White box testing has several advantages:

1. It helps to identify errors in the code logic and working of the system.
2. It helps to improve code quality by identifying areas that require optimization.
3. It helps to increase code maintainability by identifying areas that require refactoring.
4. It helps to reduce the cost of software development by identifying issues early in the development lifecycle.

However, there are some limitations to white box testing:

1. It can be time-consuming to design and execute test cases for every line of code.
2. It requires technical expertise to perform white box testing.
3. It may not uncover defects that occur due to interactions between different sections of the code.
4. It may not uncover defects that occur due to integration with other software systems.

explain black box technique in detail

Black box testing is a software testing technique that focuses on the external behavior of the software system being tested, without considering its internal implementation details. It is also known as functional testing or behavioral testing.

In black box testing, the tester is not concerned with how the software is developed or how it works internally. Instead, the tester interacts with the software using inputs and verifies whether the expected outputs are generated. The tester does not have access to the source code or any information regarding the internal workings of the software.

There are different types of black box testing techniques, such as:

1. Equivalence partitioning: This technique divides the input domain of the software into equivalence classes, where each class has a similar behavior. The goal is to reduce the number of test cases while ensuring that all possible inputs are tested.
2. Boundary value analysis: This technique tests the software by selecting input values that are at the boundaries of the input domain, as these values are more likely to cause errors.
3. Decision table testing: This technique involves creating a table that lists all possible combinations of inputs and outputs. The tester can then check each combination to verify that the expected output is generated.
4. State transition testing: This technique is used when the software system being tested has different states. The tester creates a model of the system's states and transitions between them, and then tests each transition to ensure that the system behaves correctly.
5. Exploratory testing: This technique is a manual testing approach where the tester explores the software system by using it in different ways and trying to find defects.

The advantages of black box testing include:

- It does not require knowledge of the internal workings of the software system, so testers can focus on the functionality and usability of the system.
- It is useful for finding errors that are related to incorrect inputs or outputs.
- It can help identify gaps in the software requirements.
- It can be used early in the development cycle when the software is still being developed.

The disadvantages of black box testing include:

- It may not be as effective in finding errors related to the internal implementation of the software.
- It can be time-consuming to create and execute test cases.
- It may not uncover all defects, as it only tests the system's behavior for a specific set of inputs.

History and Origin of Patterns from design pattterns

In computer science, design patterns are a specific type of pattern that refer to reusable solutions to commonly occurring problems in software design. The concept of design patterns was first introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in 1994.

However, the idea of patterns in software design can be traced back to the work of Christopher Alexander, who wrote the book "A Pattern Language" in 1977. Alexander argued that patterns can be used to create well-designed buildings and communities, and his ideas inspired the development of design patterns in software engineering.

Design patterns provide a way to solve common design problems by creating a standardized way of solving them. They help to improve software quality, reduce development time, and make it easier to maintain and extend software systems.

There are several categories of design patterns, including creational, structural, and behavioral patterns. Creational patterns deal with object creation mechanisms, while structural patterns focus on object composition and relationships. Behavioral patterns are concerned with communication between objects and responsibilities.

Some of the most commonly used design patterns include the Singleton pattern, which ensures that only one instance of a class is created; the Observer pattern, which allows objects to be notified of changes to another object; and the Decorator pattern, which allows behavior to be added to an object dynamically.

In summary, the history and origin of patterns in software design can be traced back to Christopher Alexander's work on architecture and design, and the concept of design patterns was formalized in the book "Design Patterns: Elements of Reusable Object-Oriented Software". Design patterns provide a way to solve common software design problems by creating standardized solutions.

WHAT IS DESIGN PATTERN.EXPLAIN IT'S ELEMENTS

In software engineering, a design pattern is a reusable solution to a common software design problem. It is a general, proven solution to a problem that can be applied to different situations.
There are several elements of a design pattern:

1. Pattern name: Pattern names allow designers and developers to easily locate and apply existing design solutions to new problems, making the design process more efficient and effective. Additionally, pattern names serve as a common language that can help bridge communication gaps between designers and developers, promoting collaboration and a shared understanding of the design goals and solutions.
2. Problem: This is the issue or challenge that the design pattern aims to solve. It could be a common problem in software design, such as managing dependencies or handling user input.
3. Context: The context refers to the situation in which the problem occurs. This could be a specific part of a software system or a particular set of requirements.
4. Solution: The solution is the recommended approach to solving the problem in the given context. It Describes the steps or components needed to implement the solution.
5. Consequences: The consequences are the benefits and drawbacks of using the design pattern. These could include factors such as performance, scalability, and maintainability.
6. Implementation: This is the process of applying the design pattern to a specific software system. It involves adapting the solution to the context and integrating it with the existing codebase.
7. Examples: Examples are sample code or scenarios that illustrate how the design pattern can be used in practice. They demonstrate the benefits of the design pattern and help developers understand how to apply it to their own projects.

Overall, design patterns are a valuable tool for software engineers, as they can help to improve the quality, efficiency, and maintainability of software systems. By using proven solutions to common problems, developers can avoid common pitfalls and focus on building high-quality software.

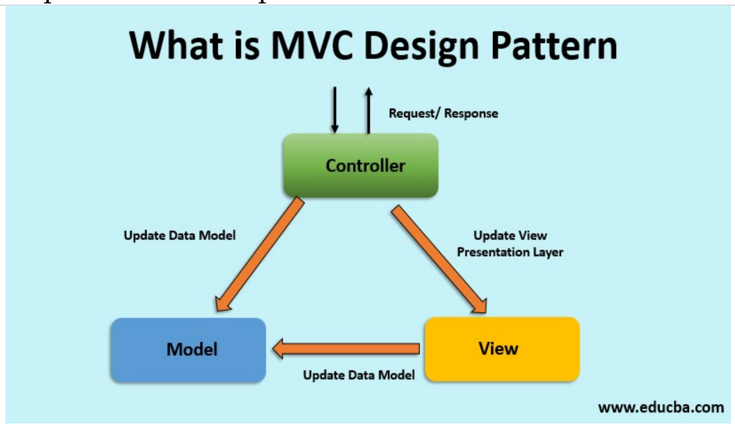
Design Patterns in MVC

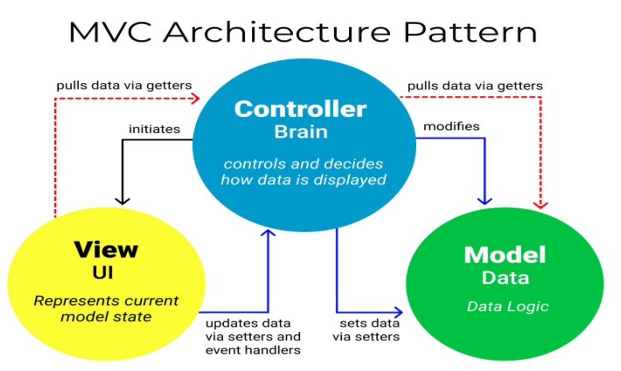
Design patterns play an important role in the implementation of Model-View-Controller (MVC) architecture, which is a widely used architectural pattern in software development.
The Model-View-Controller pattern separates an application into three interconnected components:

