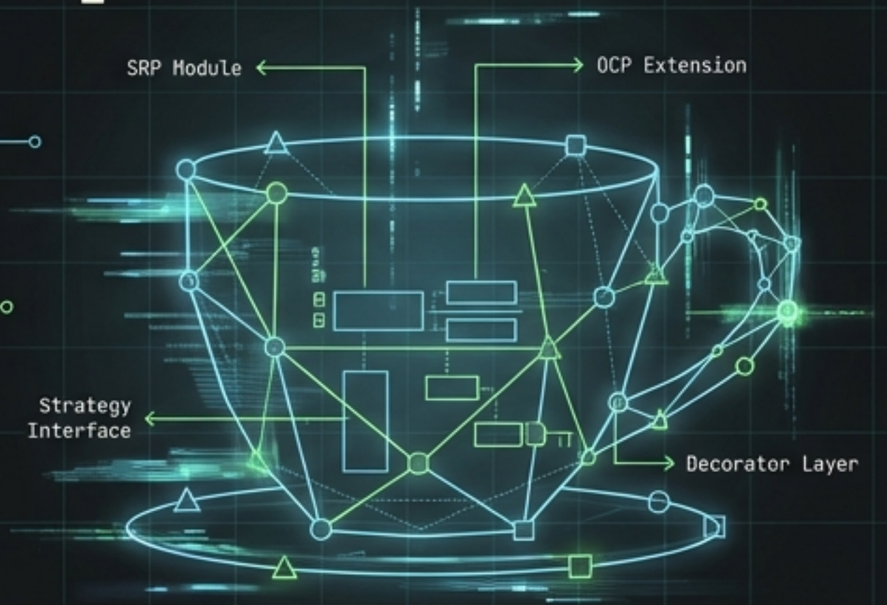
Assignment 2: The Architected Tea Shop

Implementing SRP, OCP, Strategy, and Decorator Patterns in Console Applications



Mission Profile: Design > Polish

The Product (Surface)

- Console-based Tea Shop >_
- Search & Filter (12+ Items)
- Sort (Price, Rating)
- Purchase Logic
- Payment Processing

```
$ ./tea_shop_cli
> Welcome to the Tea Shop!
> Available: Green Tea ($5.00), Black Tea ($4.50),
Herbal ($6.00)...
> Select an option:
```

The Architecture (Subsurface)

[SRP] Single Responsibility Principle (SRP)
[→ Modular Modules]

[OCP] Open/Closed Principle (OCP)
[→ Extendable without Modification]

[STRATEGY] Strategy Pattern (Payment)
[→ Pluggable Payment Methods]

[DECORATOR] Decorator Pattern (Filtering)
[→ Dynamic Filter Composition]

[INVARIANT] Invariant Enforcement
[→ Robust Data Integrity]

WARNING: This is not about UI polish. This is a test of architectural rigor.

