Ejercicio 2

Las modificaciones que se deberían de hacer son crear una interfaz y tener como firma “calcularDescuento()” ya que es lo que varia entre la clase Amateur y Profesional para poder en tiempo de ejecución cambiar de uno a otro, ya que ahora cuando creo una clase concreta de montañista amateur o profesional luego no lo puedo modificar, por ende la clase Montañita se le sumaría otra v.i. de tipo de montañista del tipo de la interfaz que mencione y dejaría de ser una clase abstracta. Luego agregar un setter para poder modificar en tiempo de ejecución el tipo y a su vez crear la Clase empresa que se encargue de que periódicamente chequee si en los últimos 3 meses los montañistas hicieron pedidos para poder cambiarlos de categoría.

Ejercicio 3

Heurísticas de asignación de responsabilidades (GRASP)

1. **Information Expert (Experto en la información)**
    - **Montañista** es quien conoce su historial de pedidos, por lo tanto es quien calcula el promedio de los últimos 3 pedidos.
    - **Pedido** es responsable de calcular su propio costo (sin delegarlo a otro), ya que conoce los ítems y su precio.
2. **Creator**
    - La clase **Montañista** es quien crea los objetos `Pedido`, ya que:
        - Usa intensamente los pedidos.
        - Contiene la relación con el historial.
        - Tiene sentido que sea quien lo cree si va a mantener su historial.
3. **Polymorphism (Polimorfismo)**
    - Se usa para modelar el comportamiento distinto entre **Amateur** y **Profesional**, con un método como `descuento()` o `calcularCosto(Pedido)` que es redefinido por cada subclase.
4. **Low Coupling (Bajo acoplamiento)**
    - Las clases están diseñadas para depender lo menos posible unas de otras:
        - Un `Pedido` no necesita saber qué tipo de montañista lo hizo.

        - El cálculo del costo depende del contexto del montañista, no del pedido en sí.

5. **High Cohesion (Alta cohesión)**
    - Cada clase tiene responsabilidades claras:
        - `Montañista`: gestionar su historial y aplicar descuentos.
        - `Pedido`: representar el pedido concreto.
        - `ItemPedido`: representar un ítem individual.
6. **Pure Fabrication (Falsificación pura)**
    - Si se introdujo una clase como `Empresa` para manejar la relación entre montañistas y pedidos, puede verse como una "fabricación pura" para desacoplar responsabilidades que no encajaban naturalmente en otras clases.

Principios SOLID

1. **S – Single Responsibility Principle (SRP)**
    - Cada clase tiene **una única razón para cambiar**:
        - `Montañista`: lógica de descuento y gestión de pedidos.
        - `Pedido`: representa un pedido histórico, cerrado, con datos fijos.
        - `ItemPedido`: solo representa una unidad de pedido.
2. **O – Open/Closed Principle (OCP)**
    - El sistema está **abierto a extensión y cerrado a modificación**:
        - Se puede agregar nuevas categorías de montañistas (Ej: "Elite") sin tocar el código de `Pedido` o `Montañista`.
3. **L – Liskov Substitution Principle (LSP)**
    - `Amateur` y `Profesional` pueden ser utilizados **en lugar de `Montañista` sin romper el sistema**.
      Ejemplo: un método que recibe `Montañista` no necesita saber si es Amateur o Profesional.