Práctica 1

Uso de patrones de diseño en orientación a objetos

Fecha última actualización: 14 de marzo de 2022.

1.1. Programación y objetivos

Esta práctica constará de 3 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 utilizar los diagramas de clase para cada ejercicio que adaptan el patrón a cada problema concreto y que se han realizado en clase de teoría. El trabajo de cada sesión debe ser dividido entre los cuatro miembros del grupo de forma equitativa. Si hay dos módulos en una sesión, dos miembros del grupo harán un módulo y los otros dos, el otro módulo.

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 arquitectónicos de distintos tipos

3. Adquirir destreza en la práctica de diseño OO

4. Aprender a aplicar patrones de diseño y arquitectónicos en distintos lenguajes OO

1.1.2. Planificación y competencias específicas

1. Haya entendido bien lo que se pide y haya hecho con sus compañeros (en clase de teoría) los diagramas que deben implementar.
2. Empiece a implementar la parte que le han asignado sus compañeros de grupo. La implementación de cada ejercicio deberá ser terminada y reunificada en un único proyecto de grupo en tiempo de trabajo fuera de la sesión y antes de la siguiente sesión de prácticas.
3. Para cada sesión, el grupo deberá generar diagramas de clase de cada ejercicio a partir de la implementación definitiva (puede hacerse de forma automática con Visual Paradigm (VP).

NOTA: En la primera sesión también se recomienda que instale las herramientas necesarias para la realización completa de la práctica, si no se ha hecho en clase de teoría. La URL para descarga de VP y obtención de licencia UGR es: https://ap.visual-paradigm.com/universidad-granada.

Sesión	**Días**	**Competencias**
S1	3-7 marzo	Aplicar patrones de diseño en Java con una GUI[1]
S2	9-14 marzo	Aplicar patrones de diseño con distintos lenguajes OO
S3	16-21 marzo	Ampliar funcionalidad y mejorar la GUI a partir de código obsoleto (*legacy code*), aplicando patrones arquitectónicos e incorporando alguna librería gráfica alternativa

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)

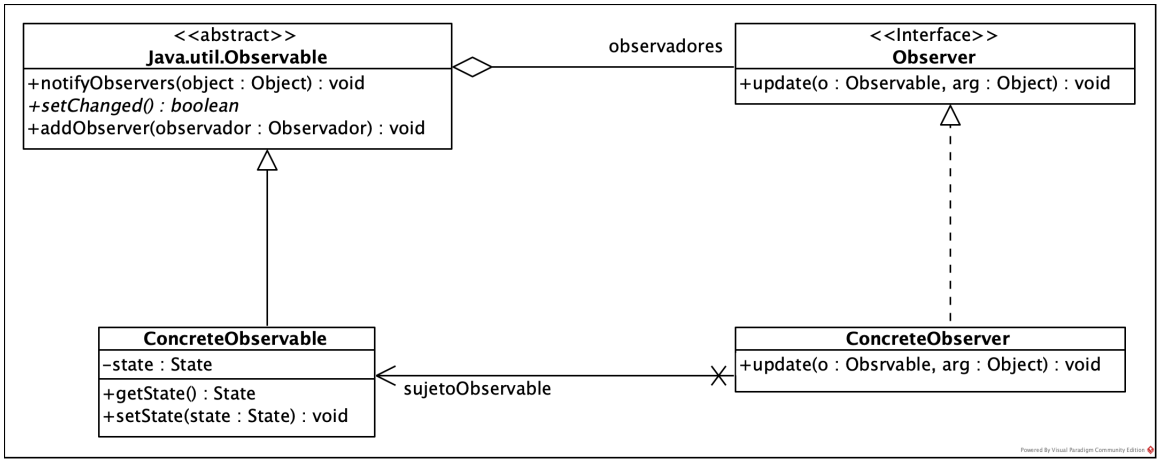

Figura 1.1: Diagrama de clases del patrón Java Observable-Observer.