1. Model - represents the data and business logic of the application
2. View - represents the user interface and displays the data from the model to the user
3. Controller - handles user input, updates the model, and updates the view accordingly

There are several design patterns that can be used in the implementation of each component of the MVC architecture:

1. Model: The Model component can be implemented using various design patterns such as Repository, DAO (Data Access Object), and Service Locator patterns. These patterns help to decouple the Model from the data access layer and allow for better organization and testing of the Model.
2. View: The View component can be implemented using the Template Method pattern. This pattern allows the View to define the structure of the user interface, while allowing subclasses to define specific implementations of the UI.
3. Controller: The Controller component can be implemented using the Command pattern. This pattern allows commands to be encapsulated into objects, which can be invoked by the Controller in response to user input.





In addition to these patterns, there are other design patterns that can be used in MVC, such as Observer, Adapter, and Facade patterns, depending on the specific requirements of the application.
Overall, the use of design patterns in MVC helps to improve the modularity, maintainability, and extensibility of the application, while also reducing code duplication and improving code organization.

Describing Design Patterns

Design patterns are reusable solutions to common software development problems that have been proven to be effective over time. They are general solutions that can be adapted to a variety of different situations and are often used to improve the quality, maintainability, and extensibility of software systems.
Design patterns can be classified into three main categories:

1. Creational Patterns: These patterns provide a way to create objects in a way that is more flexible and efficient than the traditional approach of using new operator directly. Examples of creational patterns include Singleton, Factory Method, Abstract Factory, Builder, and Prototype patterns.
2. Structural Patterns: These patterns focus on the composition of classes and objects to form larger structures. Examples of structural patterns include Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy patterns.

3. Behavioral Patterns: These patterns deal with the interaction and communication between objects and classes. Examples of behavioral patterns include Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor patterns.

Each design pattern provides a set of guidelines for solving a particular problem in software development. Design patterns are not rigid rules, but rather general templates that can be adapted to the specific requirements of a given problem. They help to improve software quality by providing solutions to common design problems and reducing the risk of errors and bugs.

Design patterns also provide a shared vocabulary that developers can use to communicate and collaborate on software development projects. By using design patterns, developers can focus on solving specific problems, rather than reinventing the wheel each time a similar problem arises.

How Design Patterns Solve Design Problems

Design patterns provide a proven and standardized approach to solving common design problems in software development. By using design patterns, developers can:

1. Improve software quality: Design patterns are tried and tested solutions to common design problems. By using these patterns, developers can improve the quality of their software by reducing the risk of errors and bugs.
2. Promote code reuse: Design patterns are reusable solutions that can be adapted to different situations. By using design patterns, developers can avoid duplicating code and promote code reuse.
3. Encourage modularity and flexibility: Design patterns help to separate concerns and promote modularity, which makes it easier to modify and extend software systems. This, in turn, makes the software more flexible and adaptable to changing requirements.
4. Simplify maintenance: Design patterns make software systems easier to maintain by providing a clear and standardized structure. This, in turn, makes it easier to modify and update the software without introducing errors or breaking existing functionality.
5. Facilitate communication and collaboration: Design patterns provide a shared vocabulary and understanding of software design that makes it easier for developers to communicate and collaborate on software development projects.

Overall, design patterns provide a set of guidelines and best practices for solving common design problems in software development. By using these patterns, developers can create software that is more reliable, maintainable, and flexible, while also promoting code reuse and collaboration.

Selecting a Design Pattern

When selecting a design pattern, there are several factors to consider:

1. The problem at hand: The first consideration when selecting a design pattern is the problem that needs to be solved. You should choose a pattern that best addresses the problem at hand and provides a solution that is flexible, extensible, and easy to maintain.
2. The system architecture: The selected pattern should be compatible with the system architecture and other design patterns that have been used in the system.
3. The team's experience and expertise: You should consider the team's experience and expertise in using design patterns. It is advisable to choose a pattern that the team is familiar with and has experience implementing.
4. The trade-offs: Each design pattern has its trade-offs. You should consider the advantages and disadvantages of each pattern before making a decision.
5. The design goals: You should consider the design goals of the system, such as scalability, maintainability, and reusability, and choose a pattern that best aligns with these goals.
6. The design patterns catalog: It is also important to be familiar with the design patterns catalog and the patterns that are available. You should consider the pattern that best fits the problem at hand and provides a solution that is compatible with the system architecture and the team's experience and expertise.

Overall, selecting a design pattern requires a thorough understanding of the problem at hand, the system architecture, the team's experience and expertise, the design goals, and the design patterns catalog. By considering these factors, you can select a pattern that best addresses the problem and provides a solution that is flexible, extensible, and easy to maintain.

Using a Design Pattern

Using a design pattern involves several steps:

1. Identify the problem: The first step in using a design pattern is to identify the problem that needs to be solved. This could be a recurring problem that has already been solved using a design pattern or a new problem that requires a customized solution.
2. Choose a pattern: Once the problem has been identified, the next step is to choose an appropriate design pattern that solves the problem. This involves selecting a pattern that best fits the problem at hand and provides a solution that is compatible with the system architecture and the team's experience and expertise.
3. Understand the pattern: Before using a design pattern, it is important to understand how it works and its advantages and disadvantages. This involves studying the pattern's structure, behavior, and implementation details.
4. Implement the pattern: Once the pattern has been chosen and understood, the next step is to implement it in the system. This involves adapting the pattern to fit the specific requirements of the problem and integrating it into the system architecture.
5. Test the pattern: After implementing the pattern, it is important to test it thoroughly to ensure that it works as expected and does not introduce any new bugs or errors.

