Desarrollo de Software
Curso 2023-2024
3º - Grado Ingeniería Informática

Guión de prácticas

Dto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

18 de marzo de 2024

Índice general

Criterios

1. Características

- Hay un total de 4 prácticas, de tres sesiones cada una de ellas.
- Cada práctica puntuará lo mismo en la nota final de prácticas (2.5 puntos).
- Las prácticas se realizarán en grupos de 4 personas.
- En cada práctica se realizará una memoria en formato PDF (explicación, diagramas de diseño, código/pseudocódigo, problemas encontrados, soluciones, etc.)
- Se valorará el uso de Github para el control de versiones y Latex para la memoria.
- Se necesita obtener una calificación superior a 4,5 en cada práctica para que se pueda hacer media con la teoría.
- Si se detecta copia en una práctica todos los integrantes tendrán una calificación de 0 en esa práctica, y por el punto anterior se tendrá la parte de prácticas suspensa.
- La parte práctica cuenta un 50 % de la nota final.

Criterios

2. Criterios de evaluación generales

Criterios para superar cada práctica:

- Calidad (se cumplen los requisitos no funcionales)
- Capacidad de trabajar en equipo (reparto equitativo de tareas)
- Implementación completa y verificabilidad (sin errores de ejecución, se cumplen los requisitos funcionales)
- Fidelidad de la implementación al patrón de diseño
- Reutilización de métodos (ausencia de código redundante)
- Validez (de acuerdo con las expectativas del cliente, el profesor en este caso)

3. Plazos de entrega y presentación de cada práctica

Cada práctica completa (código + memoria) será subida a PRADO en una fecha acordada en las clases de prácticas. En la sesión de prácticas acordada los estudiantes de cada grupo deberán realizar una defensa de la práctica. La nota de cada práctica NO será igual para todo el grupo, dependerá del trabajo individual aportado por cada integrante.

Práctica P1

Práctica 1

1. Descripción

Se realizarán programas Orientados a Objetos (OO) bajo distintos patrones de diseño creaciones y estructurales.

2. 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 distintos patrones creacionales y estructurales a problemas diversos
3. Adquirir destreza en la práctica de diseño OO
4. Aprender a adaptar los patrones a las especificidades de distintos lenguajes OO

1. Ejercicio 1. Patrón Factoría Abstracta en Java

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 para que las carreras se inicien de forma simultánea.

- Se implementarán las modalidades montaña/carretera como las dos familias/estilos de productos[1].

- 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*.

- 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*.

[1] 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.

2. Ejercicio 2. Patrón Factoría Abstracta + Patrón Prototipo (Python ó Ruby)

Descripción

Diseña e implementa una aplicación con la misma funcionalidad que la del ejercicio anterior, pero que aplique el patrón *Prototipo* junto con el patrón *Factoría Abstracta*.

Se elegirá el lenguaje de programación Python o Ruby. Para simplificar: no es necesario el uso de hebras.

3. Ejercicio 3. Patrón libre (NO usado anteriormente) estudiado en teoría en Python/Java/Ruby (a elegir)

Descripción

Diseño e implementación de una aplicación que utilice un patrón de diseño libre.

- Usar Patrón Builder o algún Patrón que NO sea creacional para no repetir.
- Se valorará complejidad del problema y solución acordada en grupo.
- Extra: Combinación de patrones.

4. Ejercicio 4. El patrón (estilo) arquitectónico *filtros de intercepción* en java. Simulación del movimiento de un vehículo con cambio automático

Para la realización de este ejercicio realizar una de las siguientes opciones:

1. Java + Interfaz Gráfica
2. Python + Interfaz Gráfica (Tkinter o similar)

A continuación se va a explicar el ejercicio en Java. Para python es análogo:

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) 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 P1.1 (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 P1.1 (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:

(a) (b)

Figura P1.1: Ejemplo de la ventana correspondiente al dispositivo de control del movimiento del vehículo (a) Estado “apagado”; (b) Estado “acelarando”.

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

La Figura P1.2 muestra un ejemplo de salpicadero sencillo.