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 Módulo I: Patrón *observador* para la monitorización de datos meteorológicos (1 punto)

Descripción

Programa en Java, utilizando el patrón de diseño *observador* (ver Figura 1.1), una aplicación con una GUI que simule la monitorización de datos meteorológicos. El programa debe crear un sujeto-observable con una *temperatura* y tres observadores. 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 (comunicación push).

Deberán cumplirse las siguientes especificaciones:

- Se usará Java como lenguaje de programación, utilizándose la clase abstracta Observable de Java.util y la interfaz Observer de Java (ver Figura 1.1). Debe tenerse en cuenta que antes de llamar al método *notifyObservers* 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 creará una GUI donde cada observador aparezca en una ventana distinta [2].

- Se definirán tres observadores, de la siguiente forma:

  1. Observador *graficaTemperatura*, debe mostrar una gráfica con las últimas 7 temperaturas recibidas (una por semana), en grados Celsius. Se trata de un observador suscrito convencional (comunicación *push*) mediante método *notifyObservers*.
  2. Observador *pantallaTemperatura*, debe mostrar la temperatura tanto en grados centígrados como en Fahrenheit. Es un observador no suscrito, es decir, se comunicará con el sujeto observable de forma asíncrona (comunicación *pull*).
  3. Otro observador suscrito, eligiendo entre una de las siguientes tres posibilidades:
     a) Observador *botonCambio*, además de mostrar la temperatura en grados Celsius, podrá cambiar la temperatura del sujeto observable a petición del usuario. Para ello se romperá parte de la filosofía de este patrón de diseño mediante la inclusión de este observador suscrito que puede producir cambios en el modelo observado.
     b) Observador *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. Para ello se debe hacer uso de algún recurso GUI más sofisticado (mapas, librerías gráficas externas, etc.).
     c) Observador *Combo*, reunirá los observadores *graficaTemperatura* y *botonCambio*. Para ello se deberá hacer uso del patrón *compuesto* (composite).

- 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.

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

1.6. SESIÓN 1ª Módulo II: El patrón (estilo) arquitectónico *filtros de intercepción* para simular el movimiento de un vehículo con cambio automático (1 punto)

Queremos representar en el salpicadero de un vehículo (el vehículo implementará 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.

Usaremos para este ejercicio el patrón arquitectónico *filtros de intercepción*. 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). 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 estilo *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. 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 $0,15$.

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

Los métodos y secuencia de ejecución para llevar a cabo los servicios de filtrado 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 motor (*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. 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.

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.

Como ejemplo de programación de los dispositivos del control del vehículo, se pueden usar los botones (JButton) *Encender*, *Acelerar*, *Frenar* y una etiqueta (JLabel) con el estado "APAGADO" / "ACELERANDO" / "FRENANDO" dentro de un objeto panel de mandos (JPanel).

Funcionamiento de los botones:

- Inicialmente la etiqueta del panel principal mostrará el texto "APAGADO" (ver Figura 1.2 (a))) y los botones, "Encender", "Acelerar", "Frenar"
- 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 1.2 (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();
```

(a) (b)

Figura 1.2: Ejemplo de la ventana correspondiente al dispositivo de control del movimiento del vehículo (a) Estado “apagado”; (b) Estado “acelarando”.

