UNIVERSIDAD AMERICANA

Metodología y Programación orientada a objetos

Primer corte evaluativo - #2

Estudiantes:
Luis Arturo Lozano Zelaya - 23020824

María Violeta Vanegas Sarria 23020383

Yazid Zahid Solís Balladares-23020856

Fernanda Concepción Gutiérrez Gómez-23021118

Docente:
Norman José Cash García

Caso de Estudio: Sistema de Gestión de Vehículos

Contexto

Una empresa de transporte necesita un sistema para gestionar diferentes tipos de vehículos, como automóviles y bicicletas. Cada tipo de vehículo tiene características comunes (como marca, modelo y año) y comportamientos específicos (como moverse o recargar combustible). El sistema debe ser flexible para permitir la adición de nuevos tipos de vehículos en el futuro y debe aprovechar los principios de programación orientada a objetos, como herencia, polimorfismo, clases abstractas e interfaces.

Tu tarea es desarrollar un programa en Java que modele este sistema, implementando los conceptos de polimorfismo, herencia, métodos sobrescritos, uso de `super`, clases abstractas e interfaces. El programa debe permitir crear vehículos, moverlos, mostrar sus detalles y, en el caso de los automóviles, gestionar el combustible.

Objetivos de Aprendizaje

1. Comprender y aplicar los conceptos de herencia para reutilizar código entre clases.

2. Implementar polimorfismo mediante la sobrescritura de métodos y el uso de interfaces.

3. Utilizar la palabra clave `super` para invocar constructores y métodos de la clase padre.

4. Diseñar e implementar clases abstractas para definir comportamientos comunes.

5. Usar interfaces para garantizar que ciertas clases implementen comportamientos específicos.

Código Fuente:

```
// Interfaz que define el comportamiento común para vehículos con capacidad de combustible
interface Combustible {
    void recargarCombustible();
    double obtenerNivelCombustible();
}

// Clase abstracta que representa un vehículo genérico
abstract class Vehiculo {
    protected String marca;
    protected String modelo;
    protected int anio;
```

```
    public Vehiculo(String marca, String modelo, int anio) {
        this.marca = marca;
        this.modelo = modelo;
        this.anio = anio;
    }

    // Método abstracto que debe ser implementado por las subclases
    public abstract void mover();

    // Método que puede ser sobrescrito
    public String obtenerDetalles() {
        return "Marca: " + marca + ", Modelo: " + modelo + ", Año: " + anio;
    }
}

// Clase que representa un Automóvil, hereda de Vehiculo e implementa Combustible
class Automovil extends Vehiculo implements Combustible {
    private double nivelCombustible;

    public Automovil(String marca, String modelo, int anio, double nivelCombustible) {
        super(marca, modelo, anio);
        this.nivelCombustible = nivelCombustible;
    }

    @Override
    public void mover() {
        if (nivelCombustible > 0) {
            System.out.println("El automóvil " + marca + " " + modelo + " está conduciendo.");
            nivelCombustible -= 0.5;
```

```
        } else {
            System.out.println("El automóvil " + marca + " " + modelo + " no tiene combustible.");
        }
    }

    @Override
    public String obtenerDetalles() {
        return super.obtenerDetalles() + ", Combustible: " + nivelCombustible + " litros";
    }

    @Override
    public void recargarCombustible() {
        nivelCombustible = 50.0;
        System.out.println("El automóvil " + marca + " " + modelo + " ha sido recargado con combustible.");
    }

    @Override
    public double obtenerNivelCombustible() {
        return nivelCombustible;
    }
}

// Clase que representa una Bicicleta, hereda de Vehiculo
class Bicicleta extends Vehiculo {
    private int numeroMarchas;

    public Bicicleta(String marca, String modelo, int anio, int numeroMarchas) {
        super(marca, modelo, anio);
        this.numeroMarchas = numeroMarchas;
```

```
    }

    @Override
    public void mover() {
        System.out.println("La bicicleta " + marca + " " + modelo + " está pedaleando con " +
numeroMarchas + " marchas.");
    }

    @Override
    public String obtenerDetalles() {
        return super.obtenerDetalles() + ", Marchas: " + numeroMarchas;
    }
}

// Clase principal para probar el sistema
public class SistemaVehiculos {
    public static void main(String[] args) {
        // Crear instancias de vehículos
        Vehiculo auto = new Automovil("Toyota", "Corolla", 2020, 40.0);
        Vehiculo bici = new Bicicleta("Trek", "Mountain", 2022, 21);

        // Probar polimorfismo
        auto.mover();
        System.out.println(auto.obtenerDetalles());
        ((Combustible) auto).recargarCombustible();
        System.out.println("Nivel de combustible: " + ((Combustible) auto).obtenerNivelCombustible());

        bici.mover();
        System.out.println(bici.obtenerDetalles());
```

Trabajo a realizar

1. Análisis del Código Base

- **Herencia en `Automóvil` y `Bicicleta`:**
  Ambas clases heredan de la clase abstracta `Vehículo`, lo que les permite reutilizar los atributos comunes (`marca`, `modelo`, `año`) y el método `obtener Detalles ()`. Así se evita repetir código, cumpliendo con el principio de reutilización.
- **Polimorfismo en `mover ()` y `obtener Detalles ()`:**
  Ambos métodos están definidos de forma diferente en cada subclase. Al declarar las variables como `Vehículo`, pero instanciarlas como `Automóvil` o `Bicicleta`, se usa polimorfismo para ejecutar el comportamiento específico de cada clase, sin cambiar el tipo base.
- **Uso de `super`:**
  La palabra clave `super` se usa para llamar al constructor de la clase padre (`Vehículo`) desde las subclases. También se utiliza en `obtener Detalles ()` para acceder al método de la clase base y agregar más información sin sobrescribir todo el contenido.
- **Rol de la interfaz `Combustible`:**
  Define un contrato para vehículos que usan combustible (métodos `recargarCombustible()` y `obtenerNivelCombustible()`). Solo `Automovil` la implementa porque las bicicletas no usan combustible. Así se evita agregar métodos innecesarios en clases que no los requieren.

