Resoluciones Guías P1

Ejercicio: Identificación de Patrones

Lee el siguiente código y responde:

```
class DatabaseConnection {
    private static instance: DatabaseConnection;
    private constructor() {}

    static getInstance(): DatabaseConnection {
        if (!this.instance) {
            this.instance = new DatabaseConnection();
        }
        return this.instance;
    }
}
```

Preguntas:

1. ¿Qué patrón de diseño se está implementando?
2. ¿Cuál es el propósito de hacer el constructor privado?
3. ¿En qué situaciones del mundo real usarías este patrón?

Solución

1. ¿Qué patrón de diseño se está implementando?

El patrón de diseño implementado es el **Singleton**.

- El **Singleton** asegura que una clase tenga **una única instancia** en toda la aplicación y proporciona un punto de acceso global a esa instancia.
- En este caso, la instancia única está almacenada en la propiedad estática `instance` de la clase.

2. ¿Cuál es el propósito de hacer el constructor privado?

El propósito de hacer el constructor privado es **evitar que la clase sea instanciada directamente desde fuera**.

- Solo se puede obtener una instancia de la clase a través del método estático `getInstance()`.
- Esto garantiza que no se puedan crear múltiples instancias de la clase, cumpliendo con el principio del **Singleton**.

3. ¿En qué situaciones del mundo real usarías este patrón?

El patrón Singleton es útil en situaciones donde se necesita una única instancia compartida de una clase. Algunos ejemplos del mundo real incluyen:

1. **Gestión de conexión a base de datos**:
   En aplicaciones donde múltiples partes del sistema necesitan conectarse a una base de datos, el Singleton asegura que haya **una sola conexión activa**.
2. **Gestión de configuración global**:
   Una clase que almacene las configuraciones de la aplicación (por ejemplo, parámetros de inicio o configuraciones de usuario) puede ser Singleton para garantizar acceso centralizado y consistente.
3. **Logger**:
   Un sistema de logging que centralice la escritura de registros en un archivo o consola puede ser Singleton, evitando múltiples instancias y problemas de concurrencia.
4. **Control de recursos compartidos**:
   Por ejemplo, un servicio que interactúe con una impresora o maneje acceso a un archivo compartido.

Ejercicio: Sistema de Notificaciones con Strategy

Diseña un sistema de notificaciones para una aplicación que puede enviar mensajes por:

- Email
- SMS
- Push notifications
- Slack

Requisitos:

1. Usa el patrón Strategy funcional (sin clases)
2. Cada estrategia debe recibir un destinatario y un mensaje
3. Implementa una función `enviarNotificacion` que acepte la estrategia como parámetro
4. Simula el envío imprimiendo en consola
5. Crea un ejemplo donde se envíe la misma notificación por diferentes canales

Bonus: Implementa un "NotificationManager" que pueda enviar por múltiples canales simultáneamente.

