Responde:

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

* Si se usa POO tradicional se tendrían que realizar cambios cada vez que se quiera agregar un nuevo pato, ya que no todos los patos con los que se quiere trabajar vuelan o graznan.

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

* Porque si se aplicara herencia, todas las subclases de ‘pato’ heredarían comportamientos que tal vez no sean adecuados dependiendo del pato. no sólo alteran el orden, generando un alto acoplamiento sino que rompen el principio Liskov de SOLID.

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

* Al usar interfaces para cada comportamiento se presentan problemas de mantenibilidad. Se tendrían que realizar cambios a flyable y quackable en todas las subclases porque las interfaces no permiten la reutilización de código. Esto forzaría al desarrollador a romper el principio Open/Closed de SOLID y el sistema, además, se volvería bastante extenso.

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

* Se crea una interfaz que mostrará el comportamiento, y clases que implementan los tipos de comportamiento, indicando si es que dicho comportamiento se cumple, no se cumple, o tiene algún otro tipo de respuesta. De esta forma los comportamientos (sean los que sean) no están ligados a las propias subclases de ‘pato’.

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

* Flexibilidad
* Menor acoplamiento
* Reutilizacion de código(comportamientos necesarios)

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

Se tendría que agregar una interfaz para el comportamiento dive (‘DiveBehavior’) y agregar implementaciones que indican si el pato puede o no bucear. (‘Dive’/ ‘CannotDive’/Floats).

