VGS2

OOP

Encapsulation:

- bundling the data (“attributes” or “fields”) and methods (behaviors)
- Essentially classes

Abstraction:

-Without abstraction, the user would have more decisions to make, is exposed to more information and complexity than is necessary to perform a task, and has to write more complex code. We can change the implementation details of EmailService without it affecting other classes

OOP

Inheritance:

- "is-a" relationship. Car is-a Vehicle, and a Bike is-a Vehicle:

Polymorphism:

- Poly = many Morph = forms
- polymorphism allows us to treat them all as instances of the base Vehicle class. The specific implementations of the Start() and Stop() methods for each vehicle type are invoked dynamically at runtime, based on the actual type of each vehicle.

OOP

Coupling:

- High coupling means that classes are tightly interconnected, making it difficult to modify or maintain them independently. Low coupling, on the other hand, indicates loose connections between classes, allowing for greater flexibility and ease of modification.

Composition:

- "has-a" relationship. the Car class is composed of an Engine, Wheels, a Chassis and Seats.

SOLID

Single Responsibility Principle:

- Single Responsibility Principle, we should separate these responsibilities into separate classes.

Open/Closed Principle (OCP) :

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.” public abstract class Shape

Liskov Substitution Principle (LSP):

- “Objects of a superclass should be replaceable with objects of its subclass without affecting the correctness of the program.

SOLID

Interface Segregation Principle (ISP):

- Clients should not be forced to depend on interfaces they do not use.
- example involving 2D and 3D shapes. Better make 2D and 3D separate interface instead of only interface shape

Dependency Inversion Principle (DIP):

- The Car class directly creates an instance of the Engine class, leading to a tight coupling between Car and Engine. - If the Engine class changes, it may affect the Car class, violating the Dependency Inversion Principle.
- To adhere to the Dependency Inversion Principle, we introduce an abstraction (interface) between Car and Engine, allowing Car to depend on an abstraction instead of a concrete implementation. public interface IEngine

Design Pattern

- Creational: the different ways to create objects.
- Structural: the relationships between those objects.
- Behavioral: the interaction or communication between those objects.

Design Patterns

Factory Method:

```
abstract class Product

{    public abstract void Use();
}

class ConcreteProductA : Product
{
   public override void Use() => Console.WriteLine("Using Product A");
}
abstract class Creator
{
   public abstract Product FactoryMethod();
}

class ConcreteCreatorA : Creator
{
   public override Product FactoryMethod() => new ConcreteProductA();
}
```

Abstract Factory

```
interface IButton
{
   void Click();
}
interface IGUIFactory
{
   IButton CreateButton();
}
class WindowsButton : IButton
{
   public void Click() => Console.WriteLine("Windows Button Clicked");
}
class WindowsFactory : IGUIFactory
{
   public IButton CreateButton() => new WindowsButton();

}
class Application
{
   private readonly IButton _button;

   public Application(IGUIFactory factory)
   {
      _button = factory.CreateButton();
 }

   public void Run()
   {
      _button.Click();
   }
```

Prototype

Copy of Object

```
// Prototype Interface
interface IPrototype<T>
{
   T Clone();
}
public Circle Clone()
   {
      return new Circle(Radius);
   }
```

Singleton

```
class Logger
{
    private static Logger _instance;
    private static readonly object _lock = new object();

    // Private constructor to prevent external instantiation
    private Logger() { }

    public static Logger GetInstance()
    {
        // Double-checked locking for thread safety
        if (_instance == null)
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new Logger();
                }
            }
        }
        return _instance;
    }
```

Adapter Pattern

The Adapter Pattern is used to enable incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that a client expects.

Example:

Let's say we have a LegacySystem that outputs text, but we want to use it in a system that expects an ILogger interface. We can adapt it using the Adapter Pattern.

Bridge Pattern

The Bridge Pattern is used to separate an abstraction from its implementation, allowing both to vary independently. This pattern is helpful when you want to decouple an abstraction from its implementation, making it easier to extend both.

