Práctica 1

Uso de patrones de diseño en orientación a objetos

Fecha última actualización: 19 de marzo de 2021.

Para esta práctica se tomará como punto de partida el caso de estudio propuesto por el grupo pequeño de teoría al que pertenece el equipo de prácticas.

1.1. Programación y objetivos

Esta práctica constará de 3 partes, una por sesión, más una sesión final para terminar todos los ejercicios o hacer los de ampliación. 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 y estilos arquitectónicos de distintos tipos
3. Adquirir destreza en la práctica de diseño OO
4. Aprender a aplicar patrones de diseño y estilos arquitectónicos 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 a los otros compañeros del grupo pequeño de teoría y al profesor 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. La URL para descarga de VisualParadigm y obtención de licencia UGR es: https://ap.visual-paradigm.com/universidad-granada.

Sesión	**Semana**	**Competencias**
S1	5-10 marzo	Aplicar los patrones *observador* y *compuesto* en una aplicación Java con una GUI[1].
S2	12-17 marzo	Aplicar tres patrones creacionales en sendas aplicaciones con hebras en Java y Ruby entendiendo las relaciones entre los mismos.
S3	19-24 marzo	Aplicar el patrón *visitante* en c++ junto con otro patrón a elegir.
S4	26 marzo-7 abril	Sesión de finalización.

1.2. Criterios de evaluación

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

- Calidad (se cumplen los requisitos no funcionales)
- 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)

1.3. Plazos de entrega y presentación de la práctica

La práctica completa será subida a PRADO en una tarea que terminará justo antes del inicio de la primera sesión de la práctica 2 (a las 15:30 horas del día de la primera sesión). Será presentada posteriormente mediante una entrevista con el profesor de prácticas.

1.4. Formato de entrega

Se deben exportar los proyectos generados en cada ejercicio y archivarlos todos juntos, poniendo también en el archivo (válido cualquier formato) la documentación extra que se pida, como por ejemplo los diagramas de clase, en formato pdf. Es preferible un único pdf con toda la documentación de todos los ejercicios de la práctica.

1.5. SESIÓN 1ª: Patrón Observador (1 punto)

1.5.1. Requisitos obligatorios (0,6 puntos)

- Se usará Java como lenguaje de programación. Debe utilizarse la clase abstracta Observable de Java.util y la interfaz Observer de Java (ver Figura 1.1). Debe tenerse en cuenta que el método *notifyObservers* tiene como argumento el *Object* con la información a publicar (no aparece en la Figura 1.1) y que antes de llamar a ese método hay que invocar al método *setChanged* para dejar constancia de que se ha producido un cambio (si no se hace, el método *notifyObservers* no hará nada).
- Se definirán dos observadores, de la siguiente forma:
  1. Observador suscrito convencional (comunicación *push*) mediante método *notifyObservers* y
  2. Observador no suscrito (comunicación *pull*).

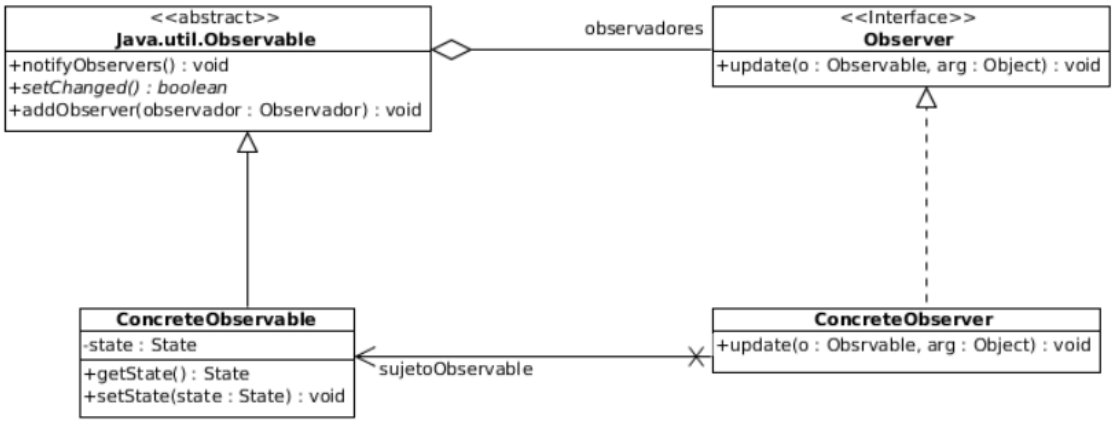

Figura 1.1: Diagrama de clases del patrón Java Observable-Observer.