Soluciones

Con Strategy Funcional

```
// Tipos
type Destinatario = {
    nombre: string;
```

```
    email?: string;
    telefono?: string;
    deviceId?: string;
    slackId?: string;
};

type Mensaje = {
    asunto: string;
    cuerpo: string;
};

type EstrategiaNotificacion = (destinatario: Destinatario, mensaje: Mensaje) =>

// Estrategias como funciones puras
const enviarPorEmail: EstrategiaNotificacion = (destinatario, mensaje) => {
    console.log(`[EMAIL] Enviando a: ${destinatario.email}`);
    console.log(`   Asunto: ${mensaje.asunto}`);
    console.log(`   Mensaje: ${mensaje.cuerpo}\n`);
};

const enviarPorSMS: EstrategiaNotificacion = (destinatario, mensaje) => {
    console.log(`[SMS] Enviando a: ${destinatario.telefono}`);
    console.log(`   ${mensaje.cuerpo}\n`);
};

const enviarPorPush: EstrategiaNotificacion = (destinatario, mensaje) => {
    console.log(`[PUSH] Enviando a device: ${destinatario.deviceId}`);
    console.log(`   ${mensaje.asunto}: ${mensaje.cuerpo}\n`);
};

const enviarPorSlack: EstrategiaNotificacion = (destinatario, mensaje) => {
    console.log(`[SLACK] Enviando a: @${destinatario.slackId}`);
    console.log(`   **${mensaje.asunto}**`);
    console.log(`   ${mensaje.cuerpo}\n`);
};

// Función principal que acepta la estrategia
const enviarNotificacion = (
    estrategia: EstrategiaNotificacion,
    destinatario: Destinatario,
    mensaje: Mensaje
): void => {
    estrategia(destinatario, mensaje);
};

// BONUS: NotificationManager
const enviarPorMultiplesCanales = (
    estrategias: EstrategiaNotificacion[],
    destinatario: Destinatario,
    mensaje: Mensaje
```

```
): void => {
    estrategias.forEach(estrategia => estrategia(destinatario, mensaje));
};

// Uso
const usuario: Destinatario = {
    nombre: "John Doe",
    email: "John@example.com",
    telefono: "+584121234123",
    deviceId: "device-abc123",
    slackId: "JohnDoe1000"
};

const notificacion: Mensaje = {
    asunto: "Bienvenido al curso",
    cuerpo: "Tu clase comienza mañana a las 10:00 AM"
};

// Envío por canal individual
console.log("Envio Individual \n");
enviarNotificacion(enviarPorEmail, usuario, notificacion);
enviarNotificacion(enviarPorSMS, usuario, notificacion);
enviarNotificacion(enviarPorPush, usuario, notificacion);
enviarNotificacion(enviarPorSlack, usuario, notificacion);

// Envío por múltiples canales simultáneamente
console.log("Envio Multiple \n");
enviarPorMultiplesCanales(
    [enviarPorEmail, enviarPorSMS, enviarPorPush],
    usuario,
    notificacion
);
```

Con Strategy POO

```
// Tipos
type Destinatario = {
    nombre: string;
    email?: string;
    telefono?: string;
    deviceId?: string;
    slackId?: string;
};

type Mensaje = {
    asunto: string;
    cuerpo: string;
};
```

```
// Interfaz Strategy
interface INotificationStrategy {
    enviar(destinatario: Destinatario, mensaje: Mensaje): void;
}

// Estrategias concretas
class EmailStrategy implements INotificationStrategy {
    enviar(destinatario: Destinatario, mensaje: Mensaje): void {
        console.log(`[EMAIL] Enviando a: ${destinatario.email}`);
        console.log(`   Asunto: ${mensaje.asunto}`);
        console.log(`   Mensaje: ${mensaje.cuerpo}\n`);
    }
}

class SMSStrategy implements INotificationStrategy {
    enviar(destinatario: Destinatario, mensaje: Mensaje): void {
        console.log(`[SMS] Enviando a: ${destinatario.telefono}`);
        console.log(`   ${mensaje.cuerpo}\n`);
    }
}

class PushStrategy implements INotificationStrategy {
    enviar(destinatario: Destinatario, mensaje: Mensaje): void {
        console.log(`[PUSH] Enviando a device: ${destinatario.deviceId}`);
        console.log(`   ${mensaje.asunto}: ${mensaje.cuerpo}\n`);
    }
}

class SlackStrategy implements INotificationStrategy {
    enviar(destinatario: Destinatario, mensaje: Mensaje): void {
        console.log(`[SLACK] Enviando a: @${destinatario.slackId}`);
        console.log(`   **${mensaje.asunto}**`);
        console.log(`   ${mensaje.cuerpo}\n`);
    }
}

// Contexto
class NotificationContext {
    constructor(private strategy: INotificationStrategy) {}

    setStrategy(strategy: INotificationStrategy): void {
        this.strategy = strategy;
    }

    enviarNotificacion(destinatario: Destinatario, mensaje: Mensaje): void {
        this.strategy.enviar(destinatario, mensaje);
    }
}

// BONUS: NotificationManager
```

```
class NotificationManager {
    private strategies: INotificationStrategy[] = [];

    agregarCanal(strategy: INotificationStrategy): this {
        this.strategies.push(strategy);
        return this; // Fluent interface
    }

    enviarPorTodosLosCanales(destinatario: Destinatario, mensaje: Mensaje): void
        this.strategies.forEach(strategy => strategy.enviar(destinatario, mensaj
    }
}

// Uso
const usuario: Destinatario = {
    nombre: "John Doe",
    email: "John@example.com",
    telefono: "+584121234123",
    deviceId: "device-abc123",
    slackId: "JohnDoe1000"
};

const notificacion: Mensaje = {
    asunto: "Bienvenido al curso",
    cuerpo: "Tu clase comienza mañana a las 10:00 AM"
};

// Envío individual cambiando estrategias
console.log("Envio Individual \n");
const notifier = new NotificationContext(new EmailStrategy());
notifier.enviarNotificacion(usuario, notificacion);

notifier.setStrategy(new SMSStrategy());
notifier.enviarNotificacion(usuario, notificacion);

notifier.setStrategy(new PushStrategy());
notifier.enviarNotificacion(usuario, notificacion);

notifier.setStrategy(new SlackStrategy());
notifier.enviarNotificacion(usuario, notificacion);

// Envío múltiple con NotificationManager
console.log("Envio Multiple \n");
const manager = new NotificationManager();
manager
    .agregarCanal(new EmailStrategy())
    .agregarCanal(new SMSStrategy())
    .agregarCanal(new PushStrategy());
```

```
manager.enviarPorTodosLosCanales(usuario, notificacion);
```