Example:

Let's say we have an Abstraction for a Shape and we want to implement it for different platforms (Windows and Mac). We can use the Bridge Pattern to separate the shape's abstraction from its rendering implementation.

interface IDrawingAPI, class WindowsDrawingAPI : IDrawingAPI, abstract class Shape { protected IDrawingAPI _drawingAPI; class Circle : Shape

Composite Pattern

Composite Pattern (Structural)

The Composite Pattern is used to treat individual objects and compositions of objects uniformly. It allows you to compose objects into tree-like structures and work with them as though they were individual objects, which is especially useful for building hierarchies like file systems or UI components.

Example:

Let's implement a File System with the Composite Pattern, where both files and directories are treated uniformly.

```
using System;

using System.Collections.Generic;

// Component Interface

interface IFileSystemComponent

{

    void Display(string indent);

}
```

Decorator Pattern

The Decorator Pattern allows you to dynamically add behavior to an object at runtime without altering its structure. It is used to extend the functionalities of a class by wrapping it with a decorator class.

Example:

Let's implement a Coffee Shop scenario, where we have a basic Coffee object, and we can decorate it with additional features like milk or sugar.

Facade Pattern

The Facade Pattern provides a simplified interface to a complex subsystem. It's like a "front desk" — instead of interacting with multiple components directly, the client communicates with a single Facade, which internally handles the complexity.

```
class HomeTheaterFacade

{

    private readonly Amplifier _amp;

    private readonly TV _tv;

    private readonly Lights _lights;



    public void WatchMovie(string movie)

……
```

Flyweight Pattern

The Flyweight Pattern is used to minimize memory usage or object creation overhead by sharing common parts of state between multiple objects. It's particularly useful when you have a large number of similar objects, and many of them can share the same data.

Example:

Let’s implement a Text Editor scenario where characters (like 'A', 'B', 'C', etc.) are reused — only their position (extrinsic state) changes, while the character style (intrinsic state) is shared.

A dictionary of all characters, but the character is only created once its get called, before not

Proxy Pattern

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It’s like a gatekeeper — the proxy controls or manages the operations before passing them to the real object.

When to use the Proxy Pattern To restrict or control access to an object (e.g., based on role). To implement lazy loading (e.g., load heavy objects only when needed). When interacting with remote objects or expensive resources. To add extra behavior like logging or caching without modifying the original object.

```
class DocumentProxy : IDocument
{
    private RealDocument _realDocument;
    private readonly string _filename;
    private readonly string _userRole;
```

Chain of Responsibility

The Chain of Responsibility Pattern allows a request to be passed along a chain of handlers, where each handler decides whether to process the request or pass it to the next in the chain. var basic = new BasicSupport();

```
    var supervisor = new SupervisorSupport();

    var manager = new ManagerSupport();

    // Build the chain

    basic.SetNext(supervisor);

    supervisor.SetNext(manager);

    // Send requests of varying severity

    basic.HandleRequest(1); // Handled by Basic

    basic.HandleRequest(2); // Handled by Supervisor

    basic.HandleRequest(3); // Handled by Manager
```

Command Pattern

The Command Pattern turns a request into a stand-alone object, allowing you to parameterize methods, queue operations, log commands, and support undoable actions. It encapsulates a request as an object.

Example Scenario:

Let’s implement a Remote Control System where each button can be assigned a command like turning a light on or off.

Interpreter Pattern

Interpreter Pattern (Behavioral)

The Interpreter Pattern defines a grammar for a language and uses an interpreter to interpret sentences in that language. It's used to evaluate expressions, often in custom languages, rules engines, or math parsers.

Example Scenario:

We’ll create a simple math expression interpreter that can handle expressions like "5 + 3" or "10 - 2".

Iterator Pattern

The Iterator Pattern provides a way to access elements of a collection sequentially without exposing its underlying structure. It helps traverse data structures like lists, trees, or custom collections in a consistent way.

```
interface IIterator

{

    bool HasNext();

    Book Next();

}

        while (iterator.HasNext())

        {

            Book book = iterator.Next();

            Console.WriteLine($"- {book.Title}");

        }
```

Mediator Pattern

The Mediator Pattern centralizes communication between objects — instead of components referring to each other directly, they communicate via a mediator object. This helps reduce dependencies between classes and improves loose coupling.

Example Scenario:

Let’s simulate a Chat Room, where users send messages to each other via a central mediator, rather than directly.