- Se creará una GUI[2].
- Para programar la simulación de la actualización de datos por parte del modelo y la petición de datos por parte del observador no suscrito, crearemos hebras en Java.

1.5.2. Requisitos opcionales (0,4 puntos)

- Se romperá parte de la filosofía de este patrón de diseño mediante la inclusión de un tercer observador suscrito que pueda producir cambios en el modelo observado.
- Se añadirá un cuarto observador que haga uso de algún recurso GUI más sofisticado (mapas, librerías gráficas externas, etc.).
- Se creará un observador compuesto haciendo uso del patrón de diseño *compuesto*.

Ejemplo (parte obligatoria): Monitorización de datos meteorológicos

Programa en Java, utilizando el patrón de diseño Observador (ver Figura 1.1), un programa con una GUI que simule la monitorización de datos meteorológicos. El programa

[2]Se puede utilizar un asistente para desarrollar la GUI, como el que incluye el IDE NetBeans.

debe crear un sujeto-observable con una *temperatura* y dos observadores: *pantallaTemperatura* 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 (solamente *graficaTemperatura*) (comunicación push). El observador *pantallaTemperatura* no está suscrito, sino que se comunicará con el sujeto observable de forma asíncrona (comunicación pull). El observador *pantallaTemperatura* debe mostrar la temperatura tanto en grados centígrados como en Fahrenheit. El observador *graficaTemperatura* debe mostrar una gráfica con las últimas 7 temperaturas recibidas (una por semana), solo en grados Celsius.

Ejemplo (parte optativa): Ampliación con información geográfica

Se añadirán dos nuevos observadores suscritos. El primero, *botonCambio*, además de mostrar la temperatura en grados Celsius, podrá cambiar la temperatura del sujeto observable a petición del usuario. El segundo, *tiempoSatelital*, mostrará la temperatura sobre una posición de un mapa y accederá por suscripción a otros sujetos observables para mostrar las temperaturas en otras posiciones del mapa. Por último, se creará otro observador suscrito como combinación de *graficaTemperatura* y *botonCambio* utilizando el patrón *compuesto* (composite).

¿Cómo crear un hilo en Java?

Para esta sesión y la siguiente, será necesario crear hebras. 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*()

1.6. SESIÓN 2ª: Patrones *Factoría Abstracta, Método Factoría* y *Prototipo* (1 punto)

Esta sesión constará de dos ejercicios.

1.6.1. Ejercicio S2E1 (obligatorio - 0,6 puntos)

- Se implementará el patrón de diseño *Factoría Abstracta* junto con el patrón de diseño *Método Factoría*.
- Se usará Java como lenguaje de programación.
- Se utilizarán hebras.
- Se usará un interfaz de usuario de texto.
- Se implementarán solo dos familias/estilos de productos aunque en el futuro se podrán añadir otras [3].

1.6.2. Ejercicio S2E2 (opcional - 0,4 puntos)

- Se implementará una solución al mismo problema considerado en el ejercicio anterior, pero haciendo los siguientes cambios:
  - Debe usarse Ruby como lenguaje de programación.
  - Debe usarse el patrón de diseño *Factoría Abstracta* junto con el patrón de diseño *Prototipo*, en vez de hacerlo junto con el patrón de diseño *Método Factoría*, como en el ejercicio anterior.

Ejemplo ejercicio S2E1 (parte obligatoria): Patrón *Factoría Abstracta* y patrón *Método Factoría*

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

[3] Considérese que el concepto de producto en patrones de diseño, tiene una definición muy abierta, para que pueda ajustarse a cada problema concreto, pero que en todo caso los productos deberán implementarse como objetos que pueden ser muy distintos entre sí, es decir, sin relación de herencia entre ellos.