Comparación de uso

Strategy Funcional

Ventajas:

- Menos boilerplate
- Más fácil de testear (funciones puras)
- Composición más flexible

Usar cuando:

- Las estrategias son simples y stateless
- Prefieres programación funcional
- Necesitas mayor flexibilidad para combinar estrategias

Strategy POO

Ventajas:

- Mejor para estrategias con estado interno
- Encapsulación clara de comportamiento

Usar cuando:

- Las estrategias necesitan mantener estado
- Tu equipo está más cómodo con POO
- Necesitas herencia de comportamiento entre estrategias

Ejercicio: Refactorización con Decorator

El siguiente código calcula el precio de pizzas con ingredientes adicionales:

```
class Pizza {
    private base: number = 10;
    private queso: boolean = false;
    private pepperoni: boolean = false;
    private champinones: boolean = false;
    private extraQueso: boolean = false;

    agregarQueso() { this.queso = true; }
    agregarPepperoni() { this.pepperoni = true; }
    agregarChampinones() { this.champinones = true; }
```

```
    agregarExtraQueso() { this.extraQueso = true; }

    calcularPrecio(): number {
        let precio = this.base;
        if (this.queso) precio += 2;
        if (this.pepperoni) precio += 3;
        if (this.champinones) precio += 2.5;
        if (this.extraQueso) precio += 1.5;
        return precio;
    }

    getDescripcion(): string {
        let desc = "Pizza base";
        if (this.queso) desc += " + queso";
        if (this.pepperoni) desc += " + pepperoni";
        if (this.champinones) desc += " + champiñones";
        if (this.extraQueso) desc += " + extra queso";
        return desc;
    }
}
```

Tarea: Refactoriza usando el patrón Decorator para:

- Eliminar los booleanos
- Hacer el código más extensible (agregar nuevos ingredientes sin modificar código existente)
- Permitir agregar el mismo ingrediente múltiples veces
- Mantener la funcionalidad de `calcularPrecio()` y `getDescripcion()`

