### **Integrantes: Joseph Daniel Tello Laiza**

**Jean Pier Condor Machacuay**

### **Parte 1: Lectura y Análisis (3 puntos)**

* + **¿Cuál es el problema principal al usar solo POO tradicional en SimUDuck app?**

El principal problema es que al usar solo POO tradicional, hace que los cambios y mantenimientos en los comportamientos, como el de volar o graznar, afectan a múltiples clases, esto provoca que haya una alta dependencia entre las clases y que se dificulte la reutilización del código, ya que si se modifica algún comportamiento, hay que modificar todas las clases que implementan ese comportamiento.

* + **¿Por qué la herencia (ej. subclases como MallardDuck) es inflexible para agregar comportamientos como fly()?**

Se obliga a todas las subclases a heredar un comportamiento o atributos que no le deberían pertenecer, obligando a agregar aclaraciones o código redundante como la anulación de métodos en el ejemplo.

Además es difícil cambiar luego la naturaleza de ciertos comportamientos como quack() o fly() ya que toda subclase heredará el método exactamente como está indicado en la superclase.

* + **¿Qué desventajas tiene usar interfaces (ej. Flyable, Quackable) para cada comportamiento?**

Comportamientos como fly() o quack() se vuelven estáticos y únicos para todos los tipos de patos que lo implementen:

* A un pato no se le debe asignar el método fly() con características ligeramente diferentes, pues solo existe una tipo de fly() para todos los patos que quieran implementarlo.
* Si queremos crear otro tipo de método quack() se debe crear como una interfaz completamente nueva, y volver a hacerlo con otros tipos de pato que deseen implementar su propia caracterización de quack().

En conclusión, el código se hace difícil de mantener si se piensan añadir nuevas funcionalidades a largo plazo, además este amontonamiento de interfaces puede hacer confusa la interpretación para otros programadores.

### **Parte 2: Diseño con Strategy (10 puntos)**

**Contexto:** En el juego, algunos patos vuelan (**fly()**), otros no, algunos graznan (**quack()**), otros hacen **squeak**, e incluso algunos no hacen sonido.

**Tarea:**

**1. Identifica los comportamientos variables** (ej. **fly, quack**) y sepáralos en **interfaces** (estrategias).

* + Ejemplo:

public interface FlyBehavior {

void fly();

}

**2. Implementa clases concretas** para cada comportamiento:

* + FlyWithWings, FlyNoWay, Quack, Squeak, MuteQuack, etc.

**3. Modifica la clase Duck** para que **delegue** estos comportamientos (usa composición en lugar de herencia).

* + - La clase base Duck debe tener:

public abstract class Duck {

FlyBehavior flyBehavior;

QuackBehavior quackBehavior;

public void performFly() {

flyBehavior.fly();

}

public void setFlyBehavior(FlyBehavior fb) {

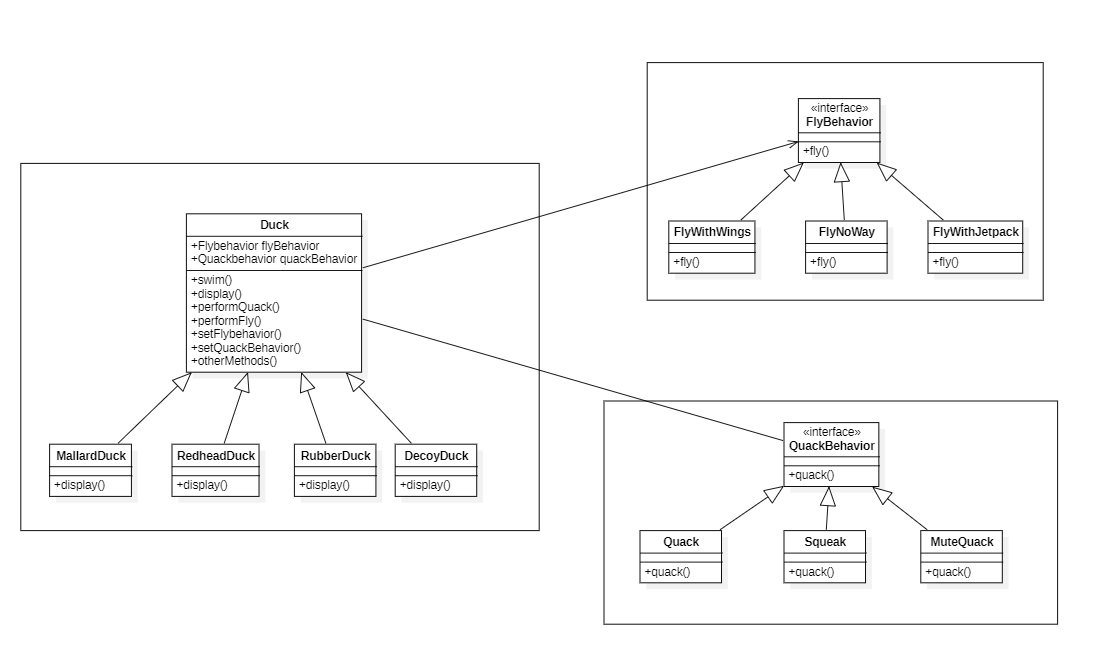
flyBehavior = fb;