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 deberían 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.

Ejemplo ejercicio S2E2 (parte optativa): Patrón *Factoría Abstracta* y patrón *Prototipo* en Ruby

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

- que aplique el patrón *Prototipo* en vez del patrón *Método Factoría*, junto con el patrón *Factoría Abstracta*,
- que no requiera programación concurrente, y
- que se implemente en Ruby[4].

[4]Debe tenerse en cuenta que en Ruby no puede declararse una clase como abstracta. La forma de

1.7. SESIÓN 3ª: Estilos arquitectónicos: el estilo *Filtros de Intercepción*

Esta sesión constará de dos ejercicios.

1.7.1. Ejercicio S3E1 (obligatorio - 0,6 puntos)

- Se implementará el estilo arquitectónico *Filtros de Intercepción* (ver Figuras 1.2 y 1.3).
- Se usará c++ como lenguaje de programación.
- Se usará una interfaz de usuario de texto donde se muestre el estado del *objetivo* para cada invocación del *cliente*.
- Se introducirán al menos dos filtros.

1.7.2. Ejercicio S3E2 (opcional - 0,4 puntos)

Se implementará una solución al mismo problema considerado en el ejercicio anterior, pero haciendo los siguientes cambios:

- Opción 1:
  - Se usará Java como lenguaje de programación y se usará Swing para hacer una GUI.
  - Se añadirá alguna funcionalidad nueva.
- Opción 2: Se usará Dart como lenguaje de programación.

El estilo arquitectónico *Filtros de Intercepción*

Las Figuras 1.2 y 1.3 muestran respectivamente un diagrama de clases y un diagrama de secuencias donde se explica este estilo arquitectónico.

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

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.

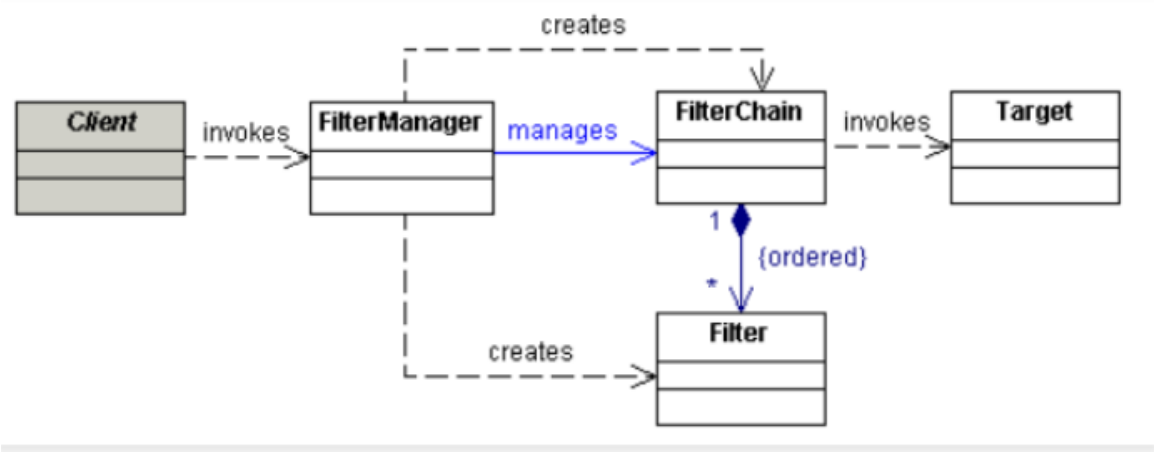

Figura 1.2: Diagrama de clases correspondiente al estilo *Filtros de intercepción*. [Fuente: (Alur et al., 2003).]

