Problema Patrones

Una empresa mediana busca reemplazar su sistema de venta de vehículos. Como parte del análisis y en base a los requerimientos actuales, se han identificado un conjunto de entidades que forman una herencia de clases: Vehículo, Automóvil y Motocicleta (Automóvil y Motocicleta son subclases de Vehículo) y AutoEléctrico (que es una subclase de Automóvil). Todos los Vehículos cuentan con una marca, un modelo y una capacidad de pasajeros; el Automóvil cuenta con un tipo de carrocería; y el AutoEléctrico cuenta con una capacidad de batería y un tiempo de autonomía. A futuro, con el incremento de las ventas, se esperan incluir otros tipos de vehículos dentro de la jerarquía; y se buscaría reemplazar el motor de base de datos con el que actualmente se cuenta por una tecnología más robusta.

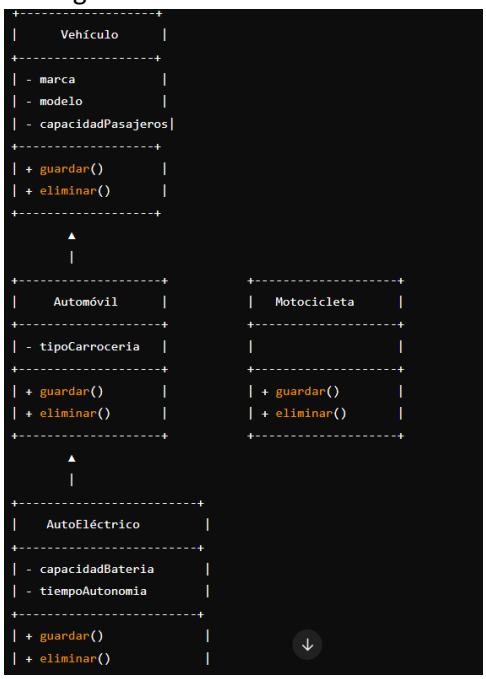
Con base al caso:
- Identificar el patrón de persistencia de objetos que utilizaría para cubrir la necesidad. Justifique su respuesta. (1.5 puntos)
- Identificar el patrón de interacción con base de datos que utilizaría para cubrir la necesidad. Justifique su respuesta. (1.5 puntos)
- Elaborar el diagrama de clases de diseño (éste debe incluir las clases con sus atributos para soportar la persistencia y la interacción contra la base de datos). (1.5 puntos)
- Diagramar las tablas necesarias para soportar la persistencia de los objetos de acuerdo con el patrón de persistencia elegido (solo incluir los nombres de las tablas y sus columnas; no es necesario incluir los tipos de datos). (1.5 puntos)

1. Patrón de Persistencia de Objetos
Patrón sugerido: Table-per-class (Una tabla por clase concreta).
Justificación: Este patrón de persistencia es útil cuando se tiene una jerarquía de clases, como es el caso de Vehículo, Automóvil, Motocicleta y AutoEléctrico. En este patrón, cada subclase concreta (Automóvil, Motocicleta, AutoEléctrico) tiene su propia tabla en la base de datos. Cada tabla almacena solo los atributos de su clase y hereda la relación con la tabla de la clase base (Vehículo) a través de claves foráneas.
Este enfoque es apropiado por varias razones:

- Permite extender fácilmente la jerarquía en el futuro sin afectar significativamente la estructura de las tablas existentes.
- Evita tener una tabla muy grande con muchos campos que serían nulos para algunas subclases (como en el patrón Table-per-hierarchy).
- Es más flexible si se prevé que en el futuro las subclases podrían tener diferentes conjuntos de datos.

2. Patrón de Interacción con la Base de Datos
Patrón sugerido: Data Mapper (Mapeador de Datos).
Justificación: El patrón Data Mapper desacopla la lógica de negocio del acceso a la base de datos mediante el uso de una capa intermedia que se encarga de mapear las clases de la aplicación a las tablas en la base de datos. Este patrón es muy adecuado cuando se tienen clases complejas con herencia, ya que permite gestionar la complejidad de los mapeos sin que la lógica de negocio se vea afectada por la estructura de la base de datos.
Con este patrón, es posible cambiar la tecnología de la base de datos en el futuro sin tener que modificar la lógica de negocio. Dado que la empresa planea reemplazar su base de datos actual por una tecnología más robusta, el Data Mapper ofrece flexibilidad para adaptarse a estos cambios.