6. Refactor and improve: Finally, after testing the pattern, it may be necessary to refactor and improve the implementation to ensure that it is scalable, maintainable, and extensible.

Overall, using a design pattern involves a systematic approach that starts with identifying the problem and selecting an appropriate pattern, understanding the pattern, implementing it in the system, testing it, and refining the implementation to ensure that it meets the design goals and requirements of the system.

Describe about creational design pattern(ABSPF)

Creational design patterns are a category of design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable for a given situation. These patterns provide ways to create objects while hiding the creation logic, thereby increasing flexibility and decoupling the client code from the actual objects being created.

There are several different types of creational design patterns, including:

1. Abstract Factory Pattern: This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. Builder Pattern: This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
3. Singleton Pattern: This pattern restricts the instantiation of a class to a single instance and provides a global point of access to that instance.
4. Prototype Pattern: This pattern creates new objects by cloning existing ones, thereby avoiding the need for complex initialization logic.
5. Factory Method Pattern: This pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Overall, creational design patterns provide a way to create objects in a flexible and decoupled manner, allowing for more maintainable and extensible code.

Explain about Abstract Factory Pattern

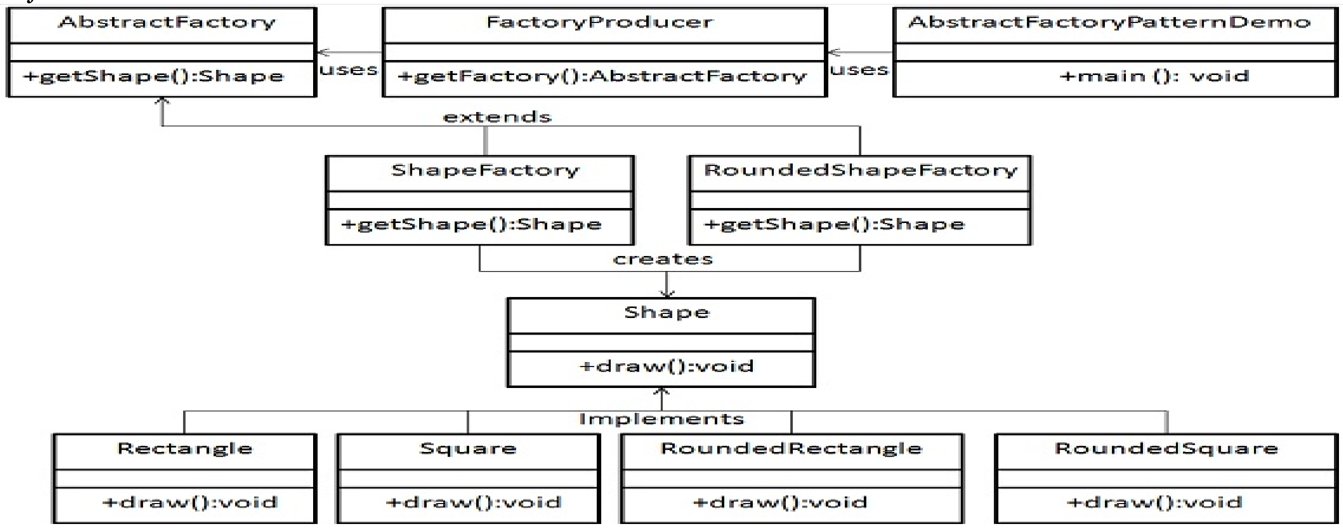
The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern encapsulates the creation of objects and is often used when a system must be independent of the way the objects it creates are composed.

In an Abstract Factory Pattern, there are typically four key components:

1. Abstract Factory: This interface declares a set of methods for creating abstract products. Concrete factories implement these methods to create specific product objects.
2. Concrete Factory: This class implements the methods declared in the abstract factory and creates specific product objects.
3. Abstract Product: This interface declares a set of methods that are common to all concrete products.
4. Concrete Product: This class implements the methods declared in the abstract product and provides specific behavior for the product.

The Abstract Factory Pattern provides a way to create families of related objects, such as a set of GUI widgets that are designed to work together. By using an abstract factory, you can create an entire family of related objects without having to worry about the specific classes of those objects. This makes it easier to maintain and modify the code over time, as new families of objects can be added or modified without affecting the existing code.

We are going to create a Shape interface and a concrete class implementing it. We create an abstract factory class AbstractFactory as next step. Factory class ShapeFactory is defined, which extends AbstractFactory. A factory creator/generator class FactoryProducer is created. AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get a AbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to AbstractFactory to get the type of object it needs.



The Abstract Factory Pattern is a useful pattern for creating families of related objects in a way that promotes loose coupling and maintains flexibility in the design.