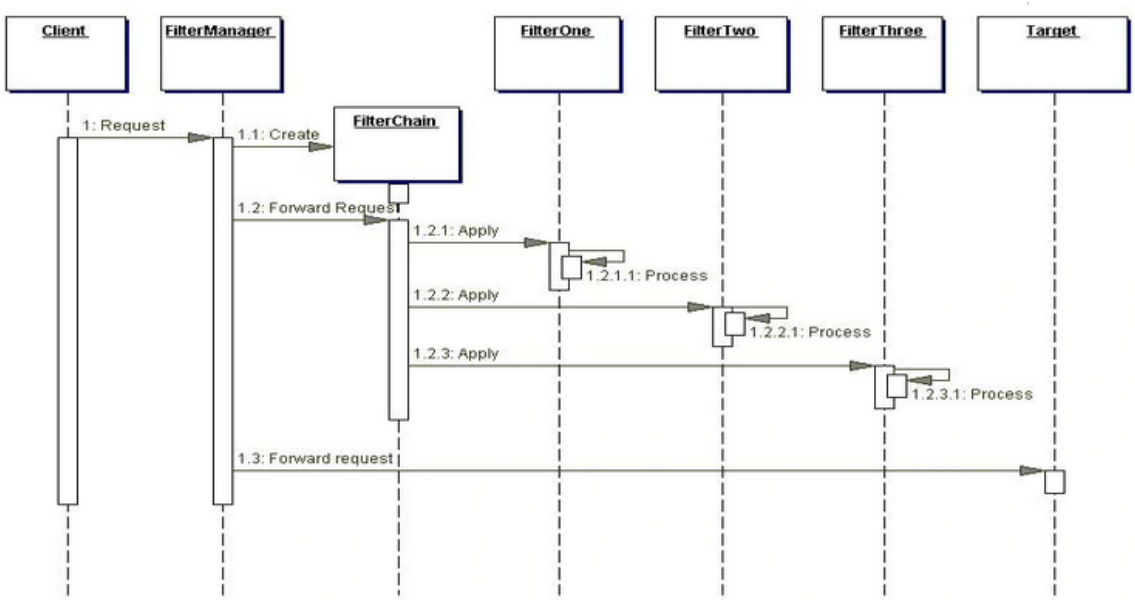

Figura 1.3: Ejemplo de diagrama de secuencias del estilo arquitectónico *Filtros de intercepción*. [Fuente: https://www.oracle.com/technetwork/java/interceptingfilter-142169.html.]

- *Objetivo* (target): Representa el objeto controlado por el cliente una vez pasado por los filtros, que se debe mostrar en la consola de texto.
- *Filtro*: Esta interfaz será implementada por al menos dos subclases. Estos filtros se aplicarán antes de que el *objetivo* ejecute sus tareas propias (método *ejecutar*) para obtener el estado final que se mostrará por consola.
- *Cliente*: Es el objeto que envía la petición a la instancia de *Objetivo*. Como estamos usando el estilo *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 *objetivo* (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*).

Ejemplo ejercicio S3E1 (parte obligatoria): Estilo arquitectónico *Filtros de Intercepción* en c++ con interfaz de usuario de texto

Simulador del movimiento de un vehículo con cambio automático Queremos representar en el salpicadero de un vehículo 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) con filtros capaces de calcular el cambio en las revoluciones como consecuencia (1) de la pendiente (aceleración por gravedad) y (2) del rozamiento.

Los cambios en el estado del motor se producirán de forma aleatoria –siempre de la misma intensidad–: sin cambio, acelerando o frenando. Por ejemplo, se puede considerar que cuando el vehículo está en estado frenando, se produce un cambio constante en las revoluciones ($-100RPM$) y cuando está en estado acelerando, en $+100RPM$.

En nuestro ejercicio, se crearán dos filtros (clases *RepercutirPendiente* y *RepercutirRozamiento*, que implementan la interfaz *Filtro* para modificar la velocidad angular en base a la pendiente y al rozamiento. A continuación se explican las entidades de modelado necesarias para programar el estilo *Filtros de intercepción* para este ejercicio.

- *Objetivo* (target): Representa el salpicadero con la velocidad del coche y distancia recorrida.
- *Filtro*: Esta interfaz es implementada por las clases *RepercutirRozamiento* y *RepercutirPendiente* 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*: Representa las acciones de control del vehículo (acelerar, frenar, neutro) que serán interceptadas por los filtros de la *CadenaFiltros* para añadir el rozamiento y la gravedad (según inclinación), antes de ejecutar la tarea propia de calcular la velocidad final del coche (método *ejecutar* de la clase *Objetivo*).
- *GestorFiltros*: Crea la cadena de filtros y se encarga de que se apliquen (método *peticionFiltros*).

