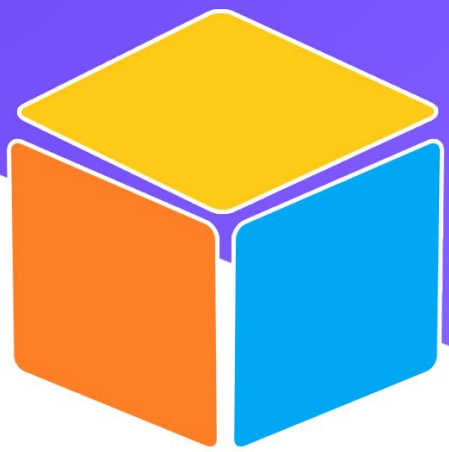
.NET Design Patterns Interview Questions & Answers



By Shailendra Chauhan

Microsoft MVP, Founder & CEO - Dot Net Tricks

DotNetTricks

.NET Design Patterns Interview Questions and Answers

Release History

- Initial Release 1.0 - 6th Sep 2021

About Dot Net Tricks

Dot Net Tricks is founded by Shailendra Chauhan (Microsoft MVP), in Jan 2010. Dot Net Tricks came into existence in the form of a blog post over various technologies including .NET, C#, SQL Server, ASP.NET, ASP.NET MVC, JavaScript, Angular, Node.js and Visual Studio, etc.

The company which is currently registered by a name of Dot Net Tricks Innovation Pvt. Ltd. came into shape in 2015. Dot Net Tricks website has an average footfall on the tune of 300k+ per month. The site has become a cornerstone when it comes to getting skilled-up on .NET technologies and we want to gain the same level of trust in other technologies. This is what we are striving for.

We have a very large number of trainees who have received training from our platforms and immediately got placement in some of the reputed firms testifying our claims of providing quality training. The website offers you a variety of free study material in the form of articles.

Unlimited Live Training Membership

Upgrade your skills set with hands-on real-time project-based training programs to build expertise on in-demand job skills and become industry competent. DotNetTricks Unlimited Live Training enables you to Become:

- **Full-stack JavaScript Developer** - JavaScript, Node.js, React, Angular
- **Full-stack .NET Developer** - .NET, MVC, ASP.NET Core, WebAPI
- **Cloud Engineer/Architect** - AWS, Microsoft Azure
- **Technical Architect** - Microservices, Design Patterns and Clean Architecture
- **DevOps Engineer** - DevOps, Docker and Kubernetes
- **Mobile Developer** - Xamarin, React Native

Learn more about Unlimited Live here: https://www.dotnettricks.com/membership

Self-Paced Training Membership

The most effective way to gain job-ready expertise for your career. Learn how to build highly scalable modern web applications & transform your coding skills. Learn to build projects and building expertise on .NET, JavaScript, Database, Cloud, DevOps, Docker, Front-end technologies and many more cutting-edge technologies which are in industry demand today.

- 5,00+ hours of unlimited access to our premium content
- Real Hands-on Labs with integrated IDE
- Full access to training, study mode quizzes, and assignments
- Step-by-step learning with exclusive Learning Paths
- Become job-ready with interview prep sessions

Learn more about Self-paced training membership here: https://www.dotnettricks.com/plus-membership

Interview Q&A eBooks

Dot Net Tricks offer a wide range of eBooks on technical interviews Q&A. industry experts and coaches write all eBooks. These eBooks will help you to prepare yourself for your next job within a short time. We offer the eBooks in the following categories:

- .NET Development
- Front-end Development
- Cloud
- DevOps
- Programming Languages
- Database - SQL and NoSQL
- Mobile Development
- ML/AI and many more...

For other eBooks, do refer to https://www.dotnettricks.com/books

Corporate Training

Dot Net Tricks has a pool of mentors who help the corporation enhance their employment skills by changing the technology landscape. Dot Net Tricks offers customized training programs for new hires and experienced employees through online and classroom mode. As a trusted and resourceful training partner, Dot Net Tricks helps the corporation achieve success with its industry-leading instructional design and customer training initiatives.

Apart from these, we also provide on-demand boot camps and personalized project consultation.

For more details about Corporate Training, do refer to https://www.dotnettricks.com/corporate-training

Technical Recruiting

We provide a full technical staffing service, which suits our client's needs. Our specialized recruiters search worldwide to find highly skilled professionals that will fit our client's needs. If you are looking for a job change, do share your resume at hr@dotnettricks.com. Dot Net Tricks will help you to find your dream job in MNCs.

Join us today, learn to code, prepare yourself for interviews, and get hired!

Dedication

This book is dedicated to my mentor Tushar Singhal who taught me programming with ease and made me a programmer. I would like to say thanks to all my family members, friends, students or followers of my articles at www.dotnettricks.com to encourage me to write this book.

-Shailendra Chauhan

Introduction

Are you preparing yourself for the .NET Design patterns interview? you are at the right place. In this book, you will get the most asked interview questions with their answers. This book will tell you exactly what you'll be asked, and how to answer them. So, get ready to crack your design pattern interview.

So, what where the author qualifications to write this book?

Shailendra Chauhan is a technical consultant, corporate trainer and Microsoft MVP having more than 12 years. Of development. He provides training and consultation over an array of technologies like Cloud, .NET, Design Patterns, Microservices, Angular, React, Node and Mobile Apps development.

So, the next question is who this book is for?

This book is best suited for beginners and professionals. It is intended for anyone who is looking to make a career .NET Architect or .Net Solution Architect. This book covers the main following topics.

- Design Patterns and Principles
- GoF Design Patterns
- Creation Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns
- Anti Patterns
- Patterns of Enterprise Application Architecture
- Code Organizational Patterns

This book is not only for interview preparation, but it also helps you to learn and implement design patterns in your real projects. This book helps you to get an in-depth knowledge of design patterns simply and elegantly.

All the best for your interview and happy programming!

About the Author

Shailendra Chauhan - An Entrepreneur, Author, Architect, Corporate Trainer, and Microsoft MVP

He is the **Founder and CEO** of DotNetTricks which is a brand when it comes to e-Learning. DotNetTricks provides training and consultation over an array of technologies like **.NET, Angular, React, Node, Cloud, Microservices, DevOps, and Mobile Apps development**. He has been awarded as **Microsoft MVP** six times in a row (2016-2021).

He has changed many lives from his writings and unique training programs. He has a number of most sought-after books to his name which have helped job aspirants in **cracking tough interviews** with ease.

Moreover, and to his credit, he has delivered **2000+ training sessions** to professionals worldwide in Microsoft .NET technologies and other technologies including JavaScript, AngularJS, Node.js, React and NoSQL Databases. In addition, he provides **Instructor-led online training**, **hands-on workshop** and **corporate training** programs.

Shailendra has a strong combination of **technical skills and solution development for complex application architecture with proven leadership and motivational skills** that have elevated him to a world-renowned status, placing him at the top of the list of most sought-after trainers.

"I always keep up with new technologies and learning new skills to deliver the best to my students," says Shailendra Chauhan, he goes on to acknowledge that the betterment of his followers and enabling his students to realize their goals are his prime objective and a great source of motivation and satisfaction.

Shailendra Chauhan - **"Follow me and you too will have the key that opens the door to success"**

How to Contact Us

Although the author of this book has tried to make this book as accurate as it possible but if there is something that strikes you as odd, or you find an error in the book please drop a line via e-mail.

The e-mail addresses are listed as follows:

- mentor@dotnettricks.com
- info@dotnettricks.com

We are always happy to hear from our readers. Please provide your valuable feedback and comments!

You can follow us on YouTube, Facebook, Twitter, Linked In and Google Plus or subscribe to RSS feed.

Table of Contents

1

Design Patterns and Principles

Q1. What are Software Design Principles?

Ans. Software design principles are a set of guidelines that helps developers to make a good software system design. The key software design principles are as:

- SOLID
- DRY
- KISS
- YAGNI

Q2. What is SOLID?

Ans. SOLID is a combination of five fundamental designing principles:

- **S**ingle Responsibility Principle (SRP)
- **O**pen/Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Q3. What is Single Responsibility Principle (SRP)?

Ans. This principle states that a class should have one and only one responsibility. There should not be more than one reason for a class to change. SRP makes the classes compact and neat where each one is responsible for a single problem, task, or concern. For example,

Explanation- In the above example, each class is responsible for handling one responsibility.

```
public class Membership
{
        public void Add()
```

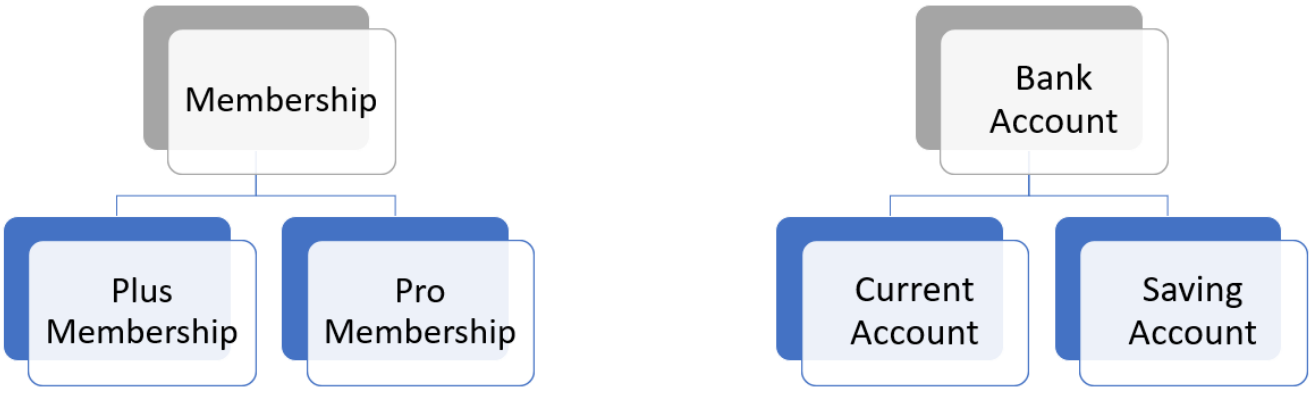
```
        {
            try
            {
                //TO DO:
            }
            catch (Exception ex)
            {
                //File.WriteAllText(@"c:\Error.txt", ex.Message);
                var logger = new FileLogger();
                logger.LogError(ex.Message);
            }
        }
}

public class FileLogger // SRP: Created a new class for error logging
{
        public void LogError(string error)
        {
            File.WriteAllText(@"c:\Error.txt", error);
        }
}
```

Q4. What is Open/Closed Principle (OCP)?

Ans. This principle states that a class should be open for extension but closed for modification.

The "closed" part of the rule states that once a class has been developed and tested, the code should only be changed to correct bugs.

The "open" part says that you should be able to extend an existing class to introduce new functionalities. In this way, you need to test only the newly created class. For example,



Explanation - A *BankAccount* base class contains all basic payment-related properties and methods. This class can be extended by different Account classes to achieve their functionalities. Hence it is open for extension but closed for modification.

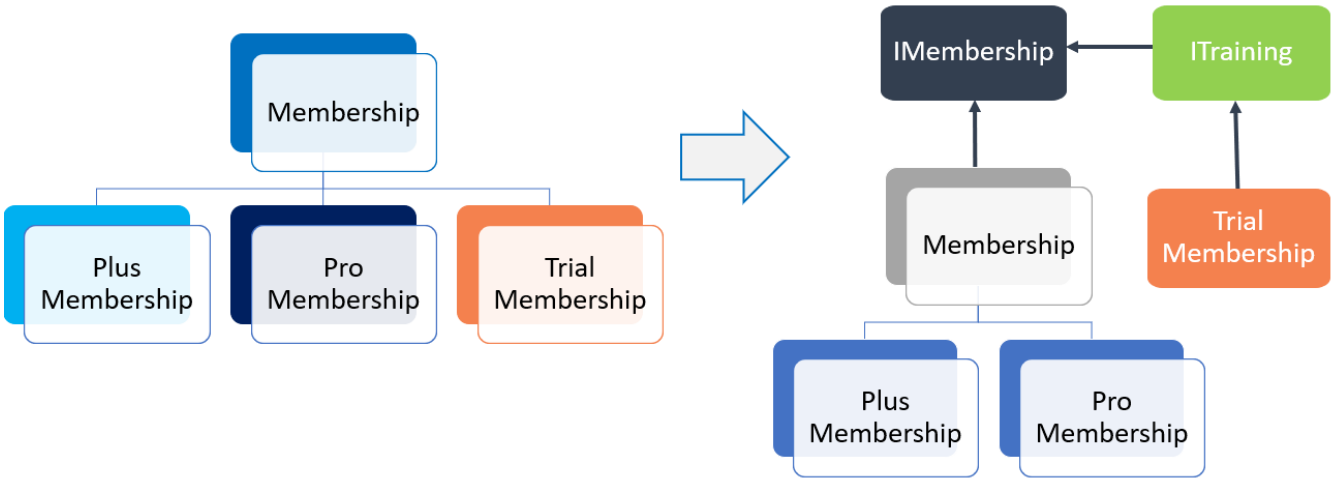
```
public class Membership
{
       //public int MembershipType { get; set; }
```

```
        public virtual int GetTraining()
        {
            return 2;
            //if (MembershipType == 1) //for Plus
            //    return 5;
            //else if (MembershipType == 2) //for Pro
            //    return 10;
            //else
            //    return 2;
        }

        public void Add()
        {
            //TO DO
        }
}
//OCP: new classes are created by inheritance
public class PlusMembership: Membership
{
        public override int GetTraining()
        {
            return 5;
        }
}

public class ProMembership: Membership //OCP
{
        public override int GetTraining()
        {
            return 10;
        }
}
```

Q5. What is Liskov Substitution Principle (LSP)?

Ans. This principle states that derived classes must be able to substitute any object for their base classes.

In simple language, objects of a parent class can be replaced with objects of its derived classes without breaking the code. To apply this principle, the behaviour of your classes becomes more important than its structure.



Example - Assume that you have an inheritance hierarchy for membership plans. Wherever you can use Membership, you should also be able to use a Plus Membership and Pro Membership, because both are subclasses of Membership.

```
public interface ITraining
{
    int GetTraining();
}

public interface IMembership : ITraining
{
    void Add();
}

public class Membership : IMembership
{
    public virtual int GetTraining()
    {
        return 2;
    }

    public virtual void Add()
    {
        // TO DO:
    }
}

public class PlusMembership : Membership
{
    public override int GetTraining()
    {
        return 5;
    }
}

public class ProMembership: Membership
{
    public override int GetTraining()
    {
        return 10;
    }
}

// LISKOV ITraining and IMembership interface created
public class TrialMembership: ITraining
{
    public int GetTraining()
    {
        return 2;
    }
}

//public class TrialMembership: Membership
//{
//    public override void Add()
//    {
```

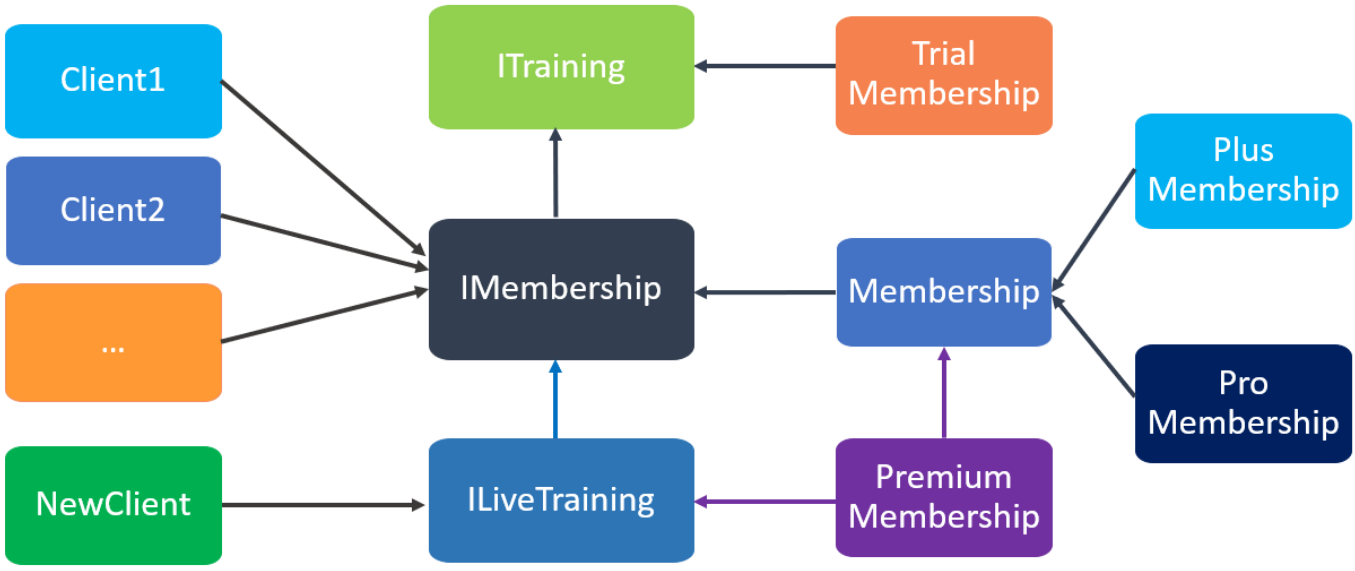
```
//        throw new NotImplementedException("Trial Membership Can't be added");
//    }
//    public override int GetTraining()
//    {
//        return 2;
//    }
//}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("LSP Principle!");

        List<Membership> membershipList = new List<Membership>();
        membershipList.Add(new PlusMembership());
        membershipList.Add(new ProMembership());

        //membershipList.Add(new TrialMembership());

        foreach (var item in membershipList)
        {
            item.Add();
        }
    }
}
```

Q6. What is Interface Segregation Principle (ISP)?

Ans. This principle states that clients of your class should not be forced to depend on methods they do not use.

Similar to the SRP, the goal of the ISP is to reduce the side effects and frequency of required changes by splitting the code into multiple, independent parts.

Example - The service interface that is exposed to the client should contain only client related methods not all.




```
public interface ITraining
{
    int GetTraining();
    //int GetLiveTraining(); //problem
}
public interface IMembership: ITraining
{
    void Add();
}
public interface ILiveTraining: IMembership
{
    int GetLiveTraining();
}

public class Membership: IMembership, ILiveTraining
{
    public virtual int GetTraining()
    {
        return 2;
    }

    public int GetLiveTraining()
    {
        return 5;
    }

    public virtual void Add()
    {
        // TO DO:
    }
}

public class PlusMembership: Membership
{
    public override int GetTraining()
    {
```

```
            return 5;
        }
    }

    public class ProMembership: Membership
    {
        public override int GetTraining()
        {
            return 10;
        }
    }

    // LISKOV ITraining and IMembership interface created
    public class TrialMembership: ITraining
    {
        public int GetTraining()
        {
            return 2;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("ISP Principle!");

            //old clients with self-paced training
            IMembership membership = new Membership();
            membership.Add();

            //new clients with live training + self-paced
            ILiveTraining membershipLive = new Membership();
            membershipLive.Add();
            membershipLive.GetLiveTraining();
        }
    }
```

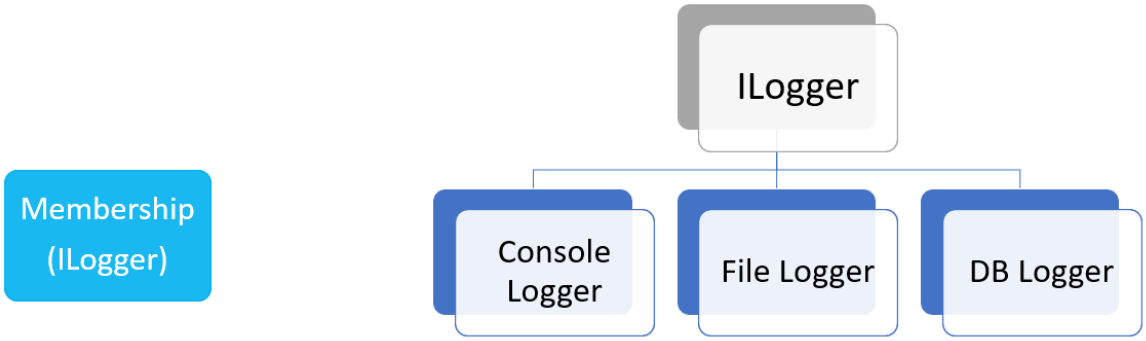
Q7. What is Dependency Inversion Principle (DIP)?

Ans. This principle states that high-level modules (concrete classes) should not depend on low-level modules (concrete classes). Instead, they should depend on abstract classes or interfaces.

This principle splits the dependency between the high-level and low-level modules by introducing abstraction between them.

In other words, the high-level module depends on the abstraction and the low-level module also depends on the same abstraction.

The Dependency Injection pattern is an implementation of this principle




```
public class Membership
{
    private ILogger logger;
    public Membership(ILogger _logger)
    {
        //TO DO: read from app.config
        //int config = 1;
        //if(config==1)
        //{
        //    logger = new FileLogger();
        //}
        //else
        //{
        //    logger = new ConsoleLogger();
        //}

        logger = _logger;
    }

    public void Add()
    {
        try
        {
            //TO DO:
        }
        catch (Exception ex)
        {
            logger.LogError(ex.Message);
        }
    }
}
public interface ILogger
{
    void LogError(string error);
}
public class FileLogger : ILogger
{
    public void LogError(string error)
    {
        File.WriteAllText(@"c:\Error.txt", error);
    }
}
public class ConsoleLogger : ILogger
{
```

```
        public void LogError(string error)
        {
            Console.WriteLine($"Error: {error}");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("DIP Principle!");

            FileLogger fileLogger = new FileLogger();
            Membership member1 = new Membership(fileLogger);

            ConsoleLogger consoleLogger = new ConsoleLogger();
            Membership member2 = new Membership(consoleLogger);
        }
    }
```

Q8. What is DRY principle?

Ans. **DRY** stand for **Don't Repeat Yourself**. This principle states that each small piece of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code.

Example – ASP.NET MVC framework works on this principle.

Q9. What is KISS principle?

Ans. **KISS** stands for **Keep it simple, Stupid**! This principle states that try to keep each small piece of software simple and unnecessary complexity should be avoided. This helps us to write easily maintainable code.

Q10. What is YAGNI principle?

Ans. **YAGNI** stands for **You ain't gonna need it**. This principle states that always implement things when you need them never implements things before you need them.

Q11. What are Design Patterns?

Ans. Design patterns provide solutions to **common problems** which **occur** in **software design**. Design patterns are about reusable designs and interactions of objects. These can be organized into 4 separate pattern groups depending on the nature of the design problem they intend to solve.

1. Gang of Four Patterns
2. Enterprise Patterns
3. SOA and Messaging Patterns
4. Model-View Patterns

Q12. What are the advantages of Design Patterns?

Ans. Design Patterns have the following main advantages in software development.

1. Provide solutions to common problems which occur in software design.
2. Provide a common platform for developers means the developer can implement these in any language.

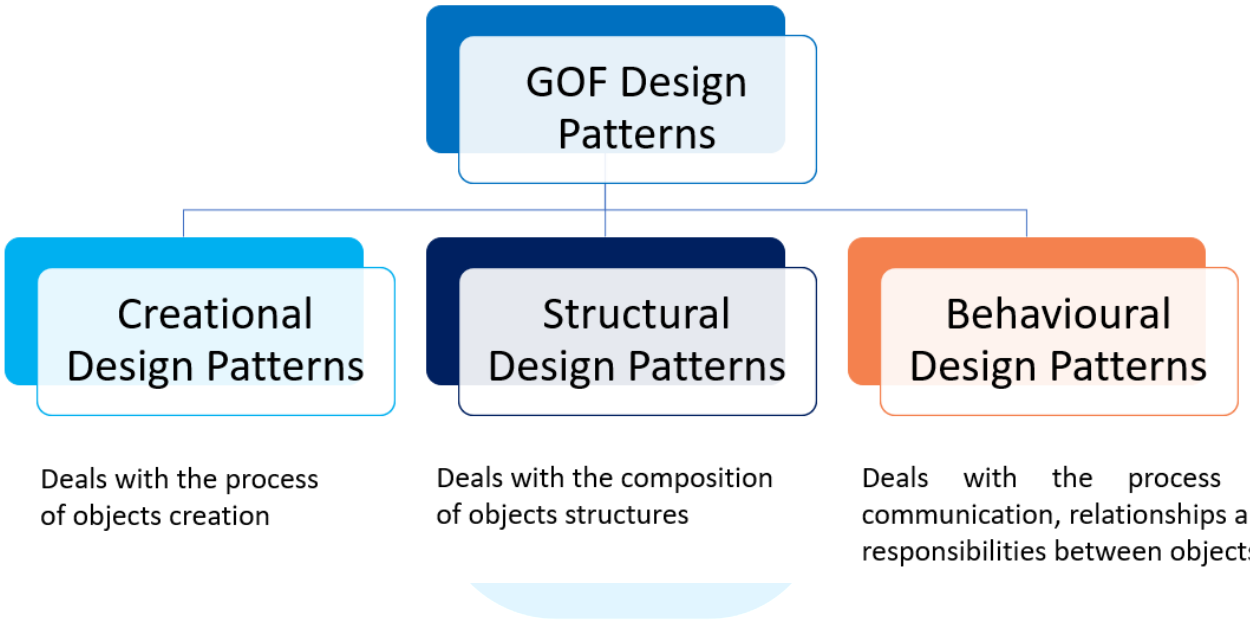
3. Provide a standard terminology and are specific to a particular scenario.

Q13. What is Gang of Four (GOF)?

Ans. In 1994, four authors **Erich** Gamma, **Richard** Helm, **Ralph** Johnson and **John** Vlissides published a book (***Design Patterns: Elements of Reusable Object-Oriented Software***) for explaining the concept of Design Pattern in Software development. These four authors are collectively known as Gang of Four (GOF).

Q14. What are GOF Design Patterns?

Ans. The **23 Design patterns** are defined by the **Gang of Four** programmers. These **23 patterns** are divided into three groups depending on the nature of the design problem they intend to solve.



Q15. What are Creational Design Patterns?

Ans. These patterns deal with the process of objects creations in such a way that they can be decoupled from their implementing system. This provides more flexibility in deciding which objects need to be created for a given use case/ scenario. There are as follows:

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton

Q16. What are Structural Design Patterns?

Ans. These patterns deal with the composition of objects structures. The concept of inheritance is used to compose interfaces and define various ways to compose objects for obtaining new functionalities. There are as follows:

1. Adapter

2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Q17. What are Behavioural Design Patterns?

Ans. These patterns deal with the process of communication, managing relationships, and responsibilities between objects. They are as follows:

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Visitor
11. Template Method

2

Creational Design Patterns

Q1. What is Factory Method pattern? Explain it with an example?

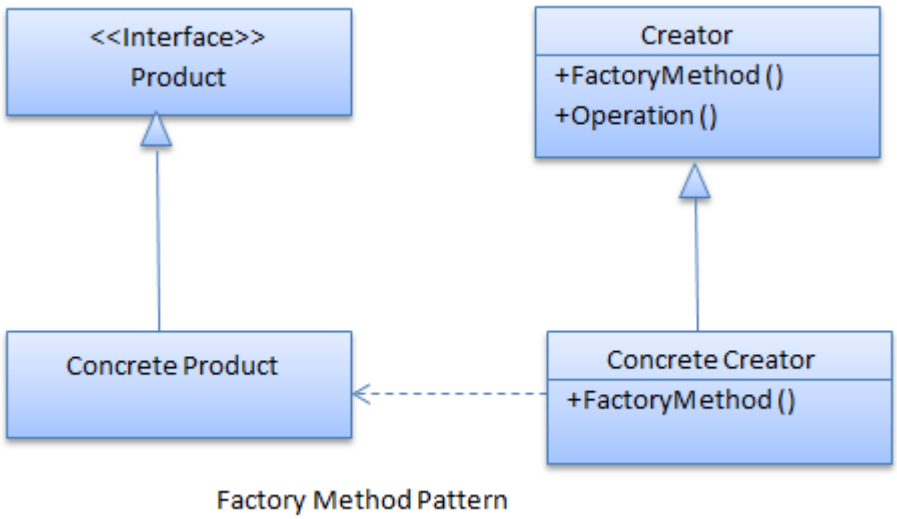
Ans. In Factory pattern, we create an object without exposing the creation logic. In this pattern, an interface is used for creating an object, but let the subclass decide which class to instantiate.

The creation of an object is done when it is required. The Factory method allows a class later instantiation to subclasses. The examples of Factory methods are given below:

1. Need to create different logger types like:
   - Console Logger
   - Database Logger
   - File Logger etc.
2. Need to create different report types like:
   - PDF
   - Word etc.

Q2. Explain Factory method pattern with UML diagram and code example?

Ans. The UML diagram for the factory design pattern is shown below:



Factory Method Pattern

The classes, interfaces and objects in the above UML class diagram are defined as follows:

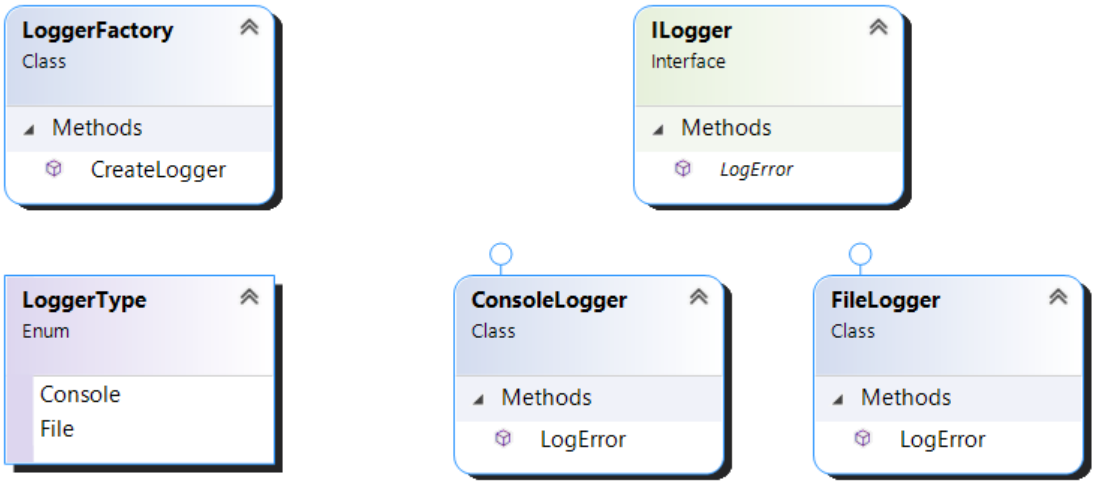
1. **Product -** This is an interface for creating the objects.
2. **ConcreteProduct -** This is a class that implements the Product interface.
3. **Creator -** This is an abstract class and declares the factory method, which returns an object of type Product.
4. **ConcreteCreator -** This is a class that implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

C# Implementation Code

```
interface Product
{
    //To DO:
}

class ConcreteProductA : Product
{
    //To DO:
}

class ConcreteProductB : Product
{
    //To DO:
}

abstract class Creator
{
    public abstract Product FactoryMethod(string type);
}

class ConcreteCreator : Creator
{
    public override Product FactoryMethod(string type)
    {
        switch (type)
        {
            case "A": return new ConcreteProductA();
            case "B": return new ConcreteProductB();
            default: throw new ArgumentException("Invalid type", "type");
        }
    }
}
```

Q3. Explain Factory method pattern with a real-life example?