Explain about Builder Pattern

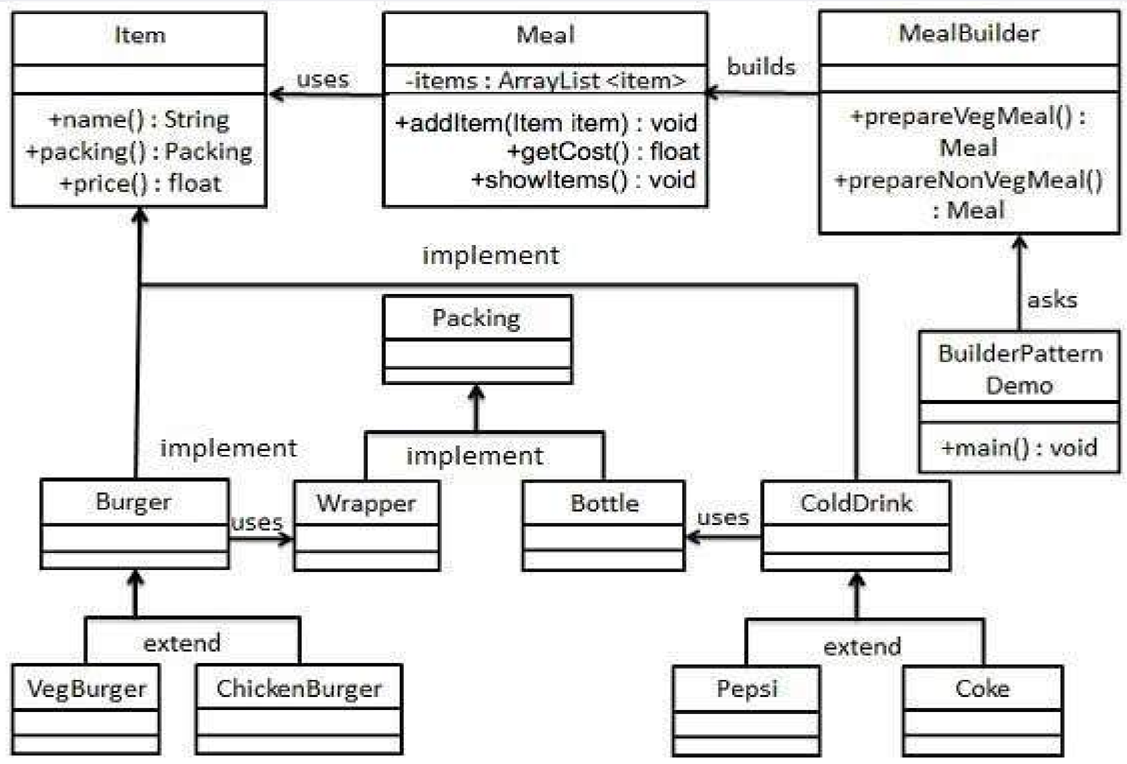
The Builder Pattern is a creational design pattern that allows you to separate the construction of a complex object from its representation, so that the same construction process can create different

representations. This pattern is particularly useful when you need to create objects that have many optional or configurable parts.
In a Builder Pattern, there are typically four key components:

1. Director: This class is responsible for defining the sequence of steps required to build a complex object. The director works with an abstract builder interface to construct the object.
2. Abstract Builder: This interface defines a set of methods for building the different parts of a complex object.
3. Concrete Builder: This class implements the abstract builder interface and provides a set of methods for building the different parts of the object.
4. Product: This class represents the complex object being built. It typically contains a collection of other objects that represent the different parts of the product.

The Builder Pattern works by separating the construction of the object into a series of steps, each of which is defined by the abstract builder interface. The director class then uses these steps to construct the object in a specific order, using a concrete builder class to implement each step. Because the construction process is separated from the representation of the object, it is possible to create different representations of the same object by using different concrete builders.

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.
We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.
We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.



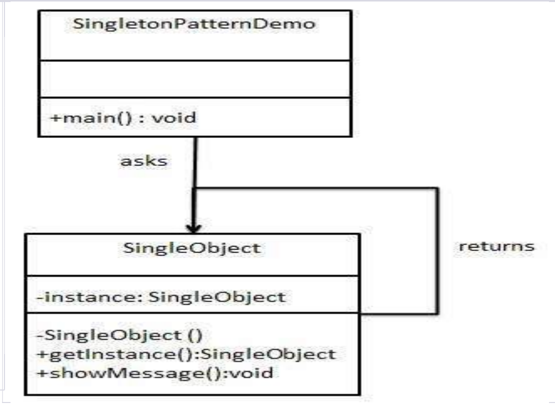
The Builder Pattern is particularly useful when you need to create objects with many optional or configurable parts, as it allows you to create a flexible and extensible construction process. It also helps to promote code reuse and makes it easier to maintain and modify the code over time.

Explain about Singleton Pattern

The Singleton Pattern is a design pattern that restricts the instantiation of a class to a single instance and provides a global point of access to it. This pattern ensures that there is only one instance of a class and provides a mechanism to access that instance throughout the application.
The Singleton Pattern is useful in scenarios where there is a need to restrict the creation of multiple instances of a class. For example, a logging class that writes log entries to a file or a database should have only one instance throughout the application. This ensures that all log entries are written to the same location and there are no conflicts.
To implement the Singleton Pattern, the class constructor is made private, so it cannot be instantiated outside the class. A static method is then created within the class that returns the instance of the class. This method is responsible for creating the instance of the class if it does not exist, and returning the instance if it already exists.
We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.
SingleObject class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.
In this implementation, the Singleton class has a private constructor, which prevents the class from being instantiated from outside the class. The static getInstance() method returns the single instance of the class. The method checks if the instance variable is null, and if it is, it creates a new instance of the Singleton class. Subsequent calls to the getInstance() method will return the same instance of the class.

Here's an example implementation of the Singleton Pattern in Java:

```
 public class Singleton {
   private static Singleton instance;
      private Singleton() {
      // private constructor
   }
      public static Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}; } }
```



Explain about prototype Pattern

The Prototype Pattern is a creational design pattern that allows you to create new objects by cloning existing ones, without relying on their concrete classes. This pattern is particularly useful when you need to create objects that have many similar properties, or when creating a new object is expensive or time-consuming.

In a Prototype Pattern, there are typically two key components:

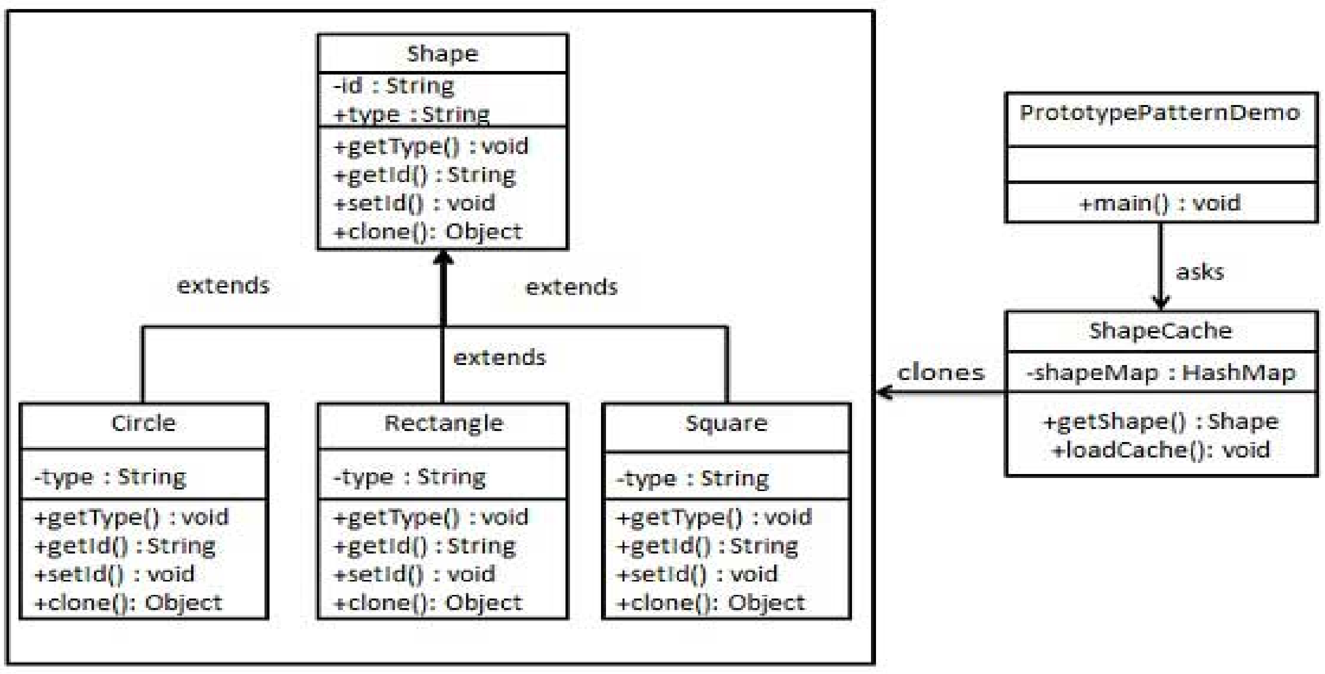
1. Prototype: This is the interface that declares the methods for cloning an object. It may also declare additional methods for configuring the cloned object.
2. Concrete Prototype: This class implements the prototype interface and provides the actual implementation of the cloning method. It may also contain additional methods for configuring the cloned object.

The Prototype Pattern works by creating a clone of an existing object, rather than creating a new object from scratch. This can be done in a number of ways, such as using a copy constructor or a clone method. Once the clone is created, it can be modified as needed to create a new, distinct object.

The Prototype Pattern is particularly useful when you need to create objects that have many similar properties, as it allows you to easily create copies of an existing object and modify them as needed. It can also be useful when creating new objects is expensive or time-consuming, as it allows you to create new objects quickly and efficiently by cloning existing ones.

It's worth noting that the Prototype Pattern can have some performance overhead, as creating a new object by cloning an existing one can be slower than creating a new object from scratch. Additionally, care must be taken to ensure that cloned objects are truly independent, and that modifying a cloned object does not inadvertently affect the original object or other cloned objects.

We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested. *PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape* object.



The Prototype Pattern provides a way to create new objects by cloning existing objects, which can be more efficient than creating new objects from scratch. It also allows for creating similar objects with different configurations or settings. The pattern can be used to reduce the cost of object creation, and to avoid the need for creating complex objects by hand.

Explain about Factory Method Pattern

The Factory Method Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This pattern is useful when you want to create objects that follow a common interface, but which can be customized or extended by subclasses.

In a Factory Method Pattern, there are typically four key components:

1. Creator: This is the superclass that defines the factory method interface for creating objects. It may also provide a default implementation of the factory method that creates a default type of object.

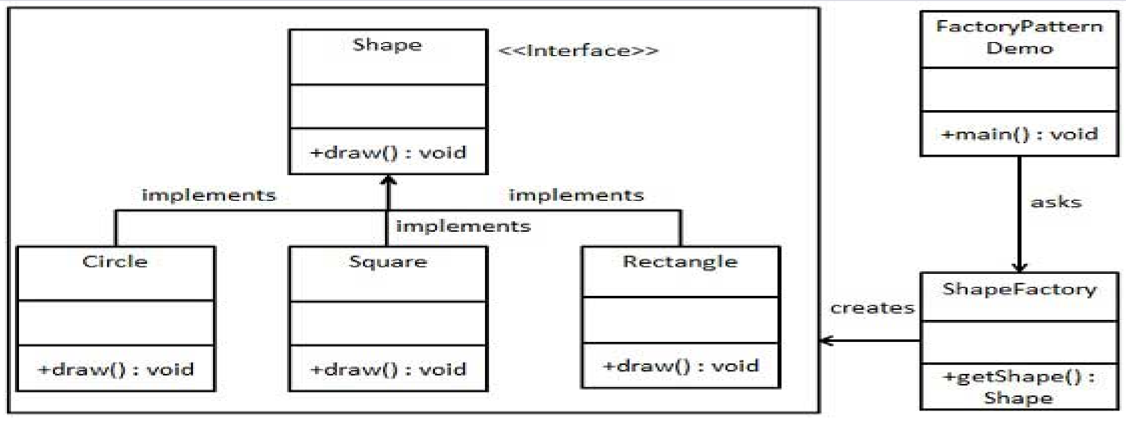
2. Concrete Creator: This is a subclass of the Creator that implements the factory method to create a specific type of object.
3. Product: This is the interface or abstract class that defines the common interface for the objects that will be created by the factory method.
4. Concrete Product: This is a subclass of the Product that provides a specific implementation of the interface.

The Factory Method Pattern works by defining a common interface for creating objects in the Creator superclass, but allowing subclasses to implement the factory method to create a specific type of object. This allows subclasses to customize or extend the behavior of the factory method, without changing the interface or behavior of the superclass.

The Factory Method Pattern is particularly useful when you want to create objects that follow a common interface, but which can be customized or extended by subclasses. It can also help to decouple client code from the specific implementations of the objects it creates, making it easier to maintain and modify the code over time.

It's worth noting that the Factory Method Pattern can have some overhead, as it requires additional classes and interfaces to be defined. Additionally, care must be taken to ensure that the factory method interface is well-designed and flexible enough to accommodate future changes and extensions.

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



The Factory Method Pattern provides a flexible and extensible way to create objects in a superclass, while allowing subclasses to alter the type of objects being created. It promotes loose coupling between the client code and the objects being created, making the code more maintainable and easier to extend.