[H]

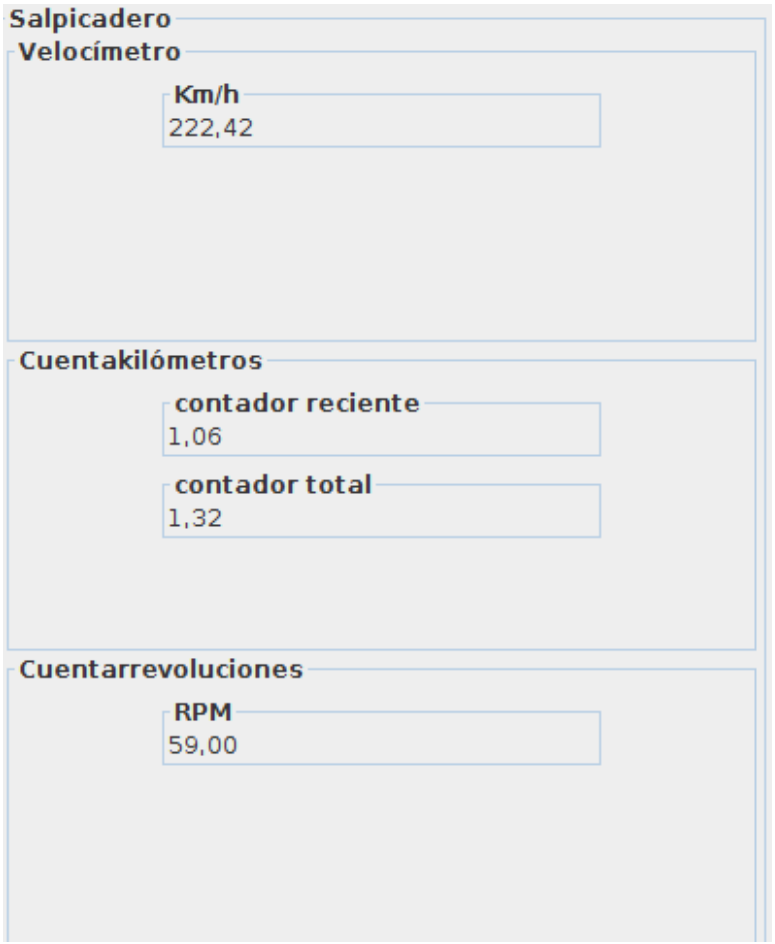

Figura P1.2: Ejemplo de la ventana correspondiente al salpicadero del vehículo

5. Ejercicio 5 (opt) +1 punto

Diseño de una aplicación de WebScraping en Python donde se utilice el Patrón *Strategy* para scrapear información en vivo de acciones en la página web https://finance.yahoo.com/quote/, donde se puede añadir el sufijo de la acción para acceder a información en vivo de ésta. Ejemplo con TESLA: https://finance.yahoo.com/quote/TSLA.

Para el Patrón Strategy se usarán 2 estrategias:

1. BeautifulSoup
2. Selenium

Se desea scrapear de la página anterior los siguientes elementos:

- Precio de cierre anterior, precio de apertura, volumen, capitalización de mercado.
- Toda esta información se guardará en un fichero .json, de forma que cada vez que se ejecute el código se borre el contenido anterior y se mostrará el nuevo.

Se valorará estilo, calidad e implementación del código.

Práctica P2

Práctica 2

1. Objetivos generales

1. Adaptar un software a una plataforma diferente, con un lenguaje de programación distinto
2. Ampliar funcionalidad de un software
3. Hacer uso de software/librerías COTs, por ejemplo añadiendo una librería gráfica para mejorar la GUI o una librería para mejorar/facilitar tareas de diseño (como la librería BLoC de Flutter)
4. Aprender a aplicar patrones arquitectónicos

2. Criterios de evaluación

Para superar esta práctica, además de los criterios de evaluación general, será necesario cumplir con todos y cada uno de los siguientes criterios:

- Capacidad de realizar mantenimiento adaptativo, reestructurando (incluso cambiando o añadiendo patrones de diseño) y refactorizando código
- Fidelidad a un patrones de diseño y patrones arquitectónicos.

Se puede considerar como una labor de reingeniería y usar esa propuesta (ver figura 2.2 del tema 2). La ingeniería inversa puede hacerse obteniendo los diagramas de clase del código obsoleto (mediante reversión de código, por ejemplo usando Visual Paradigm).

Además hay que reestructurar el código para adecuarse a la nueva funcionalidad y a las características del nuevo entorno de desarrollo y uso (lenguaje -Dart-, framework -Flutter-, dispositivos móviles). En concreto, ya que se trata de una GUI con una parte importante dedicada a la vista y el controlador, se recomienda separar el modelo del resto de elementos, usando por ejemplo el patrón arquitectónico MVC. Otra opción, que se recomienda en el caso de Flutter, es usar un patrón específico, llamado BLoC (**B**usiness **Lo**gic **C**omponent), que "saca" el modelo de los widgets con estado y lo pone aparte, separando por tanto el modelo (la lógica del software) y dejando unidas vista y controlador. En la figura **??** se muestra un diagrama con la interacción de la lógica de la aplicación y los distintos widgets.