Ans. Let's take the following example where you can create a logger by passing logger type.



C# Implementation Code

```
public class LoggerFactory
{
        public static ILogger CreateLogger(LoggerType type)
        {
            switch (type)
            {
                case LoggerType.File:
                    return new FileLogger();
                case LoggerType.Console:
                default:
                    return new ConsoleLogger();
            }
        }
}
public enum LoggerType
{
        Console,
        File
}

public interface ILogger
{
        void LogError(string error);
}

public class ConsoleLogger : ILogger
{
        public void LogError(string error)
        {
            Console.WriteLine($"Console Log: {error}");
        }
```

```
}
public class FileLogger : ILogger
{
        public void LogError(string error)
        {
            File.WriteAllText(@"c:\Error.txt", error);
        }
}

public class Client
{
        ILogger _logger;
        public Client(ILogger logger)
        {
            _logger = logger;
        }
        public void Add()
        {
            try
            {
                // TO DO:

            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }
        }
}
class Program
{
        static void Main(string[] args)
        {
            ILogger logger = LoggerFactory.CreateLogger(LoggerType.Console);
            Client client = new Client(logger);

            client.Add();
        }
}
```

Q4. When to use Factory method pattern?

Ans. The factory pattern is useful in the following cases:

- Subclasses figure out what objects should be created.
- Parent class allows later instantiation to subclasses means the creation of an object is done when it is required.
- The process of objects creation is required to centralize within the application.
- A class (creator) will not know what classes it will be required to create.

Q5. What is Abstract Factory pattern? Explain it with an example?

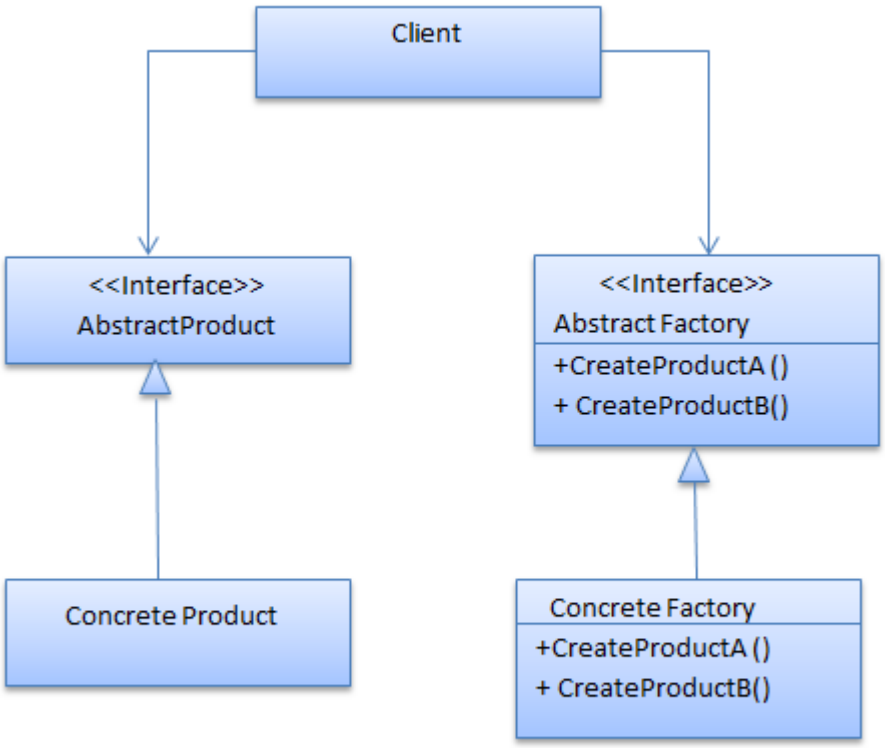
Ans. Abstract Factory patterns act as a super-factory that creates other factories. This pattern is also called a **Factory of factories**. In the Abstract Factory pattern, an interface is responsible for creating a factory of related objects, or dependent objects without specifying their concrete classes.

For example, need to support multiple database types

- SQL Server
- Oracle
- MySQL etc.

Q6. Explain Abstract factory pattern with UML diagram and code example?

Ans. The UML diagram for the Abstract factory pattern is shown below:



Abstract Factory Pattern

The classes, interfaces and objects in the above UML class diagram are defined as follows:

1. **AbstractFactory -** This is an interface that is used to create an abstract product
2. **ConcreteFactory -** This is a class that implements the AbstractFactory interface to create concrete products.
3. **AbstractProduct -** This is an interface that declares a type of product.
4. **ConcreteProduct -** This is a class that implements the AbstractProduct interface to create a product.

5. **Client -** This is a class that use AbstractFactory and AbstractProduct interfaces to create a family of related objects.

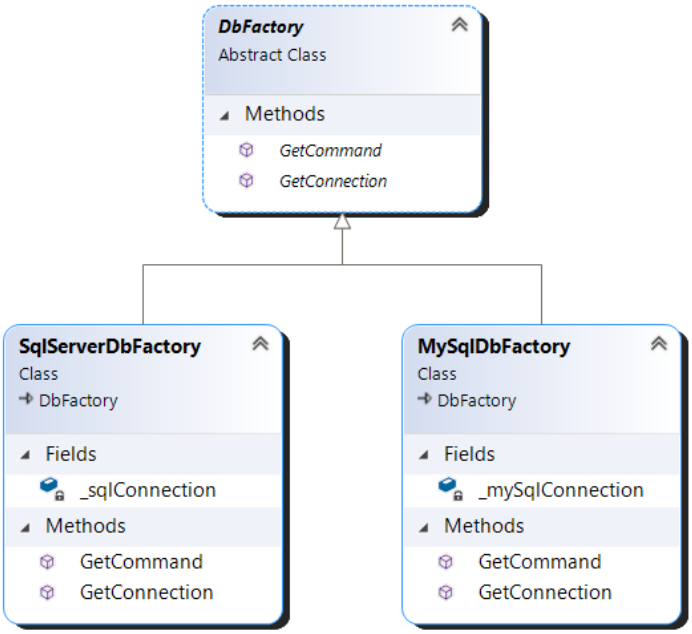
C# - Implementation Code:

```
public interface AbstractFactory
{
    AbstractProductA CreateProductA();

    AbstractProductB CreateProductB();
}

public class ConcreteFactoryA : AbstractFactory
{
    public AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}

public class ConcreteFactoryB : AbstractFactory
{
    public AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }

    public AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}

public interface AbstractProductA { }

public class ProductA1: AbstractProductA { }

public class ProductA2: AbstractProductA { }

public interface AbstractProductB { }

public class ProductB1: AbstractProductB { }

public class ProductB2: AbstractProductB { }

public class Client
{
    private AbstractProductA _productA;
    private AbstractProductB _productB;
```

```
    public Client(AbstractFactory factory)
    {
        _productA = factory.CreateProductA();
        _productB = factory.CreateProductB();
    }
}
```

Q7. Explain Abstract Factory pattern with a real-life example?

Ans. Let's take the following example where you need to create a database factory to handle database connection with SQL Server and MySQL Database.



C# - Implementation Code

```
public abstract class DbFactory
{
        public abstract DbConnection GetConnection();
        public abstract DbCommand GetCommand();
}

public class SqlServerDbFactory : DbFactory
{
        private SqlConnection _sqlConnection;
        public override DbCommand GetCommand()
        {
            SqlCommand command = new SqlCommand();
            command.Connection = _sqlConnection;
            return command;
        }

        public override DbConnection GetConnection()
```

```
        {
            string strCon = @"data source=Shailendra\SqlExpress; initial catalog=MyDB;persist
security info=True;user id=sa;password=dotnettricks;";
            _sqlConnection = new SqlConnection(strCon);
            return _sqlConnection;
        }
}
public class MySqlDbFactory : DbFactory
{
        private MySqlConnection _mySqlConnection;
        public override DbCommand GetCommand()
        {
            MySqlCommand command = new MySqlCommand();
            command.Connection = _mySqlConnection;
            return command;
        }

        public override DbConnection GetConnection()
        {
            string strCon = @"server=localhost;user id=root;password=root;database=MyDB";
            _mySqlConnection = new MySqlConnection(strCon);
            return _mySqlConnection;
        }
}

public class Client
{
        private DbFactory _dbFactory;
        public Client(DbFactory dbFactory)
        {
            _dbFactory = dbFactory;
        }
        public void Add()
        {
            // TO DO:
            var connection = _dbFactory.GetConnection();
            var command = _dbFactory.GetCommand();

            //command.CommandText = "Select * from Users";
            //connection.Open();
            //var reader = command.ExecuteReader();
            //reader.Close();
            //connection.Close();
        }
}
class Program
{
        static void Main(string[] args)
        {
            SqlServerDbFactory sqlFactory = new SqlServerDbFactory();
            Client client = new Client(sqlFactory);

            client.Add();
        }
}
```

Q8. When to use Abstract Factory pattern?

Ans. The abstract factory pattern is helpful in the following cases:

1. Create a set of related objects or dependent objects which must be used together.
2. System should be configured to work with multiple families of products.
3. The creation of objects should be independent from the utilizing system.
4. Concrete classes should be decoupled from clients.

Q9. What others design patterns can be used with Abstract Factory pattern?

1. Internally, Abstract Factory use Factory design pattern for creating objects. But it can also use Builder design pattern and prototype design pattern for creating objects. It completely depends upon your implementation for creating objects.
2. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
3. When Abstract Factory, Builder, and Prototype define a factory for creating the objects, we should consider the following points:
    - Abstract Factory use the factory for creating objects of several classes.
    - Builder uses the factory for creating a complex object by using simple objects and a step-by-step approach.
    - Prototype uses the factory for building an object by copying an existing object.

Q10. What is the difference between Factory Pattern & Abstract Factory Pattern?

Ans. The differences between Factory Pattern and Abstract Factory Pattern are given below:

Factory Pattern	Abstract Factory Pattern
Use inheritance and relies on a concrete class to create an object.	Use composition to delegate the responsibility of object creation to another class
Produce only one product	Produce a family of related products
Abstract Factory may use Singleton design pattern for creating objects.	Abstract Factory may use Factory design pattern for creating objects. It can also use Builder design pattern and prototype design pattern for creating a product. It completely depends upon your implementation for creating products.
Hides the creation process of a single object	Hides the creation process of a family of related objects

Q11. What is Singleton pattern? Explain it with an example?

Ans. Singleton pattern is one of the simplest design patterns. This pattern ensures that a class has only one instance and provides a global point of access to it.

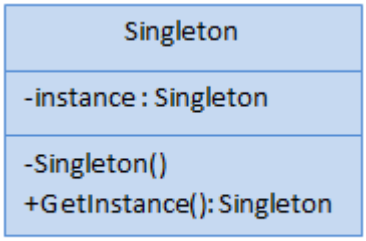
A singleton class includes:

- Static private Instance

- Private Constructor
- Public Instance Property or GetInstance() method

Q12. Explain Singleton Design pattern with UML diagram and code example?

Ans. The UML diagram for Singleton design pattern is shown below:



Singleton Pattern

The classes and objects in the above UML class diagram are as follows:

Singleton - This is a class that is responsible for creating and maintaining its own unique instance.

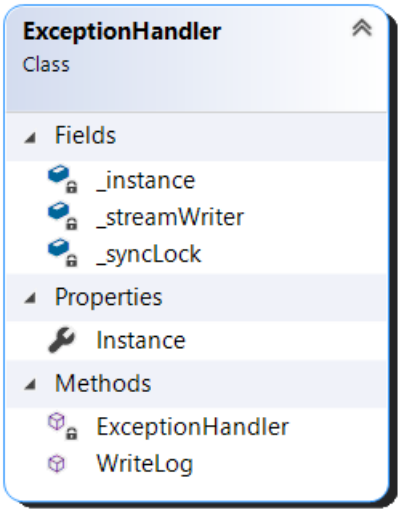
C# - Implementation Code:

```
//eager initialization of singleton
public class Singleton
{
    private static Singleton instance = new Singleton();
    private Singleton() { }

    public static Singleton GetInstance
    {
        get
        {
            return instance;
        }
    }
}
//lazy initialization of singleton
public class Singleton
{
    private static Singleton instance = null;
    private Singleton() { }

    public static Singleton GetInstance
    {
        get
        {
            if (instance == null)
                instance = new Singleton();

            return instance;
        }
    }
}
//Thread-safe (Double-checked Locking) initialization of singleton
```

```
public class Singleton
{
    private static Singleton instance = null;
    private Singleton() { }
    private static object lockThis = new object();

    public static Singleton GetInstance
    {
        get
        {
            lock (lockThis)
            {
                if (instance == null)
                    instance = new Singleton();

                return instance;
            }
        }
    }
}
```

Q13. Explain Singleton pattern with a real-life example?

Ans. Let's take the following example where we need to create a singleton object of ExceptionHandler class to handle exceptions throughout the application.



C# - Implementation Code:

```
public class ExceptionHandler
{
        // .NET guarantees thread safety for static initialization
        private static ExceptionHandler _instance = null;
        // Lock synchronization object
        private static object _syncLock = new object();
        private static StreamWriter _streamWriter;

        private ExceptionHandler()
```

```
        {
            _streamWriter = new StreamWriter(@"c:\Error.txt");
        }

        public static ExceptionHandler Instance
        {
            get
            {
                // Support multi threaded applications through
                // 'Double checked locking' pattern which (once
                // the instance exists) avoids locking each
                // time the method is invoked
                lock (_syncLock)
                {
                    if (_instance == null)
                        _instance = new ExceptionHandler();

                    return _instance;
                }
            }
        }

        public void WriteLog(string message)
        {
            _streamWriter.WriteLine(message);
            _streamWriter.Flush();
        }
}

public class Client
{
        public void Add()
        {
            try
            {
                // TO DO:
            }
            catch (Exception ex)
            {
                ExceptionHandler.Instance.WriteLog(ex.Message);
            }
        }
}
class Program
{
        static void Main(string[] args)
        {
            Client client = new Client();

            client.Add();
        }
}
```

Q14. When to use Singleton pattern?

Ans. Need a single instance of an object throughout the application. For example:

- Exception Logging
- Database Manager

Q15. What is Builder pattern? Explain it with an Example?

Ans. Builder pattern builds a complex object by using a step-by-step approach. Builder interface defines the steps to build the final object. This builder is independent from the object's creation process. A class that is known as Director, controls the object creation process.

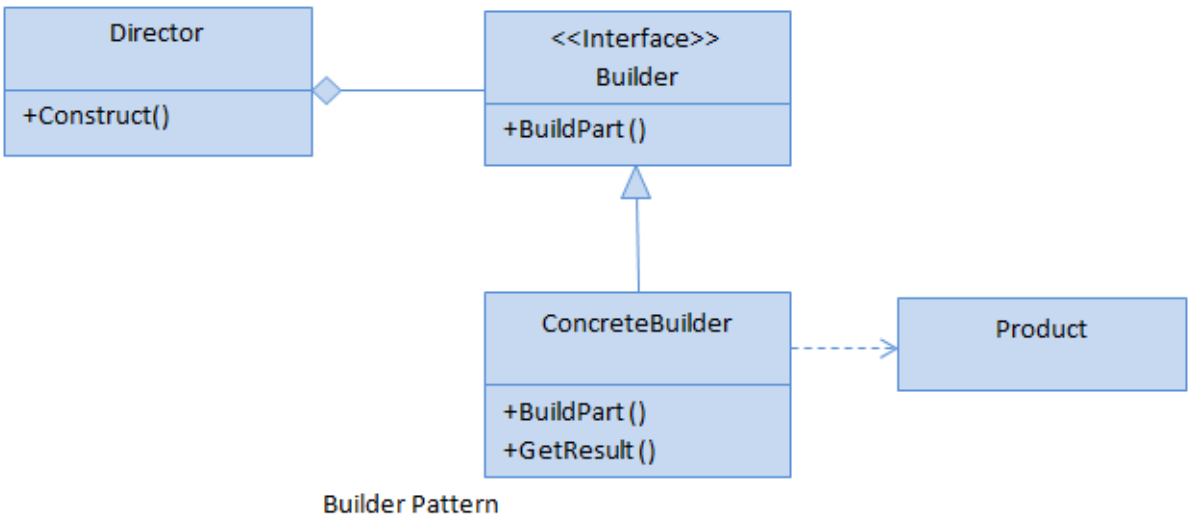
Moreover, the builder pattern describes a way to separate an object from its construction. The same construction method can create a different representation of the object.

For example, Construct Report/Email Builder Class with:

- Set Header
- Set Body
- Set Footer

Q16. Explain Builder Design pattern with UML diagram and code example?

Ans. The UML diagram for Builder design pattern is shown below:



Builder Pattern

The classes, interfaces and objects in the above UML class diagram are defined as follows:

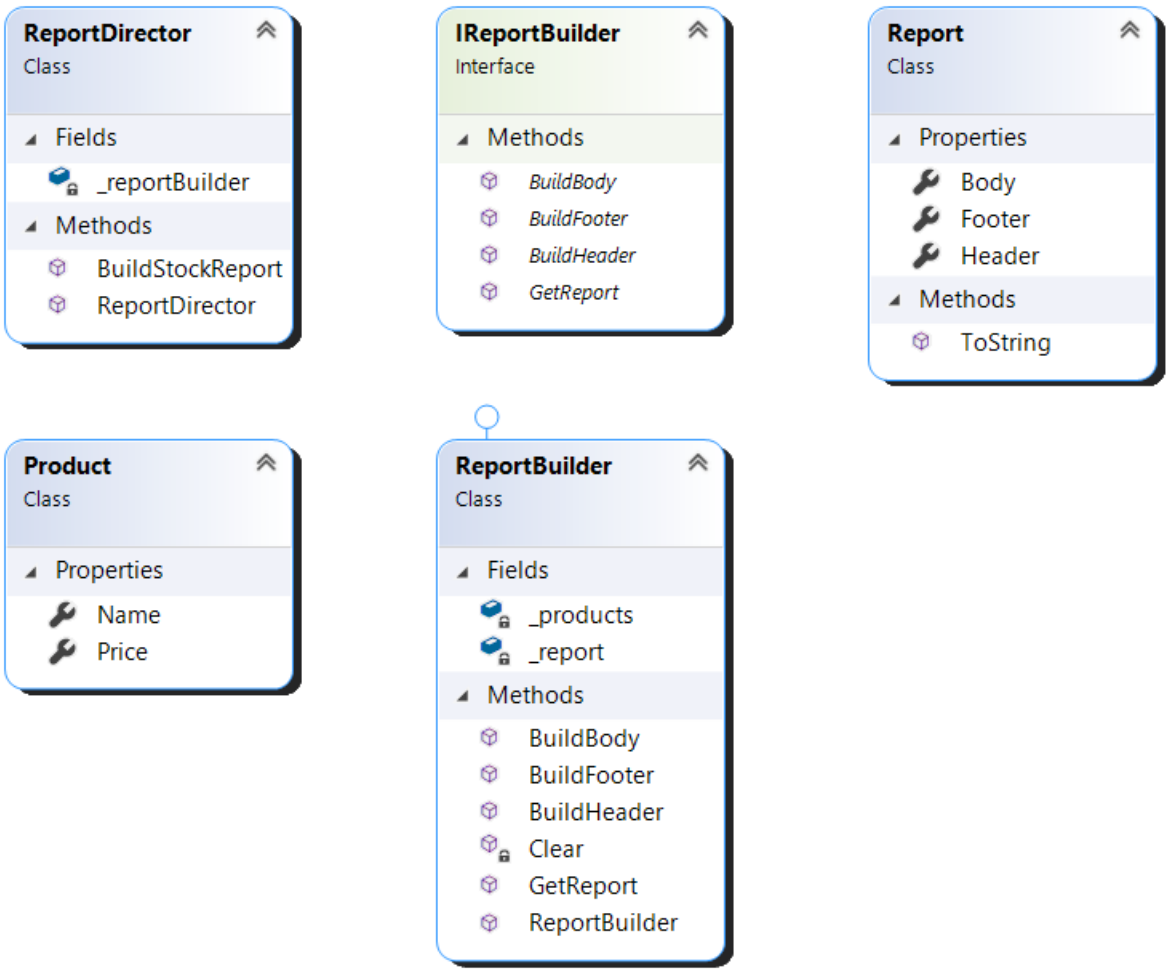
1. **Builder -** This is an interface that is used to define all the steps to create a product
2. **ConcreteBuilder -** This is a class that implements the Builder interface to create a complex product.
3. **Product -** This is a class that defines the parts of the complex object which are to be generated by the builder pattern.
4. **Director -** This is a class that is used to construct an object using the Builder interface.

C# Implementation Code:

```
public interface IBuilder
{
    void BuildPart1();
    void BuildPart2();
    void BuildPart3();
    Product GetProduct();
}

public class ConcreteBuilder : IBuilder
{
    private Product _product = new Product();

    public void BuildPart1()
    {
        _product.Part1 = "Part 1";
    }

    public void BuildPart2()
    {
        _product.Part2 = "Part 2";
    }

    public void BuildPart3()
    {
        _product.Part3 = "Part 3";
    }

    public Product GetProduct()
    {
        return _product;
    }
}

public class Product
{
    public string Part1 { get; set; }
    public string Part2 { get; set; }
    public string Part3 { get; set; }
}

public class Director
{
    public void Construct(IBuilder IBuilder)
    {
        IBuilder.BuildPart1();
        IBuilder.BuildPart2();
        IBuilder.BuildPart3();
    }
}
```

Q17. Explain Builder pattern with a real-life example?

Ans. Let's take the following example where we need a report builder to generate reports.



C# Implementation Code:

```
public class ReportDirector
{
        private readonly IReportBuilder _reportBuilder;
        public ReportDirector(IReportBuilder reportBuilder)
        {
            _reportBuilder = reportBuilder;
        }
        public void BuildStockReport()
        {
            _reportBuilder.BuildHeader();
            _reportBuilder.BuildBody();
            _reportBuilder.BuildFooter();
        }
}

public interface IReportBuilder
{
        void BuildHeader();
        void BuildBody();
        void BuildFooter();
```

```
        public Report GetReport();
}
public class ReportBuilder : IReportBuilder
{
        private Report _report;
        private IEnumerable<Product> _products;
        public ReportBuilder(IEnumerable<Product> products)
        {
            _products = products;
            _report = new Report();
        }
        public void BuildHeader()
        {
            _report.Header = $"REPORT FOR ALL THE PRODUCTS ON DATE: {DateTime.Now}\n";
        }
        public void BuildBody()
        {
            _report.Body = string.Join(Environment.NewLine, _products.Select(p => $"Product
name: {p.Name}, product price: {p.Price}"));
        }
        public void BuildFooter()
        {
            _report.Footer = "\nReport provided by the IT_PRODUCTS company.";
        }
        public Report GetReport()
        {
            var report = _report;
            Clear();
            return report;
        }
        private void Clear() => _report = new Report();
}
public class Product
{
        public string Name { get; set; }
        public double Price { get; set; }
}
public class Report
{
        public string Header { get; set; }
        public string Body { get; set; }
        public string Footer { get; set; }
        public override string ToString() =>
            new StringBuilder()
            .AppendLine(Header)
            .AppendLine(Body)
            .AppendLine(Footer)
            .ToString();
}
class Program
{
        static void Main(string[] args)
        {
            var products = new List<Product>
        {
            new Product { Name = "Monitor", Price = 200.50 },
            new Product { Name = "Mouse", Price = 20.41 },
            new Product { Name = "Keyboard", Price = 30.15}
```

```
        };
            var builder = new ReportBuilder(products);
            var director = new ReportDirector(builder);
            director.BuildStockReport();
            var report = builder.GetReport();

            Console.WriteLine(report);
        }
}
```

Q18. When to use Builder pattern?

Ans. The abstract factory pattern is helpful in the following cases:

- Need to create an object in several steps (a step-by-step approach).
- The creation of objects should be independent from the way the object's parts are assembled.
- Runtime control over the creation process is required.

Q19. What is the difference between Abstract Factory and Builder pattern?

Ans. Differences between Abstract Factory and Builder pattern are given below:

Abstract Factory Pattern	Builder Pattern
Abstract Factory Pattern will return the Instance directly.	This pattern creates the object in several steps.
It does not keep the track of its created Object.	It keeps the reference of its created Object.

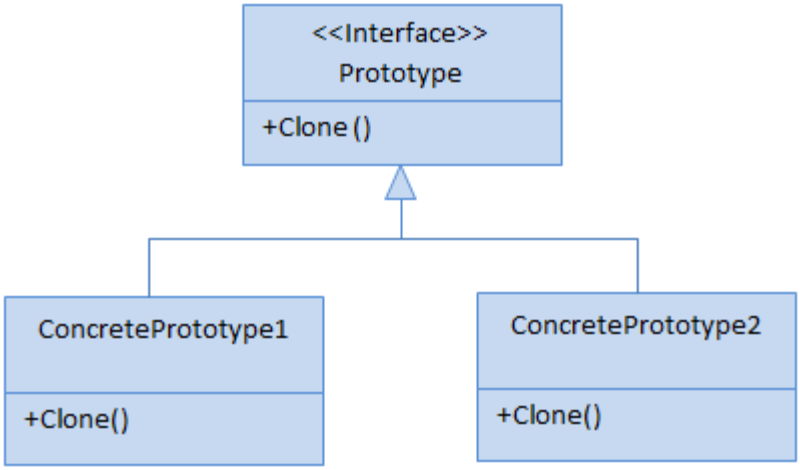
Q20. What is Prototype pattern? Explain it with an example?

Ans. Prototype pattern is used to create a duplicate object or clone of the current object to enhance performance. This pattern is used when the creation of an object is costly or complex.

For Example: An object is to be created after a costly database operation. We can cache the object, returns its clone on the next request and update the database as and when needed thus reducing database calls.

Q21. Explain Prototype Design pattern with UML diagram and code example?

Ans. The UML diagram for Prototype design pattern is shown below:



Prototype Pattern

The classes, interfaces and objects in the above UML class diagram are defined as follows:

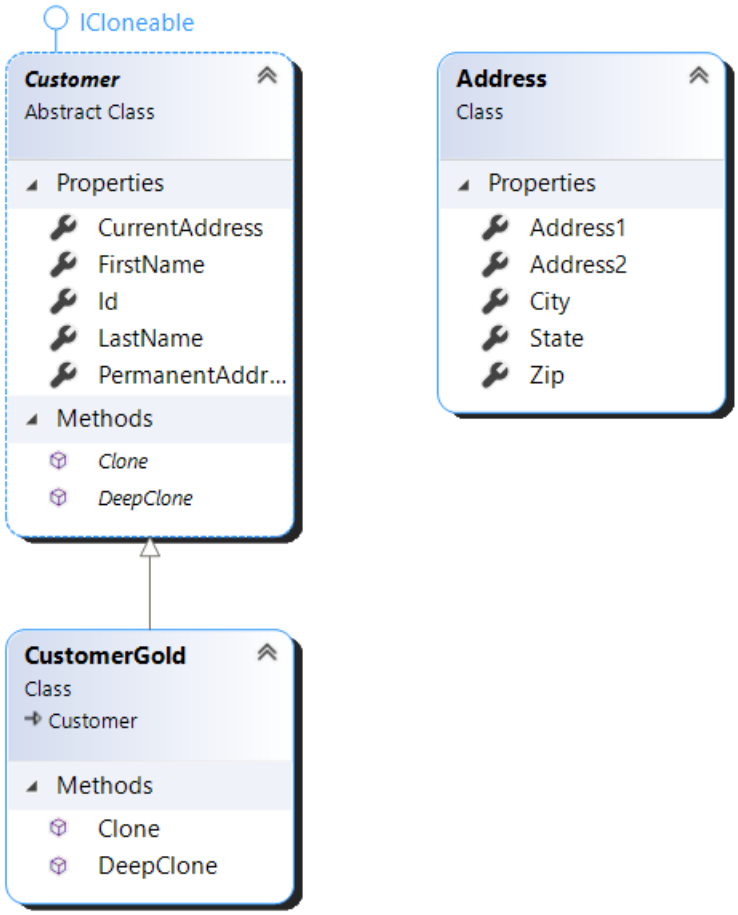
1. **Prototype -** This is an interface that is used for the types of objects that can be cloned itself.
2. **ConcretePrototype -** This is a class that implements the Prototype interface for cloning itself.

C# Implementation Code:

```
public interface Prototype
{
    Prototype Clone();
}

public class ConcretePrototypeA : Prototype
{
    public Prototype Clone()
    {
        // Shallow Copy: only top-level objects are duplicated
        return (Prototype)MemberwiseClone();

        // Deep Copy: all objects are duplicated
        //return (Prototype)this.Clone();
    }
}

public class ConcretePrototypeB : Prototype
{
    public Prototype Clone()
    {
        // Shallow Copy: only top-level objects are duplicated
        return (Prototype)MemberwiseClone();

        // Deep Copy: all objects are duplicated
        //return (Prototype)this.Clone();
    }
}
```

Q22. Explain Prototype pattern with a real-life example?

Ans. Let's take the following example where we are creating a prototype of the customer as CustomerGold same way, we can create CustomerSilver or CustomerPlatinum etc.