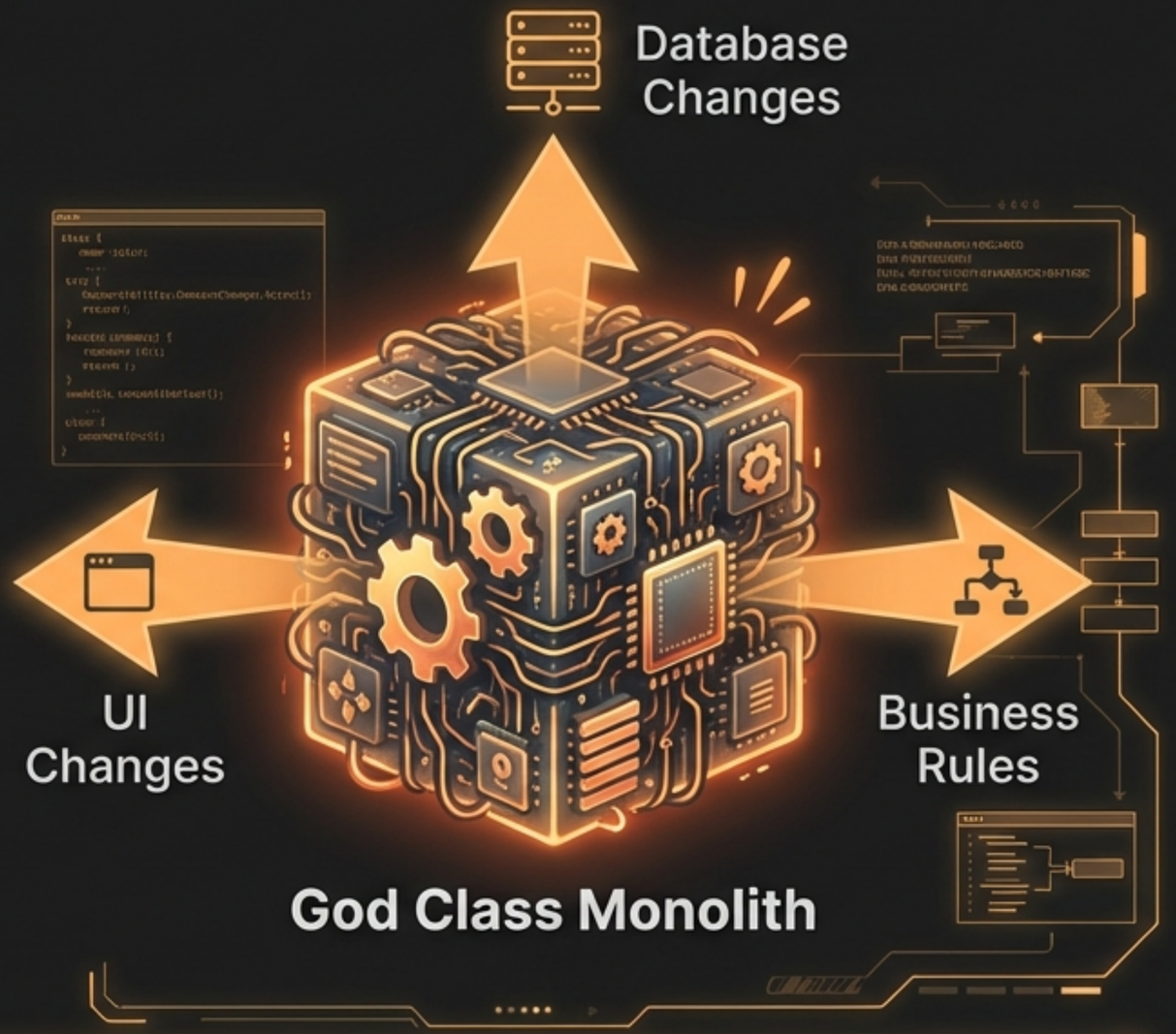
The Prime Directive: SRP

Single Responsibility Principle

A class should have **one reason** to change.

Responsibility is defined by the ACTOR requesting the change.

