Simulacro de examen T2 - Diseño orientado a objetos y diseño y realización de pruebas

Instrucciones generales

- **Duración:** 2 horas y 15 minutos.
- **Formato:** El examen consta de dos partes: una de diseño orientado a objetos y otra de diseño y realización de pruebas.
  - **Parte 1:** Diseño orientado a objetos (1 hora y 15 minutos).
  - **Parte 2:** Diseño y realización de pruebas (1 hora).

Parte 1: Diseño orientado a objetos – Creación de un Diagrama de Clases UML

Una academia de formación quiere crear una **plataforma de formación online** que permita a usuarios **inscribirse en cursos y acceder a material educativo**, así como a instructores **crear y gestionar cursos**.

Requisitos del modelo:

1️⃣ **Usuarios y Roles**

- Todo usuario registrado en la plataforma tiene un **nombre, correo electrónico y una fecha de registro**.
- Existen **dos tipos de usuarios**:
  i. **Estudiantes** → Pueden **inscribirse en cursos** y **realizar tareas**.
  ii. **Instructores** → Pueden **crear cursos, gestionar estudiantes inscritos y calificar tareas**.
- Los estudiantes pueden inscribirse en **múltiples cursos**, y los instructores pueden **crear y gestionar varios cursos**. Un instructor también puede **inscribirse como estudiante** de un curso, siempre que no sea el instructor del mismo.

2️⃣ **Cursos y Lecciones**:

- Los **cursos** tienen un **código único**, un **título**, una **descripción** y una **duración en horas**. Ademas, cada curso es impartido por uno o varios **instructores**, aunque solo uno de ellos es el **responsable** del curso. Además, en un curso puede haber matriculados hasta 30 alumnos (aquí no cuentan los posibles instructores que quieran recibir dicho curso).
- Por otra parte, un curso está formado por varias **lecciones**. Una **lección** forma parte de un curso y tiene: **un título** y **contenido textual o en video**. **Opcionalmente, una tarea asociada**, que el estudiante puede realizar. Un **estudiante puede completar una tarea** y recibir una **calificación entre 0 y 10 puntos**. Las lecciones también pueden tener un examen asociado, o un proyecto. Todos ellos serán **calificables**, pero cada uno puede tener una gestión diferente de la calificación (por ejemplo, un proyecto puede estar calificado como apto/no apto).

3️⃣ **Inscripciones y Evaluaciones**:

- Cuando un estudiante se inscribe en un curso, se crea una **inscripción** que asocia al estudiante con el curso. Se debe guardar una fecha de inscripción. Además, si un estudiante no supera el curso, puede volver a inscribirse en él.
- Para cada tarea enviada, examen o trabajo realizados, el instructor puede asignar una **calificación**.
- Un estudiante puede ver sus **calificaciones obtenidas** en cada curso.
- Se debe definir un método en `Curso` para calcular el **promedio de calificaciones** de todos los estudiantes inscritos.

5️⃣ **Datos Globales**:

- La plataforma debe llevar la cuenta de los cursos creados en ella.
- Cada vez que se crea un curso, este contador debe **aumentar automáticamente**.
- También se debe proporcionar las funcionalidades de registro de usuarios y de creación de cursos.

Instrucciones

📌 1️⃣ **Crear un diagrama de clases UML** que represente todos los elementos descritos. Crea las clases necesarias para modelar todo el sistema, escoge las relaciones entre dichas clases, define la cardinalidad adecuada en cada relación y crea las jerarquías que consideres. Añade restricciones donde sea necesario y ayúdate de las anotaciones si lo necesitas.