C# Implementation Code:

```
public abstract class Customer : ICloneable
{
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Address PermanentAddress { get; set; }
        public Address CurrentAddress { get; set; }

        public abstract object Clone();
        public abstract Customer DeepClone();
}
public class CustomerGold : Customer
{
        public override object Clone()
        {
            return this.MemberwiseClone() as Customer;
        }
```

```
        public override Customer DeepClone()
        {
            Customer customer = this.MemberwiseClone() as Customer;

            customer.PermanentAddress = new Address
            {
                Address1 = this.PermanentAddress.Address1,
                Address2 = this.PermanentAddress.Address2,
                State = this.PermanentAddress.State,
                City = this.PermanentAddress.City,
                Zip = this.PermanentAddress.Zip
            };
            customer.CurrentAddress = new Address
            {
                Address1 = this.CurrentAddress.Address1,
                Address2 = this.CurrentAddress.Address2,
                State = this.CurrentAddress.State,
                City = this.CurrentAddress.City,
                Zip = this.CurrentAddress.Zip
            };

            return customer;
        }
}
public class Address
{
        public string Address1 { get; set; }
        public string Address2 { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Zip { get; set; }
}
class Program
{
        static void Main(string[] args)
        {
            CustomerGold goldCutomer = new CustomerGold();
            goldCutomer.Id = 1;
            goldCutomer.FirstName = "Mohan";
            goldCutomer.LastName = "Kumar";
            goldCutomer.CurrentAddress = new Address
            {
                Address1 = "Noida Sector 15",
                Address2 = "Near Metro",
                State = "UP",
                City = "Noida",
                Zip = "201301"
            };
            goldCutomer.CurrentAddress = new Address
            {
                Address1 = "Lagpat Nager",
                Address2 = "Near Delhi Metro",
                State = "Delhi",
                City = "Delhi",
                Zip = "110091"
            };

            Customer clone = (Customer)goldCutomer.Clone();
```

```
            Customer deepClone = goldCutomer.DeepClone();
        }
}
```

Q23. When to use Prototype pattern?

Ans. The prototype pattern is useful in the following cases:

- The creation of each object is costly or complex.
- A limited number of state combinations exist in an object.

Q24. What is Shallow copy and Deep copy?

Ans. **Shallow Copy**- Only top-level objects are duplicated or copied. A shallow copy creates a new object by copying the non-static fields of the current object to the new object.

If a field is a **value type,** then a bit-by-bit copy of the field is performed. If a field is a **reference type,** then the reference is copied but the referred object is not. Hence the original object and its clone refer to the same object.

Deep Copy- All objects are duplicated or copied. A shallow copy creates a new object by copying the non-static fields of the current object to the new object.

If a field is a **value type,** then a bit-by-bit copy of the field is performed. If a field is a **reference type** then a new copy of the referred object is performed.

3

Structural Design Patterns

Q1. What is Adapter Design pattern? Explain with example?

Ans. The adapter pattern acts as a bridge between two incompatible interfaces. This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces.

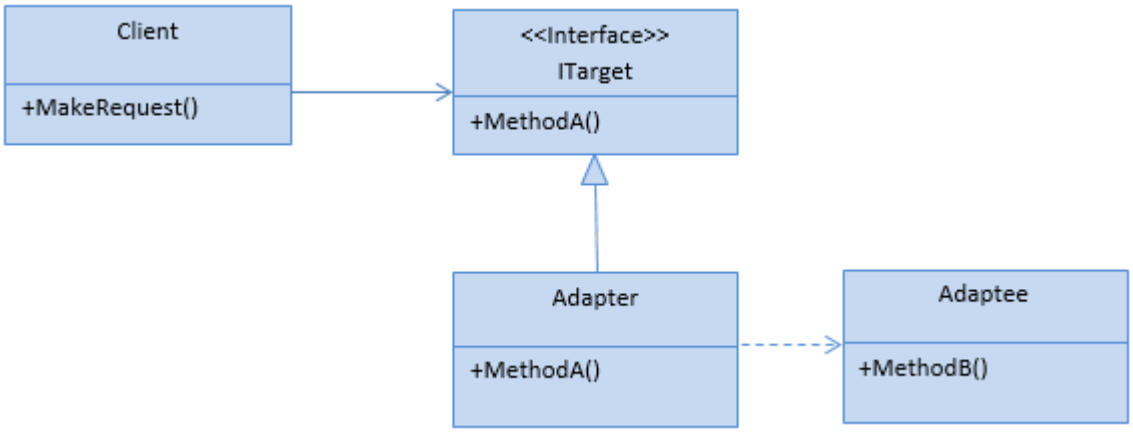
For Example, A card reader acts as an adapter between a memory card and a laptop. You plug the memory card into a card reader and card reader into the laptop so that the memory card can be read via the laptop.

The adapter pattern example includes:

- Data Transfer Object (DTO) to Business Object
- Need to convert third party objects to application types
- ADO.NET SqlAdapter, OracleAdapter, MySqlAdapter

Q2. Explain Adapter Design pattern with UML diagram and code example?

Ans. The UML diagram with adapter design pattern is shown below:



The classes, interfaces and objects in the above UML class diagram are as follows:

1. **ITarget -** This is an interface that is used by the client to achieve its functionality/request.
2. **Adapter -** This is a class that implements the ITarget interface and inherits the Adaptee class. It is responsible for communication between the Client and Adaptee.
3. **Adaptee -** This is a class which have the functionality, required by the client. However, its interface is not compatible with the client.
4. **Client -** This is a class that interacts with a type that implements the ITarget interface. However, the communication class called Adaptee, is not compatible with the client

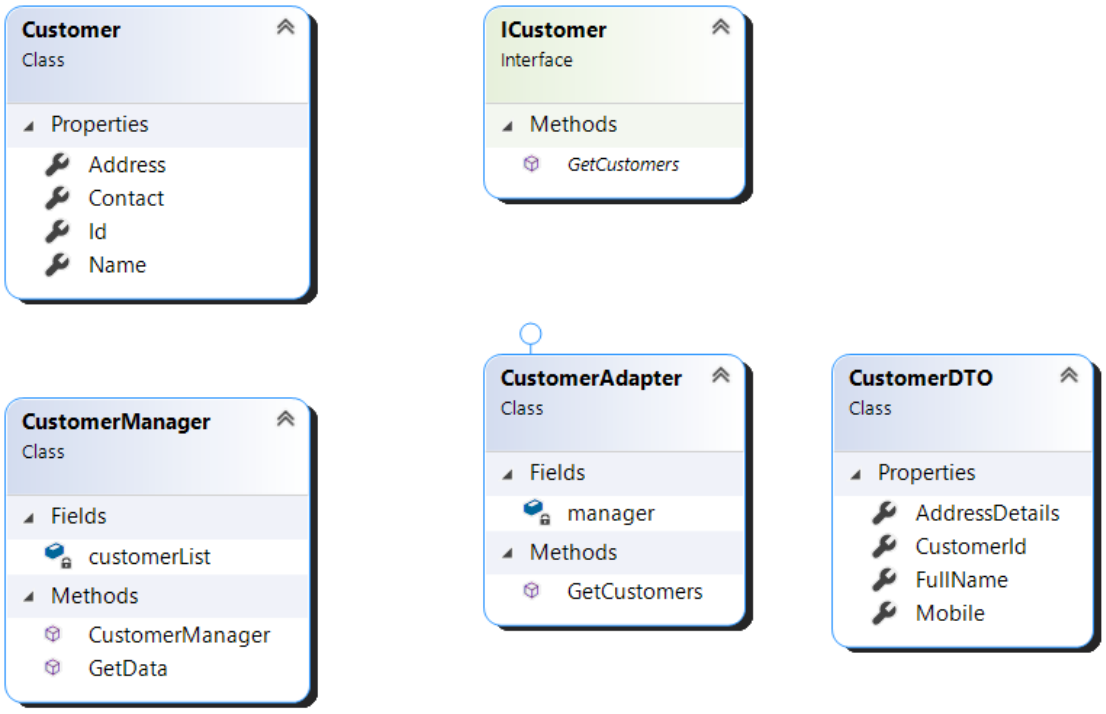
C# - Implementation Code

```
public class Client
{
    private ITarget target;

    public Client (ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}
public interface ITarget
{
    void MethodA();
}
public class Adapter: Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
public class Adaptee
```

```
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}
```

Q3. Explain Adapter pattern with a real-life example?

Ans. Let's take the following example where Customer and CustomerDTO classes are incompatible.



C# Implementation Code:

```
public class Customer
{
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string Contact { get; set; }
}
public class CustomerDTO
{
        public int CustomerId { get; set; }
        public string FullName { get; set; }
        public string AddressDetails { get; set; }
        public string Mobile { get; set; }
}
//class Adapter
//public class CustomerAdapter : CustomerManager, ICustomer
    //{
```

```
    //    public IEnumerable<CustomerDTO> GetCustomers()
    //    {
    //        var data = base.GetData();
    //        IEnumerable<Customer> customers =
JsonConvert.DeserializeObject<IEnumerable<Customer>>(data);

    //        return customers.Select(x => new CustomerDTO
    //        {
    //            CustomerId = x.Id,
    //            FullName = x.Name,
    //            AddressDetails = x.Address,
    //            Mobile = x.Contact
    //        });
    //    }
//}

//object Adapter
public class CustomerAdapter: ICustomer
{
        CustomerManager manager = new CustomerManager();
        public IEnumerable<CustomerDTO> GetCustomers()
        {
            var data = manager.GetData();
            IEnumerable<Customer> customers =
JsonConvert.DeserializeObject<IEnumerable<Customer>>(data);

            return customers.Select(x => new CustomerDTO
            {
                CustomerId = x.Id,
                FullName = x.Name,
                AddressDetails = x.Address,
                Mobile = x.Contact
            });
        }
}
public class CustomerManager
{
        private List<Customer> customerList = new List<Customer>();
        public CustomerManager()
        {
            customerList.Add(new Customer
            {
                Id = 1,
                Name = "Mohan",
                Address = "Noida",
                Contact = "989879879"
            });
        }
        public string GetData()
        {
            return JsonConvert.SerializeObject(customerList);
        }
}
public interface ICustomer
{
        IEnumerable<CustomerDTO> GetCustomers();
}
class Program
```

```
{
        static void Main(string[] args)
        {
            ICustomer customer = new CustomerAdapter();

            IEnumerable<CustomerDTO> data= customer.GetCustomers();
        }
}
```

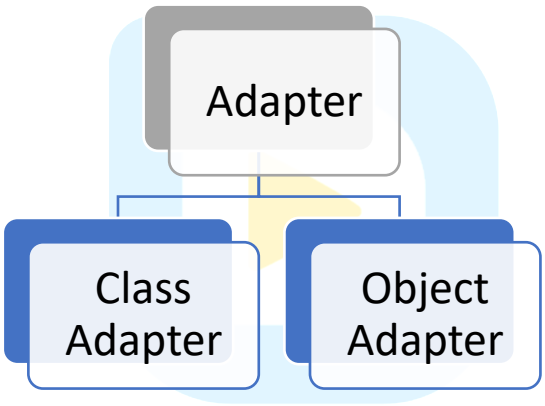
Q4. When to use Adapter Design pattern?

Ans. The adapter design pattern is useful in the following cases:

1. Allow a system to use classes of another system that is incompatible with it.
2. Allow communication between a new and already existing system that are independent to each other
3. ADO.NET SqlAdapter, OracleAdapter, MySqlAdapter are the best example of Adapter Pattern.

Q5. What are different types of Adapter design patterns?

Ans. There are two types of Adapter design patterns.



Q6. What is Bridge Design pattern? Explain with example?

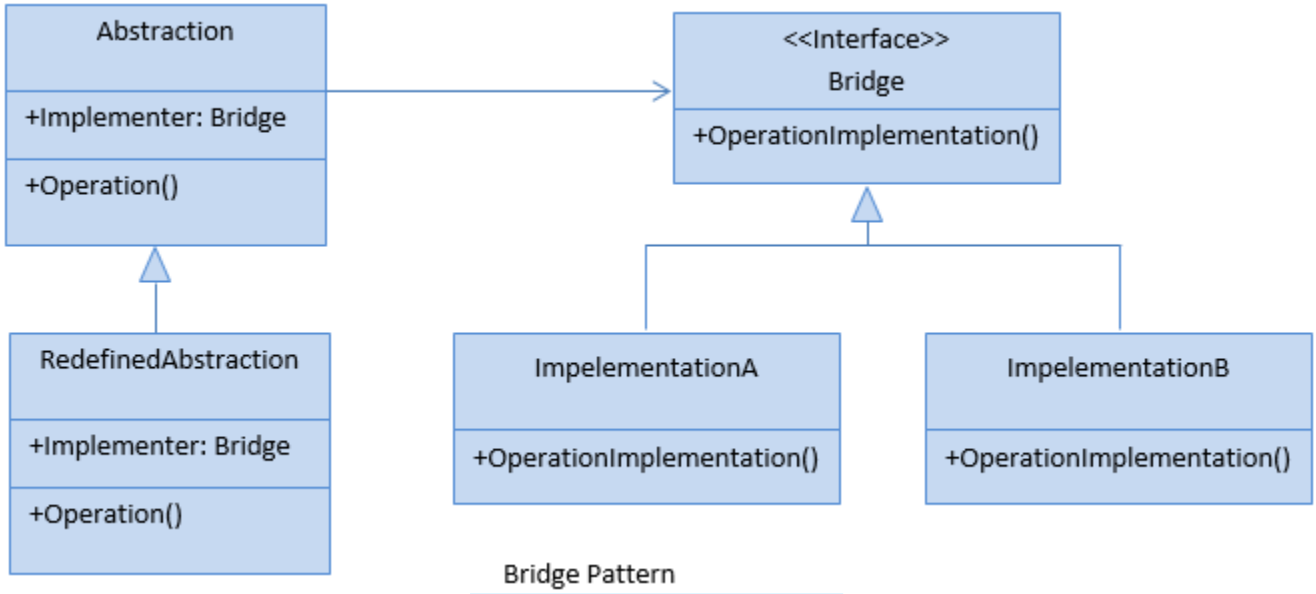
Ans. The bridge pattern is used to separate abstraction from its implementation so that both can be modified independently.

This pattern involves an interface that acts as a bridge between the abstraction class and implementer classes and also makes the functionality of the implementer independent from the abstraction. Ther bridge patterns examples are:

1. Payment Features:
   - Currently Credit Card, Debit Card
   - In Future add more
2. Messaging Features:
   - Currently SMS or Email
   - In Future add more

Q7. Explain Bridge Design pattern with UML diagram and code example?

Ans. The UML diagram with adapter design pattern is shown below:



Bridge Pattern

The classes, interfaces and objects in the above UML class diagram are as given follows:

1. **Abstraction-** This is an abstract class and containing members that define an abstract business object and its functionality. It contains a reference to an object of type Bridge. It can also act as the base class for other abstractions.
2. **Redefined Abstraction -** This is a class that inherits from the Abstraction class. It extends the interface defined by Abstraction class.
3. **Bridge -** This is an interface that acts as a bridge between the abstraction class and implementer classes and also makes the functionality of the implementer class independent from the abstraction class.
4. **ImplementationA & ImplementationB -**These are classes that implement the Bridge interface and also provide the implementation details for the associated Abstraction class.

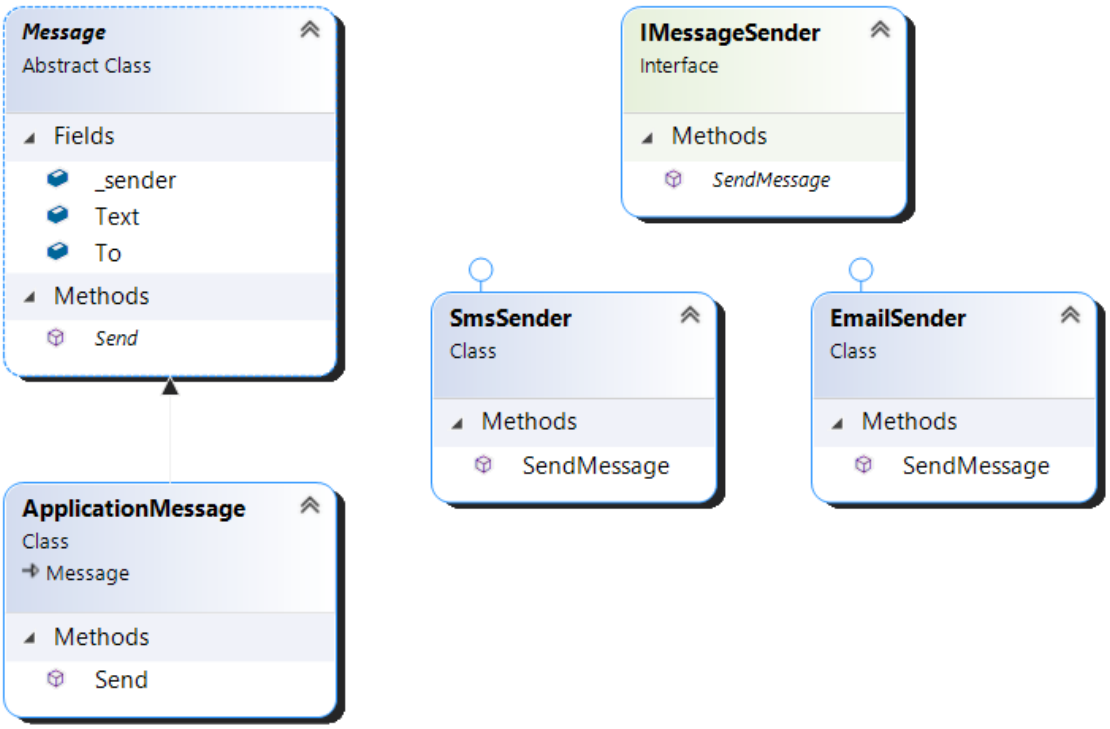
C# - Implementation Code:

```
public abstract class Abstraction
{
    public Bridge Implementer { get; set; }

    public virtual void Operation()
    {
        Console.WriteLine("ImplementationBase:Operation()");
        Implementer.OperationImplementation();
    }
}
public class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
```

```
        Console.WriteLine("RefinedAbstraction:Operation()");
        Implementer.OperationImplementation();
    }
}
public interface Bridge
{
    void OperationImplementation();
}
public class ImplementationA : Bridge
{
    public void OperationImplementation()
    {
        Console.WriteLine("ImplementationA:OperationImplementation()");
    }
}
public class ImplementationB : Bridge
{
    public void OperationImplementation()
    {
        Console.WriteLine("ImplementationB:OperationImplementation()");
    }
}
```

Q8. Explain Bridge pattern with a real-life example?

Ans. Let's take the following example where we have two implementations SmsSender and EmailSender of IMessageSender bridge.



C# - Implementation Code:

```
// 'Abstraction' abstract class
    abstract class Message
    {
        public IMessageSender _sender;
        public string Text;
        public string To;
        public abstract void Send();
    }
// 'RefinedAbstraction' class
    class ApplicationMessage : Message
    {
        public override void Send()
        {
            _sender.SendMessage(To, Text);
        }
    }
// 'Implementor' interface
    public interface IMessageSender
    {
        void SendMessage(string to, string Message);
    }
// ConcreteImplementor class
    class SmsSender : IMessageSender
    {
        public void SendMessage(string to, string Message)
        {
            Console.WriteLine($"To: {to}, Message: {Message}");
        }
    }
// ConcreteImplementor class
    class EmailSender : IMessageSender
    {
        public void SendMessage(string to, string Message)
        {
            Console.WriteLine($"To: {to}, Message: {Message}");
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            ApplicationMessage appMsg = new ApplicationMessage();
            appMsg.Text = "Hello, This is Shailendra";
            appMsg.To = "9876543210";

            appMsg._sender = new SmsSender();
            appMsg.Send();

            Console.ReadLine();
        }
    }
```

Note - Bridge pattern has nearly the same structure as the Adapter Pattern. But it is used when designing new systems instead of the Adapter pattern which is used with already existing systems.

Q9. When to use Bridge Design pattern?

Ans. The bridge design pattern is useful in the following cases:

- Abstractions and implementations should be modified independently.
- Changes in the implementation of an abstraction should have no impact on clients.
- The Bridge pattern is used when a new version of a software or system is brought out, but the older version of the software still running for its existing client. There is no need to change the client code, but the client needs to choose which version he wants to use.

Q10. What is Composite Design pattern? Explain it with examples?

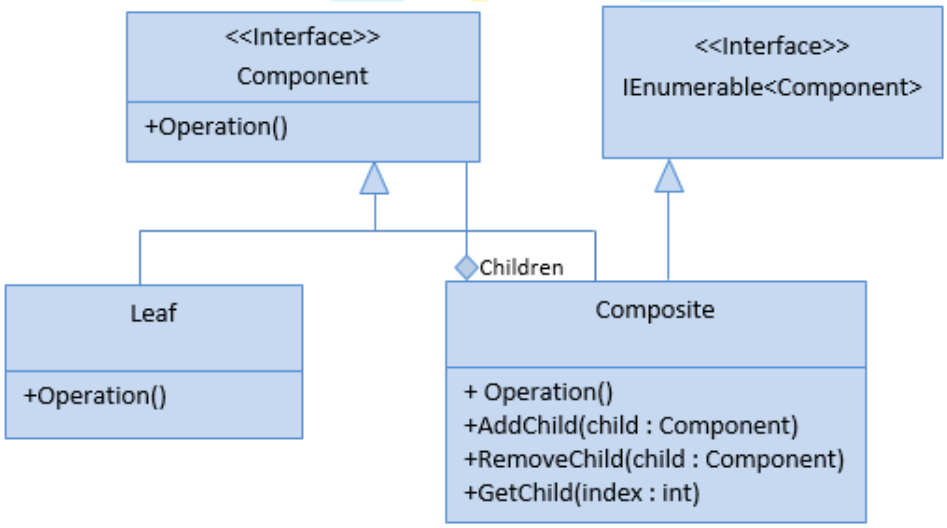
Ans. **The** Composite pattern is used when we need to treat a group of objects and a single object in the same way. Composite pattern composes objects in terms of a tree structure to represent part as well as whole hierarchies.

The examples for composite patterns are:

- File System Folders
- Managers and Employees
- Assemblies

Q11. Explain Composite Design pattern with UML diagram and code example?

Ans. This pattern creates a class that contains a group of its own objects. This class provides ways to modify its group of the same objects.



Composite Pattern

The classes, interfaces and objects in the above UML class diagram are as follows:

1. **Component-** This is an abstract class containing members that will be implemented by all objects in the hierarchy. It acts as the base class for all the objects within the hierarchy

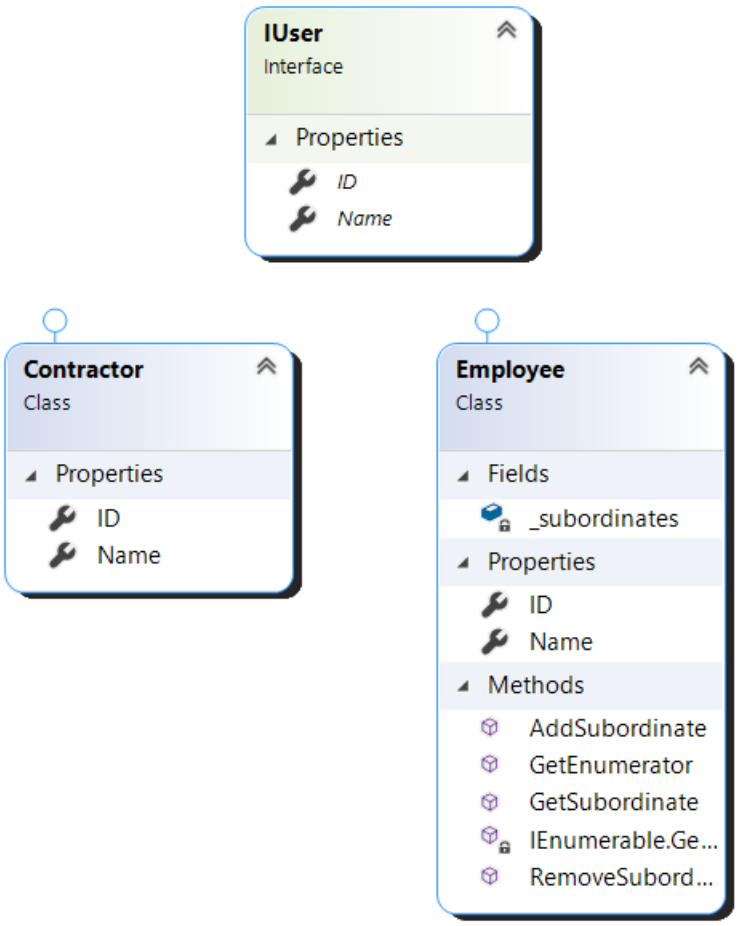
2. **Composite -** This is a class that includes Add, Remove, Find and Get methods to do operations on child components.
3. **Leaf -** This is a class that is used to define leaf components within the tree structure means these cannot have children.

C# - Implementation Code:

```
public interface Component
{
    void Operation();
}

public class Composite : Component, IEnumerable
{
    private List<Component> _children = new List<Component>();
    public void AddChild(Component child)
    {
        _children.Add(child);
    }
    public void RemoveChild(Component child)
    {
        _children.Remove(child);
    }
    public Component GetChild(int index)
    {
        return _children[index];
    }
    public void Operation()
    {
        string message = string.Format("Composite with {0} child(ren)", _children.Count);
        Console.WriteLine(message);
    }
    public IEnumerator GetEnumerator()
    {
        foreach (Component child in _children)
            yield return child;
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
public class Leaf : Component
{
    public void Operation()
    {
        Console.WriteLine("Leaf");
    }
}
```

Q12. Explain Composite pattern with a real-life example?

Ans. Let's take the following example where Employee is a composite class and IUser is a component.



C# - Implementation Code:

```
//interface
    public interface IUser
    {
        int ID { get; set; }
        string Name { get; set; }
    }
//leaf
    public class Contractor : IUser
    {
        public int ID { get; set; }
        public string Name { get; set; }
    }
//composite
    //public class Employee : IUser
    //{
    //    private List<IUser> _subordinates = new List<IUser>();

    //    public int ID { get; set; }
    //    public string Name { get; set; }

    //    public void AddSubordinate(IUser subordinate)
    //    {
    //        _subordinates.Add(subordinate);
    //    }
```

```
//    public void RemoveSubordinate(IUser subordinate)
//    {
//        _subordinates.Remove(subordinate);
//    }

//    public IEnumerable<IUser> GetSubordinates()
//    {
//        return _subordinates;
//    }
//}

public class Employee : IUser, IEnumerable<IUser>
{
    private List<IUser> _subordinates = new List<IUser>();

    public int ID { get; set; }
    public string Name { get; set; }

    public void AddSubordinate(IUser subordinate)
    {
        _subordinates.Add(subordinate);
    }

    public void RemoveSubordinate(IUser subordinate)
    {
        _subordinates.Remove(subordinate);
    }

    public IUser GetSubordinate(int index)
    {
        return _subordinates[index];
    }

    public IEnumerator<IUser> GetEnumerator()
    {
        foreach (IUser subordinate in _subordinates)
        {
            yield return subordinate;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
class Program
{
    static void Main(string[] args)
    {
        //CEO:root
        Employee Rahul = new Employee { ID = 1, Name = "Rahul" };

        //manager:nodes
        Employee Amit = new Employee { ID = 2, Name = "Amit" };
        Employee Mohan = new Employee { ID = 3, Name = "Mohan" };
```

```
            //manager:composite
            Rahul.AddSubordinate(Amit);
            Rahul.AddSubordinate(Mohan);

            Employee Rita = new Employee { ID = 4, Name = "Rita" };
            Employee Hari = new Employee { ID = 5, Name = "Hari" };

            //manager:composite
            Amit.AddSubordinate(Rita);
            Amit.AddSubordinate(Hari);

            //leafs
            Employee Kamal = new Employee { ID = 6, Name = "Kamal" };
            Contractor Sam = new Contractor { ID = 7, Name = "Sam" };

            //manager:composite
            Mohan.AddSubordinate(Kamal);
            Mohan.AddSubordinate(Sam);

            //root node
            Console.WriteLine("CEO: ID={0}, Name={1}", Rahul.ID, Rahul.Name);

            //foreach (Employee manager in Rahul.GetSubordinates())
            //{
            //    Console.WriteLine("  Manager: ID={0}, Name={1}", manager.ID, manager.Name);
            //    Console.WriteLine("   Employee(s):");
            //    foreach (var employee in manager.GetSubordinates())
            //    {
            //        Console.WriteLine("\t ID={0}, Name={1}", employee.ID, employee.Name);
            //    }
            //}

            foreach (Employee manager in Rahul)
            {
                Console.WriteLine("  Manager: ID={0}, Name={1}", manager.ID, manager.Name);
                Console.WriteLine("   Employee(s):");
                foreach (var employee in manager)
                {
                    Console.WriteLine("\t ID={0}, Name={1}", employee.ID, employee.Name);
                }
            }
        }
    }
```

Q13. When to use Composite Design pattern?

Ans. The composite design pattern is useful in the following cases:

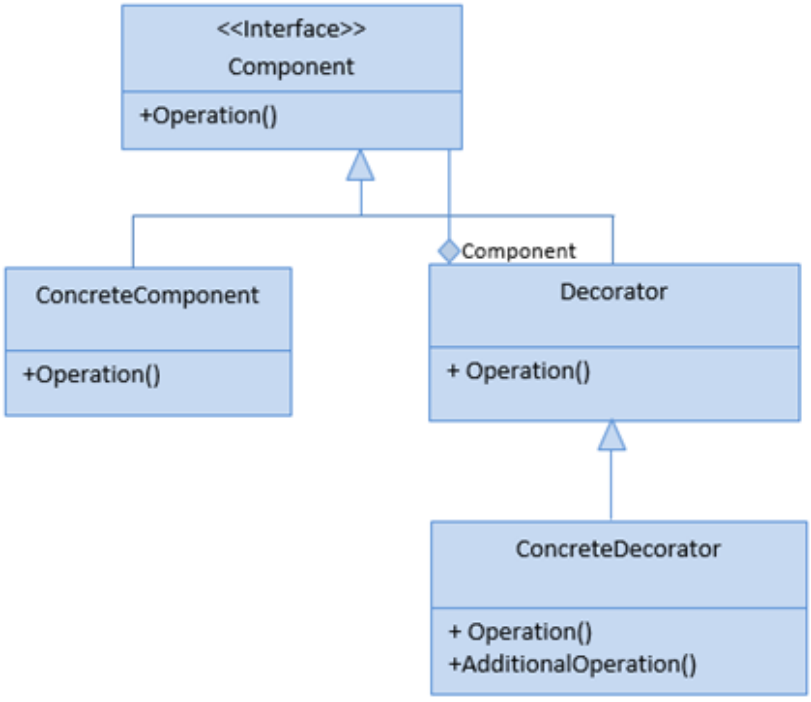
- Hierarchical representations of objects are required.
- A single object and a group of objects should be treated in the same way.
- The Composite pattern is used when data is organized in a tree structure (like as directories in a computer).

Q14. What is Decorator Design pattern? Explain it with examples?

Ans. A decorator pattern is used to add new functionality to an existing object without changing its structure. This pattern creates a decorator class that wraps the original class and add new behaviours/operations to an object at run-time. It is another way to implement inheritance.