- User Interface Actor ≠ Domain Logic Actor



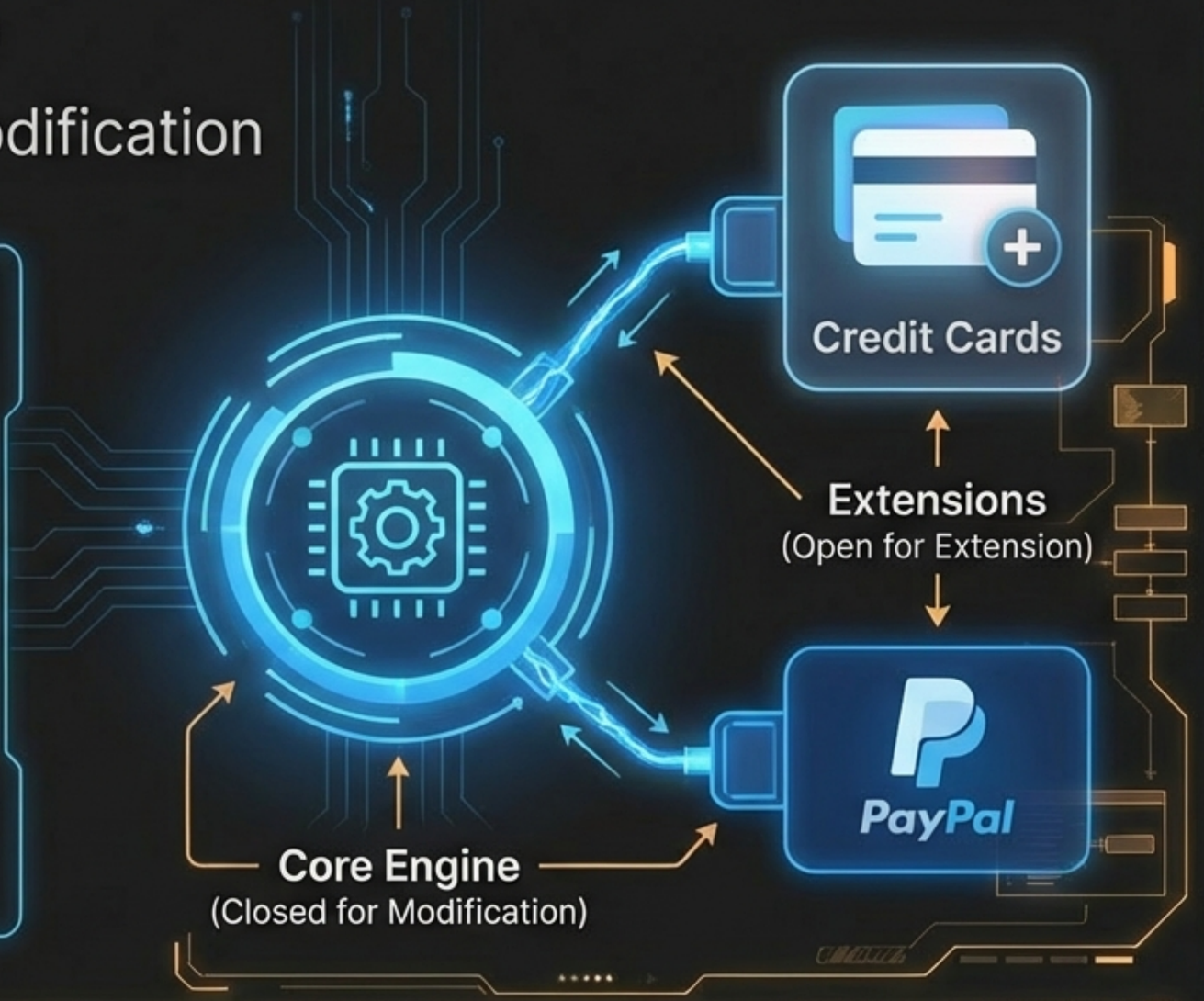
WARNING: This is not about UI polish. This is a test of architectural rigor.

Future-Proofing: OCP

Open for Extension, Closed for Modification

How do we add features without breaking the core?

- New Filter? → Add a Class.
- New Payment? → Plug in a
- The Core Engine remains untouched.



WARNING: This is not about UI polish. This is a test of architectural rigor.