Solucion

```
// Interfaz base para todos los componentes (pizzas y decoradores)
interface ComponentePizza {
    calcularPrecio(): number;
    getDescripcion(): string;
}

// Componente concreto: Pizza base
class PizzaBase implements ComponentePizza {
    calcularPrecio(): number {
        return 10;
    }

    getDescripcion(): string {
        return "Pizza base";
    }
}

// Decorator abstracto
```

```
abstract class IngredienteDecorator implements ComponentePizza {
    constructor(protected pizza: ComponentePizza) {}

    abstract calcularPrecio(): number;
    abstract getDescripcion(): string;
}

// Decoradores concretos para cada ingrediente
class QuesoDecorator extends IngredienteDecorator {
    calcularPrecio(): number {
        return this.pizza.calcularPrecio() + 2;
    }

    getDescripcion(): string {
        return this.pizza.getDescripcion() + " + queso";
    }
}

class PepperoniDecorator extends IngredienteDecorator {
    calcularPrecio(): number {
        return this.pizza.calcularPrecio() + 3;
    }

    getDescripcion(): string {
        return this.pizza.getDescripcion() + " + pepperoni";
    }
}

class ChampinonesDecorator extends IngredienteDecorator {
    calcularPrecio(): number {
        return this.pizza.calcularPrecio() + 2.5;
    }

    getDescripcion(): string {
        return this.pizza.getDescripcion() + " + champiñones";
    }
}

class ExtraQuesoDecorator extends IngredienteDecorator {
    calcularPrecio(): number {
        return this.pizza.calcularPrecio() + 1.5;
    }

    getDescripcion(): string {
        return this.pizza.getDescripcion() + " + extra queso";
    }
}

// Uso
console.log("--Ejemplos--\n");
```

```
// Pizza simple
let pizza1 = new PizzaBase();
console.log(`${pizza1.getDescripcion()}`);
console.log(`Precio: $${pizza1.calcularPrecio()}\n`);

// Pizza con varios ingredientes
let pizza2: ComponentePizza = new PizzaBase();
pizza2 = new QuesoDecorator(pizza2);
pizza2 = new PepperoniDecorator(pizza2);
pizza2 = new ChampinonesDecorator(pizza2);
console.log(`${pizza2.getDescripcion()}`);
console.log(`Precio: $${pizza2.calcularPrecio()}\n`);

// Pizza con ingredientes repetidos (doble queso)
let pizza3: ComponentePizza = new PizzaBase();
pizza3 = new QuesoDecorator(pizza3);
pizza3 = new PepperoniDecorator(pizza3);
pizza3 = new QuesoDecorator(pizza3);
pizza3 = new ExtraQuesoDecorator(pizza3);
console.log(`${pizza3.getDescripcion()}`);
console.log(`Precio: $${pizza3.calcularPrecio()}\n`);
```

Operaciones Simples en Árbol Binario

Usando el tipo `Node`:

```
type Node = {value: number; left: Node | null; rigth: Node | null;}
```

Implemente las siguientes tres funciones independientes:

```
// 1. Retorna la suma de todos los valores del árbol
function sum(node: Node | null): number

// 2. Retorna el valor máximo del árbol
function max(node: Node | null): number

// 3. Retorna la cantidad total de nodos del árbol
function count(node: Node | null): number
```

Considere los casos borde (árbol vacío, un solo nodo, etc.)

Solución

```
type Node = {
  value: number;
```

```
  left: Node | null;
  right: Node | null;
}

// 1. Suma de todos los valores
function sum(node: Node | null): number {
  if (node === null) return 0;

  return node.value + sum(node.left) + sum(node.right);
}

// 2. Valor máximo del árbol
function max(node: Node | null): number {
  if (node === null) return -Infinity;

  const maxLeft = max(node.left);
  const maxRight = max(node.right);

  return Math.max(node.value, maxLeft, maxRight);
}

// 3. Cantidad total de nodos
function count(node: Node | null): number {
  if (node === null) return 0;

  return 1 + count(node.left) + count(node.right);
}

// Uso
const tree: Node = {
  value: 10,
  left: {
    value: 5,
    left: { value: 3, left: null, right: null },
    right: { value: 7, left: null, right: null }
  },
  right: { value: 15, left: null, right: null }
};

console.log(sum(tree));   // 40 (10+5+3+7+15)
console.log(max(tree));   // 15
console.log(count(tree)); // 5
```

