Antes de adentrarnos en los patrones específicos, necesitamos reforzar los conceptos fundamentales de la Programación Orientada a Objetos que hacen posible estos patrones. **Los patrones de diseño no son trucos mágicos**, son soluciones elegantes que aprovechan las características más poderosas de la POO.

1. El Polimorfismo: La Base de la Flexibilidad

Definición: La capacidad de que objetos de diferentes tipos respondan al mismo mensaje (método) de maneras distintas.

```
interface Animal {
    hacerSonido(): string;
}

class Perro implements Animal {
    hacerSonido(): string {
        return "¡Guau!";
    }
}

class Gato implements Animal {
    hacerSonido(): string {
        return "¡Miau!";
    }
}

// Polimorfismo en acción
function hacerRuidoAnimal(animal: Animal): void {
    console.log(animal.hacerSonido()); // No sabe QUÉ animal es, pero sabe Q
}

const miPerro: Animal = new Perro();
const miGato: Animal = new Gato();

hacerRuidoAnimal(miPerro); // "¡Guau!"
hacerRuidoAnimal(miGato);  // "¡Miau!"
```

¿Por qué es crucial para los patrones?

- Te permite escribir código que funciona con "familias" de objetos sin conocer sus tipos específicos
- Es la base del patrón Strategy, Factory y muchos otros
- **Sin polimorfismo, cada vez que quisieras agregar un nuevo tipo de animal, tendrías que modificar la función `hacerRuidoAnimal` con más `if-else`. Con polimorfismo, simplemente creas una nueva clase que implemente la interfaz.**

2. Interfaces vs Clases Abstractas: Contratos vs Herencia Parcial

Interfaces: Contratos Puros

```
interface Volador {
    volar(): void;
    aterrizar(): void;
}

interface Nadador {
    nadar(): void;
}

// Un pato puede volar Y nadar
class Pato implements Volador, Nadador {
    volar(): void { console.log("El pato vuela"); }
    aterrizar(): void { console.log("El pato aterriza"); }
    nadar(): void { console.log("El pato nada"); }
}
```

¿Por qué es crucial para los patrones?

- **Las interfaces definen "qué puede hacer" un objeto sin importar "cómo lo hace"**
- Permiten que objetos completamente diferentes compartan la misma interfaz
- Son fundamentales para el patrón Strategy y Factory Method

Clases Abstractas: Herencia con Implementación Parcial

```
abstract class Vehiculo {
    protected velocidad: number = 0;

    // Método concreto: todos los vehículos aceleran igual
    acelerar(): void {
        this.velocidad += 10;
        console.log(`Velocidad: ${this.velocidad} km/h`);
    }

    // Método abstracto: cada vehículo arranca diferente
    abstract arrancar(): void;
}

class Auto extends Vehiculo {
    arrancar(): void {
        console.log("Girando la llave del auto");
    }
```

```
}

class Moto extends Vehiculo {
    arrancar(): void {
        console.log("Pateando el arranque de la moto");
    }
}
```

¿Por qué es crucial para los patrones?

- **Permiten compartir código común mientras fuerzan la implementación de comportamientos específicos**
- Son ideales para el patrón Template Method
- Reducen duplicación de código al tiempo que mantienen flexibilidad

3. Herencia y Composición: "Es un" vs "Tiene un"

Herencia: Relación "Es un"

```
class Empleado {
    protected nombre: string;
    protected salario: number;

    constructor(nombre: string, salario: number) {
        this.nombre = nombre;
        this.salario = salario;
    }

    trabajar(): string {
        return `${this.nombre} está trabajando`;
    }
}

class Programador extends Empleado {
    private lenguaje: string;

    constructor(nombre: string, salario: number, lenguaje: string) {
        super(nombre, salario);
        this.lenguaje = lenguaje;
    }

    // Sobrescribe el método padre
    trabajar(): string {
        return `${this.nombre} está programando en ${this.lenguaje}`;
    }

    // Método específico de programador
    debuggear(): string {
```

```
        return `${this.nombre} está cazando bugs`;
    }
}
```

¿Por qué es crucial para los patrones?

- **Permite reutilizar código y establecer jerarquías lógicas**
- Es fundamental para patrones como Template Method
- **Pero cuidado: el abuso de herencia crea código rígido y difícil de cambiar**

Composición: Relación "Tiene un"

```
class Motor {
    private cilindros: number;

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

    encender(): string {
        return `Motor de ${this.cilindros} cilindros encendido`;
    }
}

class Carro {
    private motor: Motor; // Composición: el carro "tiene un" motor

    constructor(cilindros: number) {
        this.motor = new Motor(cilindros); // El carro crea su motor
    }

    arrancar(): string {
        return this.motor.encender(); // Delega al motor
    }
}
```

¿Por qué es crucial para los patrones?

- **Es más flexible que la herencia: puedes cambiar el motor sin cambiar el carro**
- Es la base de patrones como Strategy, Decorator y Composite
- **"Favorece composición sobre herencia" es uno de los principios más importantes del diseño de software**

4. El Principio de Inversión de Dependencias en Acción

Problema: Código rígido y difícil de cambiar

```
// MAL: Alto acoplamiento
class EmailService {
    enviar(mensaje: string): void {
        console.log(`Enviando email: ${mensaje}`);
    }
}

class NotificacionManager {
    private emailService: EmailService = new EmailService(); // Dependencia

    notificar(mensaje: string): void {
        this.emailService.enviar(mensaje); // Solo puede enviar emails
    }
}
```

¿Por qué es crucial para los patrones?