The Search Engine

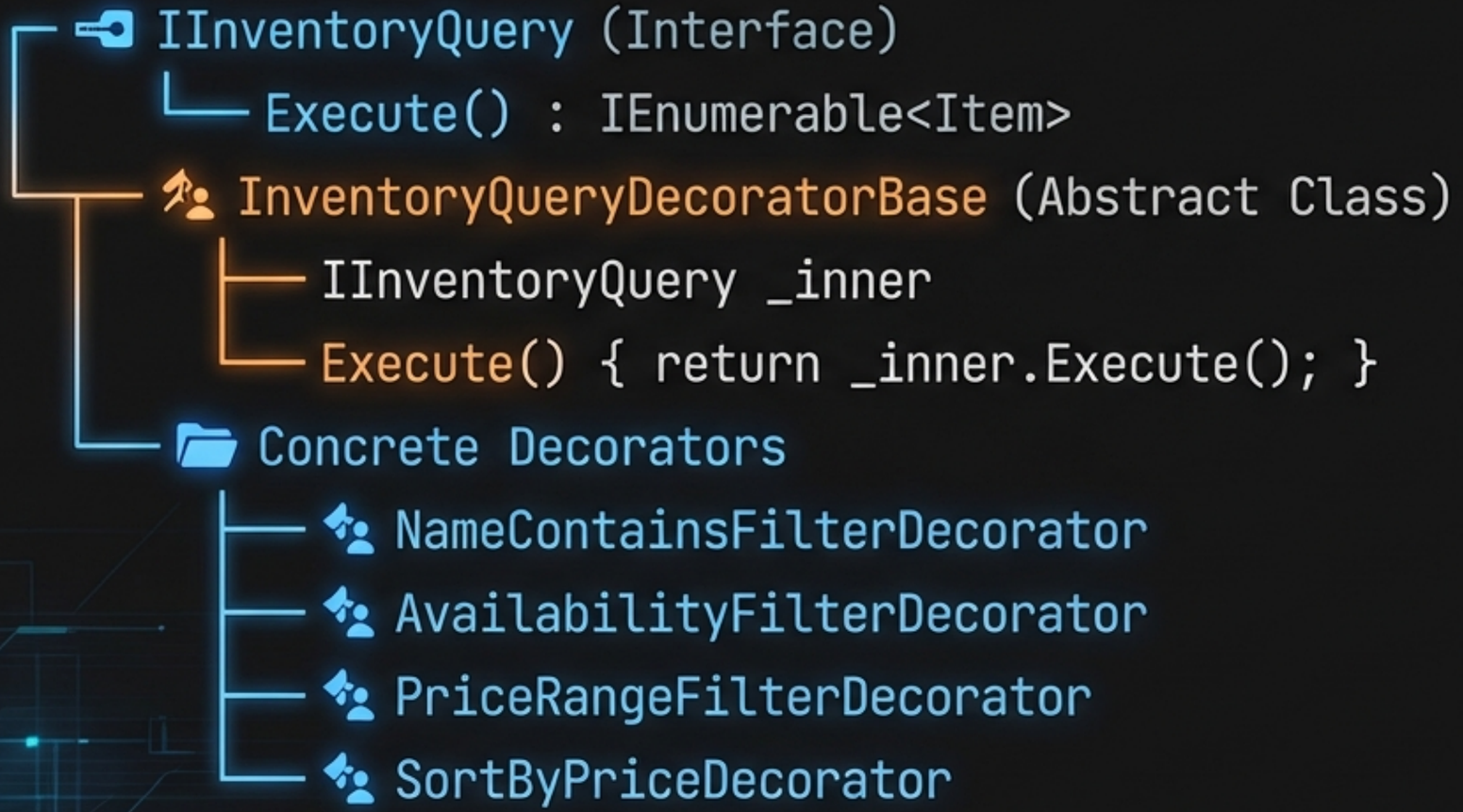
Pattern: The Decorator

Single call to **.Execute()** triggers the whole chain.

Single call to **.Execute()** triggers the whole chain.

```
var query = new Sort(new Filter(new BaseQuery()));
```