Ejercicio: Transformación de Árbol

Usando el tipo `Node`:

`type Node = {value: number; left: Node | null; rigth: Node | null;}`

Implemente:

```
function transform(node: Node | null, f: (value: number) => number): Node | null
```

Esta función debe crear un **nuevo árbol** con la misma estructura, pero aplicando la función `f` a cada valor.

Ejemplos:

- Si `f = (x) => x * 2`, duplica todos los valores
- Si `f = (x) => x % 2 === 0 ? x : 0`, reemplaza impares con 0

Importante: No debe modificar el árbol original, debe crear uno nuevo.

Solución

```
type Node = {
  value: number;
  left: Node | null;
  right: Node | null;
}

function transform(node: Node | null, f: (value: number) => number): Node | null
  if (node === null) return null;

  return {
    value: f(node.value),
    left: transform(node.left, f),
    right: transform(node.right, f)
  };
}

// Uso
const tree: Node = {
  value: 5,
  left: {
    value: 3,
    left: { value: 1, left: null, right: null },
    right: null
  },
  right: { value: 8, left: null, right: null }
};

// Ejemplo: Duplicar valores
const doubled = transform(tree, x => x * 2);
// Resultado:
//       10
```

```
//       /  \
//      6   16
//     /
//    2
```

Ejercicio: Árboles N-arios

Considere el siguiente tipo que modela un árbol n-ario (cada nodo puede tener cualquier cantidad de hijos):

```
type TreeNode = {
  value: number;
  children: TreeNode[];
}
```

Implemente la función:

```
function fold(node: TreeNode, f: (current: number, accumulated: number) => numbe
```

Esta función permite realizar operaciones como:

- Si `f = (curr, acc) => curr + acc`, calcula la suma de todos los valores
- Si `f = (curr, acc) => acc + 1`, cuenta la cantidad de nodos
- Si `f = (curr, acc) => curr * acc`, calcula el producto de todos los valores

Solución

```
type TreeNode = {
  value: number;
  children: TreeNode[];
}

function fold(
  node: TreeNode,
  f: (current: number, accumulated: number) => number,
  initial: number
): number {
  // Aplicar f al valor del nodo actual con el acumulador inicial
  let result = f(node.value, initial);

  // Fold recursivo sobre cada hijo, usando result como nuevo acumulador
  for (const child of node.children) {
    result = fold(child, f, result);
  }
```

```
  return result;
}

// Uso

//        10
//       / | \
//      5  3  8
//     /|    |
//    2 1    4
const tree: TreeNode = {
  value: 10,
  children: [
    {
      value: 5,
      children: [
        { value: 2, children: [] },
        { value: 1, children: [] }
      ]
    },
    { value: 3, children: [] },
    {
      value: 8,
      children: [
        { value: 4, children: [] }
      ]
    }
  ]
};

// Suma de todos los valores
const sum = fold(tree, (curr, acc) => curr + acc, 0);
console.log(sum); // 33

// Contar nodos
const count = fold(tree, (curr, acc) => acc + 1, 0);
console.log(count); // 7

// Producto de todos los valores
const product = fold(tree, (curr, acc) => curr * acc, 1);
console.log(product); // 9600
```

La función procesa el árbol en **pre-order** (primero el nodo, luego los hijos):

1. Aplica `f` al valor del nodo actual con el acumulador inicial
2. Itera sobre cada hijo, aplicando `fold` recursivamente
3. Cada resultado se convierte en el nuevo acumulador para el siguiente hijo
4. Retorna el resultado final

Ejercicio: Composición de Funciones

Componer funciones para crear pipelines de transformación.
Implementa dos funciones:

- `compose`, que combina funciones de derecha a izquierda.
- `pipe`, que combina funciones de izquierda a derecha.

