Ejercicio 1: Validador de entradas de usuario

Contexto:

Queremos asegurarnos de que un sistema solo acepte nombres válidos de usuario (sin números, ni símbolos, ni espacios).

Ciclo TDD:

- **RED:** Se escribe una prueba que espera que el sistema rechace nombres inválidos como “123Ana” o “Juan Pérez”.
  La prueba falla porque la validación aún no existe.

- **GREEN:** Se implementa una comprobación básica que solo acepta letras.
  Las pruebas ahora pasan.

- **REFACTOR:** Se mejora la función para permitir letras con acentos y limpiar el código repetido.

Ejercicio 2: Calculadora de precios con descuentos

Contexto:

Queremos calcular el precio final de un producto aplicando un descuento.

Ciclo TDD:

- **RED:** Se escribe una prueba que espera que un descuento del 10% sobre $100 dé $90.
  Falla porque la función no existe.

- **GREEN:** Se crea una función mínima que resta el 10%.
  Las pruebas pasan.

- **REFACTOR:** Se mejora el código para validar que el descuento no sea negativo ni superior al 100%, y se redondea el resultado.

Ejercicio 3: Sistema de autenticación simple

Contexto:

Queremos permitir que un usuario se registre y luego inicie sesión.

Ciclo TDD:

- **RED:** Se escribe una prueba que espera que un usuario pueda iniciar sesión solo si las credenciales coinciden.
  Falla porque la lógica de autenticación no existe.

- **GREEN:** Se implementa lo mínimo: guardar el usuario y comparar contraseñas.
  La prueba pasa.

- **REFACTOR:** Se mejora la seguridad (por ejemplo, usando hashes en lugar de contraseñas en texto plano) y se reorganiza el código.

Ejercicio 4: Cálculo de impuestos por tramos

Contexto:

El sistema debe calcular impuestos según niveles de ingreso (por ejemplo, sin impuesto hasta $10,000, luego 10% hasta $20,000, etc.).

Ciclo TDD:

- **RED:** Se escribe una prueba que define los resultados esperados para distintos ingresos.
  Falla porque la función aún no existe.

- **GREEN:** Se implementa una versión con condicionales básicas que pasa las pruebas.

- **REFACTOR:** Se abstrae la lógica de tramos a una estructura de datos configurable para que sea más fácil de mantener.

Ejercicio 5: Contador de palabras

Contexto:

Queremos analizar un texto y contar cuántas veces aparece cada palabra.

Ciclo TDD:

- **RED:** Se define una prueba que espera que “hola mundo hola” devuelva `{"hola": 2, "mundo": 1}`.
  Falla porque la función no está implementada.

- **GREEN:** Se agrega una lógica simple que separa las palabras y las cuenta.
  La prueba pasa.

- **REFACTOR:** Se mejora para ignorar mayúsculas/minúsculas, eliminar signos de puntuación y hacer el código más legible.

Ejercicio 6: Envios

Contexto:
Una aplicación maneja distintos métodos de envío: `EnvioExpress`, `EnvioNormal`, y `EnvioInternacional`.
El sistema actual usa un `if` gigante para calcular el costo del envío según el tipo.

Preguntas:
a) ¿Qué principio aplicarías para eliminar esos `if`?

Principio esperado:

- **Polimorfismo (GRASP)** o **Open/Closed (SOLID)**.

Por qué:

- Cada tipo de envío conoce su propia lógica de cálculo, no un método externo.
- El código que calcula el envío no cambia aunque se agregue un nuevo tipo.
- El sistema queda “abierto a extensión, cerrado a modificación”.
- Se mejora la cohesión (cada clase se encarga de una sola cosa) y se elimina la dependencia de condicionales.

Ejercicio 7: Generador de reportes

Contexto:
Un módulo genera reportes en diferentes formatos: `PDF`, `Excel`, `CSV`.
El método actual tiene un `switch(formato)` que llama a distintos bloques de código para cada tipo.

Preguntas:
a) ¿Qué principio aplicarías para mejorar este diseño?

Principio esperado:

- **Open/Closed Principle (SOLID)**.

Por qué:

- Permite agregar nuevos formatos de reporte sin alterar el código existente.
- Cada clase de formato implementa el mismo contrato (`generar()`), por lo que el cliente no necesita saber qué tipo está usando.

- Se evita romper código viejo cuando se extiende el sistema.
- También se refuerza el **Principio de Sustitución de Liskov (SOLID)**: cualquier tipo de reporte puede usarse en lugar de otro sin afectar el comportamiento esperado.

Ejercicio 8: Sistema de facturación

Contexto:
La clase `Factura` calcula el total dependiendo del tipo de cliente (`Minorista`, `Mayorista`, `Exento`).
Hoy en día se hace con varios `if`.

Preguntas:
a) ¿Qué principio usarías para eliminar los `if` y separar responsabilidades?

Principio esperado:

- **Polimorfismo (GRASP)** o **Strategy (Patrón de diseño)**.

Por qué:

- `Factura` tiene más de una responsabilidad (gestiona datos y reglas de negocio).
- Aplicando SRP, cada estrategia de cálculo se aísla en su propia clase.
- Esto permite mantener y probar cada tipo de facturación por separado.
- El uso del patrón **Strategy** elimina el `if` al delegar el cálculo a un objeto especializado.