Decorator Implementation Details



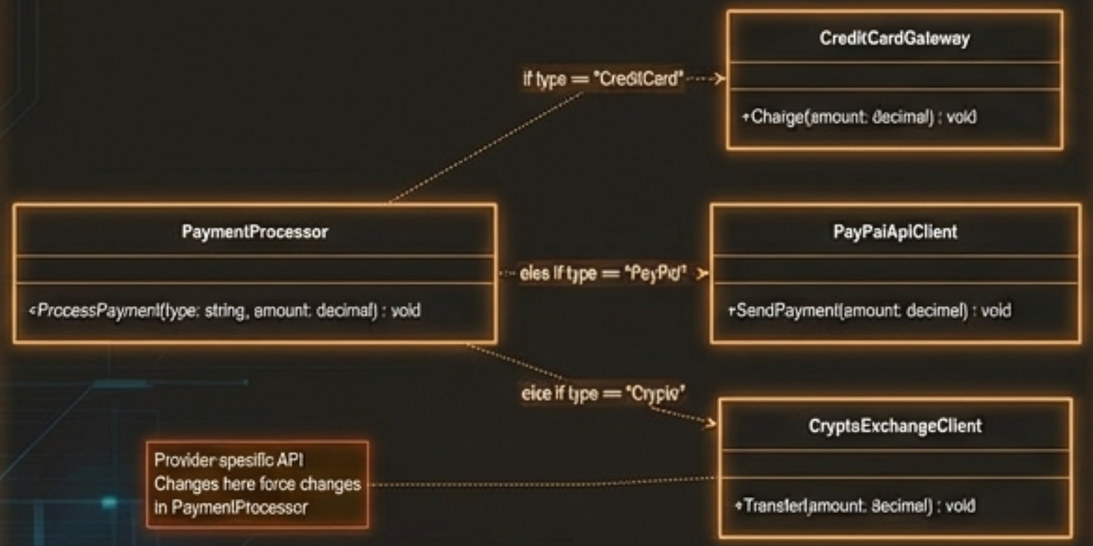
Each concrete decorator overrides **Execute()** to apply logic *before* or *after* calling the inner query.

The Checkout System

Pattern: Strategy

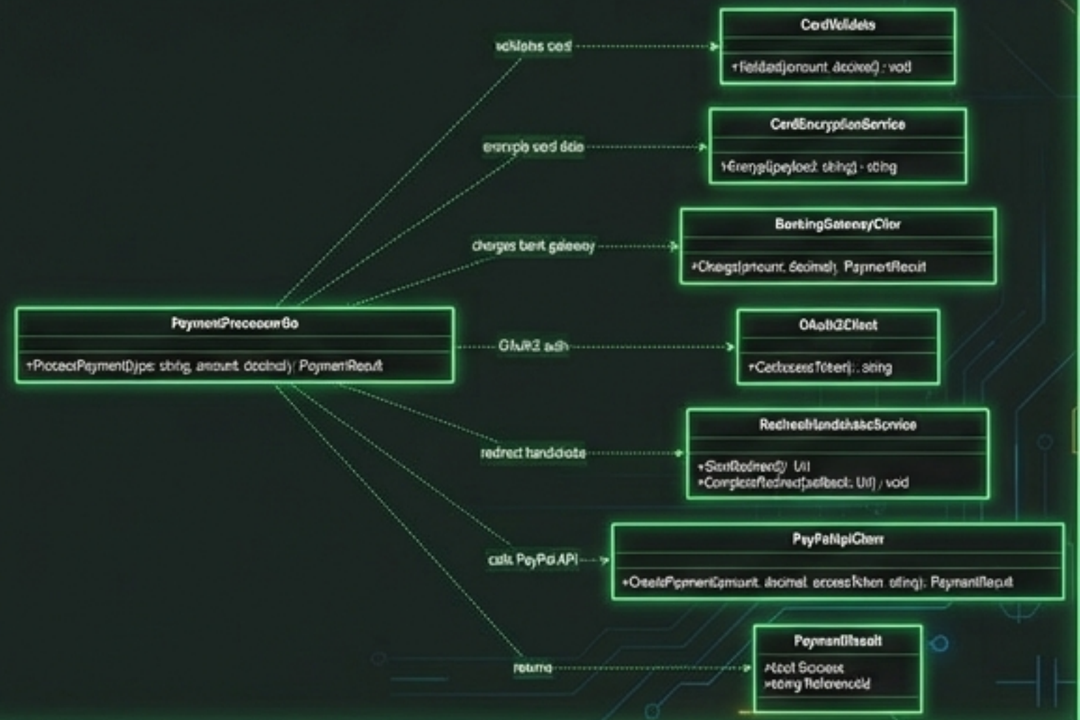
The Anti-Pattern

- Rigid, Fragile, Hard to Test.
- Uses Switch/If-Else statements.
- Modification requires breaking open the class.