```
...
}
```

La Figura 1.3 muestra un ejemplo de salpicadero sencillo.

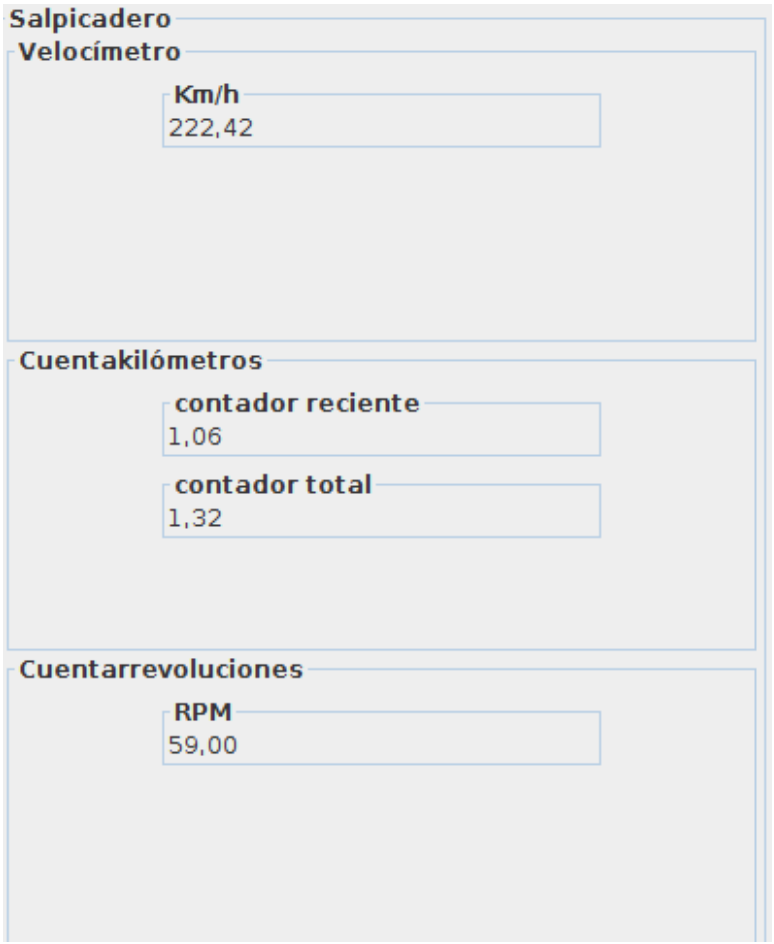

Figura 1.3: Ejemplo de la ventana correspondiente al salpicadero del vehículo

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

Este módulo constará de dos ejercicios.

1.7.1. Ejercicio S2-MI-Ej1. Patrón *Factoría Abstracta* y patrón *Método Factoría* (obligatorio - 0,6 puntos)

Descripción

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.

Deberán seguirse las siguientes especificaciones:

- 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 tanto para las carreras como para las bicicletas.
- Se usará un interfaz de usuario de texto.
- Se implementarán las modalidades montaña/carretera como las dos familias/estilos de productos[3].
- Se definirá la interfaz Java *FactoriaCarreraYBicicleta* para declarar 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*.

[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.

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

- Se definirán las clases *Bicicleta* y *Carrera* como clases abstractas que se especializarán 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*.

¿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*()

1.7.2. Ejercicio S2-MI-Ej2. Patrón *Factoría Abstracta* y patrón *Prototipo* en Ruby (opcional - 0,4 puntos)

Descripción

Diseña e implementa una aplicación con la misma funcionalidad que la del ejercicio anterior, pero en Ruby [4] y que aplique el patrón *Prototipo* en vez del patrón *Método Factoría*, junto con el patrón *Factoría Abstracta*[5].

1.8. SESIÓN 2ª Módulo II: Aplicación del patrón de comportamiento *visitante* (1 punto)

Este módulo constará de dos ejercicios, que se desarrollarán en C++.

1.8.1. Ejercicio S2-MII-E1: Patrón *visitante* básico (obligatorio - 0,6 puntos)

Utilizando este patrón 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);
```

[4] Investiga cómo se pueden crear hebras en Ruby, por ejemplo en **??**https://www.rubyguides.com/2015/07/ruby-threads/.