The examples for decorator pattern are:

1. Build Your Own Computer Order
    - RAM options
    - Storage Options
    - Processor Options
2. Build Your Own Car Order
    - Accessories Options
    - Security
    - Sports



Decorator Pattern

The classes, interfaces and objects in the above UML class diagram are as follows:

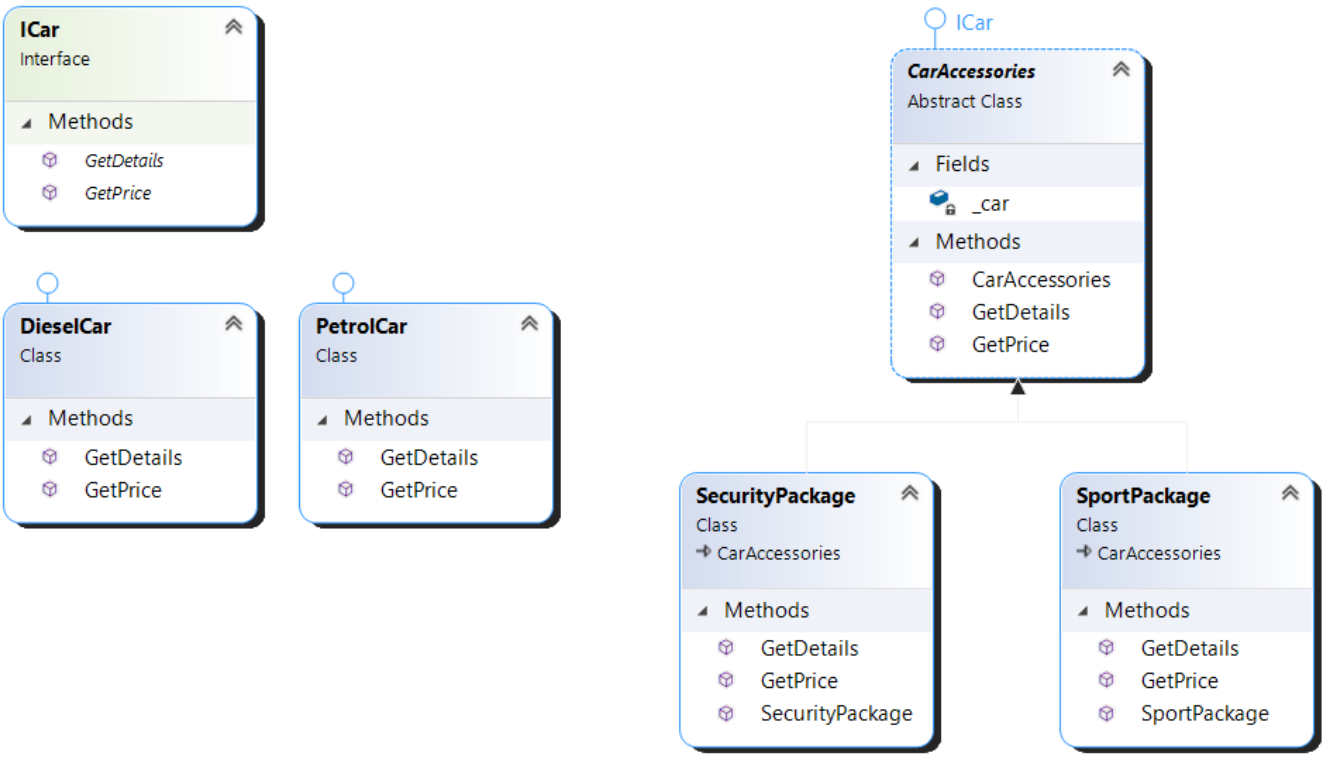
1. **Component-** This is an interface containing members that will be implemented by ConcreteClass and Decorator.
2. **ConcreteComponent -** This is a class that implements the Component interface.
3. **Decorator -** This is an abstract class that implements the Component interface and contains the reference to a component instance. This class also acts as a base class for all decorators for components.
4. **ConcreteDecorator -** This is a class that inherits from the Decorator class and provides a decorator for components.

C# - Implementation Code:

```
public interface Component
{
    void Operation();
}

public class ConcreteComponent : Component
{
    public void Operation()
    {
        Console.WriteLine("Component Operation");
    }
}

public abstract class Decorator : Component
{
    private Component _component;

    public Decorator(Component component)
    {
        _component = component;
    }

    public virtual void Operation()
    {
        _component.Operation();
    }
}

public class ConcreteDecorator : Decorator
{
    public ConcreteDecorator(Component component) : base(component) { }

    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("Override Decorator Operation");
    }
}
```

Q15. Explain Decorator pattern with a real-life example?

Ans. Let's take the following example where CarAccessories is decorator class and SecurityPackage & SportPackage are concrete decorator classes.



C# - Implementation Code:

```
public interface ICar
{
        string GetDetails();
        double GetPrice();
}
public class PetrolCar : ICar
{
        public double GetPrice()
        {
            return 100;
        }

        public string GetDetails()
        {
            return "Petrol Car";
        }
}
public class DieselCar : ICar
{
        public double GetPrice()
        {
            return 1000;
        }

        public string GetDetails()
        {
            return "Diesel Car";
```

```
        }
}
public abstract class CarAccessories : ICar
{
        private readonly ICar _car;

        public CarAccessories(ICar car)
        {
            _car = car;
        }

        public virtual double GetPrice()
        {
            return _car.GetPrice();
        }

        public virtual string GetDetails()
        {
            return _car.GetDetails();
        }
}
public class SecurityPackage : CarAccessories
{
        public SecurityPackage(ICar car) : base(car)
        {

        }

        public override string GetDetails()
        {
            return base.GetDetails() + " + Security Package";
        }

        public override double GetPrice()
        {
            return base.GetPrice() + 1;
        }

}
public class SportPackage : CarAccessories
{
        public SportPackage(ICar car) : base(car)
        {

        }

        public override string GetDetails()
        {
            return base.GetDetails() + " + Sport Package";
        }

        public override double GetPrice()
        {
            return base.GetPrice() + 10;
        }
}
class Program
{
```

```
        static void Main(string[] args)
        {
            var basicCar= new PetrolCar();
            CarAccessories upgraded = new SportPackage(basicCar);
            upgraded = new SecurityPackage(upgraded);

            Console.WriteLine($"Car: '{upgraded.GetDetails()}' Cost: {upgraded.GetPrice()}");
        }
}
```

Q16. When to use Decorator Design pattern?

Ans. The decorator design pattern is useful in the following cases:

- Add additional state or behaviour to an object dynamically.
- Make changes to some objects in a class without affecting others.

Q17. What are alternatives to Decorator Design pattern?

Ans. The alternatives to the Decorator design pattern are:

Adapter: Converts one interface to another so that it matches what the client is expecting.

Decorator: Dynamically adds responsibility to the interface by wrapping the original code.

Facade: Provides a simplified interface.

Q18. What is Façade Design pattern? Explain it with examples?

Ans. Façade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.

This pattern involves a single wrapper class that contains a set of members which are required by the client. These members access the system on behalf of the facade client and hide the implementation details.

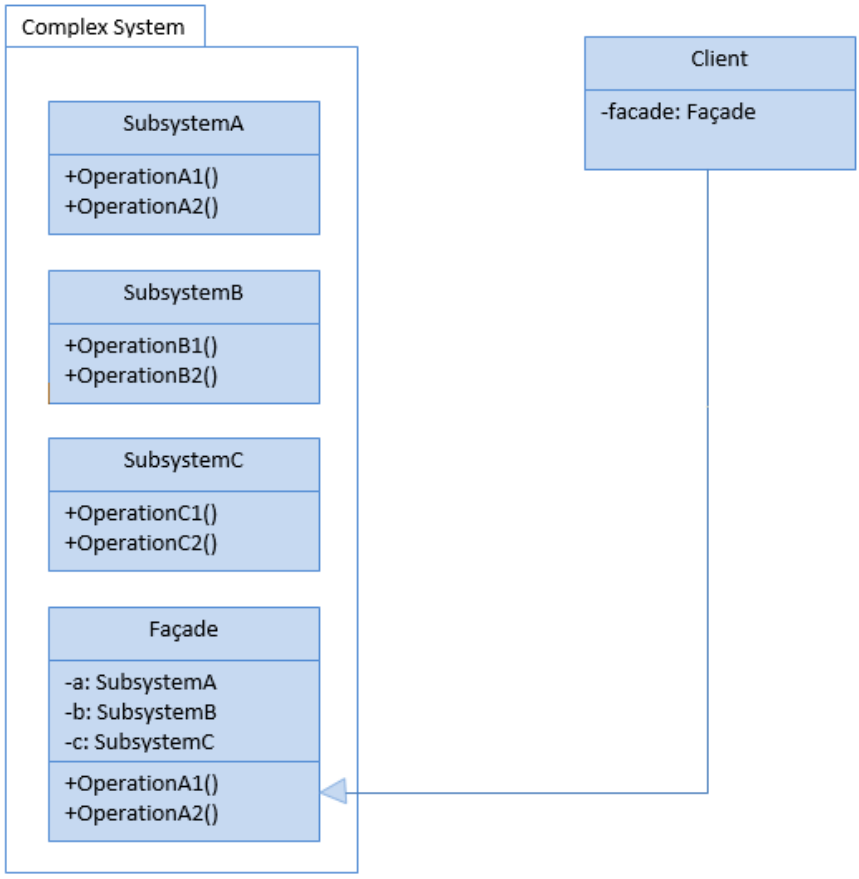
Provides a higher-level interface that makes the subsystem easier to use. It aggregate objects to simplify the process.

The examples for Façade design patterns are:

- User Registration Process
- Process An Order
- Creating User Follower Workflow

Q19. Explain Façade Design pattern with UML diagram and code example?

Ans. The UML diagram for Façade design pattern is shown below:



Façade Pattern

The classes, interfaces and objects in the above UML class diagram are defined as follows:

1. **Complex System -** A library of subsystems.
2. **SubsystemA, SubsystemB, SubsystemC -** These are classes within a complex system and offer detailed operations.
3. **Façade -** This is a wrapper class that contains a set of members which are required by the client.
4. **Client -** This is a class that calls the high-level operations in the Façade.

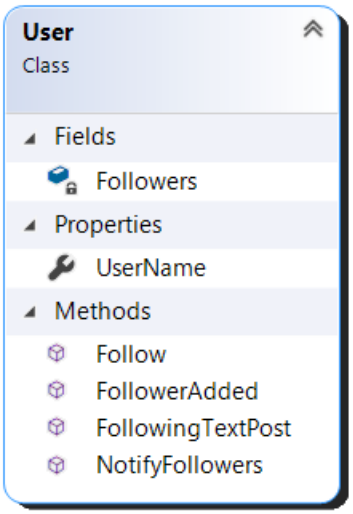
C# - Implementation Code:

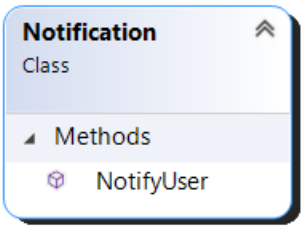
```
class SubsystemA
{
    public string OperationA1()
    {
        return "Subsystem A, Method A1\n";
    }
    public string OperationA2()
    {
        return "Subsystem A, Method A2\n";
    }
}
class SubsystemB
{
```

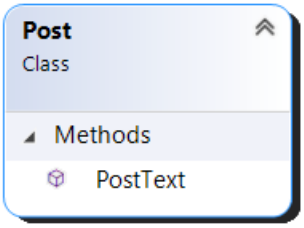
```
    public string OperationB1()
    {
        return "Subsystem B, Method B1\n";
    }

    public string OperationB2()
    {
        return "Subsystem B, Method B2\n";
    }
}
class SubsystemC
{
    public string OperationC1()
    {
        return "Subsystem C, Method C1\n";
    }

    public string OperationC2()
    {
        return "Subsystem C, Method C2\n";
    }
}

public class Facade
{
    SubsystemA a = new SubsystemA();
    SubsystemB b = new SubsystemB();
    SubsystemC c = new SubsystemC();
    public void Operation1()
    {
        Console.WriteLine("Operation 1\n" +
        a.OperationA1() +
        a.OperationA2() +
        b.OperationB1());
    }
    public void Operation2()
    {
        Console.WriteLine("Operation 2\n" +
        b.OperationB2() +
        c.OperationC1() +
        c.OperationC2());
    }
}
```

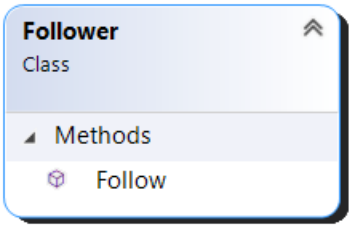
Q20. Explain Facade pattern with a real-life example?

Ans. Let's take the following example where Notification and Post are subsystems of User and the Follower is a façade to expose the details of the system.









C# - Implementation Code:

```
public class User
{
        private List<User> Followers = new List<User>();
        public string UserName { get; set; }

        public bool Follow(User userToFollow)
        {
            //TO DO:
            return true;
        }

        public bool FollowerAdded(User newFollower)
        {
            // code for notifying user that they have an added follower
            Notification notification = new Notification();
            return notification.NotifyUser(this, newFollower.UserName + " is now following
you!");
        }

        public bool FollowingTextPost(User userToFollow)
        {
            // code for posting simple text
            Post post = new Post();
```

```
            return post.PostText(this, this.UserName + " is now following " +
userToFollow.UserName);
        }

        public bool NotifyFollowers(User userToFollow)
        {
            bool result = true;
            Notification notification = new Notification();
            foreach (User follower in Followers)
            {
                // code for notifying followers of new activity
                result = notification.NotifyUser(this, this.UserName + " is now following " +
userToFollow.UserName);
                if (!result)
                {
                    break;
                }
            }
            return result;
        }

}

public class Notification
{
        public bool NotifyUser(User user, string message)
        {
            // TO DO: send email, sms
            Console.WriteLine(message);
            return true;
        }

}

public class Post
{
        public bool PostText(User user, string message)
        {
            //TO DO:
            Console.WriteLine(message);
            return true;
        }

}

//facade
public class Follower
{
        public bool Follow(User Follower, User Following)
        {
            bool result;

            //add user as follower
            result = Follower.Follow(Following);
            if (!result)
                return false;
```

```
            // notify user that they have an added follower
            result = Follower.FollowerAdded(Follower);
            if (!result)
            {
                return false;
            }

            // post that user is now following a new person
            result = Follower.FollowingTextPost(Following);
            if (!result)
            {
                return false;
            }

            // notify all of user's followers of new following activity
            result = Follower.NotifyFollowers(Following);
            if (!result)
            {
                return false;
            }

            return true;
        }

}

class Program
{
        static void Main(string[] args)
        {
            User me = new User();
            me.UserName = "proshailendra";

            User userIWantToFollow = new User();
            userIWantToFollow.UserName = "dotnettricks";

            //facade
            Follower follower = new Follower();
            var result = follower.Follow(me, userIWantToFollow);
            if (result)
            {
                Console.WriteLine("You have successfully followed " +
userIWantToFollow.UserName);
            }
            else
            {
                Console.WriteLine("Sorry, something went wrong.");
            }
        }

}
```

Q21. When to use Façade Design pattern?

Ans. A façade design pattern is useful in the following cases:

- A simple interface is required to access a complex system.
- The abstractions and implementations of a subsystem are tightly coupled.
- Need an entry point to each level of layered software.
- The facade design pattern is particularly used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable

Q22. What is Flyweight Design pattern and when to use it?

Ans. Flyweight pattern is used when there is a need to create a large number of objects of almost similar nature and they have some data in common. A few shared objects can replace many unshared ones.

The flyweight pattern uses the concepts of intrinsic and extrinsic data.

Intrinsic data is held in the properties of the shared flyweight objects. This information is stateless and generally remains unchanged, if any change occurs it would be reflected amongst all of the objects that reference the flyweight.

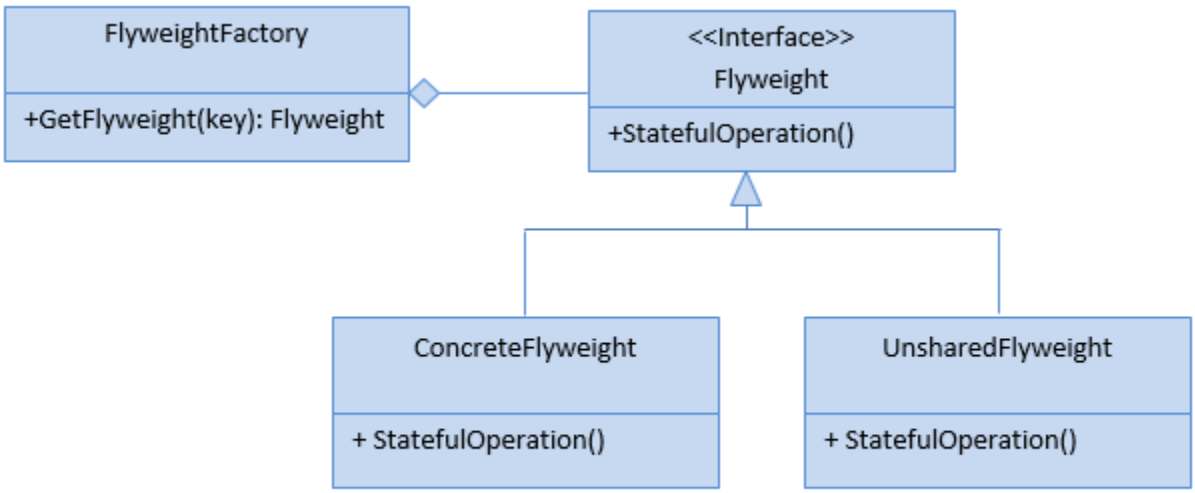
Extrinsic data is computed on the fly means at runtime and it is held outside of a flyweight object. Hence it can be stateful.

The examples for flyweight patterns are:

- Game Objects
- Social Media Posts
- List of Users

Q23. Explain Flyweight Design pattern with UML diagram and code example?

Ans. The UML diagram for Flyweight design pattern is shown below:



Flyweight Pattern

The classes, interfaces and objects in the above UML class diagram are given as follows:

1. **Flyweight -** This is an interface that defines the members of the flyweight objects.
2. **ConcreteFlyweight -** This is a class that Inherits from the Flyweight class.
3. **UnsharedFlyweight -** This is a class that Inherits from the Flyweight class and enables sharing of information, it is possible to create instances of concrete flyweight classes that are not shared.
4. **FlyweightFactory -** This is a class that holds the references of already created flyweight objects. When the GetFlyweight method is called from client code, these references are checked to determine if an appropriate flyweight object is already present or not. If present, it is returned. Otherwise, a new object is generated, added to the collection and returned.

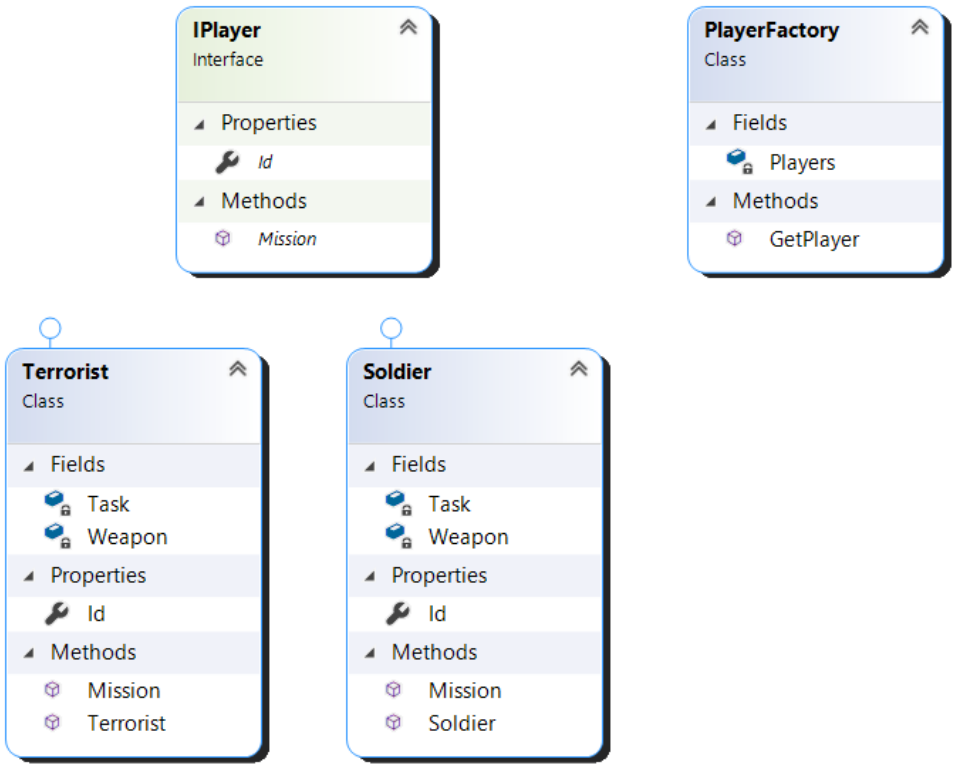
C# - Implementation Code:

```
public class FlyweightFactory
{
    private Hashtable _flyweights = new Hashtable();

    public Flyweight GetFlyweight(string key)
    {
        if (_flyweights.Contains(key))
        {
            return _flyweights[key] as Flyweight;
        }
        else
        {
            ConcreteFlyweight newFlyweight = new ConcreteFlyweight();

            // Set properties of new flyweight here.

            _flyweights.Add(key, newFlyweight);
            return newFlyweight;
        }
    }
}

public interface Flyweight
{
    void StatefulOperation(object o);
}

public class ConcreteFlyweight : Flyweight
{
    public void StatefulOperation(object o)
    {
        Console.WriteLine(o);
    }
}

public class UnsharedFlyweight : Flyweight
{
    private object _state;

    public void StatefulOperation(object o)
    {
```

```
        _state = o;
        Console.WriteLine(o);
    }
}
```

Q24. Explain Flyweight pattern with a real-life example?

Ans. Let's take the following example where PlayerFactory is a flyweight factory and IPlayer is a flyweight interface.



C# - Implementation Code:

```
//flyweight
public interface IPlayer
{
        public int Id { get; set; }
        public void Mission();
}
//concrete flyweight
public class Terrorist : IPlayer
{
        // Intrinsic Attribute : static
        private string Task;
        private string Weapon;

        // Extrinsic Attribute : dynamic
        public int Id { get; set; }
        public Terrorist()
        {
            Task = "PLANT BOMB";
```

```
            Weapon = "AK-47";
        }

        public void Mission()
        {
            //Work on the Mission
            Console.WriteLine($"Terrorist Id #{Id} with weapon ${Weapon}, Task is ${Task}");
        }
}
public class Soldier : IPlayer
{
        // Intrinsic Attribute : static
        private string Task;
        private string Weapon;

        // Extrinsic Attribute : dynamic
        public int Id { get; set; }
        public Soldier()
        {
            Task = "DIFFUSE BOMB";
            Weapon = "GUN";
        }

        public void Mission()
        {
            //Work on the Mission
            Console.WriteLine($"Soldier Id #{Id} with weapon ${Weapon}, Task is ${Task}");
        }
}
//Flyweight factory
public class PlayerFactory
{
       private static Dictionary<string, IPlayer> Players = new Dictionary<string, IPlayer>();

       public static IPlayer GetPlayer(string Type)
       {
            if (Players.ContainsKey(Type))
            {
                return Players[Type];
            }
            else
            {
                IPlayer player = null;
                switch (Type)
                {
                    case "Terrorist":
                        player = new Terrorist();
                        break;
                    case "Soldier":
                        player = new Soldier();
                        break;
                }

                //adding user to list for reusing
                Players.Add(Type, player);
                return player;
            }
        }
```

```
    }
class Program
{
        static void Main(string[] args)
        {
            IPlayer player1 = PlayerFactory.GetPlayer("Terrorist");
            player1.Id = 1;
            player1.Mission();

            IPlayer player2 = PlayerFactory.GetPlayer("Soldier");
            player2.Id = 2;
            player2.Mission();

            IPlayer player3 = PlayerFactory.GetPlayer("Soldier");
            player3.Id = 3;
            player3.Mission();
        }
}
```

Q25. When to use Flyweight Design pattern?

Ans. A flyweight design pattern is useful in the following cases:

- Flyweight is used when there is a need to create a large number of objects of almost similar nature and storage cost is high.
- A few shared objects can replace many unshared ones.
- Most of the state can be kept on disk or calculated at runtime.

Q26. What is Proxy Design pattern and when to use it?

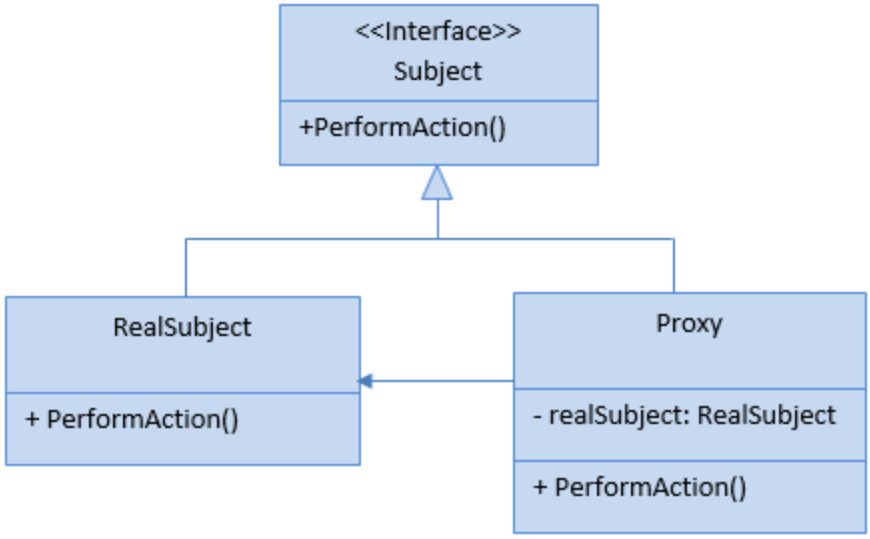
Ans. Provide a surrogate object, which references to another object. The proxy pattern involves a class, called proxy class, which represents the functionality of another class.

The examples for proxy patterns are:

- Payment Processing
- Accessing Remote Service
- Accessing Remote Database

Q27. Explain Proxy Design pattern with UML diagram and code example?

Ans. The UML diagram for the Proxy design pattern is shown below:



Proxy Pattern

The classes, interfaces and objects in the above UML class diagram are as follows:

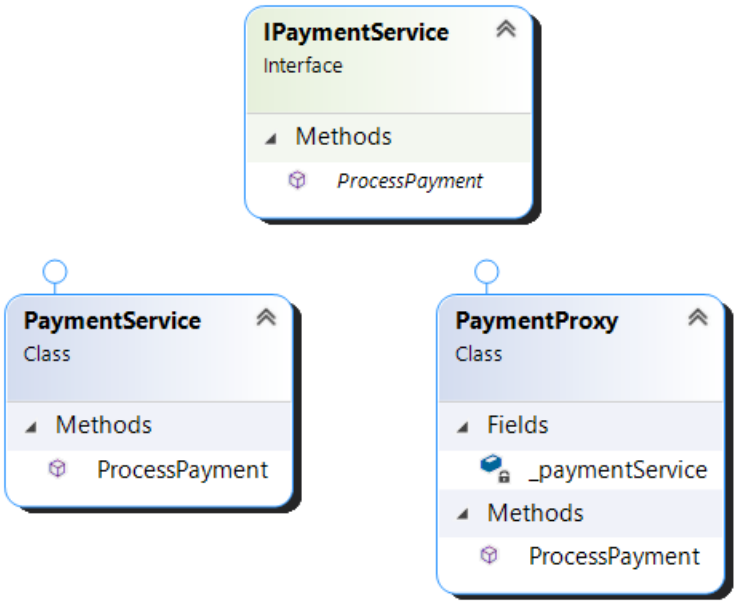
1. **Subject -** This is an interface having members that will be implemented by RealSubject and Proxy class.
2. **RealSubject -** This is a class that we want to use more efficiently by using the proxy class.
3. **Proxy -** This is a class that holds the instance of the RealSubject class and can access RealSubject class members as required.

C# - Implementation Code:

```
public interface Subject
{
    void PerformAction();
}

public class RealSubject : Subject
{
    public void PerformAction()
    {
        Console.WriteLine("RealSubject action performed.");
    }
}

public class Proxy : Subject
{
    private RealSubject _realSubject;

    public void PerformAction()
    {
        if (_realSubject == null)
            _realSubject = new RealSubject();

        _realSubject.PerformAction();
    }
}
```

Q28. Explain Proxy pattern with a real-life example?

Ans. Let's take the following example where we are creating a proxy for payment service.



C# - Implementation Code:

```
interface IPaymentService
{
        void ProcessPayment();
}
public class PaymentService : IPaymentService
{
        public void ProcessPayment()
        {
            Console.WriteLine("Payment processed!");
        }
}
public class PaymentProxy : IPaymentService
{
        private PaymentService _paymentService = new PaymentService();
        public void ProcessPayment()
        {
            _paymentService.ProcessPayment();
        }
}
class Program
{
        static void Main(string[] args)
        {
            // Create math proxy
            PaymentProxy proxy = new PaymentProxy();

            // Do the operations
            proxy.ProcessPayment();
        }
}
```

Q29. When to use Proxy Design pattern?

Ans. The proxy design pattern is useful in the following cases:

- Objects need to be created on-demand means when their operations are requested.
- Access control for the original object is required.
- Allow accessing a remote object by using a local object (it will refer to a remote object).

4

Behavioural Design Patterns

Q1. What is Chain of Responsibility Design pattern and when to use it?

Ans. The chain of responsibility pattern is used to perform a multi-step (a chain of steps) request process. It pattern decouples sender and receiver based on the type of request.

