SOFTWARE DESIGN PATTERNS

WHAT IS DESIGN PATTERN?

It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system

WHAT ALL ARE THE TYPES OF DESIGN PATTENRS?

Creational	Structural	Behavioral
• Abstract factory	• Adapter	• Iterator
• Builder	• Bridge	• Mediator
• Factory method	• Composite	• Observer
• Singleton	• Decorator	• Visitor
• Prototype	• Facade	• Strategy
• Lazy initialization	• Proxy	• Template Method
• Multiton	• Repository	• State

QUESTIONS

1. What are Design Patterns?
2. Name types of Design Patterns?

1.
SINGLETON

Ensure a class has only one instance, and provide a global point of access to it.

SINGLETON

ADVANTAGES	DISADVANTAGES
Controlled access to sole instance	Global objects can be harmful to object programming
Reduced name space	Complicates writing unit tests and following TDD
Flexibility	

SINGLETON. APPLICABILITY.

Use the Singleton pattern when..

1. there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
2. the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

SINGLETON. IMPLEMENTATION.

```
1   public sealed class Singleton
2   {
3       private static readonly Lazy<Singleton> lazy =
4           new Lazy<Singleton>(() => new Singleton());
5
6       public static Singleton Instance { get { return lazy.Value; } }
7
8       private Singleton()
9       {
10      }
11  }
```

EXAMPLES IN .NET FRAMEWORK

- System.Threading.TimerQueue.Instance
- SqlClientFactory.Instance
- Thread.CurrentThread
- AppDomain.CurrentDomain
- SynchronizationContext.Current
- log4net.LogManager.GetLogger

2.
ABSTRACT FACTORY

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

ABSTRACT FACTORY

ADVANTAGES	DISADVANTAGES
It isolates concrete classes	Supporting new kinds of products is difficult
It makes exchanging product families easy	
It promotes consistency among products	

ABSTRACT FACTORY. APPLICABILITY.

Use the Abstract factory pattern when..

1. a system should be independent of how its products are created, composed, and represented.
2. a system should be configured with one of multiple families of products.
3. a family of related product objects is designed to be used together, and you need to enforce this constraint.
4. you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

ABSTRACT FACTORY. IMPLEMENTATION.

```
interface IAbstractFactory
{
    IAbstractProductA CreateProductA();
    IAbstractProductB CreateProductB();
}
class ConcreteFactory1: IAbstractFactory
{
    public IAbstractProductA CreateProductA() => new ProductA1();
    public IAbstractProductB CreateProductB() => new ProductB1();
}
class ConcreteFactory2: IAbstractFactory
{
    public IAbstractProductA CreateProductA() => new ProductA2();
    public IAbstractProductB CreateProductB() => new ProductB2();
}

interface IAbstractProductA { }
interface IAbstractProductB { }
```

```
class ProductA1: IAbstractProductA { }
class ProductB1: IAbstractProductB { }
class ProductA2: IAbstractProductA { }
class ProductB2: IAbstractProductB { }

class Client
{
    private IAbstractProductA abstractProductA;
    private IAbstractProductB abstractProductB;
    public Client(IAbstractFactory factory)
    {
        abstractProductB = factory.CreateProductB();
        abstractProductA = factory.CreateProductA();
    }
    public void Run() { }
}
```

EXAMPLES IN .NET FRAMEWORK

- DbProviderFactory from ADO.NET
- CodeDomProvider
- SymetricAlgorithm

3.
BUILDER

Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

BUILDER

ADVANTAGES	DISADVANTAGES
It lets you vary a product's internal representation	Requires creating a separate ConcreteBuilder for each different type of product
It isolates code for construction and representation	Data members of class aren't guaranteed to be initialized
It gives you finer control over the construction process	Dependency injection may be less supported

BUILDER. APPLICABILITY.

Use the Builder pattern when..

1. the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

2. the construction process must allow different representations for the object that's constructed.

BUILDER. IMPLEMENTATION.

```
class Client
{
    void Main()
    {
        IBuilder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.Construct();
        Product product = builder.GetResult();
    }
}
class Director
{
    IBuilder builder;
    public Director(IBuilder builder) => this.builder = builder;
    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}
```

```
interface IBuilder
{
    void BuildPartA();
    void BuildPartB();
    void BuildPartC();
    Product GetResult();
}

class Product
{
    List<object> parts = new List<object>();
    public void Add(string part) => parts.Add(part);
}

class ConcreteBuilder : IBuilder
{
    Product product = new Product();
    public void BuildPartA() => product.Add("Part A");
    public void BuildPartB() => product.Add("Part B");
    public void BuildPartC() => product.Add("Part C");
    public Product GetResult() => product;
}
```

EXAMPLES IN .NET FRAMEWORK

- StringBuilder
- UriBuilder
- DbCommandBuilder
- DbConnectionStringBuilder
- DispatcherBuilder
- EndpointAddressBuilder

QUESTIONS

1. What is Abstract factory pattern?
2. What is Singleton pattern?
3. What is Builder pattern?

4.
ADAPTER

Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.