Explain about Structural Patterns(ABCDFFP)

Structural patterns are design patterns that focus on the composition of classes and objects to form larger structures or systems. They help to define relationships between different classes and objects, making it easier to manage and modify a system's architecture.
Here are some common types of structural patterns:

1. Adapter Pattern: This pattern allows incompatible classes to work together by creating a bridge between them. It translates the interface of one class into the interface expected by another class, without changing the source code of either class.
2. Bridge Pattern: This pattern decouples an abstraction from its implementation so that both can vary independently. It is useful when you want to separate an abstraction from its implementation, such as when you want to support multiple platforms or databases.
3. Composite Pattern: This pattern allows you to treat individual objects and groups of objects in the same way. It lets you create a tree-like structure of objects, where each object can have zero or more child objects.
4. Decorator Pattern: This pattern lets you add behavior to an object dynamically, without changing its class. It allows you to add new functionality to an existing object by wrapping it in a decorator object, which provides the additional behavior.
5. Facade Pattern: This pattern provides a simplified interface to a complex subsystem. It lets you hide the complexity of a system behind a simple interface, making it easier to use and understand.
6. Flyweight Pattern: This pattern is used to minimize memory usage by sharing as much data as possible between objects. It creates a set of reusable objects that can be shared across multiple contexts, reducing the number of objects that need to be created.
7. Proxy Pattern: This pattern provides a surrogate or placeholder for another object, allowing you to control access to the object. It can be used to provide additional functionality or security checks before allowing access to an object.

Each of these patterns provides a different way of structuring and organizing classes and objects within a system. By using these patterns, you can improve the modularity, flexibility, and scalability of your code, making it easier to manage and modify over time.

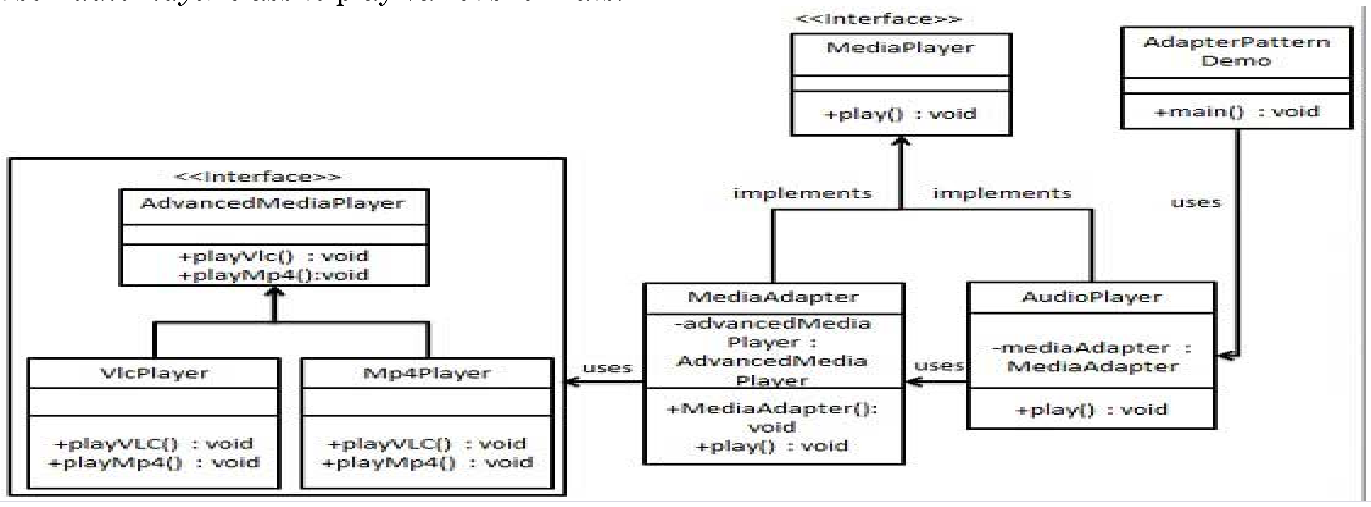
Describe about Adapter design pattern

The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together by wrapping an existing class with a new interface. This pattern is useful when you want to reuse existing code, but the interface of the existing code is not compatible with the interface required by the client code.
In an Adapter Pattern, there are typically three key components:

1. Target: This is the interface that is required by the client code, but which is not provided by the existing code.
2. Adapter: This is a class that implements the Target interface and wraps an existing class, providing the necessary interface to the client code.
3. Adaptee: This is the existing class that provides the functionality that the Adapter class wraps.

The Adapter Pattern works by creating a new class (the Adapter) that implements the Target interface required by the client code, but which wraps an existing class (the Adaptee) that provides the actual functionality. The Adapter class translates the interface of the Target interface into the interface of the Adaptee class, allowing the two to work together seamlessly.The Adapter Pattern is particularly useful when you want to reuse existing code, but the interface of the existing code is not compatible with the interface required by the client code. It can also help to decouple client code from the specific implementations of the objects it uses, making it easier to maintain and modify the code over time.

We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.We are having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.We want to make *AudioPlayer* to play other formats as well. To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.*AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.



It's worth noting that the Adapter Pattern can introduce additional overhead and complexity, as it requires an additional layer of abstraction to be added to the system. Additionally, care must be taken to ensure that the Adapter class is well-designed and provides a clear and consistent interface to the client code.

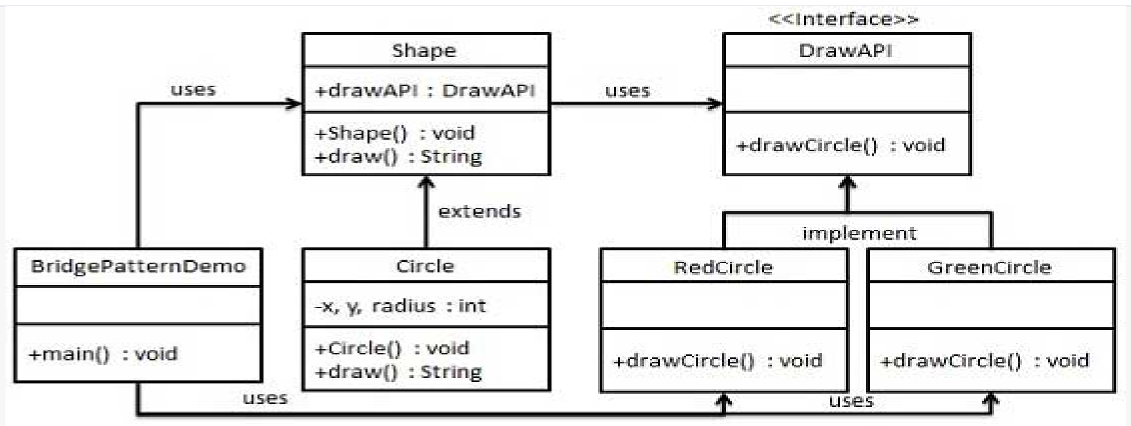
Describe about bridge design pattern

The Bridge Pattern is a structural design pattern that separates an abstraction from its implementation, allowing them to vary independently. This pattern is useful when you want to decouple an abstraction from its implementation, and allow them to evolve separately.
In a Bridge Pattern, there are typically two key components:

1. Abstraction: This is the high-level interface that clients use to interact with the system. It defines the operations that can be performed on the system, but does not specify how those operations are implemented.
2. Implementation: This is the low-level interface that provides the actual implementation of the system. It defines the operations that can be performed on the system and how those operations are implemented.