3. Diagrama de Clases de Diseño




4. Diagrama de Tablas para la Persistencia de Objetos

Cada clase concreta tendrá su propia tabla en la base de datos, con claves foráneas que representen las relaciones de herencia. Aquí están las tablas necesarias:

**Tabla Vehículo** (Clase base)

| Columna | Descripción |
|---|---|
| id | Identificador único del vehículo (PK) |
| marca | Marca del vehículo |
| modelo | Modelo del vehículo |
| capacidadPasajeros | Capacidad de pasajeros del vehículo |

**Tabla Automóvil** (Hereda de Vehículo)

| Columna | Descripción |
|---|---|
| id | Identificador único del automóvil (PK, FK de Vehículo) |
| tipoCarroceria | Tipo de carrocería del automóvil |

**Tabla Motocicleta** (Hereda de Vehículo)

| Columna | Descripción |
|---|---|
| id | Identificador único de la motocicleta (PK, FK de Vehículo) |

**Tabla AutoEléctrico** (Hereda de Automóvil)

| Columna | Descripción |
|---|---|
| id | Identificador único del auto eléctrico (PK, FK de Automóvil) |
| capacidadBateria | Capacidad de la batería (en kWh) |
| tiempoAutonomia | Tiempo de autonomía (en horas) |

Problema anti patrones

En base al listado de anti patrones de diseño vistos en clase: indique los dos (2) que considera que tienen un mayor impacto negativo al dar mantenimiento (corregir errores, modificar funcionalidades existentes o incluir nuevas funcionalidades) al software implementado. Justifique su respuesta considerando las consecuencias negativas de aplicar dichos antipatrones y el esfuerzo necesario para aplicar un refactoring.

Al considerar los antipatrones de diseño y su impacto en el mantenimiento del software, dos que tienen un mayor impacto negativo son:
1. Código Espagueti (Spaghetti Code)
El "código espagueti" se refiere a un código desorganizado, difícil de seguir y entender, con una estructura compleja y conexiones no claras entre diferentes partes del programa. Las principales consecuencias negativas de este antipatrón son:

Dificultad para entender el código: Este tipo de código suele carecer de una estructura clara y estar lleno de dependencias y saltos lógicos entre funciones o clases. Esto hace que cualquier intento de corregir errores o implementar nuevas funcionalidades requiera un esfuerzo enorme para entender cómo funciona el sistema completo.

Alta probabilidad de introducir nuevos errores: Debido a la falta de modularidad y la interdependencia de las distintas partes del código, cualquier modificación en un área puede afectar inesperadamente otras partes del sistema. Esto incrementa el riesgo de errores y obliga a realizar pruebas exhaustivas en toda la aplicación, incluso si la modificación es pequeña.

Refactorización costosa: Limpiar el "código espagueti" implica descomponerlo en componentes más manejables y crear interfaces claras. Esto requiere una reescritura significativa del código, la creación de pruebas unitarias y la eliminación de dependencias circulares, lo que demanda un gran esfuerzo.

2. El Objeto Dios (God Object)
Un "objeto Dios" es un objeto que asume demasiadas responsabilidades, violando el principio de responsabilidad única (SRP). Controla múltiples aspectos de la aplicación y conoce demasiados detalles sobre el sistema. Las principales consecuencias negativas son:

Alta complejidad: Un objeto que maneja demasiadas funciones se vuelve demasiado complejo para entender o modificar. A medida que crece su tamaño y responsabilidades, el riesgo de errores aumenta exponencialmente, ya que cualquier cambio en sus métodos puede afectar a gran parte del sistema.

Dificultad para mantener y probar: Debido a que el objeto centraliza muchas responsabilidades, las pruebas unitarias se vuelven más complicadas. Además, modificar este objeto puede requerir una comprensión exhaustiva de su comportamiento y su impacto en el sistema completo.

Refactorización costosa: Dividir un "objeto Dios" en múltiples objetos con responsabilidades claras implica rediseñar gran parte de la estructura del código. Además, este proceso requiere identificar las responsabilidades que deben delegarse y diseñar interfaces adecuadas, lo que también implica un esfuerzo considerable.
Justificación:
Ambos antipatrones dificultan enormemente el mantenimiento porque la estructura del código es poco modular y altamente acoplada. El código espagueti es difícil de modificar sin romper partes no relacionadas, mientras que el objeto Dios centraliza demasiado la lógica, complicando las modificaciones. En ambos casos, el esfuerzo de refactorización es alto porque requiere no solo reorganizar el código, sino también implementar una estructura más lógica y sostenible.