}

// ... (similar para quack)

}

**4. Crea 3 tipos de patos** (ej. **MallardDuck, RubberDuck, DecoyDuck**) con comportamientos distintos.

**5. Diagrama de clases** de la solución final.

### **Parte 3: Preguntas de Reflexión (3 puntos)**

1. **¿Cómo soluciona el patrón Strategy los problemas de la herencia?**

Se pueden establecer distintos tipos de un mismo comportamiento que luego podrán usar las distintas clases, lo cual hace más reusable el código en lugar de situaciones como en la del problema en la que debían establecer como funcionaría un mismo comportamiento en cada clase Duck.

1. **¿Qué ventaja tiene componer objetos (usar estrategias) sobre extender clases?**

Se pueden cambiar los métodos en tiempo de ejecución mediantes los Setters, si se quiere cambiar la naturaleza de un comportamiento solo se debe de agregar un nuevo Setter en lugar de tener que modificar todo el código dentro de una clase.

1. **Si mañana agregamos un nuevo comportamiento (ej. dive()), ¿qué cambios mínimos se necesitarían?**

Además de agregar las interfaces necesarias del nuevo comportamiento dive() y sus distintas clases como diveDeep() o diveSlightly(), no será necesario modificar comportamientos anteriores como fly() o quack(). Solo será necesario agregar una propiedad DiveBehavior a la clase Duck. Si luego queremos modificar la naturaleza de este comportamiento podremos usar los Setters como en los casos anteriores,

### **Parte 4: Cambio Dinámico de Comportamiento (4 puntos)**

* Modificar el sistema para que un pato pueda **alterar sus estrategias** (ej. FlyBehavior, QuackBehavior) durante la ejecución del programa.

**Instrucciones:**

#### **1. Agregar métodos de configuración en la clase Duck:** Ya deberías tener estos métodos si seguiste el diseño inicial del patrón Strategy, pero aquí los reforzamos:

public abstract class Duck {

FlyBehavior flyBehavior;

QuackBehavior quackBehavior;

// Métodos para cambiar comportamientos en tiempo de ejecución

public void setFlyBehavior(FlyBehavior fb) {

this.flyBehavior = fb;

}

public void setQuackBehavior(QuackBehavior qb) {

this.quackBehavior = qb;

}

public void performFly() {

flyBehavior.fly(); // Delega al comportamiento actual

}

public void performQuack() {

quackBehavior.quack();

}

}

#### **2. Crear nuevas estrategias (ej. Jetpack):** Añade estrategias adicionales para simular cambios dinámicos

public class FlyWithJetpack implements FlyBehavior {

@Override

public void fly() {

System.out.println("¡Volando con un jetpack! 🚀");

}

}

#### **3. Ejemplo de uso:** Patos y sus formas de vuelo

public class Main {

public static void main(String[] args) {

Duck mallard = new MallardDuck();

mallard.performFly(); // Vuela normalmente (FlyWithWings)

// ¡El pato consigue un jetpack!

mallard.setFlyBehavior(new FlyWithJetpack());

mallard.performFly(); // Ahora vuela con jetpack

// Cambio a "no volar" (por ejemplo, se agota el combustible)

mallard.setFlyBehavior(new FlyNoWay());

mallard.performFly(); // No vuela

}

}

**Responde:**

* + **¿Por qué el patrón Strategy permite este tipo de flexibilidad?**

Porque al crear distintas clases dentro de cada comportamiento permite combinar distintos comportamientos como FlyWithJetpack y MuteQuack o FlyWithJetpack y Squeak.

El hecho de que se puedan cambiar los comportamientos en tiempo de ejecución también permite una gran flexibilidad.

Además esta clasificación hace más fácil una posible modificación al código.

* + **¿Cómo harías esto con herencia tradicional? ¿Sería posible?**

Sería posible hacerlo con métodos como la creación de interfaces o superclases como el caso del ejemplo, pero su mantenimiento y entendimiento sería demasiado complicado y difícil de entender.

Github: <https://github.com/Josephtellolaiza/StrategyPOO2-Semana9.git>