📄 *Patterns Aplicados*

🧠 1. Strategy Pattern

Fuente: *Design Patterns: Elements of Reusable Object-Oriented Software (GoF) / refactoring.guru/strategy*

✓ Reseña

Strategy permite definir una familia de algoritmos y encapsularlos para que sean intercambiables en tiempo de ejecución. Evita condicionales gigantes y hace que el comportamiento sea flexible.

✓ Ejemplo mínimo

```
public interface IDiscountStrategy {
    decimal ApplyDiscount(decimal price);
}

public class SamsungDiscount : IDiscountStrategy {
    public decimal ApplyDiscount(decimal price) => price * 0.90m;
}

public class AppleDiscount : IDiscountStrategy {
    public decimal ApplyDiscount(decimal price) => price * 0.80m;
}

public class Promotion {
    private IDiscountStrategy _strategy;
    public Promotion(IDiscountStrategy strategy) => _strategy = strategy;

    public decimal Apply(decimal price) =>
_strategy.ApplyDiscount(price);
}
```

✓ Justificación

Se aplicó cuando la clase **Promotion** tenía lógica rígida con `if/marca`. Strategy elimina esa rigidez, facilita agregar nuevas marcas sin tocar el código existente y sigue el principio **OCP (Open/Closed Principle)**.

🧱 2. Factory Method

Fuente: GoF / refactoring.guru/factory-method

✓ Reseña

Factory Method delega la creación de objetos a subclases o métodos especializados. Permite controlar cómo se instancian los objetos y reduce el acoplamiento.

✓ Ejemplo mínimo

```
public abstract class MobileFactory {
    public abstract Mobile Create();
}

public class SamsungFactory : MobileFactory {
    public override Mobile Create() => new Samsung();
}
```

✓ Justificación

Se usó porque el código creaba `new Mobile()` directamente. Con Factory Method, la creación queda encapsulada y el sistema puede generar diferentes tipos de dispositivos sin modificar las clases existentes.

🧱 3. Singleton Pattern

Fuente: GoF / refactoring.guru/singleton

✓ Reseña

Singleton garantiza que una clase tenga **una única instancia global** y un punto de acceso controlado. Muy usado para manejadores centrales o servicios globales.

✓ Ejemplo mínimo

```
public class StoreManager {
    private static StoreManager _instance;
    private StoreManager() {}

    public static StoreManager Instance =>
        _instance ??= new StoreManager();
}
```

✓ Justificación

Se aplicó porque existían varias instancias de StoreManager causando inconsistencias. Con Singleton, el programa ahora usa un **único gestor de tienda**, evitando duplicados y asegurando un estado unificado.

🧱 4. Facade Pattern

Fuente: GoF / refactoring.guru/facade

✓ Reseña

Facade simplifica sistemas complejos ofreciendo una interfaz más limpia. Permite que el cliente use funciones sin conocer la lógica interna.

✓ Ejemplo mínimo

```
public class InventoryFacade {
    private readonly Inventory _inventory = new();
    private readonly Billing _billing = new();

    public void ProcessSale(string product, int qty) {
        _inventory.Decrease(product, qty);
        _billing.GenerateInvoice(product, qty);
    }
}
```

✓ Justificación

Originalmente, `InventoryAndBilling` mezclaba dos responsabilidades.
Con Facade, se separan Inventario y Facturación pero se centraliza el proceso a través de una interfaz simple.

🧱 5. Strategy Pattern (para refactorizar método largo)

(SRP + Strategy aplicado al método ProcessSale)
Fuente: refactoring.guru/srp

✓ Reseña

Además de SRP, subdividir un método complejo es parte del refactor asociado al uso de estrategias para porciones específicas del cálculo.

✓ Ejemplo mínimo

```
public class ProcessSaleHandler {
    public void Handle() {
        Validate();
        CalculateTotals();
        SaveSale();
        PrintTicket();
    }
}
```

✓ Justificación

El método original **ProcessSale** hacía demasiadas cosas.
Se dividió en acciones pequeñas con una responsabilidad única, utilizando clases distintas cuando fue necesario.

✓ Conclusión General

Cada patrón fue elegido siguiendo principios de diseño del GoF y refactoring.guru para:

- Reducir acoplamiento
- Aumentar flexibilidad
- Mejorar mantenibilidad
- Preparar el código para escalabilidad
- Evitar malas prácticas como métodos enormes, clases con demasiadas responsabilidades y lógica rígida