ADAPTER

ADVANTAGES	DISADVANTAGES
Helps achieve reusability and flexibility	Sometimes many adaptations are required along an adapter chain to reach the type which is required
It absolutely interconnects two incompatible interfaces	It unnecessarily increases the size of the code as class inheritance is less used and lot of code is needlessly duplicated between classes
Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations	

ADAPTER. APPLICABILITY.

Use the Adapter pattern when..

1. you want to use an existing class, and its interface does not match the one you need.

2. you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

ADAPTER. IMPLEMENTATION.

```
class Client
{
    public void Request(Target target) => target.Request();
}

class Target
{
    public virtual void Request() { }
}

class Adapter : Target
{
    private Adaptee adaptee = new Adaptee();
    public override void Request() => adaptee.SpecificRequest();
}

class Adaptee
{
    public void SpecificRequest() { }
}
```

EXAMPLES IN .NET FRAMEWORK

- TextReader/TextWriter
- BinaryReader/BinaryWriter
- System.Data.SqlClient.SqlDataAdapter
- ReadonlyCollection<T>

5.
DECORATOR

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

DECORATOR

ADVANTAGES	DISADVANTAGES
More flexibility than static inheritance	A decorator and its component aren't identical
Avoids feature-laden classes high up in the hierarchy	Lots of little objects

DECORATOR. APPLICABILITY.

Use the Decorator pattern when..

1. to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
2. for responsibilities that can be withdrawn.
3. extension by subclassing is impractical.

DECORATOR. IMPLEMENTATION.

```
interface IComponent
{
    void Operation();
}
class ConcreteComponent : IComponent
{
    public void Operation() { }
}
abstract class Decorator : IComponent
{
    protected IComponent component;
    public void SetComponent(IComponent component) => this.component = component;
    public override void Operation()
    {
        if (component != null)
            component.Operation();
    }
}
class ConcreteDecoratorA : Decorator
{
    public override void Operation() => base.Operation();
}
class ConcreteDecoratorB : Decorator
{
    public override void Operation() => base.Operation();
}
```

EXAMPLES IN .NET FRAMEWORK

- System.IO.BufferedStream
- System.IO.Compression.GZipStream
- System.CodeDom.Compiler.IndentedTextWriter
- System.Collections.SortedList.SyncSortedList

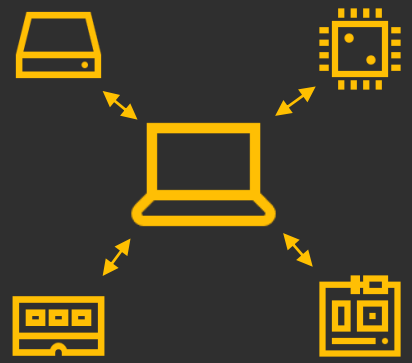
6.
FACADE

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

FACADE

ADVANTAGES	DISADVANTAGES
It shields clients from subsystem components	May make your code base bigger
It promotes weak coupling between the subsystem and its clients.	
It simplifies porting systems to other platforms	

FACADE. APPLICABILITY.



Use the Facade pattern when..

1. you want to provide a simple interface to a complex subsystem.
2. there are many dependencies between clients and the implementation classes of an abstraction.
3. you want to layer your subsystems.

FACADE. IMPLEMENTATION.

```
public class Facade
{
    SubsystemA subsystemA;
    SubsystemB subsystemB;
    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc)
    {
        subsystemA = sa;
        subsystemB = sb;
        subsystemC = sc;
    }
    public void Operation1()
    {
        subsystemA.A1();
        subsystemB.B1();
        subsystemC.C1();
    }
    public void Operation2()
    {
        subsystemB.B1();
        subsystemC.C1();
    }
}
```

```
class SubsystemA
{
    public void A1() { }
}
class SubsystemB
{
    public void B1() { }
}
class SubsystemC
{
    public void C1() { }
}
class Client
{
    public void Main()
    {
        Facade facade = new Facade(new SubsystemA(), new SubsystemB(), new SubsystemC());
        facade.Operation1();
        facade.Operation2();
    }
}
```

EXAMPLES IN .NET FRAMEWORK

- XmlSerializer
- ThreadPool.QueueUserWorkItem
- Parallel
- System.Runtime.CompilerServices.RuntimeHelper
- System.Console

QUESTIONS

1. What is Adapter pattern?
2. What is Decorator pattern?
3. What is Facade pattern?

7.
STRATEGY

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

STRATEGY

ADVANTAGES	DISADVANTAGES
Families of related algorithms	Clients must be aware of different Strategies
An alternative to subclassing	Increased number of objects
Strategies eliminate conditional statements	

STRATEGY. APPLICABILITY.