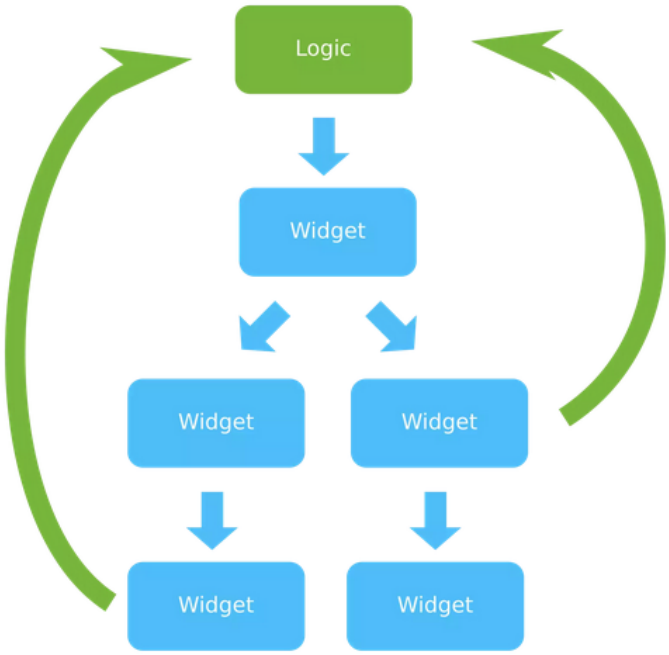

Figura P2.1: Ejemplo esquema de componentes usando el patrón BLoC en Flutter [Fuente: Web Flutter]

Por otro lado, para la refactorización se puede partir de la detección de *código hediondo*. Esto se puede hacer en parte a partir del diagrama de clases del código obsoleto (ahí pueden verse métodos con muchos argumentos, o clases con muchos métodos); pero hay otra parte que solo se detecta mirando directamente el código que va a migrarse (métodos muy largos o cadenas de mensajes).

3. Ejercicios Individuales - Taller/Actividades Formativas

Tras haber recibido el taller sobre Flutter la semana del 18 de Marzo, se realizarán 2 ejercicios para que el alumnado se familiarice con el uso de esta plataforma. Para considerarse aptos, la media de ambos ejercicios debe ser superior a 4.5, tal y como se indica en la guía docente.

1. Ejercicio Individual 1

Simulación de una calculadora. El usuario deberá introducir operación y dos elementos a través de la interfaz de usuario (UI), de forma que el modelo mostrará la solución en dicha UI. La implementación de las diferentes operaciones a considerar Suma, Producto, Potencia, etc deberán heredar de una clase Operacion.

2. Ejercicio Individual 2

Desarrollar una aplicación en Flutter que permita a los usuarios gestionar una lista de tareas. Cada tarea podrá ser marcada como completada o pendiente, y el usuario tendrá la capacidad de agregar nuevas tareas y eliminar las existentes.

Se deben crear las clases Tarea y GestorDeTareas.

- La clase Tarea debe tener al menos dos atributos: String descripcion y bool completada.
- La clase GestorDeTareas deberá contener una lista de objetos Tarea y métodos para agregar, eliminar y marcar tareas como completadas.

La interfaz debe incluir:

1. Un TextField para ingresar la descripción de la nueva tarea.
2. Un botón para agregar la tarea a la lista.
3. Mostrar la lista de tareas utilizando un widget ListView, donde cada elemento de la lista permita marcar la tarea como completada y tenga un botón para eliminarla.
4. Cada tarea en la lista debe mostrar de manera visual si está completada o pendiente.

- Al agregar una nueva tarea, el campo de texto debe limpiarse.

- Al marcar una tarea como completada, debe cambiar visualmente de manera que se diferencie de las tareas pendientes.
- Al eliminar una tarea, esta debe desaparecer de la lista.

Como requisitos técnicos, se debe utilizar StatefulWidget para gestionar el estado de la lista de tareas, así como buenas prácticas de programación estudiadas.

4. Ejercicio Grupal

Diseño adaptativo y perfectivo de la Actividad 3 de la Práctica 1 en Flutter/Dart, así como una UI con widgets para que el usuario pueda interactuar con las diferentes funcionalidades implementadas.

5. Evaluación

- Los ejercicios individuales se enseñarán el día correspondiente de la defensa de esta práctica. Justificar con diagrama UML (hecho a mano sirve) si es necesario. No hacer memoria de estos ejercicios, ya que cuentan como actividades formativas/talleres.
- Para el ejercicio 2 se debe realizar una memoria donde se detallen problemas encontrados/soluciones, widgets utilizados, mejoras (diseño perfectivo) realizadas en el software, escalabilidad, etc.