2. Extensión del Sistema

Como parte de la extensión del sistema, se agregó una nueva clase que representa un tipo de vehículo adicional. Esta nueva clase incorpora un atributo extra para indicar la potencia del motor en centímetros cúbicos, lo que permite diferenciarla de otros vehículos. También se personalizó su comportamiento al momento de moverse, mostrando un mensaje específico que incluye dicha potencia, y reduciendo el nivel de combustible cada vez que se activa esta acción.

Además, se adaptó el método para mostrar los detalles del vehículo, reutilizando la información general ya existente y agregando los datos propios de esta clase. Esta nueva clase también implementa funciones relacionadas con el manejo del combustible, como consultar el nivel actual y recargarlo por completo, estableciendo un límite de capacidad.

Finalmente, se probó la integración de esta clase dentro del sistema general, confirmando que se puede usar junto a los demás tipos de vehículos sin necesidad de modificar la estructura principal. Esto demuestra que el sistema es flexible y permite incorporar nuevas funcionalidades respetando los principios de diseño limpio y extensible.

Este cambio demuestra que el sistema permite extenderse fácilmente con nuevos tipos de vehículos, cumpliendo el principio de **abierto a extensión, cerrado a modificación**.

3. Implementación de una Nueva Interfaz

Se creó una nueva interfaz llamada `Mantenimiento` con el propósito de definir una acción común para vehículos que requieren mantenimiento regular. Esta interfaz contiene un único método: **`realizarMantenimiento().`**

Las clases **`Automovil`** **y** **`Motocicleta`** fueron modificadas para implementar esta interfaz, ya que son vehículos que, por su naturaleza mecánica, requieren revisiones periódicas. Cada clase implementa el método con un mensaje específico: en el caso del automóvil, se indica que se está revisando el motor, mientras que en la motocicleta se menciona el cambio de aceite.

En el método **`Main`** del sistema, se incluyeron validaciones usando **`instanceof`** para identificar qué vehículos implementan la interfaz `Mantenimiento`, y luego se invocó el método **`realizarMantenimiento()`** de forma polimórfica. Esto refuerza la flexibilidad del sistema al permitir extender comportamientos comunes sin depender del tipo exacto de cada vehículo.

Esta implementación demuestra el uso adecuado de interfaces para agregar funcionalidades específicas sin alterar la estructura general del sistema.

4. Polimorfismo avanzado

Se creó el método **`procesarVehiculos()`** que recibe un arreglo de objetos tipo **`Vehiculo.`** Usando polimorfismo, se llaman los métodos **`mover()`** y **`obtenerDetalles()`** sin importar si es un automóvil, bicicleta o motocicleta.
Si el vehículo implementa la interfaz **`Combustible,`** también se invocan los métodos **`obtenerNivelCombustible()`** y **`recargarCombustible().`**
El método fue probado desde **`main`** con un arreglo que incluye los tres tipos de vehículos, confirmando que el sistema responde correctamente a cada comportamiento específico.

5. Reflexión escrita

Durante el desarrollo de este proyecto en grupo, aplicamos varios conceptos de la programación orientada a objetos, pero uno de los más útiles fue el polimorfismo. Esta característica nos permitió trabajar con diferentes tipos de objetos de manera general, usando métodos en común que cada clase podía personalizar según sus necesidades. Gracias a eso, no tuvimos que hacer muchas modificaciones al código base al momento de agregar nuevas clases o funcionalidades, lo cual fue clave para mantener el sistema limpio y ordenado.

El polimorfismo también nos permitió repartir mejor las tareas del equipo. Cada integrante pudo trabajar en una clase diferente sin preocuparse por cómo encajaría con el resto del programa, ya que todos seguíamos una estructura común definida por la clase base y las interfaces. Esto facilitó la colaboración y evitó conflictos en el código. Por ejemplo, al tener un método como `mover()`, cada clase lo implementaba a su manera, pero desde el programa principal se podía llamar a ese método sin importar el tipo de objeto.

Además, implementamos interfaces para definir comportamientos específicos, como el mantenimiento o el uso de combustible, y eso también fue posible gracias al polimorfismo. Pudimos identificar en tiempo de ejecución si un objeto tenía ciertas capacidades, y aplicar métodos adicionales solo cuando correspondía. Todo esto nos ayudó a evitar escribir código repetido o innecesariamente complejo.

En conclusión, el polimorfismo no solo nos facilitó el diseño del sistema, sino que también hizo más eficiente el trabajo en equipo. Nos permitió crear un programa flexible, fácil de extender, y que puede seguir creciendo en el futuro sin tener que rehacer lo que ya funciona. Este concepto fue clave para lograr una solución bien estructurada y profesional dentro del contexto de nuestra clase de programación.