Criterios de evaluación de la Parte 1

Criterio	Ponderación
Modelado correcto de entidades y relaciones	30%
Uso adecuado de herencia, interfaces y clases abstractas	30%
Definición correcta de multiplicidades y asociaciones	20%
Uso adecuado de atributos o métodos estáticos	10%
Uso adecuado de anotaciones y restricciones	5%
Claridad y organización del diagrama	5%

Parte 2: Diseño y realización de pruebas

Ejercicio 2 – Diseño e Implementación de Pruebas para un Sistema de Parking

Un parking utiliza un sistema automático para calcular el precio a pagar según el tiempo de estacionamiento y condiciones especiales. Para ello, disponemos de la clase `Parking` con un método `calcularTarifa()` que implementa las siguientes **reglas de tarificación:**

1. **Tarifa base**: Se cobra **2€/hora completa**.
2. **Fracción de hora**: Si el vehículo estuvo estacionado más de **15 minutos adicionales**, se cobra **una hora extra**.
3. **Máximo diario**: Si el estacionamiento supera **10 horas**, se cobra un **importe fijo de 18€**.
4. **Clientes abonados**: Tienen un **20% de descuento** sobre la tarifa final.
5. **Horario nocturno (22:00 - 06:00)**: Durante este periodo, la tarifa **se reduce al 50%**.
6. **Entrada inválida**: Si los valores de **horas o minutos son negativos** o si los minutos superan 59, se debe lanzar una excepción.

Por otra parte, el sistema permite registrar a nuevos clientes mediante el método `registrarAbonado()`. Esta es la descripción completa de este método:

Firma del método:

```
public boolean registrarAbonado(String matricula, boolean esVIP)
```

El método permite registrar una matrícula en el sistema de abonados del parking, indicando si el cliente es VIP o no.

Parámetros:

Parámetro	Tipo	Descripción
`matricula`	`String`	Matrícula del vehículo a registrar. Debe ser una cadena no vacía ni nula, y con el formato 0000BBB, donde 0000 son dígitos y BBB son letras no vocales
`esVIP`	`boolean`	Indica si el cliente es VIP (`true`) o no (`false`).

Valor de retorno:

Retorno	Descripción
`true`	Se ha registrado correctamente la matrícula.
`false`	La matrícula ya estaba registrada previamente.

Requisitos Funcionales

1. **Registro de una matrícula válida**
   ✅ Si la matrícula **no está registrada**, se almacena en el sistema y el método devuelve `true` .
2. **Registro de un abonado ya existente**
   ✅ Si la matrícula **ya está registrada**, el método no la sobrescribe y devuelve `false` .
3. **Restricciones sobre la matrícula**
   ❌ Si la matrícula es **nula (`null`)**, **una cadena vacía (`""`)**, o **contiene solo espacios (`"  "`)**, el método debe lanzar una excepción:

   ```
   throw new IllegalArgumentException("Matrícula inválida");
   ```

   ❌ Si la matrícula **no cumple con el formato esperado (`0000BBB`)**, el método debe lanzar una excepción.

   ```
   throw new IllegalArgumentException("Formato de matrícula incorrecto");
   ```

4. **Independencia del estado VIP**
   ✅ El valor de `esVIP` **no afecta al registro** en cuanto a la validez de la matrícula.