[5] Debe tenerse 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.

```
  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, o bien tiene declarado un método de ligadura dinámica (virtual) y no se implementa, o bien hereda un método de este tipo y no lo implementa, o ambas condiciones.

Las subclases de *VisitanteEquipo* definirán algoritmos concretos que se aplican sobre la estructura de objetos que se obtiene de instanciar las subclases de *ComponenteEquipo*. 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.

1.8.2. Ejercicio S2-MII-Ej2: Categorías de clientes (opcional - 0,4 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.

1.9. SESIÓN 3ª: Amplicación de funcionalidad y mejora de la GUI a partir de código obsoleto (*legacy code*), aplicando patrones arquitectónicos e incorporando alguna librería gráfica alternativa (1 punto)

Esta sesión constará de un único módulo con dos ejercicios, uno de ellos optativo. Los miembros del grupo deberán dividir el trabajo de implementación a partir del diseño arquitectónico realizado. Deberá además mejorarse la GUI, usando alguna librería gráfica externa[6] para diseñar componentes lo más parecidos posible a los elementos a los que representan, como por ejemplo, el velocímetro que debería aparecer como un círculo o semicírculo con una aguja señalando la velocidad actual. Como ejemplo puede descargarse applet que implementa un problema similar[7].

1.9.1. Ejercicio S3-MI-Ej1: Simulador de control automático para la conducción de un vehículo (SCACV) (obligatorio - 0,8 puntos)

A partir del ejercicio de simulación del movimiento de un vehículo automático (Ejercicio S1-MII-Ej1), debe añadirse nueva funcionalidad para desarrollar un SCACV y monitorizar los nuevos parámetros, modificando asimismo la GUI siguiendo las especificaciones que se detallan a continuación. Los patrones a usar no se proponen, sino que deberán ser elegidos por los desarrolladores del proyecto. Los criterios de selección y forma de aplicación serán tenidos en cuenta para la evaluación.

Descripción

El SCACV debe ser controlable mediante una palanca de cuatro posiciones y el pedal del freno, existiendo también el pedal del acelerador. Inicialmente el vehículo funcionará en modo manual, y el conductor usará los pedales del freno y del acelerador. Es importante que la velocidad de "crucero" del vehículo se mantenga, una vez alcanzada, ya que una velocidad superior a la necesaria implica un desperdicio de potencia consumida por el motor (y actualmente una retirada de puntos del carnet), por lo cual ha de mantenerse a toda costa. Para ello, el conductor pondrá la palanca del SCACV en la posición acelerar a la

[6] Por ejemplo, Java "SteelSeries".

[7] Para ejecutar el applet puede usarse el programa appletviewer que forma parte del Java Development Kit (JDK).