Use the Strategy pattern when..

1. many related classes differ only in their behavior.
2. you need different variants of an algorithm.
3. an algorithm uses data that clients shouldn't know about.
4. a class defines many behaviors, and these appear as multipleconditional statements in its operations.

STRATEGY. IMPLEMENTATION.

```
1   public interface IStrategy
2   {
3       void Algorithm();
4   }
5
6   public class ConcreteStrategy1 : IStrategy
7   {
8       public void Algorithm() { }
9   }
10
11  public class ConcreteStrategy2 : IStrategy
12  {
13      public void Algorithm() { }
14  }
15
16  public class Context
17  {
18      public IStrategy ContextStrategy { get; set; }
19      public Context(IStrategy _strategy) => ContextStrategy = _strategy;
20      public void ExecuteAlgorithm() => ContextStrategy.Algorithm();
21  }
```

EXAMPLES IN .NET FRAMEWORK

- IComparer<T>
- IEqualityComparer<T>
- IErrorHandler (WCF)
- IDispatchMessageFormatter (WCF)
- MessageFilter (WCF)

8.
MEDIATOR

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.

MEDIATOR

ADVANTAGES	DISADVANTAGES
It limits subclassing	It centralizes control
It decouples colleagues	
It simplifies object protocols	

MEDIATOR. APPLICABILITY.

Use the Mediator pattern when..

1. a set of objects communicate in well-defined but complex ways.
2. reusing an object is difficult because it refers to and communicates with many other objects.
3. a behavior that's distributed between several classes should be customizable without a lot of subclassing.

MEDIATOR. IMPLEMENTATION.

```
interface IMediator
{
    void Send(string msg, Colleague colleague);
}

abstract class Colleague
{
    protected IMediator mediator;
    public Colleague(IMediator mediator) => this.mediator = mediator;
}

class ConcreteColleague1 : Colleague
{
    public ConcreteColleague1(IMediator mediator) : base(mediator) { }
    public void Send(string message) => mediator.Send(message, this);
    public void Notify(string message) { }
}
```

```
class ConcreteColleague2 : Colleague
{
    public ConcreteColleague2(IMediator mediator) : base(mediator) { }
    public void Send(string message) => mediator.Send(message, this);
    public void Notify(string message) { }
}

class ConcreteMediator : IMediator
{
    public ConcreteColleague1 Colleague1 { get; set; }
    public ConcreteColleague2 Colleague2 { get; set; }
    public void Send(string msg, Colleague colleague)
    {
        if (Colleague1 == colleague)
            Colleague2.Notify(msg);
        else
            Colleague1.Notify(msg);
    }
}
```

EXAMPLES IN .NET FRAMEWORK

- EventAggregator
- BlockingCollection
- Any forms in WinForms

9.
OBSERVER

Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

OBSERVER

ADVANTAGES	DISADVANTAGES
Abstract coupling between Subject and Observer	Unexpected updates
Support for broadcast communication	

OBSERVER. APPLICABILITY.

Use the Observer pattern when..

1. an abstraction has two aspects, one dependent on the other.
2. a change to one object requires changing others, and you don't know how many objects need to be changed.
3. an object should be able to notify other objects without making assumptions about who these objects are

OBSERVER. IMPLEMENTATION.

```
interface IObservable
{
    void AddObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}

class ConcreteObservable : IObservable
{
    private List<IObserver> observers;
    public ConcreteObservable() => observers = new List<IObserver>();
    public void AddObserver(IObserver o) => observers.Add(o);
    public void RemoveObserver(IObserver o) => observers.Remove(o);
    public void NotifyObservers() => observers.ForEach(observer => observer.Update());
}

interface IObserver
{
    void Update();
}

class ConcreteObserver :IObserver
{
    public void Update() { }
}
```

EXAMPLES IN .NET FRAMEWORK

- EventHandler<T>
- IEventProcessor
- AppDomainSetup.AppDomainInitializer,
  HttpConfiguration.Initializer

QUESTIONS

1. What is Strategy pattern?
2. What is Mediator pattern?
3. What is Observer pattern?