A continuación se muestra el **Código de la clase `Parking` incluyendo los métodos `calcularTarifa()` y `registrarAbonado()`:**

```java
import java.util.HashMap;
import java.util.Map;

public class Parking {
    private static final String FORMATO_MATRICULA = "^[0-9]{4}[BCDFGHJKLMNPRSTVWXYZ]{3}$"; // E
    private Map<String, Boolean> abonados = new HashMap<>(); // Cada matrícula se asocia a un bo
    private String nombre;
    private int plazas;

    /**
     * Registra un vehículo como abonado en el sistema de parking.
     *
     * @param matricula Matrícula del vehículo a registrar. Debe cumplir con el formato "0000BBB
     * @param esVIP Indica si el abonado es VIP (true) o no (false).
     * @return true si el registro es exitoso, false si la matrícula ya estaba registrada.
     * @throws IllegalArgumentException si la matrícula es nula, vacía o no cumple el formato.
     */
    public boolean registrarAbonado(String matricula, boolean esVIP) {
        if (matricula == null || matricula.trim().isEmpty()) {
            throw new IllegalArgumentException("Matrícula inválida");
        }

        if (!matricula.matches(FORMATO_MATRICULA)) {
            throw new IllegalArgumentException("Formato de matrícula incorrecto");
        }

        if (abonados.containsKey(matricula)) {
            return false; // La matrícula ya está registrada
        }

        abonados.put(matricula, esVIP);
        return true; // Registro exitoso
    }

    /**
     * Verifica si un vehículo está registrado como abonado.
     *
     * @param matricula Matrícula del vehículo a verificar.
     * @return true si el vehículo está registrado como abonado VIP, false en caso contrario.
     */
    public boolean esAbonadoVip(String matricula) {
        return abonados.getOrDefault(matricula, false);
    }
```

```java
    /**
     * Calcula la tarifa a pagar por el estacionamiento de un vehículo.
     *
     * @param horas Número de horas completas de estacionamiento.
     * @param minutos Fracción de hora adicional (0-59).
     * @param esAbonado Indica si el vehículo es abonado.
     * @param tarifaNocturna Indica si el estacionamiento se realizó en horario nocturno (22:00
     * @return Tarifa a pagar por el estacionamiento.
     * @throws IllegalArgumentException si los valores de horas o minutos son inválidos.
     */
    public double calcularTarifa(int horas, int minutos, boolean esAbonado, boolean tarifaNoctu
        if (horas < 0 || minutos < 0 || minutos >= 60) {
            throw new IllegalArgumentException("Tiempo inválido");
        }

        double tarifa = 0;

        // Si el tiempo supera las 10 horas, se cobra tarifa fija
        if (horas >= 10) {
            tarifa = 18;
        } else {
            tarifa = horas * 2.0;  // 2€/hora

            // Si hay fracción mayor a 15 min, se cobra una hora más
            if (minutos > 15) {
                tarifa += 2.0;
            }

            // Si es horario nocturno (22:00 - 06:00), se cobra al 50%
            if (tarifaNocturna) {
                tarifa *= 0.5;
            }
        }

        // Descuento para abonados
        if (esAbonado) {
            tarifa *= 0.8;
        }

        return tarifa;
    }
}
```

Encontrarás este código en el siguiente enlace de GitHub.

Diseño de casos de prueba

Parte 1: Pruebas de Caja Negra

Realiza el diseño de pruebas de **caja negra** para el método `registrarAbonado()` siguiendo los siguientes pasos:

- Definir particiones de equivalencia y valores límite para cada regla.
- Identificar los escenarios relevantes y sus valores de entrada.
- Define los casos de prueba con los datos de entrada y salida esperados.

Parte 2: Pruebas de Caja Blanca

Realiza el diseño de pruebas de **caja blanca** para el método `calcularTarifa()` siguiendo los siguientes pasos:

- Obtener el **grafo de flujo**. Créalo utilizando Draw.io (versión de escritorio o extensión de vscode).
  - Etiqueta los nodos para definir más cómodamente los caminos.
  - Numera las regiones resultantes.
- Calcular la **complejidad ciclomática** utilizando las 3 fórmulas conocidas.
- Definir los **caminos de prueba**.
- Diseñar los **casos de prueba** con los datos de entrada y salida esperados.

Implementación de pruebas en JUnit

Parte 3: Creación de tests en JUnit

- Crea una copia del proyecto proporcionado en tu equipo. En la carpeta `src/test/java` debes:
  - **Implementar** las pruebas unitarias en **JUnit 5** a partir del diseño de tests de caja negra para el método `registrarAbonado()` .
  - **Implementar** las pruebas unitarias en **JUnit 5** a partir del diseño de tests de caja blanca para el método `calcularTarifa()` .
  - Crea una **clase Suite** que permita centralizar la ejecución de todas las pruebas.
- **Ejecuta** todos los tests, comprueba que todos pasan y realiza **capturas de pantalla** de la ejecución.
- Realiza una ejecución de test con cobertura de código, comprueba que se alcanza una cobertura del 100% y realiza capturas de pantalla de la ejecución.

Criterios de evaluación de la Parte 2

Criterio	Ponderación
Pruebas de caja negra para `registrarAbonado()`	25%
Pruebas de caja blanca para `calcularTarifa()`	25%
Implementación de tests en JUnit 5	25%
Ejecución de tests y capturas de pantalla	20%
Cobertura de código con JUnit 5	5%