vez que suelta el pedal de aceleración y una vez alcanza la velocidad deseada, cambiará la palanca del SCACV a la posición de modo automático (estado "manteniendo"), cuando el vehículo alcance la velocidad deseada. Así, el vehículo mantendrá la velocidad de crucero. Para ello debe tenerse en cuenta el autómata de la Figura 1.4.

Descripción del SCACV y resto de dispositivos de control

El subsistema de control automático de velocidad del vehículo es controlado inicialmente por el conductor que puede poner la palanca en alguna de sus 4 posiciones conmutables: "acelerar", "apagado", "reiniciar" y "mantener".

- "Acelerar": manteniendo la palanca en esta posición la velocidad del motor se incrementa continuamente.
- "Mantener" o "modo automático": la velocidad actual del vehículo es memorizada por el sistema y el vehículo mantiene esta velocidad de forma constante. Hay que tener en cuenta que sólo si se apaga y se vuelve a encender el motor se cancela la última velocidad memorizada.
- "Reiniciar": el vehículo recupera la última velocidad de crucero almacenada (la que tenía la última vez que estuvo en "modo automático"). Una vez alcanzada, se modificará automáticamente la palanca del SCACV a la posición de "Mantener".
- "Apagado": se vuelve al modo de control manual de la velocidad del vehículo. El conductor ha de poder seleccionar esta posición si el SCACV estaba activado (cuando la palanca estaba en posición "Modo automático") o estaba en posición "Acelerar". También se pone automáticamente en esta posición si la palanca estaba en posición "Modo automático" o en posición "Reiniciar" y pisa el freno.

Además de este subsistema, el conductor también dispone de los siguientes controles relacionados con la marcha del vehículo:

- *Encendido/apagado del motor: Para arrancar-apagar el motor al inicio-fin de la ejecución del programa respectivamente.*
- *Frenar/soltar freno: Frena o deja de frenar el vehículo respectivamente. El frenado del vehículo, en el caso de estar activado el "Modo automático" o el de "Reiniciar" del SCACV, provocará automáticamente el cambio de la palanca del SCACV a la posición de "Apagado" (modo de control manual de la velocidad del vehículo).*
- *Acelerar/soltar acelerador (pedal): Acelera o deja de acelerar el vehículo respectivamente.*

Autómata que reproduce los estados del vehículo

En la Figura 1.4 se muestra un autómata con los 5 estados (nodos) del vehículo:

- motor **apagado o no arrancado (nodo transparente, vehículo parado o en desaceleración con motor apagado)**
- motor arrancado (nodos coloreados)
  - **encendido (vehículo parado o en desaceleración por rozamiento con motor arrancado)**
  - **frenando**
  - **acelerando**
  - **manteniendo: este es el único estado que no se alcanza con la conducción manual y por tanto que no se representó en el ejercicio del módulo 2 de la sesión 1ª**

Descripción de los estados del diagrama y su relación con los estados del SCACV