The Solution

- Modular, Encapsulated, Polymorphic.
- Depends on Interfaces.
- New methods = New classes.



Strategy Implementation Details

```
interface IPaymentStrategy {
    void Process(decimal amount);
}
```

```
abstract class PaymentStrategyBase : IPaymentStrategy {
    protected void Validate(decimal amount);
    public abstract void Process(decimal amount);
}
```

```
class CreditCardStrategy : PaymentStrategyBase {
    public override veid Process(decimal amount) {
        // Logic: Card Num Validation
        Validate(amount);
        // Specific Credit Card processing
    }
}
```

```
class ApplePayStrategy : PaymentStrategyBase {
    public override void Process(decimal amount) {
        // Logic: Device Token
        Validate(amount);
        // Specific Apple Pay processing
    }
}
```

```
class CryptoCurrencyStrategy : PaymentStrategyBase {
    public override void Process(decimal amount) {
        // Logic: Wallet Address
        Validate(amount);
        // Specific Crypto processing
    }
}
```

The Separation of Powers

DOMAIN LAYER (Pure Logic)

- Inventory Rules
- Payment Strategies
- Decorators

NO `Console.WriteLine()` or `ReadLine()` allowed.

THE IRON CURTAIN

USER INTERFACE LAYER (Dirty I/O)

- Prompts & Menus
- Builders
- Input Parsing

References Domain.

The Data Layer

State Management & Invariants

The Repository (InventoryRepository)

- In-Memory Storage (List<T>).
- Private State (List is NOT public).
- Mutation only via methods (e.g., ReduceQuantity).

The Inventory Item

- Name (String)
- Price (Decimal)
- Quantity (Int >= 0)
- **StarRating** (Custom Class, enforces 1-5 range)

⚠ Database = Forbidden. JSON = Forbidden.

The Application Flow

1

User Interface

- 'InventoryQueryBuilder' collects inputs.
- 'Constructs' the Decorator Chain.

2

Domain Execution

- 'Application' calls 'query.Execute()'.
- 'Repository' filters data in memory.

3

Purchase Flow

- 'PaymentBuilder' selects Strategy.
- 'Application' calls 'RunCheckout(strategy)'.

UI handles Construction. Domain handles Execution.

File Organization

No God Files. One Class Per File.

C# Structure

```
Assignment2Solution
├── Domain
│   ├── Inventory
│   └── Payment
└── UserInterface
```

Java Structure

```
edu.kennesaw.teashop
├── domain
│   ├── inventory
│   └── payment
└── userinterface
```

Correct namespace/package structure is mandatory.

The Entry Point (Main)

Keep it Short and Boring.

```
public static void Main()
{
    // 1. Create Dependencies
    var repo = new InventoryRepository();

    // 2. Wire Application
    var app = new Application(repo);

    // 3. Run
    app.Run();
}
```

Max 20-30 lines.
No loops. No logic.
Just wiring.

Constraints & FAQ

Strictly Forbidden (X)

- **NO** Databases or External Libraries.
- **NO** `if/else` for payment selection.
- **NO** filtering logic inside UI classes.
- **NO** generic `Object` types.

Required / Encouraged (Check)

- **YES** Public GitHub Repository.
- **YES** `.gitignore` file.
- **YES** Docker (Optional but Recommended).
- **YES** Unit Tests (Optional).

Definition of Done

Functionality

- [ ] Inventory of 12+ items
- [ ] 4 Working Filters (Name, Stock, Price, Rating)
- [ ] 2 Working Sorts (Price, Rating)
- [ ] 3 Payment Strategies

Logistics

- [ ] Source code in /src
- [ ] README.md with screenshots
- [ ] Incremental Commit History

“A simple, boring, well-designed solution is better than a clever, tangled one.”