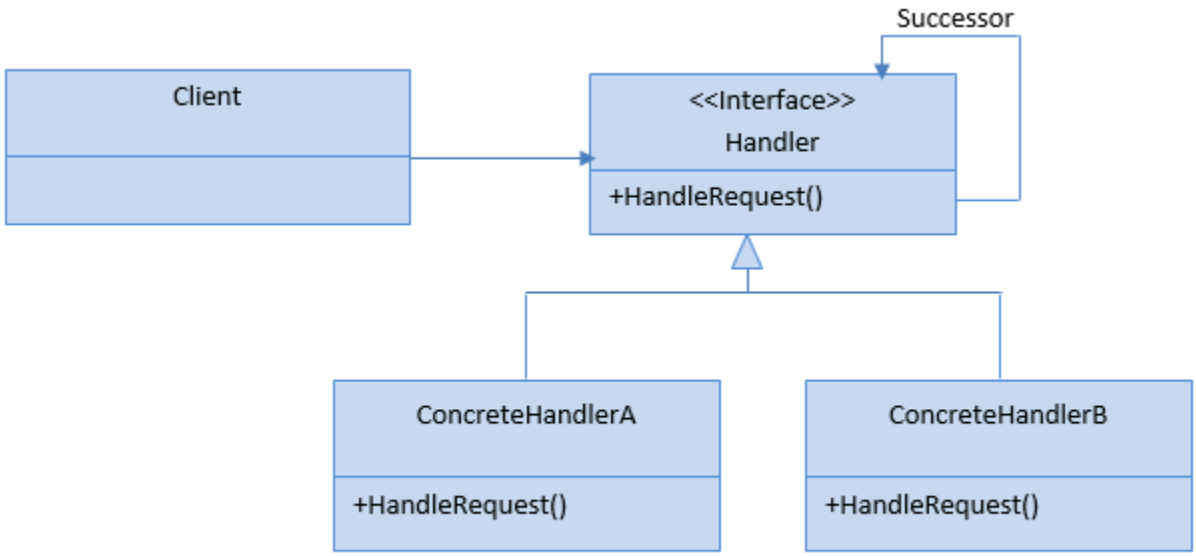
In this pattern, normally each receiver (handler) contains a reference to another receiver. If one receiver cannot handle the request then it passes the same to the next receiver and so on.

The examples for the chain of responsibility pattern are:

- Multi-Steps Request Process
- Approval Process

Q2. Explain the chain of responsibility pattern with UML diagram and code example?

Ans. The UML diagram for the chain of responsibility design pattern is shown below:



Chain of Responsibility Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

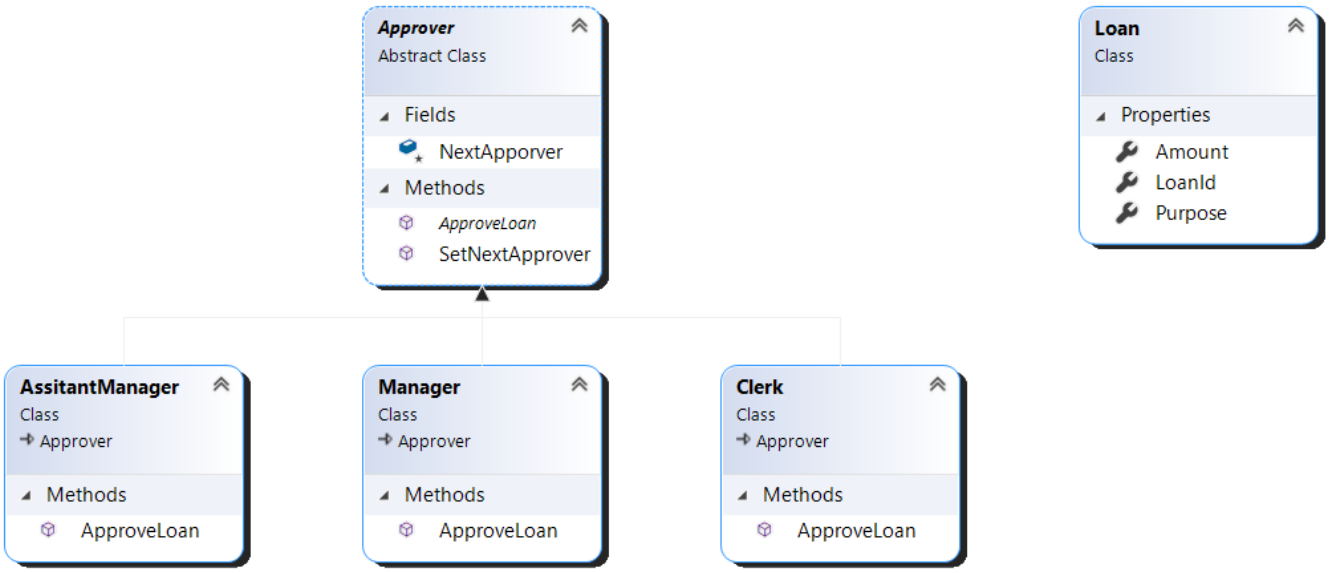
1. **Client -** This is the class that generates the request and passes it to the first handler in the chain of responsibility.
2. **Handler -** This is the abstract class that contains a member that holds the next handler in the chain and an associated method to set this successor. It also has an abstract method that must be implemented by concrete classes to handle the request or pass it to the next object in the pipeline.
3. **ConcreteHandlerA & ConcreteHandlerB -** These are concrete handlers classes inherited from the Handler class. These include the functionality to handle some requests and pass others to the next item in the chain of requests.

C# - Implementation Code:

```
public abstract class Handler
{
    protected Handler _successor;

    public abstract void HandleRequest(int request);

    public void SetSuccessor(Handler successor)
    {
        _successor = successor;
    }
}

public class ConcreteHandlerA : Handler
{
    public override void HandleRequest(int request)
    {
        if (request == 1)
            Console.WriteLine("Handled by ConcreteHandlerA");
        else if (_successor != null)
            _successor.HandleRequest(request);
    }
}

public class ConcreteHandlerB : Handler
{
    public override void HandleRequest(int request)
    {
        if (request > 10)
            Console.WriteLine("Handled by ConcreteHandlerB");
        else if (_successor != null)
            _successor.HandleRequest(request);
    }
}
```

Q3. Explain Chain of Responsibility pattern with a real-life example?

Ans. Let's take the following example where we are creating a chain of approvers for a loan.



C# - Implementation Code:

```
public class Loan
{
        public Guid LoanId { get; set; }
        public string Purpose { get; set; }
        public int Amount { get; set; }
}
public abstract class Approver
{
        protected Approver NextApporver;

        public void SetNextApprover(Approver nextApprover)
        {
            this.NextApporver = nextApprover;
        }
        public abstract bool ApproveLoan(Loan loan, ref string reason);
}
public class Clerk : Approver
{
        public override bool ApproveLoan(Loan loan, ref string reason)
        {
            if (loan.Amount <= 1000)
            {
                //clerk can approve
                return true;
            }
            else
            {
                //for >1k, go for assistant manager llevel
                if (NextApporver != null)
                {
                    return NextApporver.ApproveLoan(loan, ref reason);
                }
                return true;
```

```
            }
        }
}
public class AssitantManager : Approver
{
        public override bool ApproveLoan(Loan loan, ref string reason)
        {
            if (loan.Amount <= 10000)
            {
                //AssitantManager can approve
                return true;
            }
            else
            {
                //for >10k, go for  manager llevel
                if (NextApporver != null)
                {
                    return NextApporver.ApproveLoan(loan, ref reason);
                }
                return true;
            }
        }
}
public class Manager : Approver
{
        public override bool ApproveLoan(Loan loan, ref string reason)
        {
            if (loan.Amount <= 100000)
            {
                //for <=100k, go for  manager level
                if (NextApporver != null)
                {
                    return NextApporver.ApproveLoan(loan, ref reason);
                }
                return true;
            }
            else
            {
                reason = "This is too much!";
                return false;
            }
        }
}
class Program
{
        static void Main(string[] args)
        {
            Loan loan = new Loan
            {
                LoanId = Guid.NewGuid(),
                Purpose = "Home Loan",
                Amount = 10000
            };

            Approver clerk = new Clerk();
            Approver assistanManager = new AssitantManager();
            Approver manager = new Manager();
```

```
            //set chain of handler
            clerk.SetNextApprover(assistanManager);
            assistanManager.SetNextApprover(manager);

            string reason = "";
            if (clerk.ApproveLoan(loan, ref reason))
            {
                reason = "Apprroved." + reason;
            }
            else
            {
                reason = "Disapprroved." + reason;
            }
            Console.WriteLine(reason);
        }
}
```

Q4. When to use chain of responsibility pattern?

Ans. chain of responsibility pattern is useful in the following cases:

- A set of handlers to handle a request.
- A scenario within you need to pass a request to one handler among a list of handlers at run-time based on certain conditions.
- The exception handling system in C# is a good example of this pattern. Since an exception is thrown by a piece of code in C# is handled by a set of try-catch blocks. Here catch blocks act as possible handlers to handle the exception.

Q5. What is Command Design pattern? Explain it with an example?

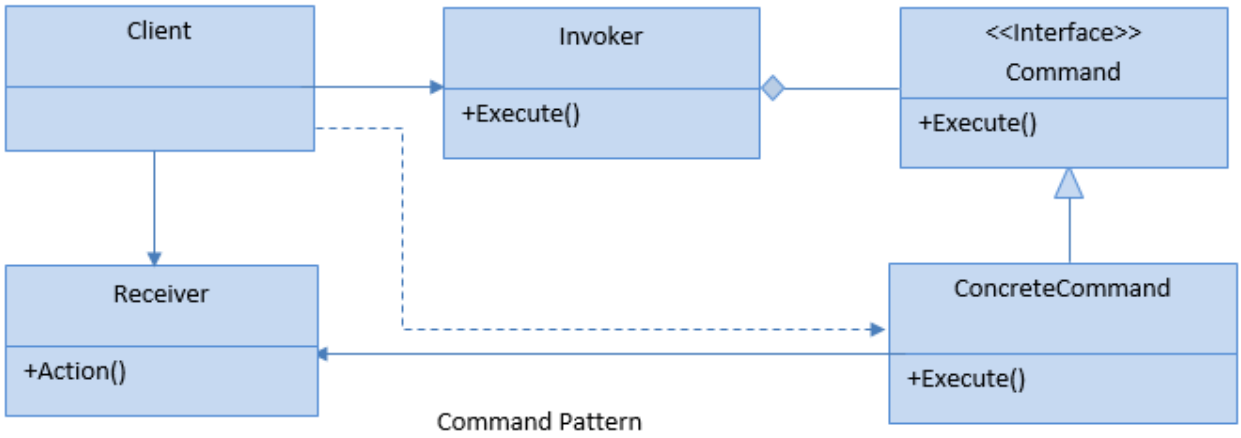
Ans. In this pattern, a request is wrapped under an object as a command and passed to the invoker object. The invoker object passes the command to the appropriate object which can handle it and that object executes the command.

The examples for command design patterns are:

- Menu systems in applications such as Editor, IDE etc.
- Update UI via commands
- Auditing and logging of all changes via commands

Q6. Explain command pattern with UML diagram and code example?

Ans. The UML diagram for the command design pattern is shown below:



Command Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Client -** This is the class that creates and executes the command object.
2. **Invoker -** Asks the command to act.
3. **Command** – This is an interface that specifies the *Execute* operation.
4. **ConcreteCommand** – This is a class that implements the *Execute* operation by invoking operation(s) on the Receiver
5. **Receiver –** This is a class that performs the *Action* associated with the request.

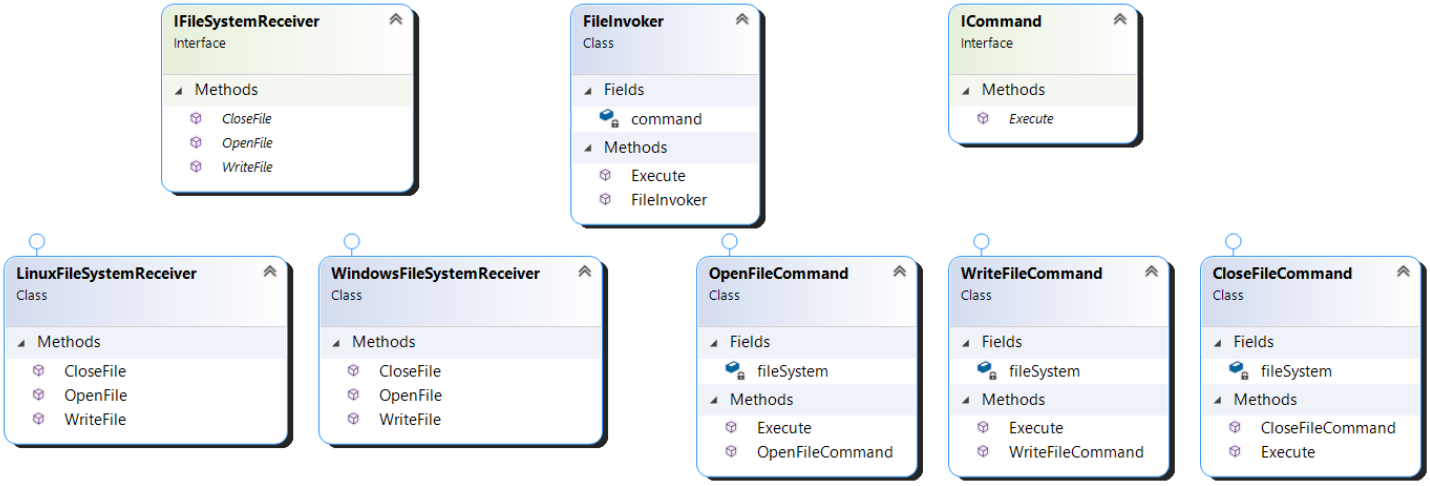
C# - Implementation Code:

```
public class Client
{
    public void RunCommand()
    {
        Invoker invoker = new Invoker();
        Receiver receiver = new Receiver();
        ConcreteCommand command = new ConcreteCommand(receiver);
        command.Parameter = "Dot Net Tricks !!";
        invoker.Command = command;
        invoker.ExecuteCommand();
    }
}

public class Receiver
{
    public void Action(string message)
    {
        Console.WriteLine("Action called with message: {0}", message);
    }
}

public class Invoker
{
    public ICommand Command { get; set; }

    public void ExecuteCommand()
    {
```

```
        Command.Execute();
    }
}

public interface ICommand
{
    void Execute();
}

public class ConcreteCommand : ICommand
{
    protected Receiver _receiver;
    public string Parameter { get; set; }

    public ConcreteCommand(Receiver receiver)
    {
        _receiver = receiver;
    }
    public void Execute()
    {
        _receiver.Action(Parameter);
    }
}
```

Q7. Explain Command pattern with a real-life example?

Ans. Let's take the following example where we are executing various commands on a file at different OS.



C# - Implementation Code:

```
public interface IFileSystemReceiver
    {
        void OpenFile();
        void WriteFile();
        void CloseFile();
    }
public class LinuxFileSystemReceiver : IFileSystemReceiver
    {
```

```
        public void CloseFile()
        {
            Console.WriteLine("Closing file in Linux OS");
        }

        public void OpenFile()
        {
            Console.WriteLine("Opening file in Linux OS");
        }

        public void WriteFile()
        {
            Console.WriteLine("Writing file in Linux OS");
        }
    }
public class WindowsFileSystemReceiver : IFileSystemReceiver
    {
        public void CloseFile()
        {
            Console.WriteLine("Closing file in Windows OS");
        }

        public void OpenFile()
        {
            Console.WriteLine("Opening file in Windows OS");
        }

        public void WriteFile()
        {
            Console.WriteLine("Writing file in Windows OS");
        }
    }
public class FileInvoker
    {
        ICommand command;
        public FileInvoker(ICommand c)
        {
            command = c;
        }

        public void Execute()
        {
            command.Execute();
        }
    }
public interface ICommand
    {
        void Execute();
    }
public class OpenFileCommand : ICommand
    {
        private IFileSystemReceiver fileSystem;
        public OpenFileCommand(IFileSystemReceiver fs)
        {
            fileSystem = fs;
        }
        public void Execute()
        {
```

```
            fileSystem.OpenFile();
        }
    }
public class WriteFileCommand : ICommand
    {
        private IFileSystemReceiver fileSystem;
        public WriteFileCommand(IFileSystemReceiver fs)
        {
            fileSystem = fs;
        }
        public void Execute()
        {
            fileSystem.WriteFile();
        }
    }
public class CloseFileCommand : ICommand
    {
        private IFileSystemReceiver fileSystem;
        public CloseFileCommand(IFileSystemReceiver fs)
        {
            fileSystem = fs;
        }
        public void Execute()
        {
            fileSystem.CloseFile();
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            //create the receiver object
            IFileSystemReceiver fs = new WindowsFileSystemReceiver();

            //define
            FileInvoker fileInvoker;

            //Create comand and associate with receiver
            OpenFileCommand openFileCommand = new OpenFileCommand(fs);
            fileInvoker = new FileInvoker(openFileCommand);
            fileInvoker.Execute();

            WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
            fileInvoker = new FileInvoker(writeFileCommand);
            fileInvoker.Execute();

            CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
            fileInvoker = new FileInvoker(closeFileCommand);
            fileInvoker.Execute();
        }
    }
```

Q8. When to use command Design pattern?

Ans. Command design pattern is useful in following cases:

- Need to implement callback functionalities.

- Need to support Redo and Undo functionality for commands.
- Sending requests to different receivers can handle it in different ways.
- Need for auditing and logging of all changes via commands.

Q9. What is Interpreter Design pattern? Explain it with examples?

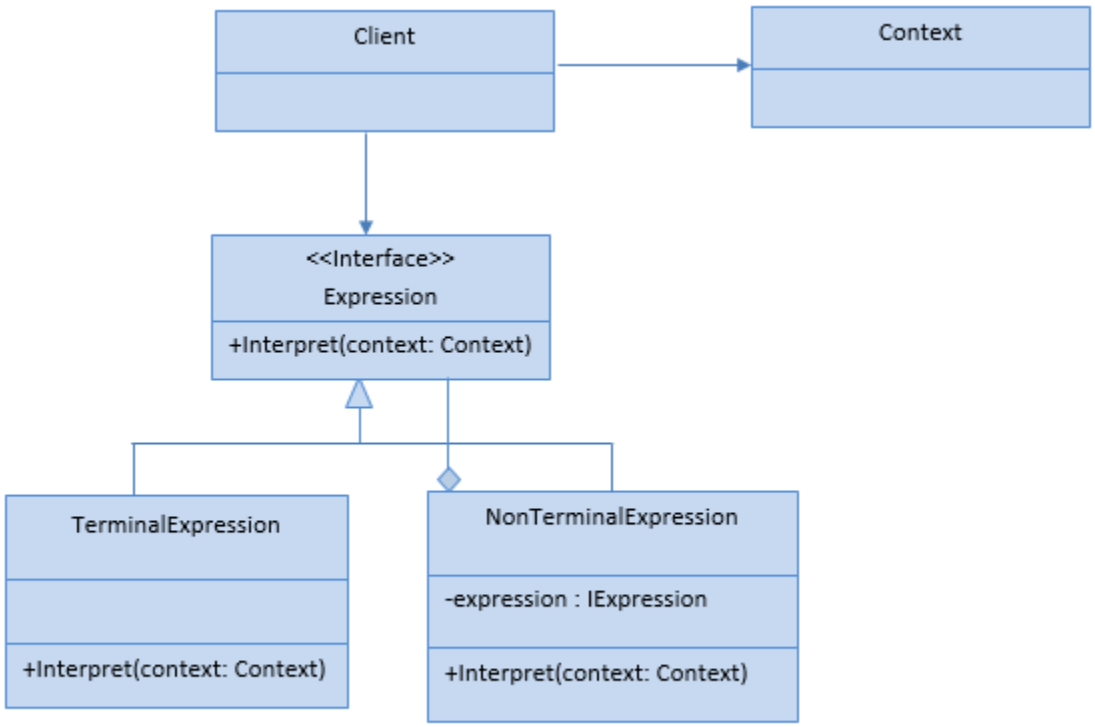
Ans. This pattern evaluates/interprets the instructions (Source Code) written in a language grammar or notation. Involves implementing an expression interface that tells to interpret a particular context.

The examples for interpreter pattern are:

- Language Compiler
- Language Interpreter/Translator
- String/Data Parsing

Q10. Explain Interpreter pattern with UML diagram and code example?

Ans. The UML diagram for interpreter design pattern is shown below:



Interpreter Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Client -** This is the class that builds the abstract syntax tree for a set of instructions in the given grammar. This tree builds with the help of instances of NonTerminalExpression and TerminalExpression classes.
2. **Context –** This is a class that contains information (input and output), which is used by the Interpreter.

3. **Expression –** This is an interface that defines the Interpret operation, which must be implemented by each subclass.
4. **NonTerminal** – This is a class that implements the Expression. This can have other instances of Expression.
5. **Terminal** – This is a class that implements the Expression.

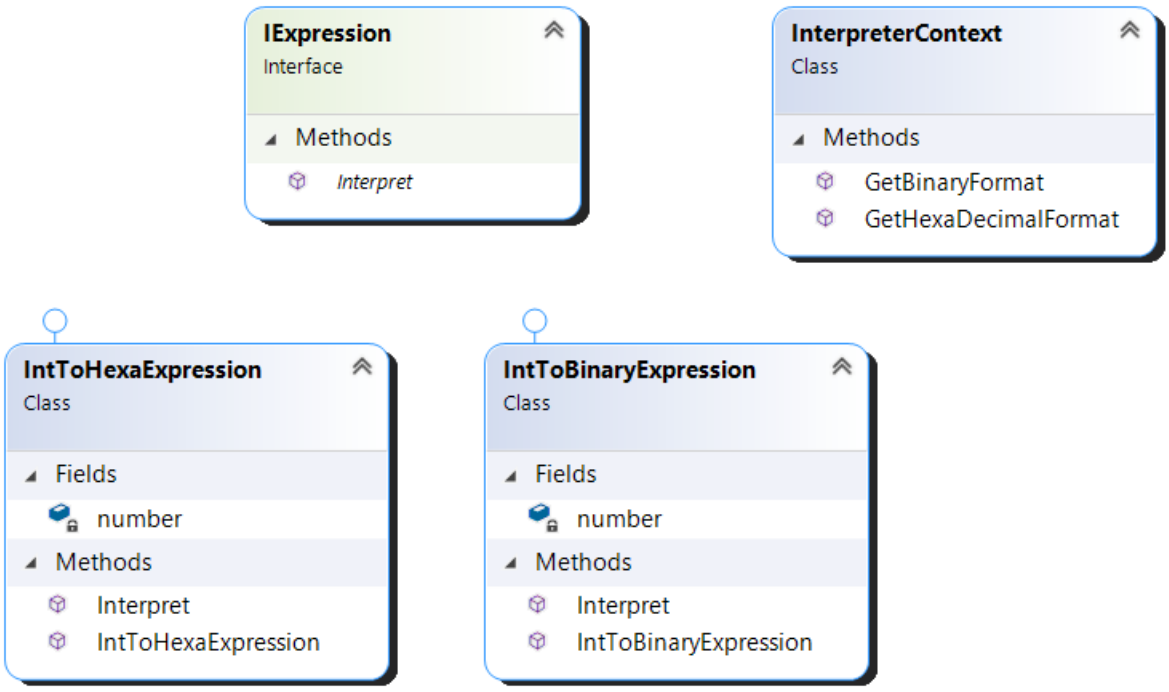
C# Implementation Code:

```
public class Client
{
    public void BuildAndInterpretCommands()
    {
        Context context = new Context("Dot Net context");
        NonterminalExpression root = new NonterminalExpression();
        root.Expression1 = new TerminalExpression();
        root.Expression2 = new TerminalExpression();
        root.Interpret(context);
    }
}

public class Context
{
    public string Name { get; set; }

    public Context(string name)
    {
        Name = name;
    }
}

public interface IExpression
{
    void Interpret(Context context);
}

public class TerminalExpression : IExpression
{
    public void Interpret(Context context)
    {
        Console.WriteLine("Terminal for {0}.", context.Name);
    }
}

public class NonterminalExpression : IExpression
{
    public IExpression Expression1 { get; set; }

    public IExpression Expression2 { get; set; }

    public void Interpret(Context context)
    {
        Console.WriteLine("Nonterminal for {0}.", context.Name);
        Expression1.Interpret(context);
        Expression2.Interpret(context);
    }
```

```
}
```

Q11. Explain Interpreter pattern with a real-life example?

Ans. Let's take the following example where we are converting a number into Hexa decimal and Binary format.



C# - Implementation Code:

```
public interface IExpression
    {
        string Interpret(InterpreterContext ic);
    }
public class IntToBinaryExpression : IExpression
    {
        int number;
        public IntToBinaryExpression(int num)
        {
            number = num;
        }
        public string Interpret(InterpreterContext ic)
        {
            return ic.GetBinaryFormat(number);
        }
    }
public class IntToHexaExpression : IExpression
    {
        int number;
        public IntToHexaExpression(int num)
        {
            number = num;
        }
```

```
        public string Interpret(InterpreterContext ic)
        {
            return ic.GetHexaDecimalFormat(number);
        }
    }
public class InterpreterContext
    {
        public string GetBinaryFormat(int number)
        {
            return Convert.ToString(number, 2);
        }

        public string GetHexaDecimalFormat(int number)
        {
            return Convert.ToString(number, 16);
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            InterpreterContext ic = new InterpreterContext();
            int number = 2;

            IExpression exp = new IntToBinaryExpression(number);
            string binaryValue = exp.Interpret(ic);

             exp = new IntToHexaExpression(number);
            string hexaValue = exp.Interpret(ic);

            Console.WriteLine(binaryValue);
            Console.WriteLine(hexaValue);
        }
    }
```

Q12. When to use Interpreter Design pattern?

Ans. Interpreter design pattern is useful in following cases:

- Need to interpret a grammar that can be represented as a large syntax trees.
- Parsing tools are available.
- Efficiency is not a concern.

Q13. What is Iterator Design pattern and when to use it?

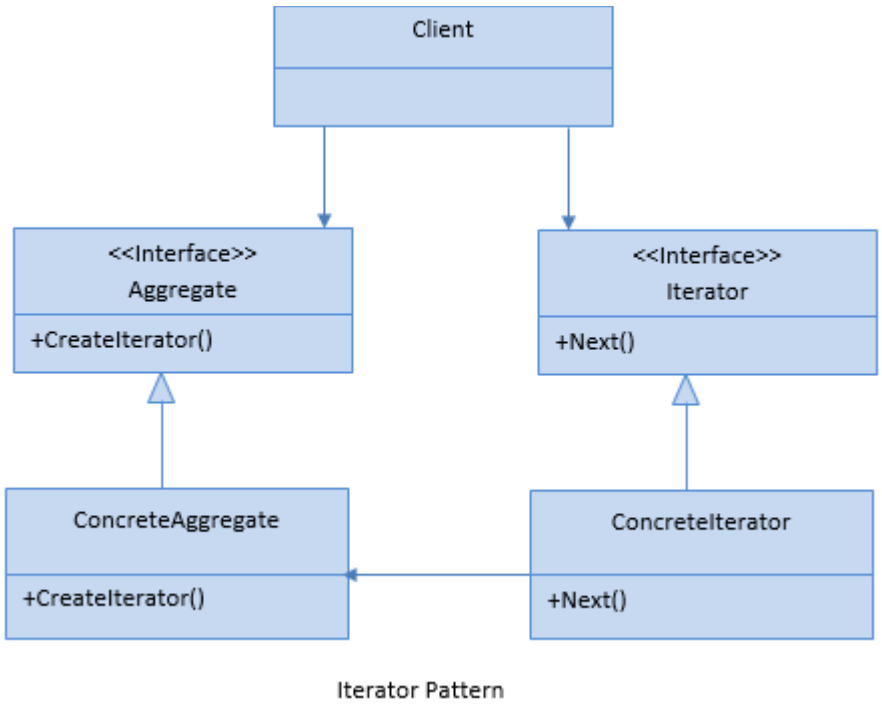
Ans. This pattern provides a way to access the elements from a collection in a sequential manner without exposing its underlying structure. It provides read-only access and no access to inner collection.

The examples for iterator patterns are:

- Collection Iteration
- Collection Elements Ordering (Asc, Desc)

Q14. Explain Iterator pattern with UML diagram and code example?

Ans. The UML diagram for the iterator design pattern is shown below:



Iterator Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Client** - This is the class that contains an object collection and uses the *Next* operation of the iterator to retrieve items from the aggregate in an appropriate sequence.
2. **Iterator –** This is an interface that defines operations for accessing the collection elements in a sequence.
3. **ConcreteIterator –** This is a class that implements the Iterator interface.
4. **Aggregate –** This is an interface that defines an operation to create an iterator.
5. **ConcreteAggregate –** This is a class that implements an Aggregate interface.

C# Implementation Code:

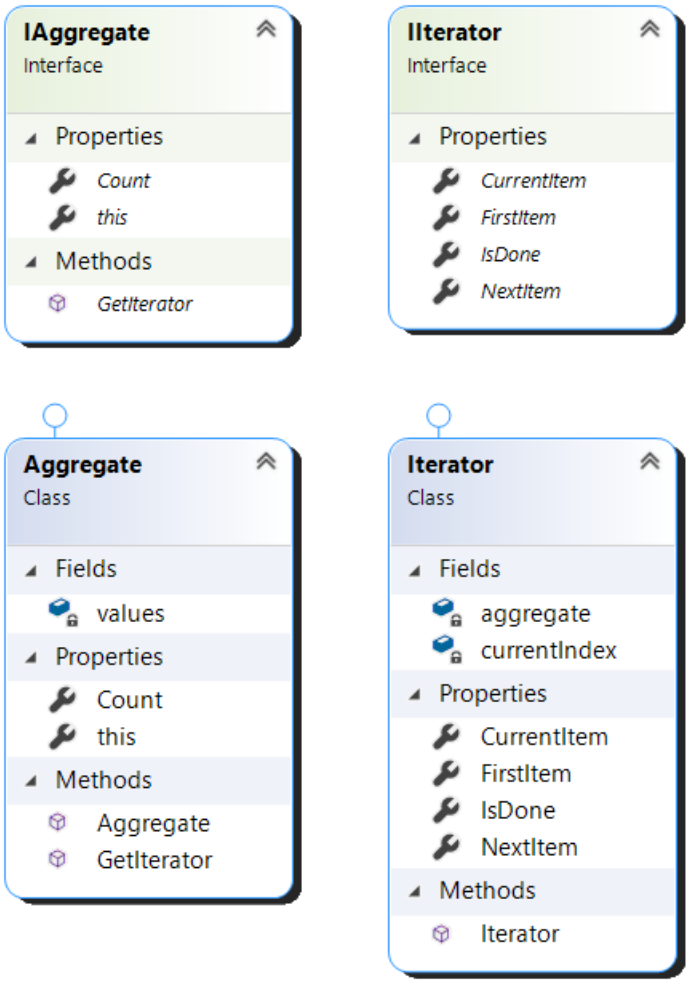
```
public class Client
{
    public void UseIterator()
    {
        ConcreteAggregate aggr = new ConcreteAggregate();
        aggr.Add("One");
        aggr.Add("Two");
        aggr.Add("Three");
        aggr.Add("Four");
        aggr.Add("Five");

        Iterator iterator = aggr.CreateIterator();
        while (iterator.Next())
        {
            string item = (string)iterator.Current;
```

```
            Console.WriteLine(item);
        }
    }
}

public interface Aggregate
{
    Iterator CreateIterator();
}

public class ConcreteAggregate : Aggregate
{
    private ArrayList items = new ArrayList();

    public Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }

    public object this[int index]
    {
        get { return items[index]; }
    }

    public int Count
    {
        get { return items.Count; }
    }

    public void Add(object o)
    {
        items.Add(o);
    }
}
public interface Iterator
{
    object Current { get; }
    bool Next();

}

public class ConcreteIterator : Iterator
{
    private ConcreteAggregate aggregate;
    int index;

    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this.aggregate = aggregate;
        index = -1;
    }

    public bool Next()
    {
        index++;
```

```
        return index < aggregate.Count;
    }

    public object Current
    {
        get
        {
            if (index < aggregate.Count)
                return aggregate[index];
            else
                throw new InvalidOperationException();
        }
    }
}
```

Q15. Explain Iterator pattern with a real-life example?