El filtro que calcula el rozamiento, considera una disminución constante del mismo, p.e. -1. El filtro que calcula la aceleración por la inclinación del plano, calcula de forma aleatoria el cambio que se produce, p.e. un valor en el intervalo [-2,2] (los números negativos son cuestas). 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 FiltroRepercutirPendiente*.- Actualiza y devuelve las revoluciones añadiendo la cantidad *incremento* (un atributo del filtro, puede ser negativa). 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 *decremento*, que representa 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 no todos lo necesiten. 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.

Ejemplo ejercicio S3E2 (parte optativa, opción 1): Estilo arquitectónico *Filtros de Intercepción* en Java y GUI con Java Swing

Simulador del movimiento de un vehículo con cambio automático. Ejemplo en Java Swing Al ejercicio anterior se le añadirá el control de la velocidad del vehículo por parte del usuario (acelerar y frenar, además de encender y apagar el vehículo) y se utilizará una hebra que simule todo el proceso de cálculo de la velocidad a partir del estado del motor y de los filtros.

Se diseñará una GUI con salpicadero y mandos básicos del conductor. 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.

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"
- El botón *Encender* puede ser 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.

(a) (b)

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

- 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".

La Figura 1.5 muestra un ejemplo de salpicadero sencillo.

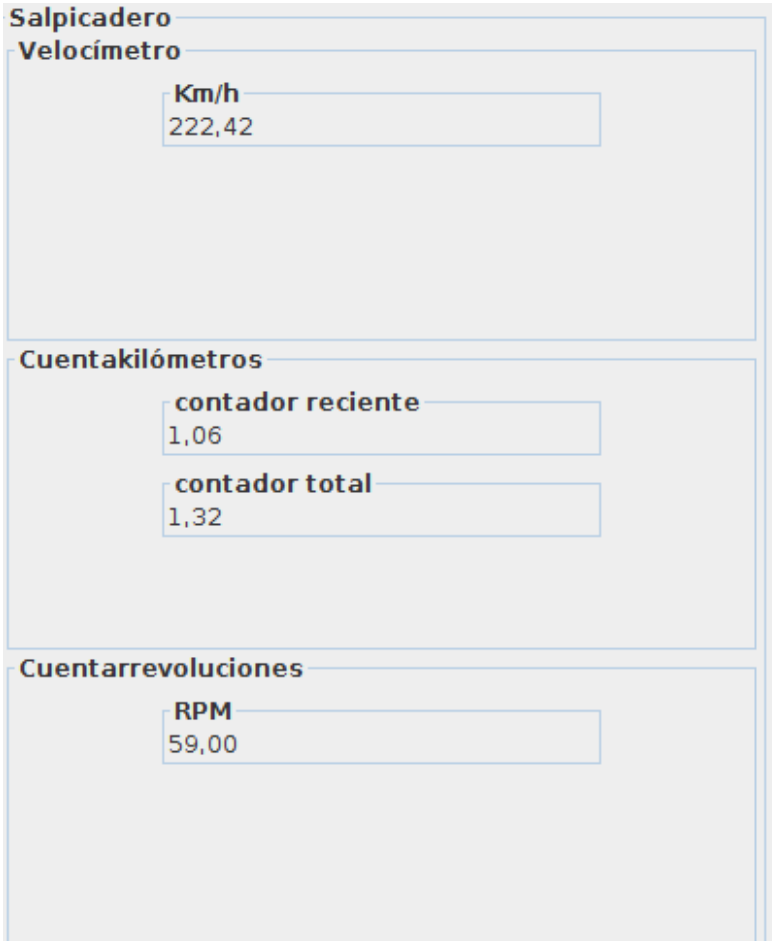

Figura 1.5: Ejemplo de la ventana correspondiente al salpicadero del vehículo

Bibliografía

Deepak Alur, Malks Dan, and Crupi John. *Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition*. Pearson, 2003. URL https://learning.oreilly.com/library/view/core-j2eetm-patterns/0131422464/pr02.html.