Ambas funciones deben trabajar específicamente con funciones que operan sobre números (`(x: number) => number`).

Ejemplo:

```
const duplicar = (x: number) => x * 2;
const sumar3 = (x: number) => x + 3;
const elevarAlCuadrado = (x: number) => x ** 2;

// Operación con compose: de derecha a izquierda
const operacion1 = compose([duplicar, sumar3, elevarAlCuadrado]);
console.log(operacion1(2));
// duplicar(sumar3(elevarAlCuadrado(2))) = duplicar(sumar3(4)) = duplicar(7) = 1

// Operación con pipe: de izquierda a derecha
const operacion2 = pipe([elevarAlCuadrado, sumar3, duplicar]);
console.log(operacion2(2));
// duplicar(sumar3(elevarAlCuadrado(2))) = 14
```

Requisitos:

1. Las funciones deben aceptar únicamente funciones que operen sobre números.
2. Deben funcionar con cualquier cantidad de funciones.
3. Ambas implementaciones (`compose` y `pipe`) deben ser independientes.

Solución

```
// Compose: Combina funciones de derecha a izquierda
function compose(funcs: ((x: number) => number)[]): (x: number) => number {
  return (x: number) => {
    let result = x;
    for (let i = funcs.length - 1; i >= 0; i--) {
      result = funcs[i](result);
    }
    return result;
  };
}
```

```
// Pipe: Combina funciones de izquierda a derecha
function pipe(funcs: ((x: number) => number)[]): (x: number) => number {
  return (x: number) => {
    let result = x;
    for (let i = 0; i < funcs.length; i++) {
      result = funcs[i](result);
    }
    return result;
  };
}

// Ejemplo de funciones
const duplicar = (x: number) => x * 2;
const sumar3 = (x: number) => x + 3;
const elevarAlCuadrado = (x: number) => x ** 2;

// Uso de compose y pipe
const operacion1 = compose([duplicar, sumar3, elevarAlCuadrado]);
console.log(operacion1(2)); // 14

const operacion2 = pipe([elevarAlCuadrado, sumar3, duplicar]);
console.log(operacion2(2)); // 14
```

Ejercicio: Sistema de Tarifas UrbanRide

Eres desarrollador en "*UrbanRide*", una aplicación de transporte urbano que necesita calcular tarifas dinámicas para sus viajes. El sistema debe ser lo suficientemente flexible para adaptarse a diferentes tipos de viajes y aplicar recargos según las circunstancias.

Requisitos del Sistema:

1. **Cálculo Base de Tarifas:**
   - **Viajes Cortos** (< 8 km): Tarifa base de $1.50 + $0.80 por kilómetro
   - **Viajes Medios** (≥ 8 km y < 12 km): Tarifa base de $1.80 + $0.70 por kilómetro
   - **Viajes Largos** (≥ 12 km): Tarifa base de $2.00 + $0.60 por kilómetro
2. **Recargos Aplicables:**
   - **Nocturno**: +15% sobre el total
   - **Aeropuerto**: +$3.00 fijo
   - **Fin de Semana**: +25% sobre el total
3. **Funcionalidad Requerida:**
   - Seleccionar automáticamente la estrategia de tarifa base según la distancia
   - Permitir aplicar múltiples recargos de forma encadenada
   - Mostrar desglose detallado del costo (base + cada recargo)
   - Calcular el precio final total