Ans. Let's take following example where, we are iterating a list of elements.



C# - Implementation Code:

```
public interface IAggregate
    {
        IIterator GetIterator();
        string this[int index] { get; set; }
        int Count { get; }
    }
public class Aggregate : IAggregate
    {
        List<string> values = null;
        public Aggregate()
        {
            values = new List<string>();
        }
        public string this[int index]
        {
            get
            {
                if (index < values.Count)
                {
                    return values[index];
                }
                else
                {
                    return string.Empty;
                }
            }
            set
            {
                values.Add(value);
            }
        }

        public int Count
        {
            get
            {
                return values.Count;
            }
        }

        public IIterator GetIterator()
        {
            return new Iterator(this);
        }
    }
public interface IIterator
    {
        string FirstItem { get; }
        string CurrentItem { get; }
        string NextItem { get; }
        bool IsDone { get; }
    }
public class Iterator : IIterator
    {
        IAggregate aggregate = null;
        int currentIndex = 0;
```

```
        public Iterator(IAggregate aggregate)
        {
            this.aggregate = aggregate;
        }

        public string FirstItem
        {
            get
            {
                currentIndex = 0;
                return aggregate[currentIndex];
            }
        }

        public string NextItem
        {
            get
            {
                currentIndex += 1;

                if (IsDone == false)
                {
                    return aggregate[currentIndex];
                }
                else
                {
                    return string.Empty;
                }
            }
        }

        public string CurrentItem
        {
            get
            {
                return aggregate[currentIndex];
            }
        }

        public bool IsDone
        {
            get
            {
                if (currentIndex < aggregate.Count)
                {
                    return false;
                }
                return true;
            }
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            Aggregate collection = new Aggregate();
```

```
            collection[0] = "10";
            collection[1] = "11";
            collection[2] = "21";
            collection[3] = "31";
            collection[4] = "41";
            collection[5] = "51";

            IIterator iter = collection.GetIterator();
            string s = iter.FirstItem;
            while (!iter.IsDone)
            {
                Console.WriteLine(s);
                s = iter.NextItem;
            }
        }
    }
```

Q16. When to use Iterator Design pattern?

Ans. The iterator design pattern is useful in the following cases:

- Allows accessing the elements of a collection object in a sequential manner without knowing its underlying structure.
- Multiple or concurrent iterations are required over collections elements.
- Provides a uniform interface for accessing the collection elements.

Q17. What is Mediator Design pattern and when to use it?

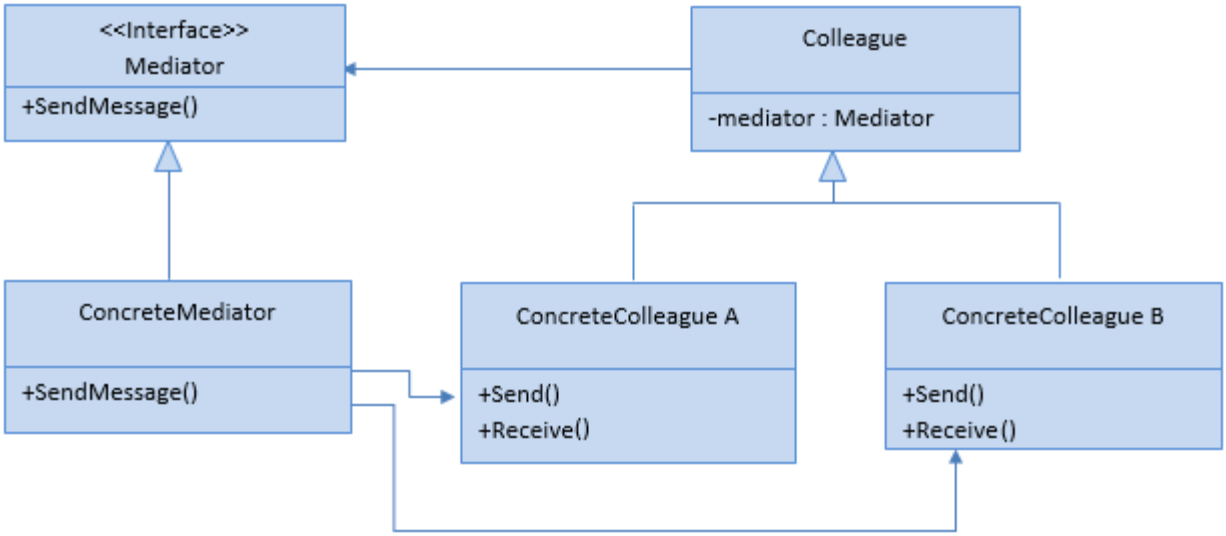
Ans. This pattern Allows multiple objects to communicate with each others without knowing each other structure. This pattern defines an object which encapsulates how the objects will interact with each other and supports easy maintainability of the code by loose coupling.

The examples for mediator pattern are:

- Messaging
- Chat

Q18. Explain Mediator pattern with UML diagram and code example?

Ans. The UML diagram for mediator design pattern is shown below:



Mediator Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Mediator -** This is an interface that defines operations which can be called by colleague objects for communication.
2. **ConcreteMediator –** This is a class that implement the communication operations of the Mediator interface.
3. **Colleague –** This is a class that defines a single, protected field that holds a reference to a mediator.
4. **ConcreteColleagueA/B –** These are the classes that communicate with each other via the mediator.

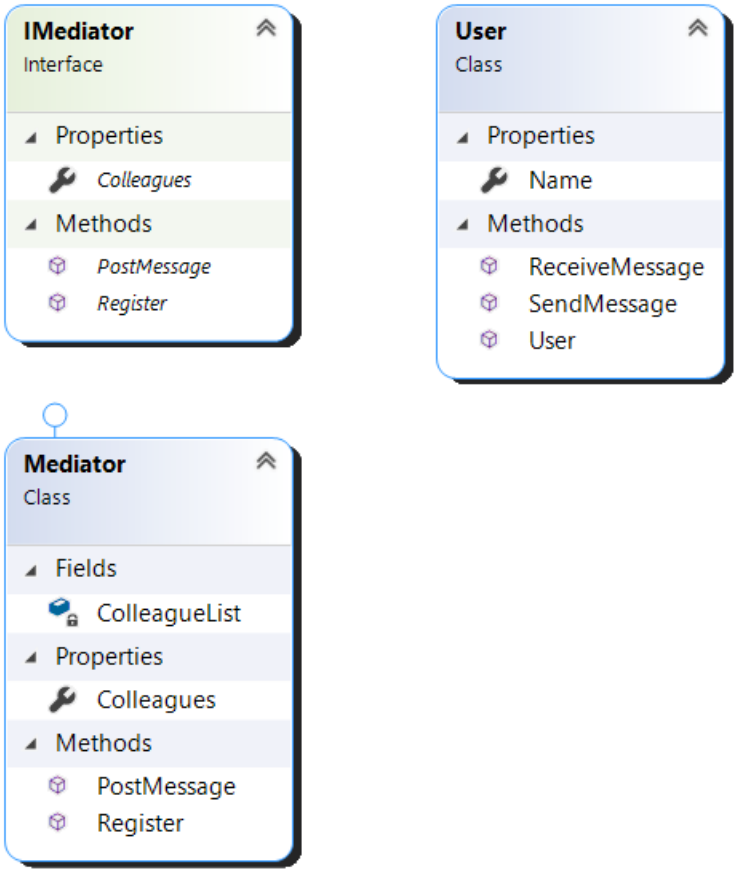
C# Implementation Code:

```
public abstract class Colleague
{
    protected IMediator _mediator;

    public Colleague(IMediator mediator)
    {
        _mediator = mediator;
    }
}

public class ConcreteColleagueA : Colleague
{
    public ConcreteColleagueA(IMediator mediator) : base(mediator) { }

    public void Send(string msg)
    {
        Console.WriteLine("A send message:" + msg);
        _mediator.SendMessage(this, msg);
    }

    public void Receive(string msg)
```

```
    {
        Console.WriteLine("A receive message:" + msg);
    }
}

public class ConcreteColleagueB : Colleague
{
    public ConcreteColleagueB(IMediator mediator) : base(mediator) { }

    public void Send(string msg)
    {
        Console.WriteLine("B send message:" + msg);
        _mediator.SendMessage(this, msg);
    }

    public void Receive(string msg)
    {
        Console.WriteLine("B receive message:" + msg);
    }
}

public interface IMediator
{
    void SendMessage(Colleague caller, string msg);
}

public class ConcreteMediator : IMediator
{
    public ConcreteColleagueA Colleague1 { get; set; }

    public ConcreteColleagueB Colleague2 { get; set; }

    public void SendMessage(Colleague caller, string msg)
    {
        if (caller == Colleague1)
            Colleague2.Receive(msg);
        else
            Colleague1.Receive(msg);
    }
}
```

Q19. Explain the Mediator pattern with a real-life example?

Ans. Let's take the following example where Users are sending and receiving messages through the mediator class.



C# - Implementation Code:

```
public interface IMediator
    {
        List<User> Colleagues { get; }
        void PostMessage(User sender, string message);
        void Register(User colleague);
    }
public class Mediator : IMediator
    {
        private List<User> ColleagueList = new List<User>();
        public List<User> Colleagues
        {
            get
            {
                return ColleagueList;
            }
        }

        public void PostMessage(User sender, string message)
        {
            foreach (var user in ColleagueList)
            {
                if (user != sender)
                {
                    user.ReceiveMessage(message, sender.Name);
                }
```

```
            }
        }

        public void Register(User colleague)
        {
            ColleagueList.Add(colleague);
        }
    }
public class User
    {
        public string Name { get; set; }
        public User(string name)
        {
            Name = name;
        }

        public void SendMessage(IMediator mediator, string message)
        {
            mediator.PostMessage(this, message);
        }

        public void ReceiveMessage(string message, string sender)
        {
            Console.WriteLine(Name + ", received message from -" + sender + " : " + message);
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            User mohan = new User("Mohan");
            User rama = new User("Rama");
            User pavan = new User("Pavan");
            User rohit = new User("Rohit");

            IMediator mediator1 = new Mediator();
            mediator1.Register(mohan);
            mediator1.Register(rama);
            mediator1.Register(pavan);
            mohan.SendMessage(mediator1, "Hi Folks!");

            IMediator mediator2 = new Mediator();
            mediator2.Register(rama);
            mediator2.Register(rohit);

            rama.SendMessage(mediator2, "A private message");
        }
    }
```

Q20. When to use Mediator Design pattern?

Ans. The mediator design pattern is useful in the following cases:

- Communication between multiple objects is well defined but potentially complex.
- When too many relationships exist and a common point of control or communication is required.
- Some objects can be grouped and customized based on behaviours.

Q21. What is Memento Design pattern and when to use it?

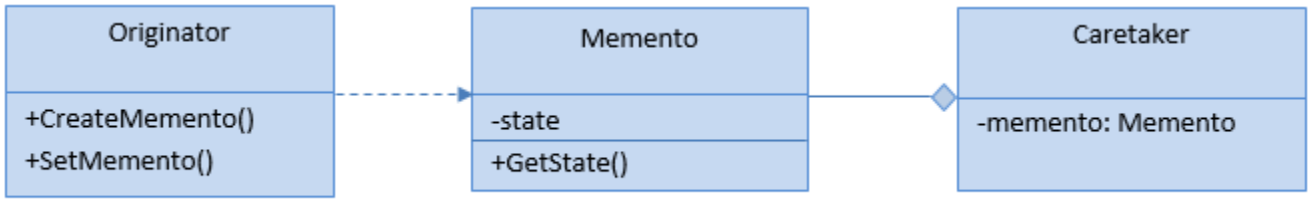
Ans. This pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

The examples for memento patterns are:

- Undo Operation
- Transaction Rollback

Q22. Explain Memento pattern with UML diagram and code example?

Ans. The UML diagram for the memento design pattern is shown below:



Memento Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Originator -** This is a class that creates a memento object containing a snapshot of the Originator's current state. It also restores the Originator to a previously stored state by using *SetMemento* operation.
2. **Memento –** This is a class that holds the information about the Originator's saved state.
3. **Caretaker –** This is a class that is used to hold a Memento object for later use. This acts as a store only; it never examines or modify the contents of the Memento object.

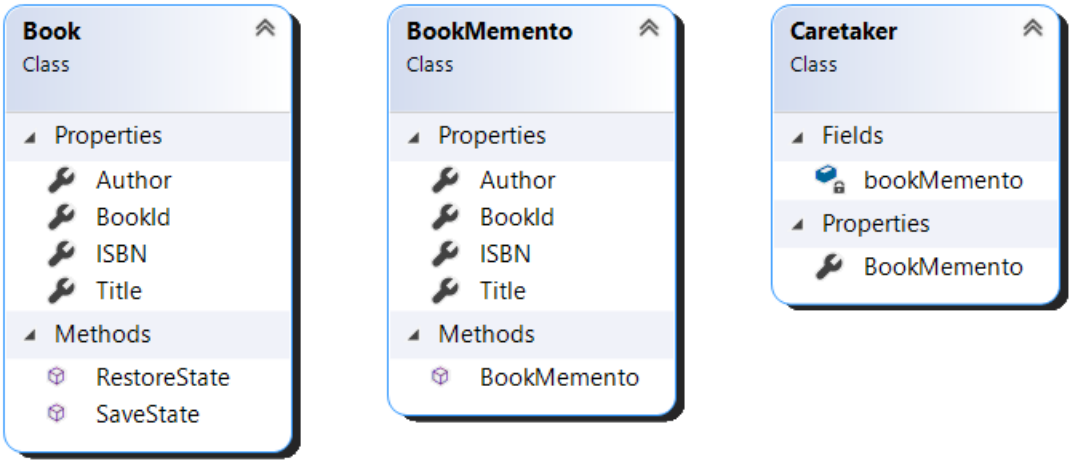
C# Implementation Code:

```
public class Originator
{
    private string state;

    public Memento CreateMemento()
    {
        return new Memento(state);
    }

    public void SetMemento(Memento memento)
    {
        this.state = memento.GetState();
    }
}

public class Memento
{
    private string state;

    public Memento(string state)
```

```
    {
        this.state = state;
    }

    public string GetState()
    {
        return state;
    }
}

public class Caretaker
{
    public Memento Memento { get; set; }
}
```

Q23. Explain Memento pattern with a real-life example?

Ans. Let's take the following example where we have Book class is used to save and restore states.



C# - Implementation Code:

```
//originator
 public class Book
    {
        public int BookId { get; set; }
        public string Title { get; set; }
        public string ISBN { get; set; }
        public string Author { get; set; }

        public void RestoreState(BookMemento bookMemento)
        {
            BookId = bookMemento.BookId;
            Title = bookMemento.Title;
            ISBN = bookMemento.ISBN;
            Author = bookMemento.Author;
        }
        public BookMemento SaveState()
        {
            return new BookMemento(BookId, Title, ISBN, Author);
        }
```

```
    }
public class BookMemento
    {
        public int BookId { get; set; }
        public string Title { get; set; }
        public string ISBN { get; set; }
        public string Author { get; set; }

        public BookMemento(int bookId, string title, string isbn, string author)
        {
            BookId = bookId;
            Title = title;
            ISBN = isbn;
            Author = author;
        }
    }
public class Caretaker
    {
        private BookMemento bookMemento;

        public BookMemento BookMemento
        {
            get
            {
                return bookMemento;
            }
            set
            {
                bookMemento = value;
            }
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            Book book = new Book();
            Caretaker careTaker = new Caretaker();

            book.BookId = 1;
            book.Title = "Design Pattern Book";
            book.ISBN = "567GJLJ6565";
            book.Author = "Shailendra";

            //Display book information
            Console.WriteLine($"BookId: {book.BookId}, Title: {book.Title}, ISBN: {book.ISBN},
Author: {book.Author}");

            //Save State
            careTaker.BookMemento = book.SaveState();
            //change
            book.Title = "Design Pattern Book 2.0";

            //Display new book information
            Console.WriteLine($"BookId: {book.BookId}, Title: {book.Title}, ISBN: {book.ISBN},
Author: {book.Author}");

            //Undo
```

```
            book.RestoreState(careTaker.BookMemento);
            //Display book information
            Console.WriteLine($"BookId: {book.BookId}, Title: {book.Title}, ISBN: {book.ISBN},
Author: {book.Author}");
        }
    }
```

Q24. When to use Memento Design pattern?

Ans. The memento design pattern is useful in the following cases:

- The state of an object needs to be saved and restored at a later time.
- The state of an object cannot be exposed directly by using an interface without exposing implementation.

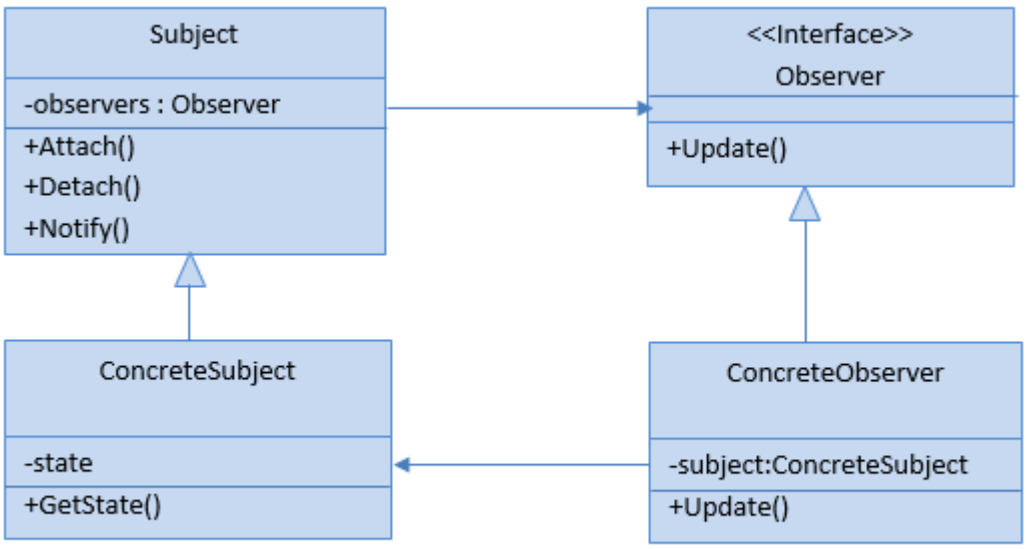
Q25. What is Observer Design pattern? Explain it with examples?

Ans. This pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

This pattern allows a single object, known as the subject, to publish changes to its state and other observer objects that depend upon the subject are automatically notified of any changes to the subject's state.

Q26. Explain Observer pattern with UML diagram and code example?

Ans. The UML diagram for observer design pattern is shown below:



Observer Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Subject -** This is a class that contains a private collection of the observers that are subscribed to a subject for notification by using *Notify* operation.

2. **ConcreteSubject –** This is a class that maintains its own state. When a change is made to its state, the object calls the base class's Notify operation to indicate this to all of its observers.
3. **Observer –** This is an interface that defines an operation *Update*, which is to be called when the subject's state changes.
4. **ConcreteObserver –** This is a class that implements the Observer interface and examines the subject to determine which information has changed.

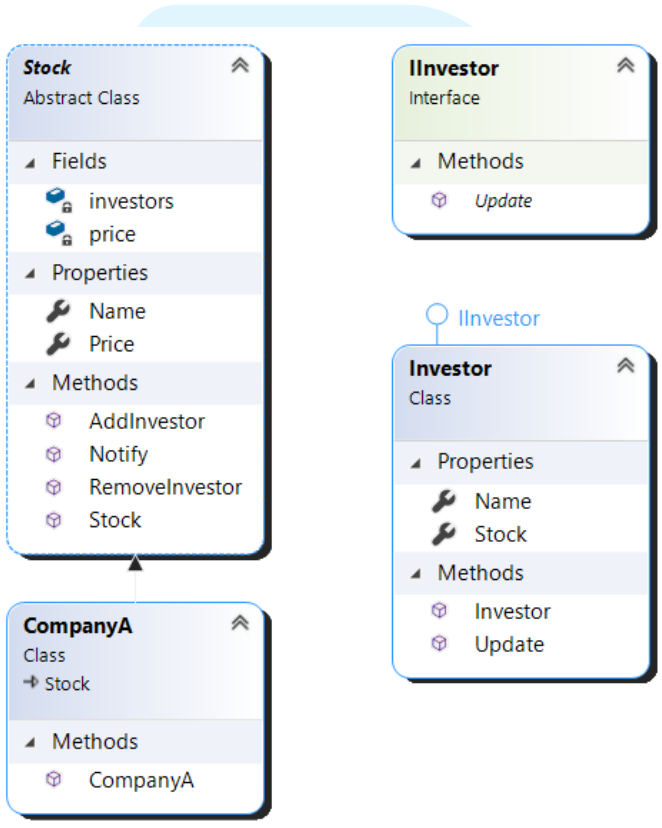
C# Implementation Code:

```
public abstract class Subject
{
    private ArrayList observers = new ArrayList();

    public void Attach(IObserver o)
    {
        observers.Add(o);
    }

    public void Detach(IObserver o)
    {
        observers.Remove(o);
    }

    public void Notify()
    {
        foreach (IObserver o in observers)
        {
            o.Update();
        }
    }
}

public class ConcreteSubject : Subject
{
    private string state;

    public string GetState()
    {
        return state;
    }

    public void SetState(string newState)
    {
        state = newState;
        Notify();
    }
}

public interface IObserver
{
    void Update();
}
```

```
public class ConcreteObserver : IObserver
{
    private ConcreteSubject subject;

    public ConcreteObserver(ConcreteSubject sub)
    {
        subject = sub;
    }

    public void Update()
    {
        string subjectState = subject.GetState();
        Console.WriteLine(subjectState);
    }
}
```

Q27. Explain Observer pattern with a real-life example?

Ans. Let's take the following example where an Investor observing the CompanyA stock and get notified when changed.



C# - Implementation Code:

```
public interface IInvestor
{
    void Update(Stock stock);
}
```

```
public class Investor : IInvestor
{
    public string Name { get; set; }
    public Stock Stock { get; set; }

    public Investor(string name)
    {
        Name = name;
    }

    public void Update(Stock stock)
    {
        Console.WriteLine($"Notification to {Name}: Price of {stock.Name} change to
{stock.Price}");
    }
}

public abstract class Stock
{
    private double price;
    public string Name { get; private set; }
    public double Price
    {
        get
        {
            return price;
        }
        set
        {
            if (price != value)
            {
                price = value;
                Notify();
            }
        }
    }

    private List<IInvestor> investors = new List<IInvestor>();
    public Stock(string name, double price)
    {
        Name = name;
        Price = price;
    }

    public void AddInvestor(IInvestor investor)
    {
        investors.Add(investor);
    }
    public void RemoveInvestor(IInvestor investor)
    {
        investors.Remove(investor);
    }
    public void Notify()
    {
        foreach (IInvestor investor in investors)
        {
            investor.Update(this);
        }
```

```
        Console.WriteLine("");
    }
}
public class CompanyA: Stock
{
    public CompanyA(double price) : base("CompanyA", price)
    {

    }
}
class Program
    {
        static void Main(string[] args)
        {
            CompanyA company = new CompanyA(120.0);
            company.AddInvestor(new Investor("Mohan"));
            company.AddInvestor(new Investor("Rohit"));

            company.Price = 119.0;
        }
    }
```

Q28. When to use Observer Design pattern?

Ans. The Observer design pattern is useful in the following cases:

- Changes in the state of an object need to be notified to a set of dependent objects, not all of them.
- Notification capability is required.
- The object sending the notification does not need to know about the receivers' objects.

Q29. What is State Design pattern and when to use it?

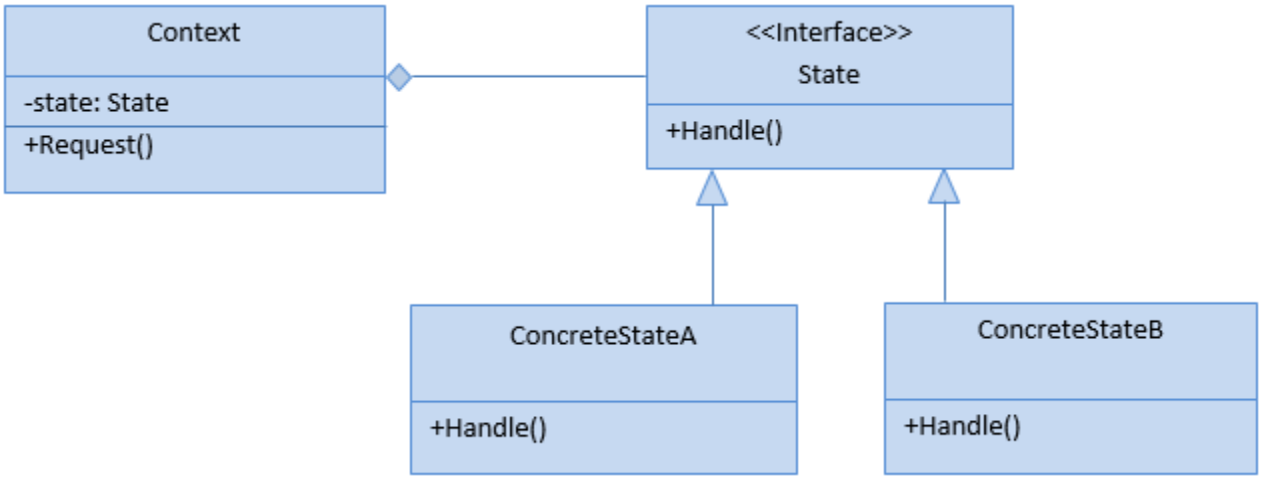
Ans. This pattern is used to alter the behaviour of an object when its internal state changes. Here, an object is created which represent various states and a context object whose behaviour varies as its state object changes.

The examples for state design pattern are:

- Navigation
- Validations

Q30. Explain State pattern with UML diagram and code example?

Ans. The UML diagram for state design pattern is shown below:



This pattern seems as a dynamic version of the Strategy pattern.

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Context -** This is a class that holds a concrete state object that provides the behaviour according to its current state. This is used by the clients.
2. **State –** This is an interface that is used by the Context object to access the changeable functionality.
3. **ConcreteStateA/B –** These are classes that implement State interface and provide the real functionality that will be used by the Context object. Each concrete state class provides behaviour that applies to a single state of the Context object.

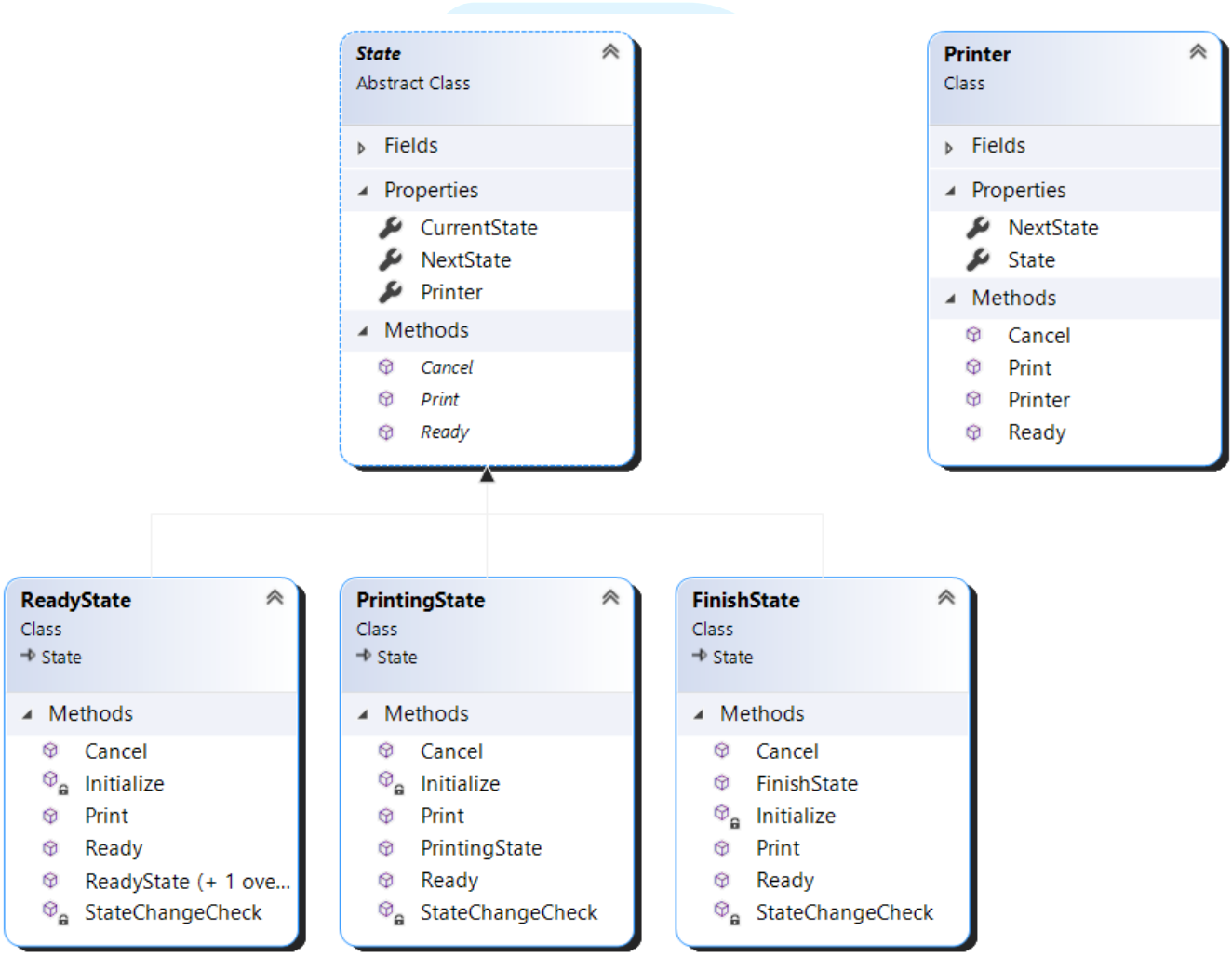
C# Implementation Code:

```
public class Context
{
    private IState state;

    public Context(IState newstate)
    {
        state = newstate;
    }

    public void Request()
    {
        state.Handle(this);
    }

    public IState State
    {
        get { return state; }
        set { state = value; }
    }
}
public interface IState
{
    void Handle(Context context);
}
```

```
public class ConcreteStateA : IState
{
    public void Handle(Context context)
    {
        Console.WriteLine("Handle called from ConcreteStateA");
        context.State = new ConcreteStateB();
    }
}
public class ConcreteStateB : IState
{
    public void Handle(Context context)
    {
        Console.WriteLine("Handle called from ConcreteStateB");
        context.State = new ConcreteStateA();
    }
}
```

Q31. Explain State pattern with a real-life example?