The Bridge Pattern works by separating the abstraction and implementation components into separate hierarchies, and providing a bridge between the two. The Abstraction hierarchy defines the high-level interface that clients use to interact with the system, while the Implementation hierarchy defines the low-level interface that provides the actual implementation of the system. The Bridge provides a link between the two hierarchies, allowing them to vary independently and enabling the system to evolve over time.
The Bridge Pattern is particularly useful when you want to decouple an abstraction from its implementation, and allow them to evolve separately. It can also help to reduce the impact of changes to the system by limiting the scope of changes to a specific component.

We have a *DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implementing the *DrawAPI* interface. *Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, our demo class will use *Shape* class to draw different colored circle.



It's worth noting that the Bridge Pattern can introduce additional complexity, as it requires an additional layer of abstraction to be added to the system. Additionally, care must be taken to ensure that the Abstraction and Implementation hierarchies are well-designed and provide a clear and consistent interface to the client code.

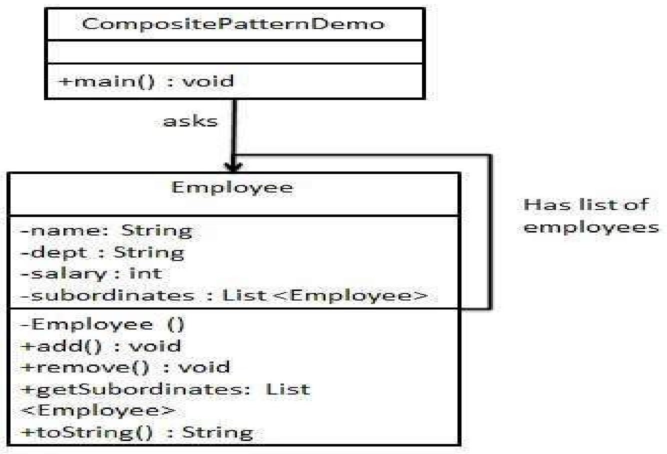
Describe about composite design pattern

Composite design pattern is a structural pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. This pattern allows you to treat individual objects and compositions of objects uniformly.

The Composite pattern consists of the following components:

1. Component: The interface that defines the common operations for both the Composite and Leaf classes.
2. Composite: A class that implements the Component interface and contains a collection of child Components.
3. Leaf: A class that implements the Component interface and represents a leaf node in the tree.

We have a class *Employee* which acts as composite pattern actor class. *CompositePatternDemo*, our demo class will use *Employee* class to add department level hierarchy and print all employees.



The Composite pattern is useful when you have a tree-like structure that contains leaf nodes and composite nodes. It can simplify code by allowing you to treat individual objects and compositions of objects uniformly. It can also make it easier to add or remove objects from the tree structure.

Describe about decorator design pattern

The Decorator design pattern is a structural pattern that allows you to add functionality to an object at runtime without changing the underlying class. It is useful when you want to add or remove features from an object dynamically, without having to modify the original code.

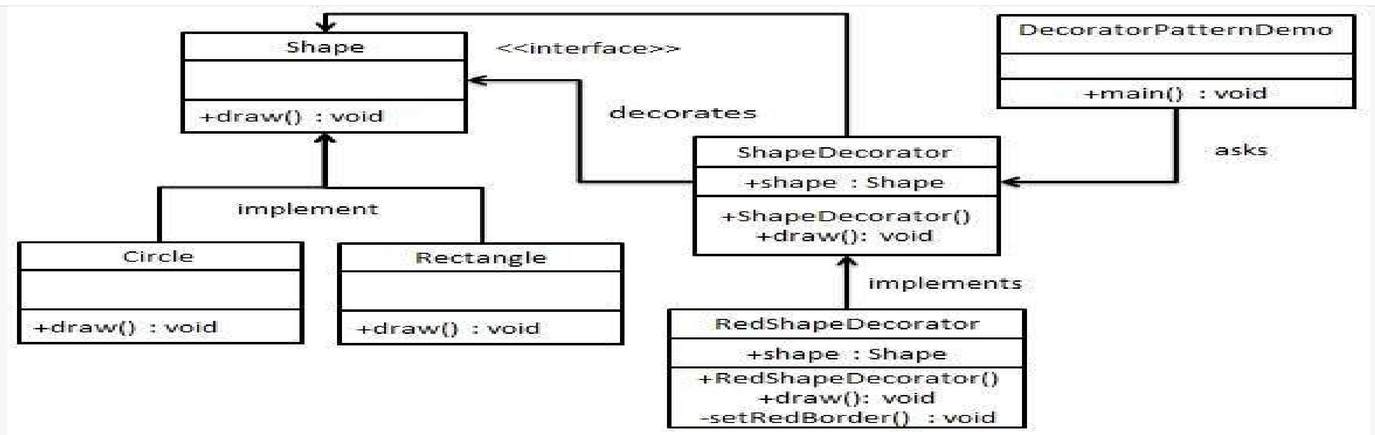
The Decorator pattern consists of the following components:

1. Component: The interface that defines the common operations for both the ConcreteComponent and Decorator classes.
2. ConcreteComponent: The class that implements the Component interface.
3. Decorator: The abstract class that implements the Component interface and contains a reference to a Component object. It also defines the common operations for all concrete decorators.
4. ConcreteDecorator: The class that extends the Decorator class and adds additional functionality to the Component object.

