# Design Patterns Explained

This paper provides clear explanations of common design patterns, complete with UML diagrams and real-world analogies.

## 1. Strategy Pattern

**Definition:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

**Real-World Analogy:** Think of different payment methods at a store. Whether you pay by credit card, PayPal, or cash, the end result (payment) is the same, but the strategy (how you pay) differs.

**UML Diagram:**

```
classDiagram
    class PaymentContext {
        -PaymentStrategy strategy
        +setStrategy(PaymentStrategy)
        +executePayment()
    }
    class PaymentStrategy {
        <>
        +pay()
    }
    class CreditCardPayment {
        +pay()
    }
    class PayPalPayment {
        +pay()
    }
    class CashPayment {
        +pay()
    }
    PaymentStrategy <|.. CreditCardPayment
    PaymentStrategy <|.. PayPalPayment
    PaymentStrategy <|.. CashPayment
    PaymentContext o-- PaymentStrategy
```

## 2. Factory Method Pattern

**Definition:** Provides an interface for creating objects but allows subclasses to decide which class to instantiate.

**Real-World Analogy:** A restaurant kitchen receiving orders. The kitchen (factory) knows how to create different dishes (products), but the specific chef (concrete factory) decides how to prepare each dish.

**UML Diagram:**

```
classDiagram
    class Restaurant {
        <>
        +createDish()
        +serveDish()
    }
    class ItalianRestaurant {
        +createDish()
    }
    class ChineseRestaurant {
        +createDish()
    }
    class Dish {
        <>
        +prepare()
    }
    class Pizza {
        +prepare()
    }
    class NoodleDish {
        +prepare()
    }
    Restaurant <|-- ItalianRestaurant
    Restaurant <|-- ChineseRestaurant
    Dish <|.. Pizza
    Dish <|.. NoodleDish
```

## 3. Abstract Factory Pattern

**Definition:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**Real-World Analogy:** A furniture manufacturer that creates different styles of furniture (modern, vintage, etc.). Each style includes matching chairs, tables, and sofas.

**UML Diagram:**

```
classDiagram
    class FurnitureFactory {
        <>
        +createChair()
        +createTable()
        +createSofa()
    }
    class ModernFurnitureFactory {
        +createChair()
        +createTable()
        +createSofa()
    }
    class VintageFurnitureFactory {
        +createChair()
```

```
        +createTable()
        +createSofa()
    }
    class Chair {
        <>
    }
    class Table {
        <>
    }
    class Sofa {
        <>
    }
    FurnitureFactory <|.. ModernFurnitureFactory
    FurnitureFactory <|.. VintageFurnitureFactory
```

## 4. Façade Pattern

**Definition:** Provides a unified interface to a set of interfaces in a subsystem, making it easier to use.

**Real-World Analogy:** A car's dashboard. You don't need to understand the complex systems (engine, fuel injection, electrical) to drive - the dashboard provides a simple interface.

**UML Diagram:**

```
classDiagram
    class CarFacade {
        -Engine engine
        -FuelSystem fuelSystem
        -ElectricalSystem electrical
        +startCar()
        +stopCar()
    }
    class Engine {
        +start()
        +stop()
    }
    class FuelSystem {
        +pumpFuel()
        +stopFuel()
    }
    class ElectricalSystem {
        +initializeSystems()
        +shutdown()
    }
    CarFacade --> Engine
    CarFacade --> FuelSystem
    CarFacade --> ElectricalSystem
```

## 5. Decorator Pattern

**Definition:** Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing.

**Real-World Analogy:** Customizing a coffee order. You start with a basic coffee and can "decorate" it with extra shots, milk, sugar, etc.

**UML Diagram:**

```
classDiagram
    class Coffee {
        <>
        +cost()
        +description()
    }
    class BasicCoffee {
        +cost()
        +description()
    }
    class CoffeeDecorator {
        <>
        -Coffee coffee
        +cost()
        +description()
    }
    class ExtraShotDecorator {
        +cost()
        +description()
    }
    class MilkDecorator {
        +cost()
        +description()
    }
    Coffee <|.. BasicCoffee
    Coffee <|.. CoffeeDecorator
    CoffeeDecorator <|-- ExtraShotDecorator
    CoffeeDecorator <|-- MilkDecorator
```

## 6. Singleton Pattern

**Definition:** Ensures a class has only one instance and provides a global point of access to it.

**Real-World Analogy:** A country's government. There can only be one active government at a time, and everyone refers to that same government.

**UML Diagram:**

```
classDiagram
    class Government {
        -static instance: Government
        -Government()
```

```
        +static getInstance(): Government
        +makeDecision()
    }
```

## 7. Observer Pattern

**Definition:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Real-World Analogy:** Social media followers. When a celebrity posts something (subject), all followers (observers) get notified.

**UML Diagram:**

```
classDiagram
    class Subject {
        +attach(Observer)
        +detach(Observer)
        +notify()
    }
    class Observer {
        <>
        +update()
    }
    class Celebrity {
        -observers: List
        +post()
    }
    class Follower {
        +update()
    }
    Subject <|-- Celebrity
    Observer <|.. Follower
    Celebrity --> Observer
```

## 8. Model-View-Controller (MVC)

**Definition:** Separates an application into three interconnected components: Model (data), View (user interface), and Controller (business logic).

**Real-World Analogy:** A restaurant where the kitchen (Model) prepares food, the dining room (View) presents it to customers, and the waiter (Controller) coordinates between them.

**UML Diagram:**

```
classDiagram
    class Model {
        -data
        +getData()
```

```
        +updateData()
    }
    class View {
        +render()
        +updateDisplay()
    }
    class Controller {
        +handleInput()
        +updateModel()
        +refreshView()
    }
    Controller --> Model
    Controller --> View
    View ..> Model
```

---

## Summary

These design patterns represent proven solutions to common software design problems. Each pattern serves a specific purpose:

- **Strategy**: Flexible algorithm selection
- **Factory Method**: Object creation delegation
- **Abstract Factory**: Related object family creation
- **Façade**: Simplified interface to complex system
- **Decorator**: Dynamic feature addition
- **Singleton**: Single instance guarantee
- **Observer**: Event notification system
- **MVC**: Separation of concerns

Understanding these patterns helps in creating more maintainable, flexible, and robust software systems.