Ans. Let's take the following example where we are dealing with a printer state.



C# - Implementation Code:

```
public class Printer
    {
        private State state;
        private string name;

        public Printer(string name)
        {
            this.name = name;
            this.state = new ReadyState(this);
        }

        public string NextState
        {
            get { return state.NextState; }
        }

        public State State
        {
            get { return state; }
            set { state = value; }
        }

        public void Ready()
        {
            state.Ready();
            Console.WriteLine($"Name: {name}, CurrentState:{state.CurrentState},
NextState:{state.NextState}");
        }
        public void Print()
        {
            state.Print();
            Console.WriteLine($"Name: {name}, CurrentState:{state.CurrentState},
NextState:{state.NextState}");
        }
        public void Cancel()
        {
            state.Cancel();
            Console.WriteLine($"Name: {name}, CurrentState:{state.CurrentState},
NextState:{state.NextState}");
        }
    }
public abstract class State
    {
        protected Printer printer;
        protected string nextState;
        protected string currentState;
        protected int level;

        public Printer Printer
        {
            get { return printer; }
            set { printer = value; }
        }
        public string CurrentState
```

```
        {
            get { return currentState; }
            set { currentState = value; }
        }

        public string NextState
        {
            get { return nextState; }
            set { nextState = value; }
        }

        public abstract void Ready();
        public abstract void Print();
        public abstract void Cancel();
    }
public class ReadyState : State
    {
        public ReadyState(State state)
        {
            currentState = "Ready State";
            nextState = "Printing State";
            printer = state.Printer;
            Initialize();
        }

        public ReadyState(Printer printer)
        {
            currentState = "Ready State";
            nextState = "Printing State";
            this.printer = printer;
            Initialize();
        }

        private void Initialize()
        {
            level = 0;
            StateChangeCheck();
        }

        public override void Cancel()
        {
            level = 2;
            StateChangeCheck();
        }

        public override void Ready()
        {
            level = 0;
            StateChangeCheck();
        }

        public override void Print()
        {
            level = 1;
            StateChangeCheck();
        }

        private void StateChangeCheck()
```

```
        {
            switch (level)
            {
                case 0:
                    printer.State = this;
                    break;
                case 1:
                    printer.State = new PrintingState(this);
                    break;
                case 2:
                    printer.State = new FinishState(this);
                    break;
            }
        }
    }
public class PrintingState: State
    {
        public PrintingState(State state)
        {
            currentState = "Print State";
            this.nextState = "Finish State";
            this.printer = state.Printer;
            Initialize();
        }

        private void Initialize()
        {
            level = 1;
        }

        public override void Print()
        {
            level = 1;
            StateChangeCheck();
        }

        public override void Cancel()
        {
            level = 2;
            StateChangeCheck();
        }

        public override void Ready()
        {
            level = 0;
            StateChangeCheck();
        }

        private void StateChangeCheck()
        {
            switch (level)
            {
                case 0:
                    printer.State = new ReadyState(this);
                    break;
                case 1:
                    printer.State = this;
                    break;
```

```
                case 2:
                    printer.State = new FinishState(this);
                    break;
            }
        }
    }
public class FinishState : State
    {
        public FinishState(State state)
        {
            this.currentState = "Finish State";
            this.nextState = "Ready State";
            this.printer = state.Printer;
            Initialize();
        }

        private void Initialize()
        {
            level = 0;
        }

        public override void Print()
        {
            level = 1;
            StateChangeCheck();
        }

        public override void Cancel()
        {
            level = 2;
            StateChangeCheck();
        }

        public override void Ready()
        {
            level = 0;
            StateChangeCheck();
        }

        private void StateChangeCheck()
        {
            switch (level)
            {
                case 0:
                    printer.State = new ReadyState(this);
                    break;
                case 1:
                    printer.State = new PrintingState(this);
                    break;
                case 2:
                    printer.State = this;
                    break;
            }
        }
    }
```

Q32. When to use State Design pattern?

Ans. The Strategy design pattern is useful in the following cases:

- The behaviour of an object is changed based on its state.
- Preserve flexibility in assigning requests to handlers.
- An object is becoming complex, with many conditional behaviours.

Q33. What is Strategy Design pattern? Explain it with an example?

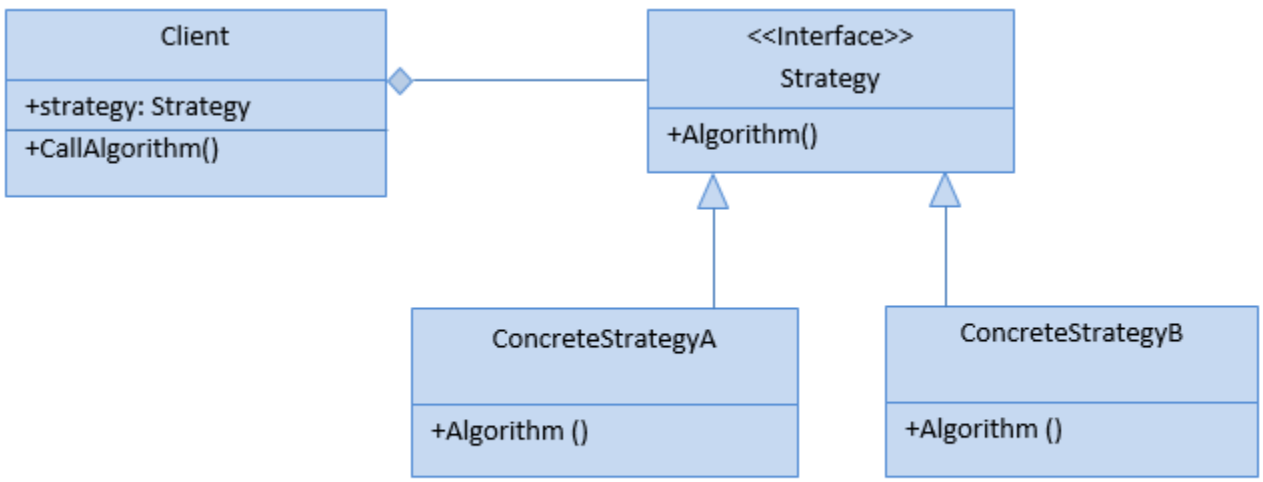
Ans. This pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. It allows a client to choose an algorithm from a family of algorithms at run-time and gives it a simple way to access it.

The examples for strategy pattern are:

- Discount Strategy
- Payment Strategy

Q34. Explain Strategy pattern with UML diagram and code example?

Ans. The UML diagram for strategy design pattern is shown below:



Strategy Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

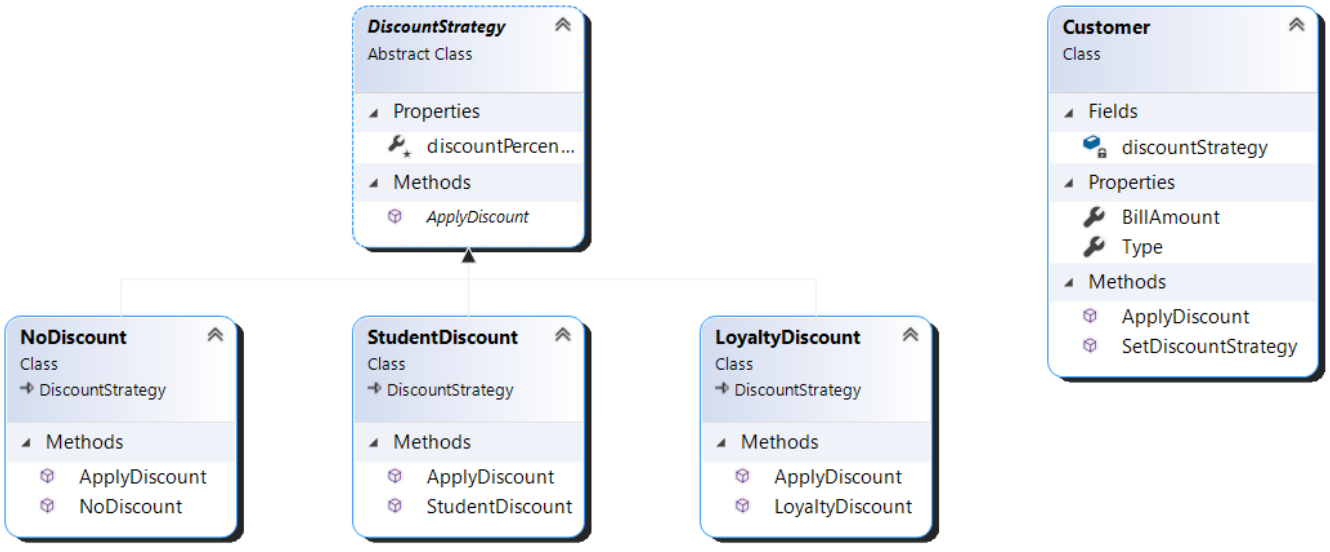
1. **Context -** This is a class that contains a property to hold the reference of a Strategy object. This property will be set at run-time according to the algorithm that is required.
2. **Strategy –** This is an interface that is used by the Context object to call the algorithm defined by a ConcreteStrategy.
3. **ConcreteStrategyA/B –** These are classes that implement the Strategy interface.

C# Implementation Code:

```
public class Client
{
    public IStrategy Strategy { get; set; }

    public void CallAlgorithm()
    {
        Console.WriteLine(Strategy.Algorithm());
    }
}

public interface IStrategy
{
    string Algorithm();
}

public class ConcreteStrategyA : IStrategy
{
    public string Algorithm()
    {
        return "Concrete Strategy A";
    }
}

public class ConcreteStrategyB : IStrategy
{
    public string Algorithm()
    {
        return "Concrete Strategy B";
    }
}
```

Q35. Explain Strategy pattern with a real-life example?

Ans. Let's take the following example where we are handing a product discount strategy.



C# - Implementation Code:

```
public class Customer
    {
        private DiscountStrategy discountStrategy;
        public string Type { get; set; }
        public decimal BillAmount { get; set; }

        public void SetDiscountStrategy(DiscountStrategy discountStrategy)
        {
            this.discountStrategy = discountStrategy;
        }

        public decimal ApplyDiscount(decimal sale)
        {
            return discountStrategy.ApplyDiscount(sale);
        }
    }
public abstract class DiscountStrategy
    {
        protected double discountPercentage { get; set; }
        public abstract decimal ApplyDiscount(decimal sale);
    }
public class NoDiscount: DiscountStrategy
    {
        public NoDiscount()
        {
            discountPercentage = 0;
        }
        public override decimal ApplyDiscount(decimal sale)
        {
            return (decimal)(100 - discountPercentage) * sale / 100;
        }
    }
public class StudentDiscount: DiscountStrategy
    {
        public StudentDiscount()
        {
            discountPercentage = 25;
        }

        public override decimal ApplyDiscount(decimal sale)
        {
            return (decimal)(100 - discountPercentage) * sale / 100;
        }
    }
public class LoyaltyDiscount: DiscountStrategy
    {
        public LoyaltyDiscount()
        {
            discountPercentage = 15;
        }

        public override decimal ApplyDiscount(decimal sale)
        {
            return (decimal)(100 - discountPercentage) * sale / 100;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Customer customer = new Customer();
            customer.Type = "Visiting";
            customer.BillAmount = 1200;

            if (customer.Type == "Student")
            {
                customer.SetDiscountStrategy(new StudentDiscount());
            }
            else if (customer.Type == "Loyal")
            {
                customer.SetDiscountStrategy(new LoyaltyDiscount());
            }
            else
            {
                customer.SetDiscountStrategy(new NoDiscount());
            }

            decimal finalBill = customer.ApplyDiscount(customer.BillAmount);
            Console.WriteLine($"Actual Bill: {customer.BillAmount}, Bill After
Discount:{finalBill}");
        }
    }
```

Q36. When to use Strategy Design pattern?

Ans. The Strategy design pattern is useful in the following cases:

- There are multiple strategies for a given problem and the selection criteria of a strategy is defined run-time.
- Many related classes only differ in their behaviours.
- The strategies use the data to which the client has no access.

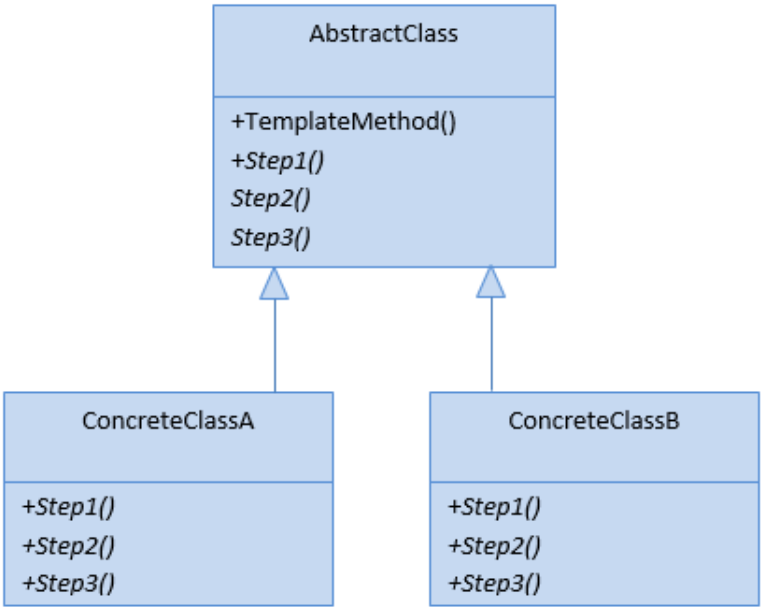
Q37. What is Template Method Design pattern and when to use it?

Ans. This pattern is used to define the basic steps of an algorithm and allow the implementation of the individual steps to be changed. This pattern looks similar to the strategy design pattern, but it changes the parts of an algorithm rather than replacing an entire algorithm.

The example for template pattern is an abstract class that defined the template method and this method includes the steps which are implemented by the subclasses.

Q38. Explain Template pattern with UML diagram and code example?

Ans. The UML diagram for template design pattern is shown below:



The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **AbstractClass -** This is an abstract class that contains template method and abstract operations for each of the steps that may be implemented by subclasses.
2. **ConcreteClassA/B –** These are subclasses which inherit the AbstractClass and override the abstract class operations.

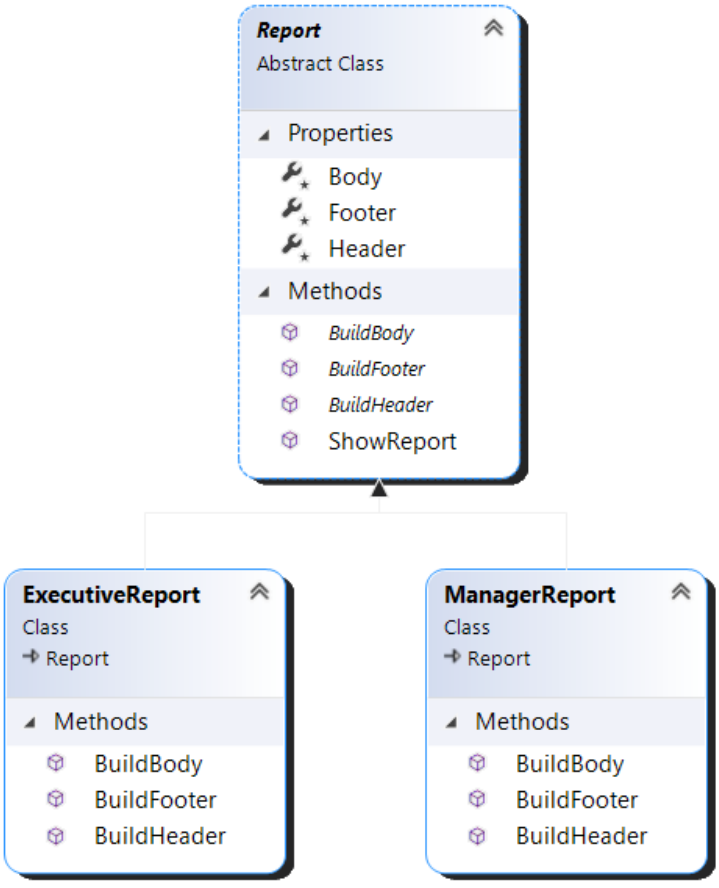
C# Implementation Code:

```
public abstract class AbstractClass
{
    public void TemplateMethod()
    {
        Step1();
        Step2();
        Step3();
    }
    public abstract void Step1();

    public abstract void Step2();

    public abstract void Step3();
}
public class ConcreteClassA : AbstractClass
{
    public override void Step1()
    {
        Console.WriteLine("Concrete Class A, Step 1");
    }

    public override void Step2()
    {
        Console.WriteLine("Concrete Class A, Step 2");
```

```
    }

    public override void Step3()
    {
        Console.WriteLine("Concrete Class A, Step 3");
    }
}
public class ConcreteClassB : AbstractClass
{
    public override void Step1()
    {
        Console.WriteLine("Concrete Class B, Step 1");
    }
    public override void Step2()
    {
        Console.WriteLine("Concrete Class B, Step 2");
    }
    public override void Step3()
    {
        Console.WriteLine("Concrete Class B, Step 3");
    }
}
```

Q39. Explain Template pattern with a real-life example?

Ans. Let's take the following example where we have templates for generating reports.



C# - Implementation Code:

```
public abstract class Report
    {
        protected string Header { get; set; }
        protected string Body { get; set; }
        protected string Footer { get; set; }

        public void ShowReport()
        {
            BuildHeader();
            BuildBody();
            BuildFooter();
        }

        public abstract void BuildHeader();
        public abstract void BuildBody();
        public abstract void BuildFooter();
    }
public class ExecutiveReport: Report
    {
        public override void BuildHeader()
        {
            Header = $"REPORT HEADER ON DATE: {DateTime.Now}\n";
            Console.WriteLine(Header);
        }
        public override void BuildBody()
        {
            Body = $"REPORT FOR ALL EXECUTIVE ON DATE: {DateTime.Now}\n";
            Console.WriteLine(Body);
        }
        public override void BuildFooter()
        {
            Footer = "Report by ABC Inc.";
            Console.WriteLine(Footer);
        }
    }
public class ManagerReport : Report
    {
        public override void BuildHeader()
        {
            Header = $"REPORT HEADER ON DATE: {DateTime.Now}\n";
            Console.WriteLine(Header);
        }
        public override void BuildBody()
        {
            Body = $"REPORT FOR ALL MANAGERS ON DATE: {DateTime.Now}\n";
            Console.WriteLine(Body);
        }
        public override void BuildFooter()
        {
            Footer = "Report by ABC Inc.";
            Console.WriteLine(Footer);
        }
    }
class Program
    {
        static void Main(string[] args)
```

```
        {
            Report exReport = new ExecutiveReport();
            exReport.ShowReport();

            Console.WriteLine("\n----------------------");
            Report mrReport = new ManagerReport();
            mrReport.ShowReport();
        }
    }
```

Q40. When to use Template Design pattern?

Ans. The template design pattern is useful in the following cases:

- An abstract view of an algorithm is required, but implementation varies according to the subclasses.
- Common behaviours of subclasses can be used to make to a common class.

Q41. What is Visitor Design pattern? Explain it with an example?

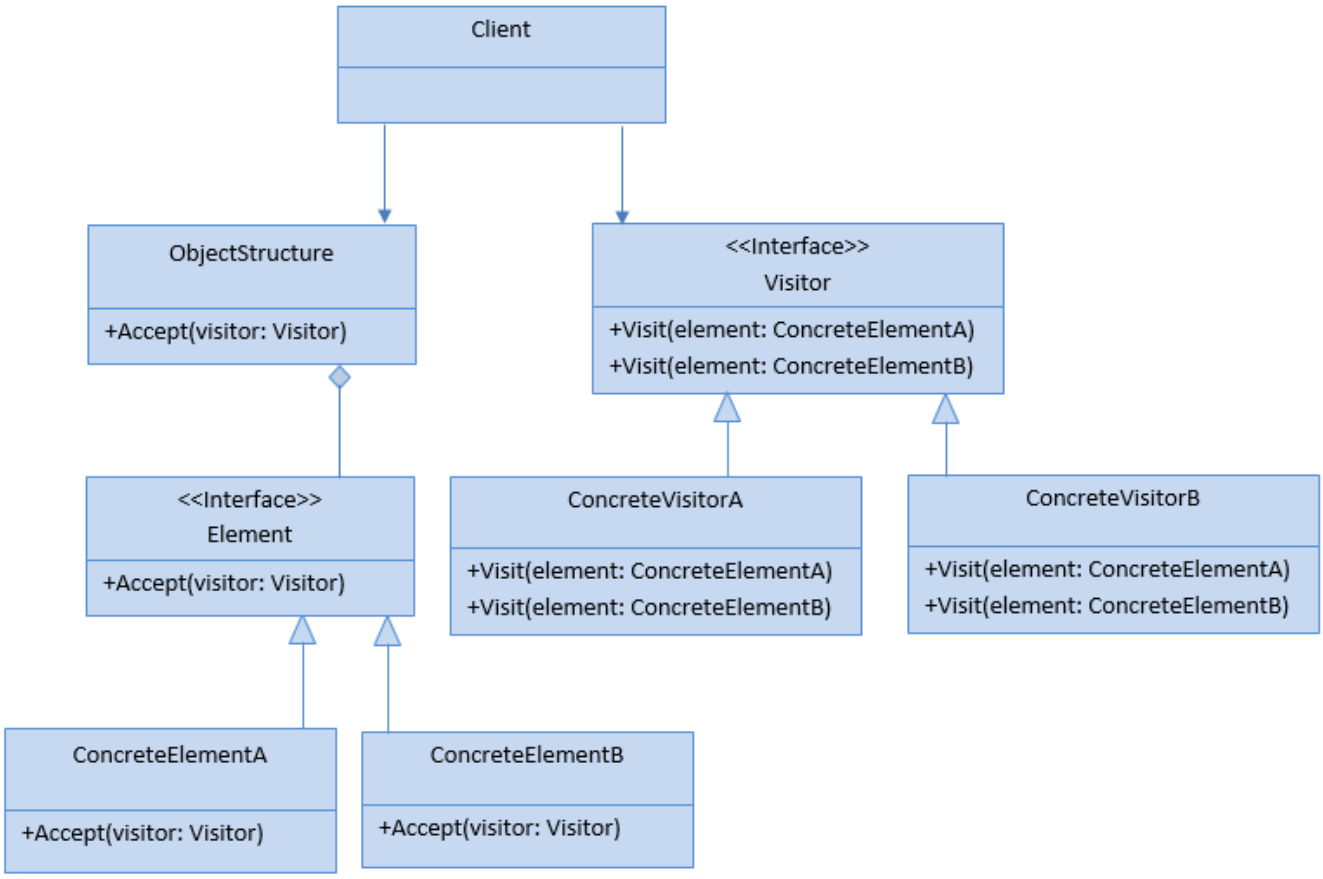
Ans. This pattern is used to create and perform new operations on a set of objects without changing the object structure or classes. This pattern enables loose coupling and the addition of new operations without changing the existing structure.

The examples for visitor pattern are:

- Add Functions to an object Dynamically
- Separate Object from Functions

Q42. Explain Visitor pattern with UML diagram and code example?

Ans. The UML diagram for visitor design pattern is shown below:



Visitor Pattern

The classes, interfaces and objects in the above UML class diagram are as defined follows:

1. **Client -** This is a class that has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate operations.
2. **ObjectStructure –** This is a class that holds all the elements which can be used by visitors.
3. **Element -** This is an interface that specifies the Accept operation.
4. **ConcreteElementA/B -** These are classes that implement the Element interface and holds the real information.
5. **Visitor –** This is an interface that specifies the Visit operations for concrete visitors.
6. **ConcreteVisitorA/B –** These are sub classes that implement the Visitor interface.

C# Implementation Code:

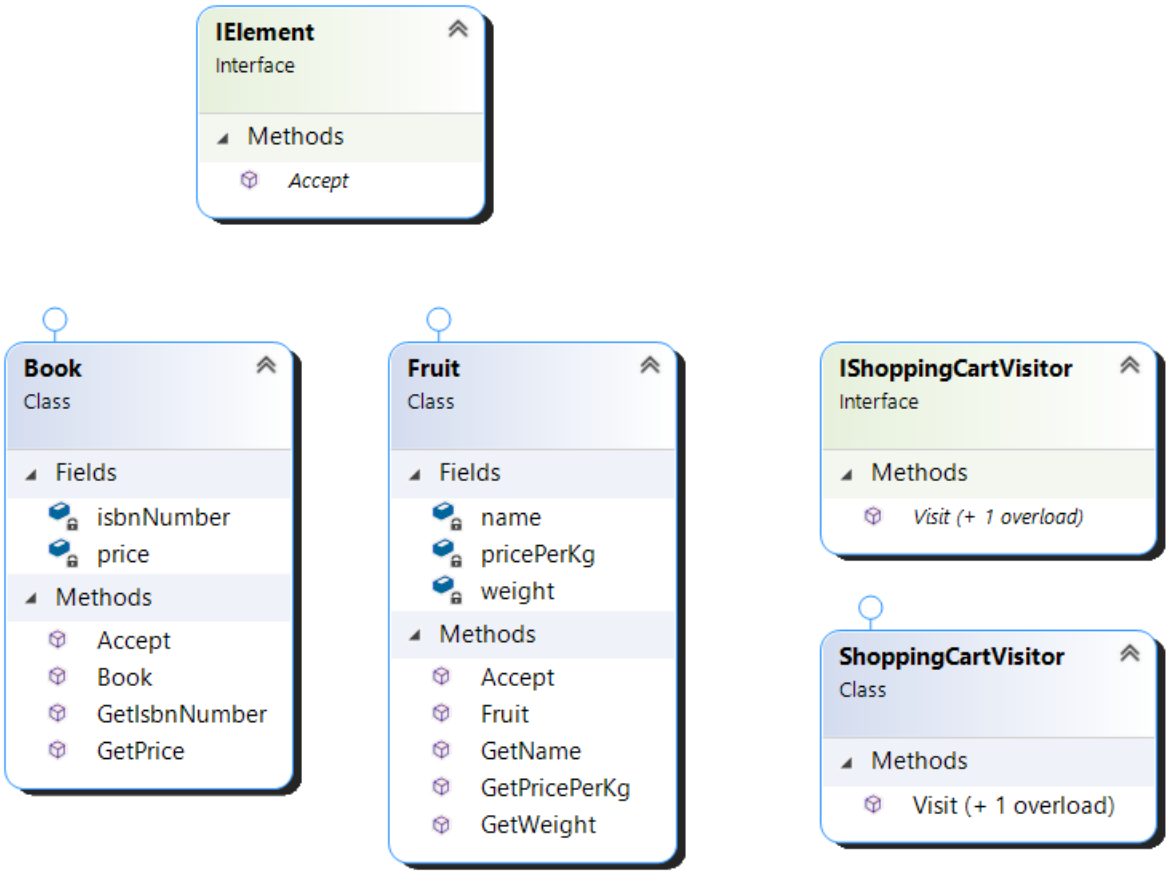
```
public class ObjectStructure
{
    public List<Element> Elements { get; private set; }

    public ObjectStructure()
    {
        Elements = new List<Element>();
    }
```

```
    public void Accept(Visitor visitor)
    {
        foreach (Element element in Elements)
        {
            element.Accept(visitor);
        }
    }
}

public interface Element
{
    void Accept(Visitor visitor);
}

public class ConcreteElementA : Element
{
    public void Accept(Visitor visitor)
    {
        visitor.Visit(this);
    }

    public string Name { get; set; }
}
public class ConcreteElementB : Element
{
    public void Accept(Visitor visitor)
    {
        visitor.Visit(this);
    }

    public string Title { get; set; }
}

public interface Visitor
{
    void Visit(ConcreteElementA element);

    void Visit(ConcreteElementB element);
}

public class ConcreteVisitorA : Visitor
{
    public void Visit(ConcreteElementA element)
    {
        Console.WriteLine("VisitorA visited ElementA : {0}", element.Name);
    }

    public void Visit(ConcreteElementB element)
    {
        Console.WriteLine("VisitorA visited ElementB : {0}", element.Title);
    }
}
public class ConcreteVisitorB : Visitor
{
```

```
    public void Visit(ConcreteElementA element)
    {
        Console.WriteLine("VisitorB visited ElementA : {0}", element.Name);
    }

    public void Visit(ConcreteElementB element)
    {
        Console.WriteLine("VisitorB visited ElementB : {0}", element.Title);
    }
}
```

Q43. Explain Visitor pattern with a real-life example?

Ans. Let's take the following example where we are separating the functions of the shopping cart.



