciones, EstadoMotor estadoMotor)`, en la subclase de *Filtro FiltroCalcularVelocidad*.- Actualiza y devuelve las revoluciones añadiendo la cantidad *incrementoVelocidad* (un atributo del *filtro*, puede ser negativa o 0), que debe previamente haberse asignado teniendo en cuenta el estado del mo-

tor (*acelerando*, *frenando*, *apagado*, *encendido*). Debe tenerse en cuenta un máximo en la velocidad, por encima del cual nunca se puede pasar, por ejemplo 5000 RPM.

2. `double ejecutar(double revoluciones, EstadoMotor estadoMotor)`, en la subclase *Filtro FiltroRepercutirRozamiento*.- Actualiza y devuelve las revoluciones quitando una cantidad fija considerada como la disminución de revoluciones debida al rozamiento.

3. `double ejecutar(double revoluciones, EstadoMotor estadoMotor)`, en la clase *Objetivo*.- Modifica los parámetros de movimiento del vehículo en el *salpicadero* (velocidad angular, lineal y distancia recorrida). Gracias a los filtros, utiliza como argumento las revoluciones recalculadas teniendo en cuenta el estado del vehículo y el rozamiento.

   - Por simplificar la implementación, se han puesto dos argumentos en el método *ejecutar*, aunque el segundo sólo sea requerido para el primero de los filtros. El patrón, de forma genérica define un sólo argumento tipo *Object* que puede corresponder a otra clase con toda la información que necesitemos.
   - *EstadoMotor* puede implementarse como enumerado o bien considerarse como entero.

Se puede considerar *incrementoVelocidad* como 0 cuando el vehículo está en estado apagado o encendido. Si está en estado frenando, *incrementoVelocidad* será -100 RPM y cuando está en estado acelerando, *incrementoVelocidad* será $+100$ RPM.

Como ejemplo de programación de los dispositivos del control del vehículo, se pueden usar los botones *Encender*, *Acelerar*, *Frenar* y las etiquetas "APAGADO" / "ACELERANDO" / "FRENANDO" 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, BotonFrenar;
  private javax.swing.JLabel EtiqMostrarEstado;
  public PanelBotones(){ ...}; //constructor
  synchronized private void
      BotonAcelerarActionPerformed(java.awt.event.ActionEvent evt){...};
synchronized private void
      BotonFrenarActionPerformed(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 1.4 (a))) y las etiquetas de los botones, "Encender", "Acelerar", "Frenar"

(a) (b)

Figura 1.4: Ejemplo de la ventana correspondiente al dispositivo de control del movimiento del vehículo (a) Estado "apagado"; (b) Estado "acelarando".

- 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 *Acelerar'* cambia el texto de la etiqueta del panel principal a "ACELERANDO" (ver Figura 1.4 (b)), pero sólo si el motor está encendido; si no, no hace caso a la pulsación del usuario.
- La pulsación del botón *Frenar* cambia el texto de la etiqueta del panel principal a "FRENANDO", pero sólo si el motor está encendido; si no, no hace caso a la pulsación del usuario.
- Los botones *Acelerar* y *Frenar* serán de selección de tipo conmutador *JToggleButton*, cambiando de color y de texto ("Acelerar" / "Frenar") cuando se pulsan.
- Los botones *Acelerar* y *Frenar* no pueden estar presionados simultáneamente (el conductor no puede pisar ambos pedales a la vez).
- Si ahora se pulsa el botón que muestra ahora la etiqueta "Apagar", la etiqueta del panel principal volverá a mostrar el texto inicial "APAGADO".

En cuanto al *salpicadero*, puede programarse como un *JPanel* que contiene un velocímetro, un cuentarrevoluciones y un cuentakilómetros (que a su vez también pueden definirse como clases que hereden de *JPanel*), tal y como aparece en el código siguiente:

```
public class Salpicadero extends JPanel {
Velocimetro velocimetro=new Velocimetro();
CuentaKilometros cuentaKilometros=new CuentaKilometros();
CuentaRevoluciones cuentaRevoluciones=new CuentaRevoluciones();
...
}
```

La Figura 1.5 muestra un ejemplo de salpicadero sencillo.

Para que el ejercicio sea considerado correcto hay que añadirle el siguiente requerimiento:

Incluir una clase anónima (*WindowAdapter*) con Swing (Java) para terminar bien la ejecución de la clase *Objetivo* correspondiente:

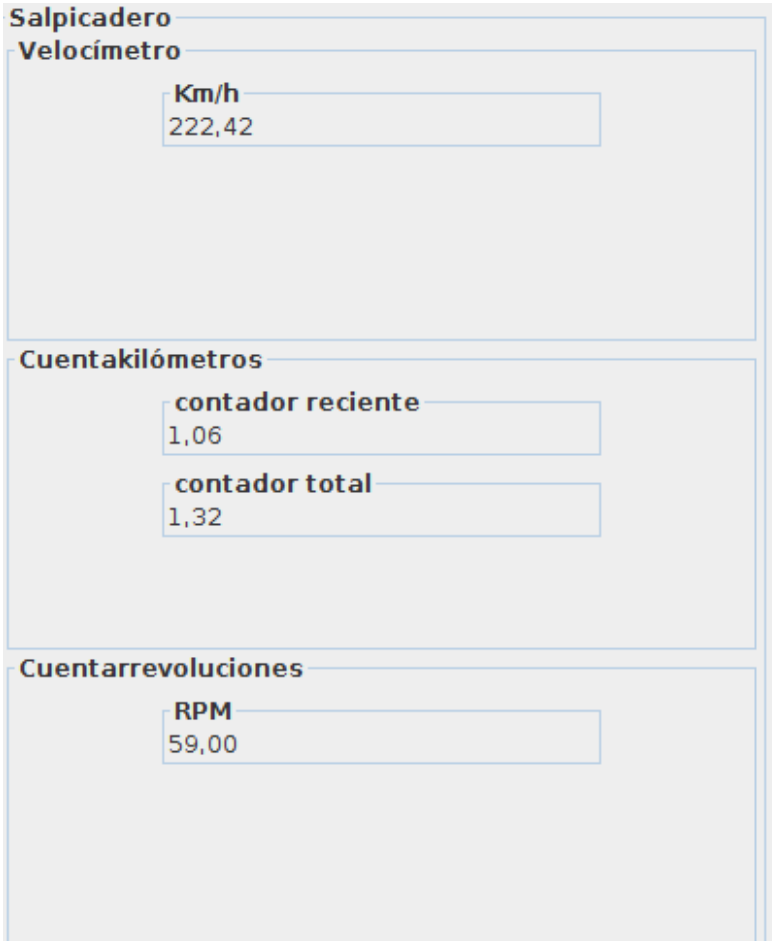

Figura 1.5: Ejemplo de la ventana correspondiente al salpicadero del vehículo

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