**Nombre: Santiago Acosta Calvo**

**Seguimiento 3**

**Diseño con Interfaces y Clases Abstractas**

# 1. Criterios para decidir entre una interfaz y una clase abstracta

## Caso 1: Ecosistema de Organismos

Clase abstracta: Organismo

Razón: Se utiliza una clase abstracta porque todos los organismos comparten atributos y métodos comunes como `isAlive()` y `realizarAccion()`. Además, permite compartir código y definir comportamientos genéricos, dejando que cada subclase implemente acciones específicas.

## Caso 2: Gestión de Dispositivos Electrónicos

Clase abstracta: DispositivoBase

Interfaz: Recargable

Razón: La clase abstracta agrupa atributos y métodos comunes como encendido/apagado. La interfaz `Recargable` añade funcionalidades específicas, permitiendo que solo algunos dispositivos implementen la capacidad de recarga.

## Caso 3: Plataforma de Pagos

Clase abstracta: Procesador

Razón: La clase abstracta encapsula la lógica compartida entre métodos de pago, mientras que métodos como `realizarPago()` se definen de forma abstracta para que cada subclase los implemente de manera específica.

## Caso 4: Sistema Académico

Clase abstracta: Participante

Razón: Todos los participantes (estudiantes, profesores, administrativos) comparten atributos como `ID` y `nombre`, y requieren implementar el método `evaluarDesempeno()`. La clase abstracta garantiza la estructura común y obliga a su implementación.

# 2. Impacto del polimorfismo en la flexibilidad de la solución

Manejo genérico de objetos: En el Caso 1, una lista de `Organismo` puede contener distintos tipos (plantas, herbívoros, carnívoros). Al llamar a `realizarAccion()`, se ejecuta la versión correspondiente sin necesidad de conocer el tipo exacto.

Extensibilidad: En el Caso 4, se puede agregar una nueva clase (por ejemplo, `Tutor`) heredando de `Participante` e implementando `evaluarDesempeno()` sin modificar el código existente.

Interfaz consistente: En el Caso 2, la interfaz `Recargable` estandariza el comportamiento de carga entre distintos dispositivos, sin afectar su jerarquía de herencia.

Evitar código duplicado: En el Caso 3, `Procesador` permite compartir lógica común como validaciones o logs, evitando replicar ese código en cada tipo de pago.

# 3. Desafíos al combinar interfaces y clases abstractas

Herencia múltiple implícita:  
 - Problema: En el Caso 2, los dispositivos necesitaban heredar de `DispositivoBase` y ser recargables.  
 - Solución: Crear una clase abstracta `DispositivoRecargable` que herede de `DispositivoBase` e implemente `Recargable`.

Conflictos entre métodos similares:  
 - Problema: En el Caso 3, un método en `Procesador` podía entrar en conflicto con métodos definidos en interfaces.  
 - Solución: Priorizar la lógica de las clases abstractas para evitar duplicidades y mantener coherencia.

Validaciones comunes:  
 - Problema: En el Caso 4, era necesario validar datos como calificaciones o salarios.  
 - Solución: Centralizar atributos comunes en `Participante` e implementar validaciones en métodos de subclases como `agregarCalificacion()`.

Organización de responsabilidades:  
 - Problema: En el Caso 1, decidir si acciones como “cazar” debían estar en una interfaz o clase abstracta.  
 - Solución: Usar la clase abstracta para lógica común (por ejemplo, gestión de energía) y métodos abstractos para acciones específicas.