Frequently asked and likely interview questions on design principles and patterns in .NET for a candidate with 8+ years of experience, along with accurate and detailed answers:

**### 1. What are the SOLID principles? Explain each one briefly.**

\*\*Answer:\*\*

The SOLID principles are a set of design principles for object-oriented programming that make systems more understandable, flexible, and maintainable:

1. \*\*Single Responsibility Principle (SRP):\*\* A class should have only one reason to change, meaning it should have only one job or responsibility.

2. \*\*Open/Closed Principle (OCP):\*\* Software entities (classes, modules, functions) should be open for extension but closed for modification.

3. \*\*Liskov Substitution Principle (LSP):\*\* Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

4. \*\*Interface Segregation Principle (ISP):\*\* No client should be forced to depend on methods it does not use. This means interfaces should be small and specific to each client.

5. \*\*Dependency Inversion Principle (DIP):\*\* High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

**### 2. Can you explain the Factory Design Pattern and provide an example in .NET?**

\*\*Answer:\*\*

The Factory Design Pattern is a creational pattern that provides a way to create objects without specifying the exact class of the object that will be created. This pattern defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

\*\*Example in .NET:\*\*

```csharp

public interface IProduct

{

void DoSomething();

}

public class ConcreteProductA : IProduct

{

public void DoSomething()

{

Console.WriteLine("Product A");

}

}

public class ConcreteProductB : IProduct

{

public void DoSomething()

{

Console.WriteLine("Product B");

}

}

public class ProductFactory

{

public static IProduct CreateProduct(string type)

{

if (type == "A")

return new ConcreteProductA();

else if (type == "B")

return new ConcreteProductB();

else

throw new ArgumentException("Invalid type");

}

}

// Usage

var productA = ProductFactory.CreateProduct("A");

productA.DoSomething();

```

**### 3. What is the Singleton Design Pattern and how do you implement it in .NET?**

\*\*Answer:\*\*

The Singleton Design Pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is useful when exactly one object is needed to coordinate actions across the system.

\*\*Implementation in .NET:\*\*

```csharp

public class Singleton

{

private static Singleton \_instance;

private static readonly object \_lock = new object();

private Singleton() { }

public static Singleton Instance

{

get

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Singleton();

}

return \_instance;

}

}

}

public void DoSomething()

{

Console.WriteLine("Singleton instance");

}

}

// Usage

var singleton = Singleton.Instance;

singleton.DoSomething();

```

**### 4. Explain the Dependency Injection (DI) pattern and its advantages.**

\*\*Answer:\*\*

Dependency Injection (DI) is a design pattern used to implement IoC (Inversion of Control). It allows a class to receive its dependencies from an external source rather than creating them itself.

\*\*Advantages:\*\*

- \*\*Decoupling:\*\* Reduces the dependency between classes by injecting dependencies from outside.

- \*\*Testability:\*\* Makes it easier to test classes in isolation by injecting mock dependencies.

- \*\*Maintainability:\*\* Simplifies code maintenance by managing dependencies centrally.

- \*\*Flexibility:\*\* Allows changing implementations without modifying the dependent class.

\*\*Example in .NET using a DI container:\*\*

```csharp

public interface IService

{

void Serve();

}

public class Service : IService

{

public void Serve()

{

Console.WriteLine("Service Called");

}

}

public class Client

{

private readonly IService \_service;

public Client(IService service)

{

\_service = service;

}

public void Start()

{

\_service.Serve();

}

}

// Setting up DI in .NET Core

public class Program

{

public static void Main(string[] args)

{

var serviceProvider = new ServiceCollection()

.AddTransient<IService, Service>()

.BuildServiceProvider();

var client = new Client(serviceProvider.GetService<IService>());

client.Start();

}

}

```

**### 5. What is the Repository Pattern and how is it implemented in .NET?**

\*\*Answer:\*\*

The Repository Pattern is used to encapsulate the logic required to access data sources. It provides a central place to interact with the data source, thus abstracting the data access logic from the business logic.

\*\*Implementation in .NET:\*\*

```csharp

public interface IRepository<T>

{

IEnumerable<T> GetAll();

T GetById(int id);

void Add(T entity);

void Update(T entity);

void Delete(int id);

}

public class Repository<T> : IRepository<T> where T : class

{

private readonly DbContext \_context;

private readonly DbSet<T> \_dbSet;

public Repository(DbContext context)

{

\_context = context;

\_dbSet = context.Set<T>();

}

public IEnumerable<T> GetAll()

{

return \_dbSet.ToList();

}

public T GetById(int id)

{

return \_dbSet.Find(id);

}

public void Add(T entity)

{

\_dbSet.Add(entity);

\_context.SaveChanges();

}

public void Update(T entity)

{

\_context.Entry(entity).State = EntityState.Modified;

\_context.SaveChanges();

}

public void Delete(int id)

{

T entity = \_dbSet.Find(id);

\_dbSet.Remove(entity);

\_context.SaveChanges();

}

}

// Usage example

public class Product { public int Id { get; set; } public string Name { get; set; } }

public class ProductRepository : Repository<Product>

{

public ProductRepository(DbContext context) : base(context) { }

}

```

**### 6. What is the difference between Abstract Factory and Factory Method patterns?**

\*\*Answer:\*\*

- \*\*Factory Method Pattern:\*\* Defines an interface for creating an object but lets subclasses decide which class to instantiate. It relies on inheritance and the subclass's ability to override the method that creates the object.

- \*\*Abstract Factory Pattern:\*\* Provides an interface for creating families of related or dependent objects without specifying their concrete classes. It involves a higher level of abstraction compared to Factory Method and typically uses composition to achieve object creation.

\*\*Factory Method Example:\*\*

```csharp

public abstract class Creator

{

public abstract IProduct FactoryMethod();

}

public class ConcreteCreatorA : Creator

{

public override IProduct FactoryMethod()

{

return new ConcreteProductA();

}

}

```

\*\*Abstract Factory Example:\*\*

```csharp

public interface IAbstractFactory

{

IProductA CreateProductA();

IProductB CreateProductB();

}

public class ConcreteFactory1 : IAbstractFactory

{

public IProductA CreateProductA()

{

return new ProductA1();

}

public IProductB CreateProductB()

{

return new ProductB1();

}

}

```

**### 7. How do you implement the Observer Pattern in .NET?**

\*\*Answer:\*\*

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

\*\*Implementation in .NET:\*\*

```csharp

public interface IObserver

{

void Update();

}

public interface ISubject

{

void Attach(IObserver observer);

void Detach(IObserver observer);

void Notify();

}

public class ConcreteSubject : ISubject

{

private readonly List<IObserver> \_observers = new List<IObserver>();

public void Attach(IObserver observer)

{

\_observers.Add(observer);

}

public void Detach(IObserver observer)

{

\_observers.Remove(observer);

}

public void Notify()

{

foreach (var observer in \_observers)

{

observer.Update();

}

}

public string SubjectState { get; set; }

}

public class ConcreteObserver : IObserver

{

private readonly ConcreteSubject \_subject;

public ConcreteObserver(ConcreteSubject subject)

{

\_subject = subject;

}

public void Update()

{

Console.WriteLine("Observer Updated with state: " + \_subject.SubjectState);

}

}

// Usage

var subject = new ConcreteSubject();

var observer1 = new ConcreteObserver(subject);

var observer2 = new ConcreteObserver(subject);

subject.Attach(observer1);

subject.Attach(observer2);

subject.SubjectState = "State 1";

subject.Notify();

```