Se puede observar en la figura que los mismos estados se repiten con distintos colores, lo que significa que el estado del SCACV (posición de la palanca y si la velocidad de crucero ha sido almacenada desde la última vez que se arrancó el vehículo) es distinto según la Tabla 1.1:

		Velocidad almacenada	
		No	Sí
Posición de la palanca	Apagado		
	Reiniciar	-	
	Acelerar		
	Mantener	-	

Tabla 1.1: Colores usados en el autómata para representar los distintos estados del SCACV.

1.9.2. Ejercicio S3-MI-Ej2: Ampliación del simulador del vehículo automático con SCACV con un subsistema de monitorización del consumo (optativo - 0,2 puntos)

Este ejercicio ampliará el ejercicio anterior.

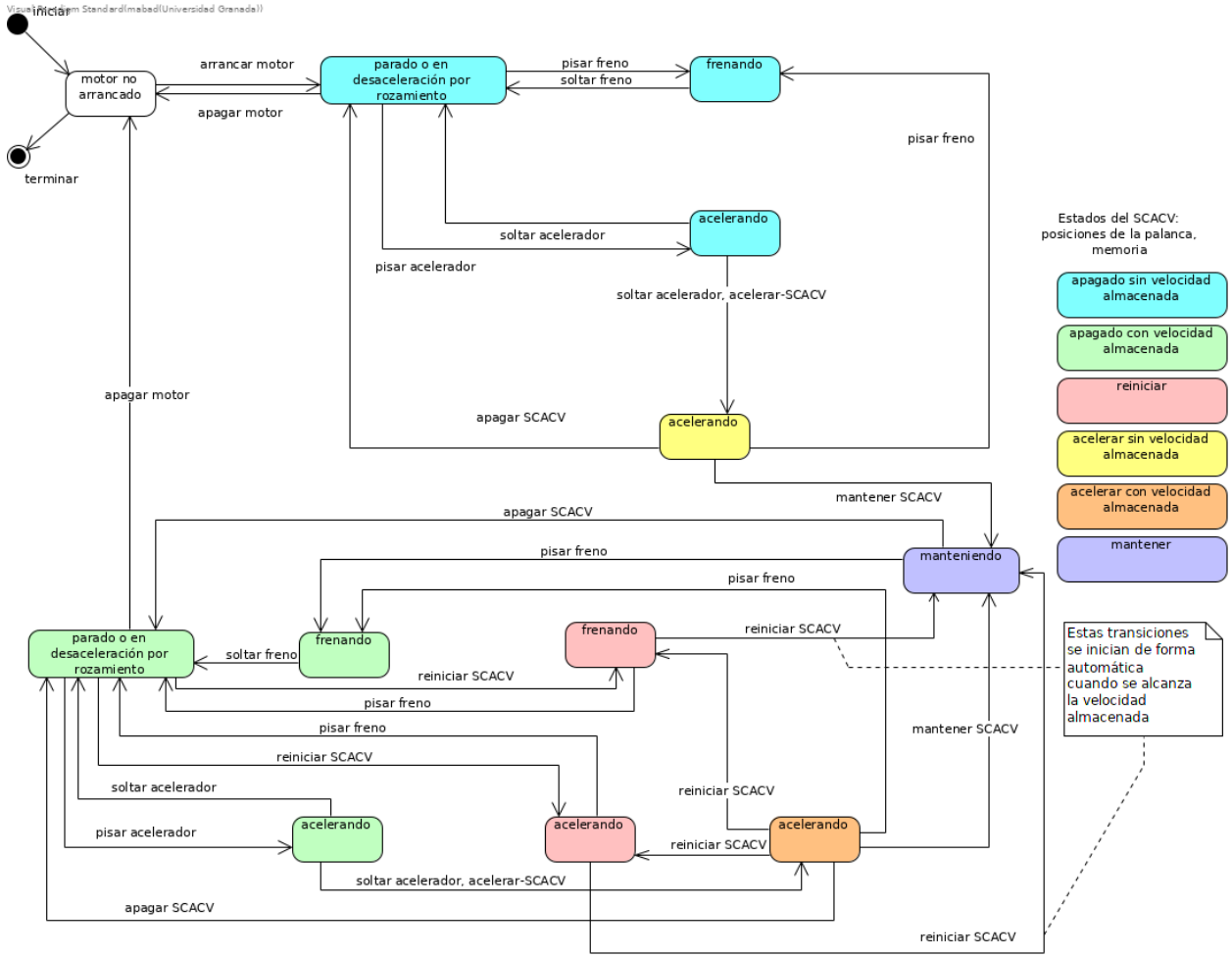

Figura 1.4: Autómata finito que representa los estados del vehículo y el SCACV y las transiciones entre ellos.

Descripción

Para ello será necesario guardar cada vez que se actualiza la monitorización: (1) el instante en el que se realiza y (2) las vueltas (revoluciones) producidas por el eje desde la última actualización (a partir del instante anterior y la velocidad anterior). Para cada consumible deberán también almacenarse el tiempo y las rotaciones totales del eje desde el último cambio/recarga, y además:

- *Consumo de combustible*: debe almacenarse el nivel de combustible alcanzado cuando se reposta o se inicia la aplicación (asignado de forma aleatoria dentro de un rango). El consumo debe calcularse en función del número de revoluciones del eje desde la última vez que se actualizó (por ejemplo, puede considerarse que el consumo en litros es de $rot \times rot \times 5 \times 10^{-10}$, siendo rot el número de revoluciones (vueltas) del eje desde la última vez que se actualizó. Cuando el depósito baje del 15 % deberá aparecer un mensaje indicando que se debe repostar. Se añadirá un botón para realizar el repostaje, que deberá hacerse con el coche apagado.

- *Consumo aceite, pastillas de freno y revisión general*: El sistema proporciona notificaciones de mantenimiento a sus usuarios: cada 5×10^6 rotaciones del eje aparecerá un mensaje en la pantalla que le avisará de la necesidad de cambiar el aceite del motor; cada 10^8 rotaciones para cambio del pastillas y cada 10^9 rotaciones para efectuar una revisión general del sistema. Además se incluirán 3 botones para hacer el cambio de aceite, de pastillas de freno y revisión general respectivamente. Esos botones requerirán para estar activos que el motor del vehículo esté apagado y el vehículo parado, de forma que sea accesible para su mantenimiento. Las acciones que deben realizarse al ser pulsados serán las de actualizar, respectivamente (1) número de rotaciones acumuladas en la fecha del ultimo engrase, (2) número de rotaciones en la fecha del cambio pastillas de freno y (3) número de rotaciones en la última fecha de revisión general.