C# - Implementation Code:

```
public interface IElement
    {
        int Accept(IShoppingCartVisitor visitor);
     }
public class Book: IElement
    {
        private int price;
        private string isbnNumber;
```

```
        public Book(int cost, string isbn)
        {
            price = cost;
            isbnNumber = isbn;
        }

        public int GetPrice()
        {
            return price;
        }

        public string GetIsbnNumber()
        {
            return isbnNumber;
        }

        public int Accept(IShoppingCartVisitor visitor)
        {
            return visitor.Visit(this);
        }
    }
public class Fruit: IElement
    {
        private int pricePerKg;
        private int weight;
        private string name;

        public Fruit(int priceKg, int wt, string nm)
        {
            pricePerKg = priceKg;
            weight = wt;
            name = nm;
        }

        public int GetPricePerKg()
        {
            return pricePerKg;
        }

        public int GetWeight()
        {
            return weight;
        }

        public string GetName()
        {
            return name;
        }

        public int Accept(IShoppingCartVisitor visitor)
        {
            return visitor.Visit(this);
        }
    }
public interface IShoppingCartVisitor
    {
        int Visit(Book book);
        int Visit(Fruit fruit);
```

```
    }
public class ShoppingCartVisitor : IShoppingCartVisitor
    {
        public int Visit(Book book)
        {
            int cost= book.GetPrice();
            Console.WriteLine($"Book ISBN: {book.GetIsbnNumber()} cost =INR{cost}");
            return cost;
        }

        public int Visit(Fruit fruit)
        {
            int cost = fruit.GetPricePerKg() * fruit.GetWeight();
            Console.WriteLine($"Name: {fruit.GetName()}, cost = INR{cost}");
            return cost;
        }
    }
class Program
    {
        static void Main(string[] args)
        {
            IElement[] items = new IElement[]
            {
                new Book(20, "ISBN-1234"),
                new Book(100, "ISBN-5678"),
                new Fruit(10, 2, "Banana"),
                new Fruit(5, 5, "Apple")
            };

            int total = CalculatePrice(items);
            Console.WriteLine($"Cart Total Cost = INR{total}");
        }
        private static int CalculatePrice(IElement[] items)
        {
            IShoppingCartVisitor visitor = new ShoppingCartVisitor();
            int sum = 0;
            foreach (IElement item in items)
            {
                sum = sum + item.Accept(visitor);
            }
            return sum;
        }
    }
```

Q44. When to use visitor Design pattern?

Ans. The visitor design pattern is useful in the following cases:

- An object structure has many unrelated operations to perform on it.
- An object structure cannot change but you need to perform new operations on it.
- The operations need to perform on the concrete classes of an object structure.

5
Anti Patterns

Q1. What are anti-patterns?

Ans. Anti-patterns are the opposite of software design Patterns. Anti-patterns are the outcomes of bad practices, over-engineering and implementation of design patterns without having enough knowledge or not understanding the context of the problem.

Some common anti-patterns are:

- God Objects
- Copy and Paste
- Lava Flow
- Boat Anchor

Q2. What are God objects?

Ans. An object that knows too much or does too much is called a God object. God object is just opposite to the Single responsibility Principle. God objects are costly to maintain. If you change anything in a God object, you have to test the whole object functionality.

For example, a simple example for GOD object us as follows:

```
public class Employee
{
        public void Get(string Id) { }
        public void Save(EmployeeModel employee) { }
        public void Update(EmployeeModel employee) { }
        public void Delete(string Id) { }
        private class EmployeeModel
        {
            public string Id { get; set; }
            public string Name { get; set; }
            public string Address { get; set; }
            public string Designation { get; set; }
            public double Salary { get; set; }
        }
}
```

Q3. What is Copy and Paste pattern?

Ans. When you write your code, you take the help of Goole, Stackoverflow or some articles to complete your code. This pattern is good if you are doing it to boost your development with a good understanding of it.

But it becomes Anti-pattern when you do it with your own code. If you are doing copy and paste of your own code to multiple modules or places, then it is an Anti-pattern. You should move such code to a class and reuse that one.

Q4. What is Lava flow?

Ans. The lava flow occurs when a code is working well but it is not documented or difficult to change or understand. This code exists in a system because it works.

Q5. What is Boat anchor?

Ans. The Boat anchor occurs when a part of the code is kept in the sytem despite it is no longer in use. This happens since developers believe that might need it in future. This belief seems always wrong.

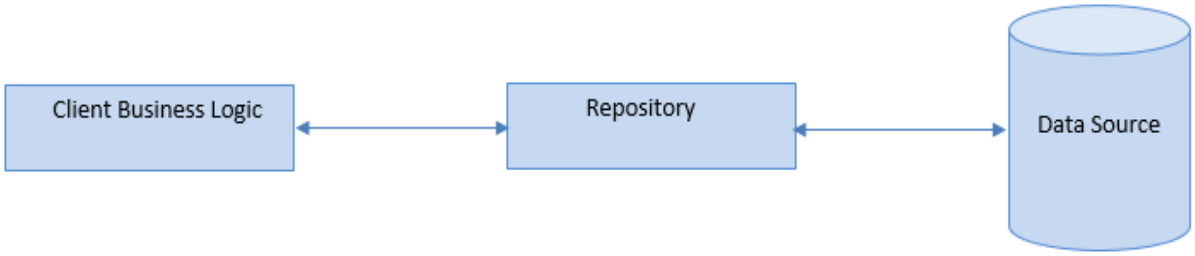
6

Patterns of Enterprise Application Architecture

Q1. What is Repository pattern?

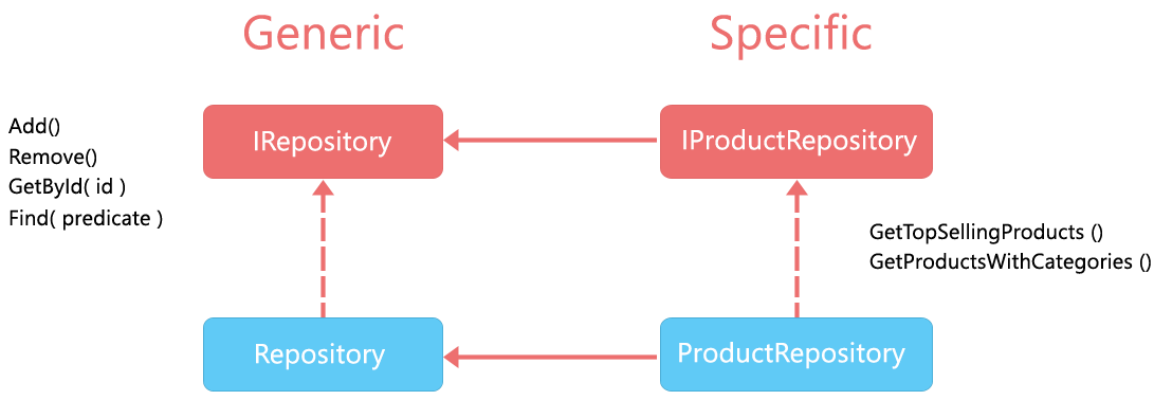
Ans. This pattern is used to create an abstract layer between the business access layer (BAL) and data access layer (DAL). A repository manages the interaction between business logic and underlying data source or Web service.

This pattern queries the data from the data source and maps it to the business entity and persists changes in the business entity to the data source.



Q2. How to implement Repository pattern?

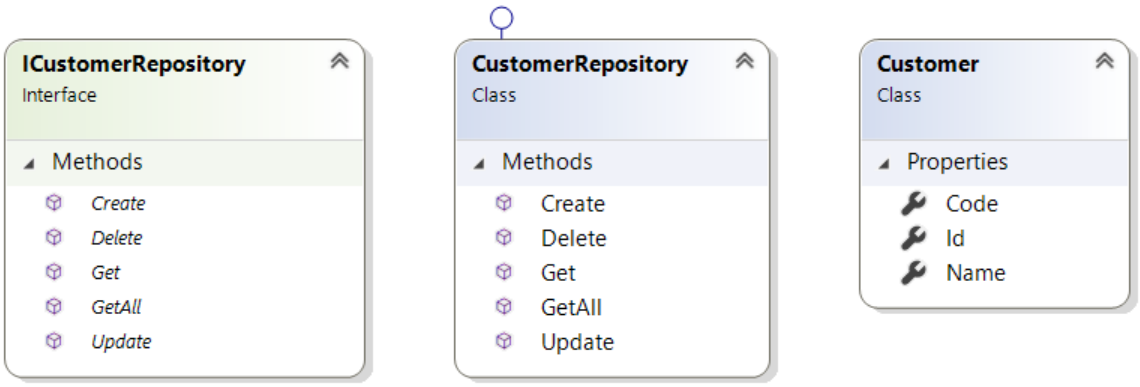
Ans. You can implement repository pattern using Generic repository and Specific or Silo repository pattern as shown in the given diagram.



In generic repository, you need to write commonly used methods. In specific repository add methods that are specific to an entity like product.

Q3. Explain Repository pattern with real-world example?

Ans. Let's take the following example where we need to perform CRUD operations on the Customer entity.



C# - Implementation Code:

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Code { get; set; }
}
public interface ICustomerRepository
{
    void Create(Customer customer);
    Customer Get(int id);
    IEnumerable<Customer> GetAll();
    void Update(Customer customer);
    void Delete(int id);
}
public class CustomerRepository : ICustomerRepository
{
    public void Create(Customer customer)
    {
        //Add customer
    }
    public void Delete(int id)
    {
        // Delete customer
    }
    public Customer Get(int id)
    {
        //Get customer
        return new Customer();
    }
    public IEnumerable<Customer> GetAll()
    {
        // Get All Customer
        return new List<Customer>();
```

```
    }
    public void Update(Customer customer)
    {
        //Update customer
    }
}
```

C# Test Code

```
static void Main(string[] args)
{
    CustomerRepository customerRepository = new CustomerRepository();
    var cutomers = customerRepository.GetAll();
}
```

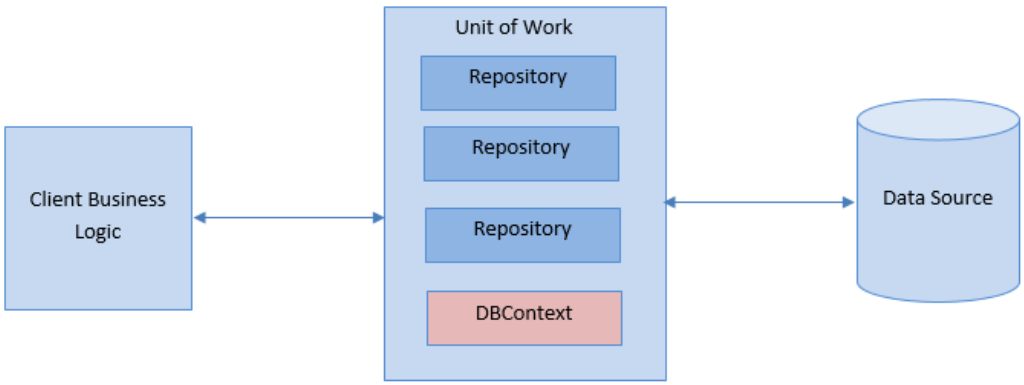
Q4. When to use Repository Design pattern?

Ans. This design pattern is useful in the following cases:

1. Need to facilitate automated unit testing or test-driven development (TDD).
2. Need to make code clean and easy to maintain by separating business logic from data or service access logic.
3. Need to make a centralised system for managing data access rules and logic, where you may access the data source from different locations.
4. Need to implement and make centralize caching strategy for the data source.

Q5. What is Unit of Work pattern and when to use it?

Ans. This pattern makes a list of objects, keeps track of all the changes to the objects by a business transaction and commit or rollback the changes as a whole or unit. It is commonly used with the repository pattern.



By default, Entity Framework supports Unit of Work Pattern.

Entity Framework

<< repository >>
DbSet

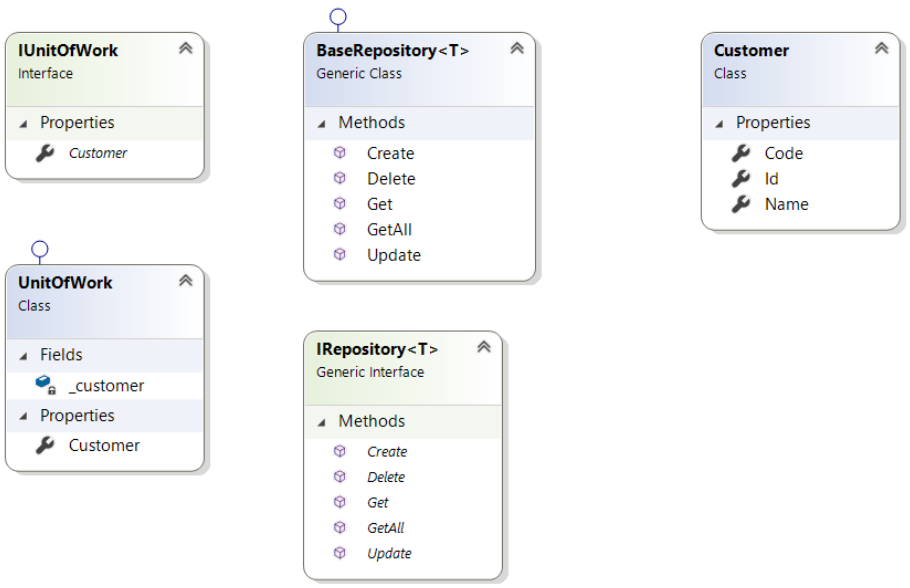
Add(obj)
Remove(obj)
Find(id)
Where(predicate)

<< unit of work >>
DbContext

DbSet
DbSet
DbSet
SaveChanges()

Q6. Explain Unit of Work pattern with a real-world example?

Ans. Let's take the following example where we need to manage business transactions for database operations on multiple tables.



C# - Implementation Code:

```
public interface IUnitOfWork
{
    IRepository<Customer> Customer { get; }
}

public class UnitOfWork : IUnitOfWork
{
    IRepository<Customer> _customer;
    public IRepository<Customer> Customer
    {
        get
        {
            return _customer ?? (_customer = new BaseRepository<Customer>());
        }
    }
}
```

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Code { get; set; }
}

public interface IRepository<T> where T : class
{
    void Create(T entityToCreate);
    T Get(int id);
    IEnumerable<T> GetAll();
    void Update(T entityToUpdate);
    void Delete(int id);
}

public class BaseRepository<T> : IRepository<T> where T : class
{
    public void Create(T entityToCreate)
    {

    }

    public void Delete(int id)
    {

    }

    public T Get(int id)
    {
        return null;
    }

    public IEnumerable<T> GetAll()
    {
        Console.WriteLine("GetAll");
        return null;
    }

    public void Update(T entityToUpdate)
    {

    }
}
```

C# Test Code

```
static void Main(string[] args)
{
    UnitOfWork unitOfWork = new UnitOfWork();
    var cutomer = new BaseRepository<Customer>();
    cutomer.GetAll();
}
```

Q7. When to use Unit of Work Design pattern?

Ans. This design pattern is useful in the following cases:

- In order to execute database operations, Insert, Update and Delete on the different entities within a transaction as a Unit.
- Expendable support for different data sources like SQL Server, Oracle, DB2, Web Services etc.

Q8. What is Lazy load pattern?

Ans. In this pattern, all the objects (classes) are initialized when they are required. This design pattern is very useful when the cost of object initialization is very high and usage of an object is very low or rare. This helps you to improve the performance of the application by avoiding unnecessary computation and memory consumption.

Q9. What are different ways to implement Lazy load pattern?

Ans. There are four ways to implement the Lazy Loading pattern:

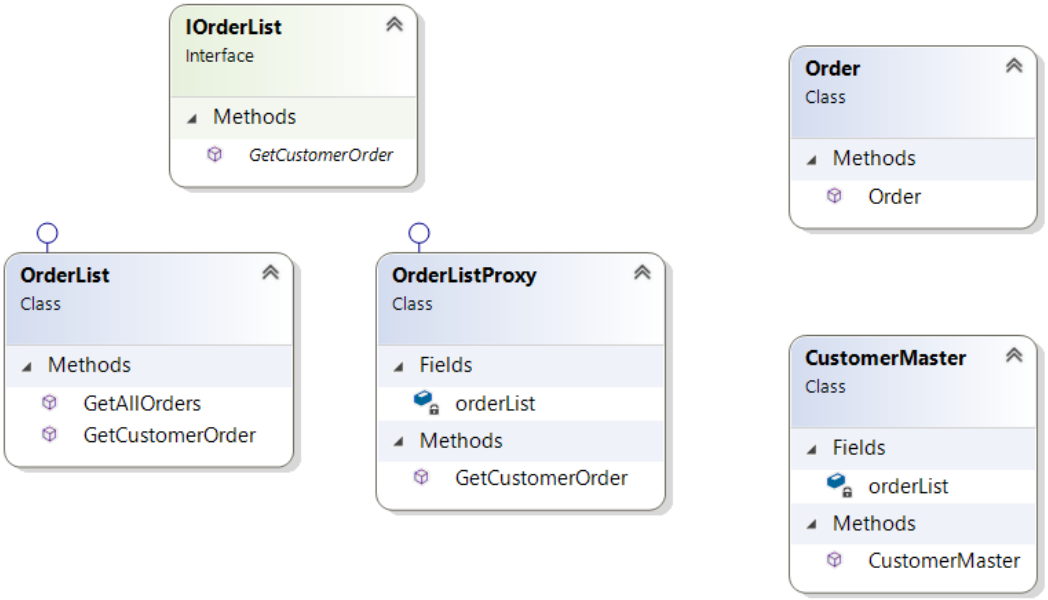
1. Lazy Initialization

In this technique, checking the value of a class field when it is being used. If the value of the class instance is null then the proper value being initialized before it returns.

```
public class Order
{
    public Order(int CustomerId)
    {

    }
    // Implement your order Class;
}

public class CustomerOrderManagement
{
    public int CustomerId { get; set; }

    public Order customerOrder;
    public Order Orders
    {
        get
        {
            if(customerOrder == null)
            {
                customerOrder = new Order(CustomerId);
            }
            return customerOrder;
        }
    }

    // Implementation of Customer Order Management class
}
```

2. Virtual Proxy

The representation of this implementation is very similar to Lazy initialization. This uses a proxy object instead of the real object. When a user tries to access the proxy object first, it creates a new instance of the real object.




```
public interface IOrderList
{
    List<Order> GetCustomerOrder(int id);
}

public class CustomerMaster
{
    IOrderList orderList;
    public CustomerMaster(IOrderList list)
    {
        orderList = list;
    }
}
public class OrderList: IOrderList
{
    public List<Order> GetCustomerOrder(int id)
    {
        return GetAllOrders();
    }
    public List<Order> GetAllOrders()
    {
        return null;
    }
}
public class OrderListProxy : IOrderList
{
    private IOrderList orderList;
    public List<Order> GetCustomerOrder(int id)
    {
        if(orderList == null) {
            orderList = new OrderList();
        }
        return orderList.GetCustomerOrder(id);
    }
}
public class Order
```

```
{
    public Order(int CustomerId)
    {

    }
    // Implement your order Class;
}
```

3. Value Holder

It is a generic object holder that handles the lazy loading behaviour and returns the required object. The generic holder returns the required object instance based on type. The main drawback of this approach is that the user must know that a value holder is expected.

```
public class ValueHolder<T>
{
    private T _value;
    private readonly Func<object, T> _valueProvider;

    // Constructor
    public ValueHolder(Func<object, T> valueProvider)
    {
        _valueProvider = valueProvider;
    }

    // We'll use the signature "GetValue" for convention
    public T GetValue(object parameter)
    {
        if (_value == null)
            _value = _valueProvider(parameter);
        return _value;
    }
}
```

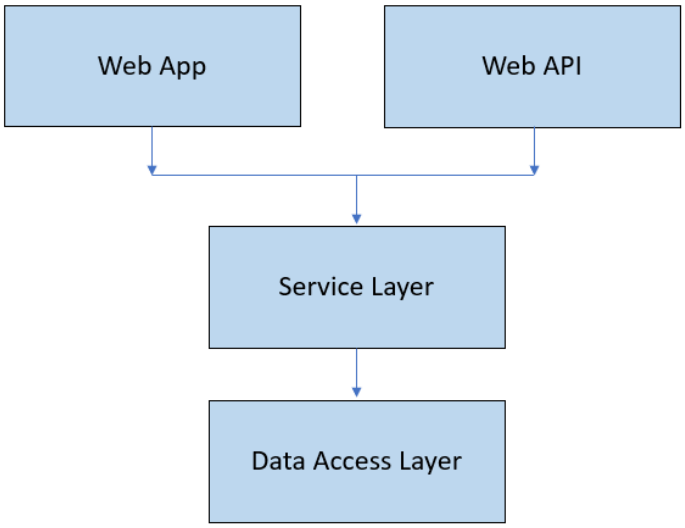
4. Ghost

A ghost is a partially loaded object. It is the same as the real object but it is not in its full state. It means that the object may be empty or may contain just some fields. The ghost initializes itself when the user tries to access the fields which are not loaded yet.

Q10. What is service layer pattern?

Ans. In the service layer pattern, business logic is written into service. The service is a class that contains the domain logic of your application. This pattern is more useful when you support multiple types of applications such as web-based applications, mobile native app, etc for the same functionalities.

The service layer is shared between the application. This helps you to avoid duplicating the service (business) logic in your application.



Q11. What are davantages of service layer pattern?

Ans. The main advantages of service layer pattern are:

- Code Reusability
- Maintainability
- Reduce Code Duplication

7

Code Organization Patterns

Q1. What is Namespace Pattern?

Ans. The namespace is used to organize the classes in a logical group. It also helps to keep a set of names separate from another. You can also define the same name class in a different namespace. You can define a nested namespace for a hierarchical system. Namespaces play an important role in managing a big project to write cleaner code.

```
namespace AppBusinessLayer
{
    public class EmployeeBL
    {

    }
}
```

```
namespace AppDataAccessLayer
{
    public class EmployeeDAL
    {

    }
}
```

Q2. What is Module Pattern?

Ans. This pattern is used to implements the concept of software modules. A module is a small piece of related classes or screens. This pattern can be implemented by defining a namespace for related classes or put all related classes into one folder.

Q3. What is Sandbox Pattern?

Ans. Sandbox is an isolated environment in which users can run programs or files without impacting the system, platform or, application. It is used to run and test new code. This pattern is helpful to execute malicious code without harming the device, networks or other connected devices.

Q4. What is MVC pattern?

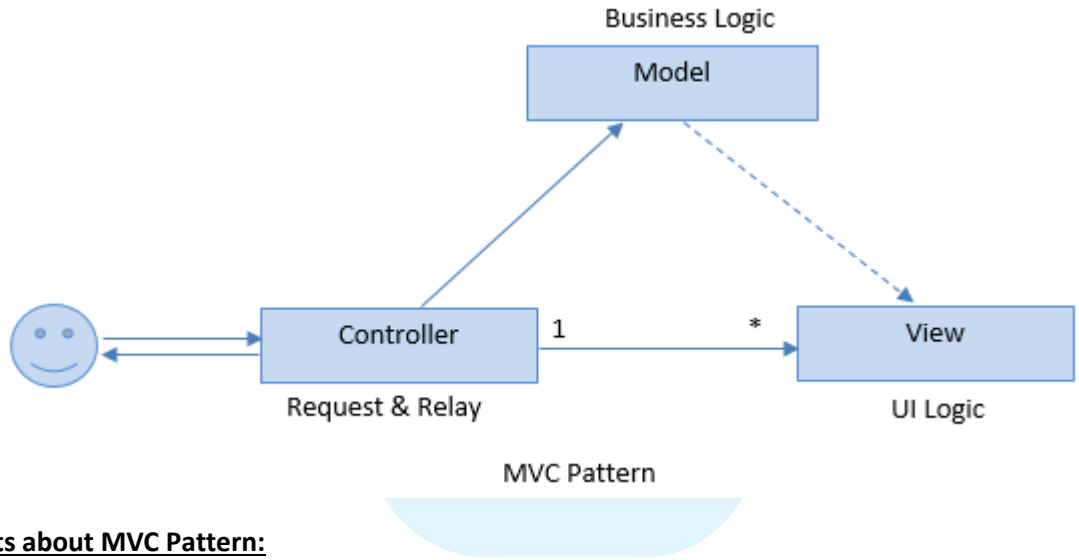
Ans. The MVC (Model-View-Controller) pattern was introduced in 1970s. This pattern forces a separation of concerns within an application.

For example, separating data access logic and business logic from the UI. This design pattern splits an application into three main parts: Model, View and Controller.

Model - The Model represents a set of classes that describes the business logic and data. It also defines business rules for data means how the data can be changed and manipulated.

View – The View represents the UI components like CSS, jQuery, html etc. It is only responsible for displaying the data that is received from the controller as the result. This also transforms the model(s) into UI.

Controller- The Controller is responsible for controlling the application logic and acts as the coordinator between the View and the Model. This receives input from users via the View, then process the user's data with the help of Model and passing the results back to the View



Key Points about MVC Pattern:

1. The user interacts with the Controller.
2. There is one-to-many relationship between Controller and View means one controller can map to multiple views.
3. A controller has a reference to View and View can have a reference to the model.
4. A controller can talk to View but View cannot talk to the controller.

Today, this pattern is commonly used by many popular frameworks like as Ruby on Rails, Spring Framework, Apple iOS Development and ASP.NET MVC.

Q5. When to use MVC pattern?

Ans. This design pattern is useful in the following cases:

1. You need to decouple view (UI), model (Business Logic) and controller (user interaction) from each others.
2. This allows to map many views with a given model.
3. Make code clean, testable and easy to maintain.

Q6. What is MVP pattern?

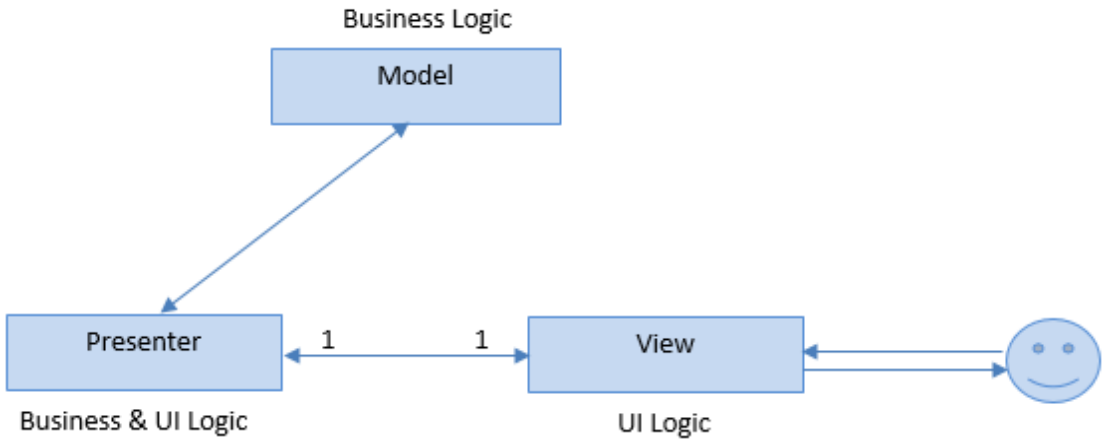
Ans. This pattern is similar to MVC pattern in which the controller has been replaced by the presenter. This design pattern splits an application into three main parts: Model, View and Presenter.

Model - The Model represents a set of classes that describes the business logic and data. It also defines business rules for data means how the data can be changed and manipulated.

View – The View represents the UI components like CSS, jQuery, html etc. It is only responsible for displaying the data that is received from the presenter as the result. This also transforms the model(s) into UI.

Presenter- The Presenter is responsible for handling all UI events on behalf of the view. This receives input from users via the View, then process the user's data with the help of the Model and passing the results back to the View. Unlike view and controller, view and presenter are completely decoupled from each other's and communicate to each other by an interface.

Also, the presenter does not manage the incoming request traffic as a controller.



MVP Pattern

This pattern is commonly used with ASP.NET Web Forms applications which require to create automated unit tests for their code-behind pages. This is also used with windows forms.

Key Points about MVP Pattern:

1. User interacts with the View.
2. There is one-to-one relationship between View and Presenter means one View is mapped to only one Presenter.
3. A view has a reference to Presenter but View has not to reference to Model.
4. Provides two communications between View and Presenter.

Q7. When to use MVP pattern?

Ans. This design pattern is useful in the following cases:

1. You need to decouple view (UI), model (Business Logic) and presenter (user interaction) from each others.
2. This allows to map many views with a given model.

3. Make code clean, testable and easy to maintain

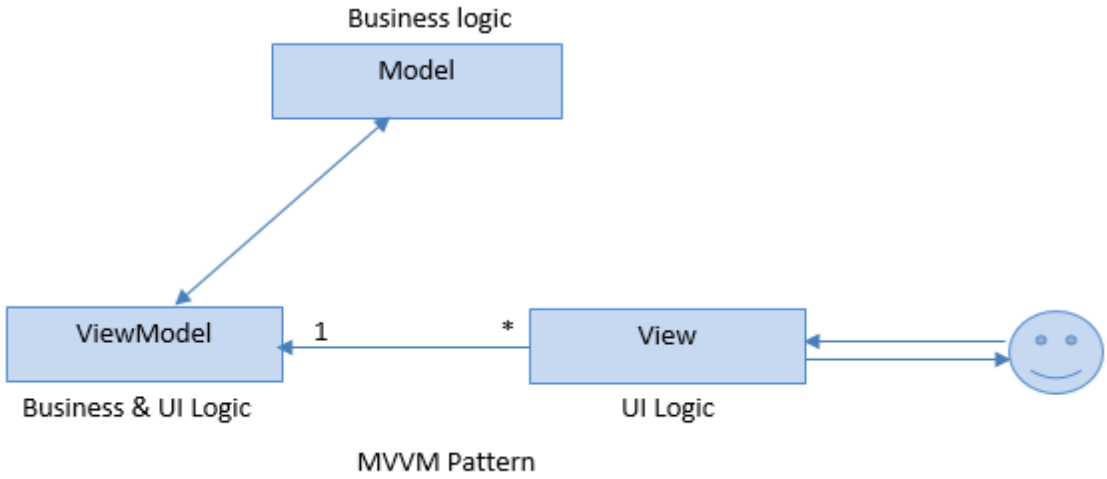
Q8. What is MVVM pattern?

Ans. MVVM stands for Model-View-View Model. This pattern supports two-way data binding between the view and View model. This enables automatic propagation of changes, within the state of view model to the View. Typically, the view model uses the observer pattern to notify changes in the view model to model.

Model - The Model represents a set of classes that describes the business logic and data. It also defines business rules for data means how the data can be changed and manipulated.

View – The View represents the UI components like CSS, jQuery, html etc. It is only responsible for displaying the data that is received from the controller as the result. This also transforms the model(s) into UI.

View Model - The View Model is responsible for exposing methods, commands, and other properties that help to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.



MVVM Pattern

This pattern is commonly used by the WPF, Silverlight, Caliburn, nRoute etc.

Key Points about MVVM Pattern:

1. A user interacts with the View.
2. There is many-to-one relationship between View and ViewModel means many Views can be mapped to one ViewModel.
3. A view has a reference to ViewModel but View Model has no information about the View.
4. Supports two-way data binding between View and ViewModel.

Q9. When to use MVVM pattern?

Ans. This design pattern is useful in the following cases:

1. You need to decouple view (UI) and model (Business Logic) from each other's.
2. This allows to map many views with a given view model.
3. Make code clean, testable and easy to maintain.

References

This book has been written by referring to the following sites:

1. https://www.dotnettricks.com/learn/designpatterns- DotNetTricks - Design Patterns
2. https://stackoverflow.com/questions/tagged/design-patterns - Stack Overflow - Design Patterns