Solución

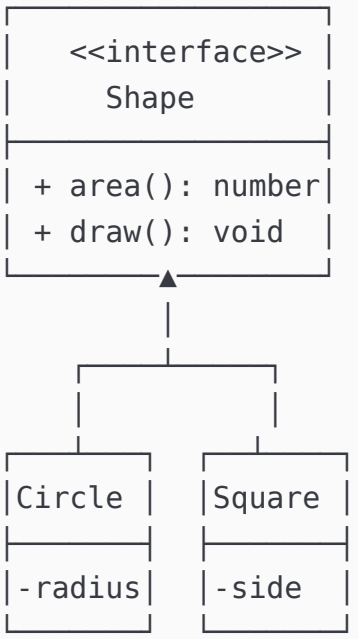
```
// Interfaz para las estrategias de tarifa base
interface TarifaBase {
  distancia: number;
  calcular(): number;
  obtenerDescripcion(): string;
}

// Estrategia para viajes cortos
class TarifaCorta implements TarifaBase {
  distancia: number;

  constructor(distancia: number) {
    this.distancia = distancia;
  }

  calcular(): number {
    const tarifaBase = 1.5;
    const costoPorKm = 0.8;
    return tarifaBase + this.distancia * costoPorKm;
  }

  obtenerDescripcion(): string {
    return "Tarifa Corta";
  }
}

// Estrategia para viajes medios
class TarifaMedia implements TarifaBase {
  distancia: number;

  constructor(distancia: number) {
    this.distancia = distancia;
  }

  calcular(): number {
    const tarifaBase = 1.8;
    const costoPorKm = 0.7;
    return tarifaBase + this.distancia * costoPorKm;
  }

  obtenerDescripcion(): string {
    return "Tarifa Media";
  }
}

// Estrategia para viajes largos
class TarifaLarga implements TarifaBase {
```

```
  distancia: number;

  constructor(distancia: number) {
    this.distancia = distancia;
  }

  calcular(): number {
    const tarifaBase = 2.0;
    const costoPorKm = 0.6;
    return tarifaBase + this.distancia * costoPorKm;
  }

  obtenerDescripcion(): string {
    return "Tarifa Larga";
  }
}

// Fabrica Simple para obtener la estrategia adecuada
class FabricaTarifa {
  static obtenerTarifa(distancia: number): TarifaBase {
    if (distancia < 8) {
      return new TarifaCorta(distancia);
    } else if (distancia >= 8 && distancia < 12) {
      return new TarifaMedia(distancia);
    } else {
      return new TarifaLarga(distancia);
    }
  }
}

// Clase abstracta para los recargos (decoradores)
abstract class Recargo implements TarifaBase {
  protected tarifa: TarifaBase;
  distancia: number;

  constructor(tarifa: TarifaBase) {
    this.tarifa = tarifa;
    this.distancia = tarifa.distancia;
  }

  abstract calcular(): number;
  abstract obtenerDescripcion(): string;
}

// Recargo nocturno (+15%)
class RecargoNocturno extends Recargo {
  calcular(): number {
    const tarifaBase = this.tarifa.calcular();
    return tarifaBase * 1.15; // +15%
  }
```

```
  obtenerDescripcion(): string {
    return `${this.tarifa.obtenerDescripcion()} + Recargo Nocturno (15%)`;
  }
}

// Recargo aeropuerto (+$3.00 fijo)
class RecargoAeropuerto extends Recargo {
  calcular(): number {
    const tarifaBase = this.tarifa.calcular();
    return tarifaBase + 3.0; // +$3.00 fijo
  }

  obtenerDescripcion(): string {
    return `${this.tarifa.obtenerDescripcion()} + Recargo Aeropuerto ($3.00)`;
  }
}

// Recargo fin de semana (+25%)
class RecargoFinDeSemana extends Recargo {
  calcular(): number {
    const tarifaBase = this.tarifa.calcular();
    return tarifaBase * 1.25; // +25%
  }

  obtenerDescripcion(): string {
    return `${this.tarifa.obtenerDescripcion()} + Recargo Fin de Semana (25%)`;
  }
}

// Uso
let tarifa: TarifaBase = FabricaTarifa.obtenerTarifa(15);
tarifa = new RecargoNocturno(tarifa);
tarifa = new RecargoAeropuerto(tarifa);
tarifa = new RecargoFinDeSemana(tarifa);

const costoFinal = tarifa.calcular();
const descripcion = tarifa.obtenerDescripcion();

console.log(costoFinal);
console.log(descripcion);
```