Memento Pattern

memento Pattern (Behavioral)

The Memento Pattern lets you capture and restore an object’s internal state without exposing its internals. It’s used to implement features like undo/redo, checkpoints, or state snapshots.

Example Scenario:

Let’s build a Text Editor with undo functionality, where we can save and restore the editor's state using mementos.

Observer Pattern

The Observer Pattern allows an object (subject) to notify multiple observers (listeners) about state changes without knowing who or what those observers are. This pattern is ideal for event-driven systems where one event needs to trigger multiple actions.

State Pattern

The State Pattern allows an object to change its behavior when its internal state changes, appearing as if the object has changed its class. It’s useful for managing an object’s state transitions in an elegant way, such as in finite state machines.

Example Scenario:

Let’s create a traffic light system that behaves differently based on its current state (Red, Green, or Yellow).

Strategy pattern

The Strategy Pattern allows a client to choose an algorithm from a family of algorithms at runtime. The idea is to define a family of algorithms (strategies), encapsulate each one, and make them interchangeable. This pattern is useful when you want to select an algorithm at runtime based on different conditions.

Template pattern

The Template Method Pattern defines the skeleton of an algorithm in a method, allowing subclasses to provide specific implementations for certain steps without changing the overall structure of the algorithm. It lets you define a step-by-step algorithm while allowing for variability in certain parts of the process.

Example Scenario:

Let’s create a Cooking Process where the basic steps for preparing a dish are predefined, but subclasses can customize specific steps (like cooking time or ingredients).

Visitor Pattern

The Visitor Pattern lets you separate algorithms from the objects on which they operate. It allows you to add new operations to existing object structures without modifying their classes. This is particularly useful when you need to perform operations on objects of different types in a complex object structure.

Example Scenario:

Let’s consider a shopping cart system where we have different types of items (like books and electronics). We want to apply different discounts based on the type of item, but without modifying the classes for each item.

Microservices

Vertikales Scaling (Scaling Up):

Bedeutung: Erhöhen der Ressourcen eines einzelnen Servers, z. B. durch mehr CPU, RAM oder Festplattenspeicher.

Beispiel: Du fügst einem Server mehr Speicher oder eine leistungsstärkere CPU hinzu.

Vorteil: Einfach zu implementieren, da keine zusätzliche Infrastruktur benötigt wird.

Nachteil: Es gibt physikalische Grenzen, wie viel du einem einzelnen Server hinzufügen kannst.

Horizontales Scaling (Scaling Out):

Bedeutung: Hinzufügen weiterer Server zum System, um die Last auf mehrere Maschinen zu verteilen.

Beispiel: Du fügst zusätzliche Server zu einem Cluster hinzu, um mehr Anfragen zu bearbeiten.

Vorteil: Unbegrenzte Skalierbarkeit, da immer mehr Server hinzugefügt werden können.

Nachteil: Komplexere Implementierung, da Daten und Prozesse zwischen Servern synchronisiert werden müssen.

Zusammenfassung:

Vertikal = größere Maschine (mehr Leistung pro Maschine).

Horizontal = mehr Maschinen (Lastverteilung).

Indexe in SQL

In SQL sind Indexe spezielle Datenstrukturen, die verwendet werden, um den Zugriff auf Daten in einer Datenbank schneller und effizienter zu gestalten. Sie werden häufig auf Tabellenkolonnen erstellt, die häufig in Abfragen verwendet werden, besonders bei WHERE-Klauseln, JOINs oder ORDER BY-Anweisungen.

Wesentliche Merkmale von Indexen:

Schnellere Abfragen: Indexe ermöglichen schnellere Leseoperationen (SELECT-Abfragen), indem sie die Menge an Daten reduzieren, die durchsucht werden muss.

Ähnlich wie ein Buch-Index: Ein Index in einer Datenbank funktioniert ähnlich wie der Index in einem Buch, der es ermöglicht, schnell die Seite zu finden, auf der ein bestimmtes Thema besprochen wird.

Verbrauch von Speicherplatz: Während Indexe die Abfragegeschwindigkeit verbessern, benötigen sie zusätzlichen Speicherplatz und können die Schreiboperationen (wie INSERT, UPDATE, DELETE) verlangsamen, da der Index bei jeder Änderung aktualisiert werden muss.