- **Este código está "casado" con EmailService. Si mañana quieres agregar SMS, tienes que modificar NotificacionManager**
- Viola el principio abierto/cerrado: no está cerrado para modificación

Solución: Inversión de dependencias

```
// BIEN: Bajo acoplamiento
interface ServicioNotificacion {
    enviar(mensaje: string): void;
}

class EmailService implements ServicioNotificacion {
    enviar(mensaje: string): void {
        console.log(`📧 Email: ${mensaje}`);
    }
}

class SMSService implements ServicioNotificacion {
    enviar(mensaje: string): void {
        console.log(`📱 SMS: ${mensaje}`);
    }
}

class NotificacionManager {
    constructor(private servicio: ServicioNotificacion) {} // Inyección de d

    notificar(mensaje: string): void {
        this.servicio.enviar(mensaje); // Funciona con cualquier implementac
    }
}
```

```
// Uso flexible
const emailManager = new NotificacionManager(new EmailService());
const smsManager = new NotificacionManager(new SMSService());
```

¿Por qué es crucial para los patrones?

- **Permite que el código dependa de abstracciones, no de implementaciones concretas**
- Es fundamental para patrones como Strategy, Factory, y Dependency Injection
- **Hace que el código sea extensible: puedes agregar WhatsAppService sin tocar una línea de código existente**

5. Encapsulación: Controlando el Acceso

```
class CuentaBancaria {
    private saldo: number = 0; // Privado: nadie puede tocarlo directamente

    // Método público controlado
    depositar(cantidad: number): void {
        if (cantidad > 0) {
            this.saldo += cantidad;
            console.log(`Depositado: $${cantidad}. Saldo: $${this.saldo}`);
        } else {
            throw new Error("La cantidad debe ser positiva");
        }
    }

    // Método público controlado
    retirar(cantidad: number): boolean {
        if (cantidad > 0 && cantidad <= this.saldo) {
            this.saldo -= cantidad;
            console.log(`Retirado: $${cantidad}. Saldo: $${this.saldo}`);
            return true;
        }
        return false;
    }

    // Solo lectura del saldo
    getSaldo(): number {
        return this.saldo;
    }
}
```

¿Por qué es crucial para los patrones?

- **Protege el estado interno y garantiza que el objeto mantenga su integridad**

- Es fundamental para Singleton (controla la creación de instancias)
- **Sin encapsulación, cualquier código podría modificar directamente el saldo y crear estados inconsistentes**

Ejercicio Integrador: Sistema de Formas Geométricas

Ahora que hemos repasado los conceptos, veamos cómo se integran:

```
// 1. Interface para el contrato
interface Figura {
    calcularArea(): number;
    dibujar(): void;
}

// 2. Clase abstracta para comportamiento común
abstract class FiguraConColor implements Figura {
    protected color: string;

    constructor(color: string) {
        this.color = color;
    }

    // Método concreto común
    mostrarInfo(): void {
        console.log(`Figura ${this.color} con área: ${this.calcularArea()}`)
    }

    // Métodos abstractos que deben implementar las subclases
    abstract calcularArea(): number;
    abstract dibujar(): void;
}

// 3. Implementaciones concretas
class Circulo extends FiguraConColor {
    constructor(private radio: number, color: string) {
        super(color);
    }

    calcularArea(): number {
        return Math.PI * this.radio * this.radio;
    }

    dibujar(): void {
        console.log(`Dibujando círculo ${this.color} de radio ${this.radio}`
    }
}

class Rectangulo extends FiguraConColor {
```

```
    constructor(private ancho: number, private alto: number, color: string)
        super(color);
    }

    calcularArea(): number {
        return this.ancho * this.alto;
    }

    dibujar(): void {
        console.log(`Dibujando rectángulo ${this.color} de ${this.ancho}x${t
    }
}

// 4. Polimorfismo en acción
class DibujoManager {
    private figuras: Figura[] = [];

    agregarFigura(figura: Figura): void {
        this.figuras.push(figura);
    }

    dibujarTodas(): void {
        this.figuras.forEach(figura => {
            figura.dibujar(); // Polimorfismo: cada figura se dibuja diferen
        });
    }

    calcularAreaTotal(): number {
        return this.figuras.reduce((total, figura) => {
            return total + figura.calcularArea(); // Polimorfismo: cada figu
        }, 0);
    }
}

// Uso del sistema
const manager = new DibujoManager();
manager.agregarFigura(new Circulo(5, "rojo"));
manager.agregarFigura(new Rectangulo(10, 20, "azul"));

manager.dibujarTodas();
console.log(`Área total: ${manager.calcularAreaTotal()}`);
```

¿Por qué es crucial esta integración para los patrones?

- **Este ejemplo muestra cómo todos los conceptos trabajan juntos**
- `DibujoManager` no sabe qué tipos específicos de figuras maneja (polimorfismo)
- Puede agregar nuevas figuras sin modificar el manager (abierto/cerrado)
- Cada figura controla cómo se dibuja y calcula (encapsulación)

- Las figuras comparten comportamiento común pero implementan lo específico (herencia + abstracción)

Material realizado con IA