Ejercicio: Implementación desde UML

Implementa el siguiente diagrama de clases en TypeScript:



Requisitos:

- Implementa la interfaz `Shape`
- Implementa las clases `Circle` y `Square`
- Agrega un método `describe()` que retorne una descripción de la forma
- Crea un array de formas mixtas y calcula el área total

Solución

```
// Definimos la interfaz Shape
interface Shape {
  area(): number;
  draw(): void;
  describe(): string; // Método adicional para descripción
}

// Clase Circle que implementa Shape
class Circle implements Shape {
  private radius: number;

  constructor(radius: number) {
    this.radius = radius;
  }

  // Calcula el área del círculo
  area(): number {
    return Math.PI * this.radius * this.radius;
  }

  // Dibuja el círculo (simulado con un console.log)
  draw(): void {
    console.log("Dibujando un círculo...");
```

```
  }

  // Describe el círculo con su radio
  describe(): string {
    return `Círculo con radio ${this.radius}`;
  }
}

// Clase Square que implementa Shape
class Square implements Shape {
  private side: number;

  constructor(side: number) {
    this.side = side;
  }

  // Calcula el área del cuadrado
  area(): number {
    return this.side * this.side;
  }

  // Dibuja el cuadrado (simulado con un console.log)
  draw(): void {
    console.log("Dibujando un cuadrado...");
  }

  // Describe el cuadrado con su lado
  describe(): string {
    return `Cuadrado con lado ${this.side}`;
  }
}

// Crea un array de formas mixtas
const shapes: Shape[] = [
  new Circle(5), // Círculo con radio 5
  new Square(4), // Cuadrado con lado 4
  new Circle(3), // Círculo con radio 3
  new Square(2), // Cuadrado con lado 2
];

// Calcula el área total
let totalArea = 0;
for (const shape of shapes) {
  console.log(shape.describe());
  shape.draw(); // Dibuja cada forma
  totalArea += shape.area(); // Suma el área
}

console.log(`El área total de las formas es: ${totalArea.toFixed(2)}`);
```

Ejercicio: De Imperativo a Funcional

Refactoriza el siguiente código imperativo para usar programación funcional (map, filter, reduce):

```
const productos = [
    { nombre: "Laptop", precio: 1200, categoria: "tecnologia" },
    { nombre: "Mouse", precio: 25, categoria: "tecnologia" },
    { nombre: "Silla", precio: 150, categoria: "muebles" },
    { nombre: "Teclado", precio: 75, categoria: "tecnologia" }
];

// Código imperativo
let productosTecnologia = [];
for (let i = 0; i < productos.length; i++) {
    if (productos[i].categoria === "tecnologia") {
        productosTecnologia.push(productos[i]);
    }
}

let preciosConDescuento = [];
for (let i = 0; i < productosTecnologia.length; i++) {
    preciosConDescuento.push(productosTecnologia[i].precio * 0.9);
}

let total = 0;
for (let i = 0; i < preciosConDescuento.length; i++) {
    total += preciosConDescuento[i];
}

console.log(total);
```

Requisitos:

- Usa `filter`, `map` y `reduce`
- Todo debe hacerse en una sola expresión encadenada
- El código debe ser más legible que el original

Solución

```
const productos = [
    { nombre: "Laptop", precio: 1200, categoria: "tecnologia" },
    { nombre: "Mouse", precio: 25, categoria: "tecnologia" },
    { nombre: "Silla", precio: 150, categoria: "muebles" },
    { nombre: "Teclado", precio: 75, categoria: "tecnologia" }
];
```

```
// Refactorización funcional
const total = productos
    .filter(producto => producto.categoria === "tecnologia") // Filtra productos
    .map(producto => producto.precio * 0.9) // Aplica el descuento del 10%
    .reduce((suma, precio) => suma + precio, 0); // Suma los precios con descuen

console.log(total); // Salida: 1188
```