The Decorator Pattern works by creating a chain of decorator objects that each add new behavior to the original object. Each decorator object implements the Component interface and contains a reference to the original object. When a method is called on the decorated object, the decorator object adds its own behavior before passing the call on to the original object. This allows you to add functionality to an object at runtime without modifying its original code.The Decorator Pattern is particularly useful when you want to add functionality to an object without modifying its original code. It can also help to simplify client code by providing a consistent interface for working with decorated objects.

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We will then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable.*RedShapeDecorator* is concrete class implementing *ShapeDecorator*.

DecoratorPatternDemo, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.



Decorator Pattern can introduce additional overhead and complexity, as it requires a chain of decorator objects to be created. Additionally, care must be taken to ensure that the Component interface is well-designed and provides a clear and consistent interface to the client code.

The Decorator pattern is useful when you want to add or remove features from an object dynamically, without having to modify the original code. It can also make it easier to add new features to an object in the future, since you can simply create a new decorator instead of modifying the existing code.

Describe about facade design pattern

The Facade Pattern is a structural design pattern that provides a simplified interface to a complex subsystem. It encapsulates a group of individual classes or interfaces, making them easier to use and reducing their overall complexity.

In a Facade Pattern, there are typically two key components:

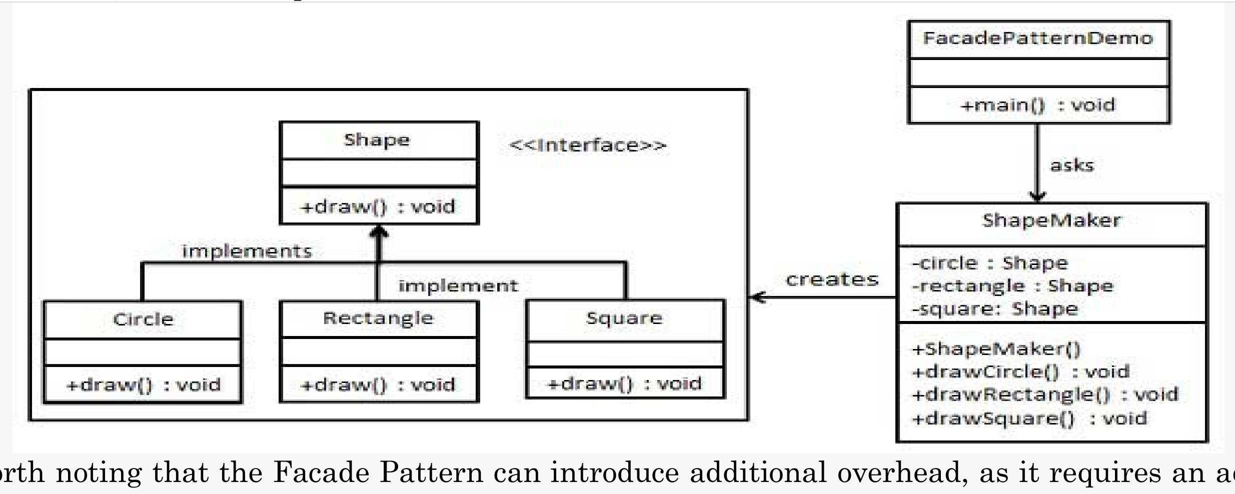
1. Facade: This is a class that provides a simplified interface to a complex subsystem. It encapsulates a group of individual classes or interfaces and provides a unified interface for client code to interact with.
2. Subsystem: This is a group of individual classes or interfaces that implement the functionality of the system. They are often complex and have a high degree of interdependence.

The Facade Pattern works by creating a simplified interface for client code to interact with. The Facade class encapsulates the complexity of the subsystem, and provides a unified interface for client code to interact with. This simplifies the client code, reduces the complexity of the system, and makes it easier to maintain and modify.

The Facade Pattern is particularly useful when you have a complex system with many individual classes or interfaces, and you want to simplify the interface for client code to interact with. It can also help to reduce the coupling between client code and the subsystem, making it easier to modify or replace individual components of the system.

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



It's worth noting that the Facade Pattern can introduce additional overhead, as it requires an additional layer of abstraction to be added to the system. Additionally, care must be taken to ensure that the Facade interface is well-designed and provides a clear and consistent interface to the client code.

Describe about flyweight design pattern

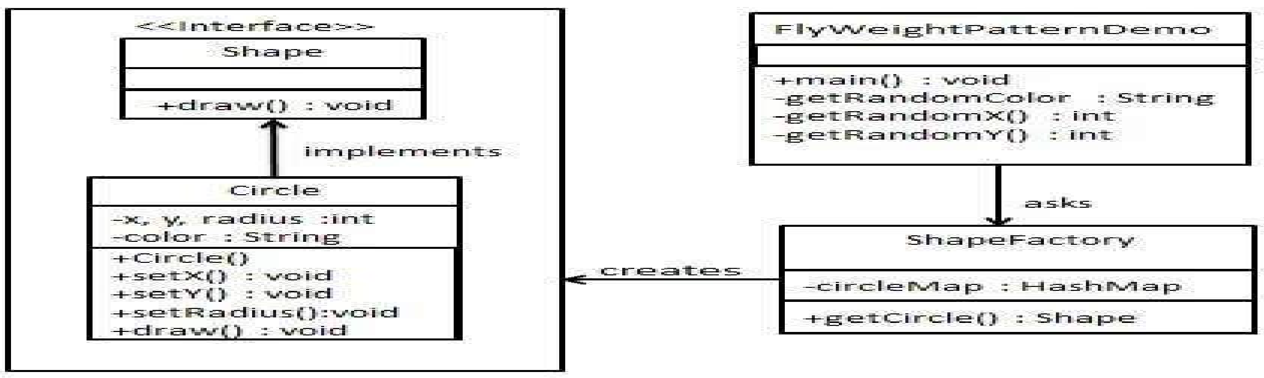
The Flyweight design pattern is a structural pattern that allows you to use shared objects to reduce memory usage and improve performance. It achieves this by sharing common data between multiple objects, rather than duplicating that data in each object.

The Flyweight pattern consists of the following components:

1. FlyweightFactory: A factory that creates and manages flyweight objects. It ensures that flyweight objects are shared between multiple clients and are not duplicated unnecessarily.
2. Flyweight: An interface that defines the common data that can be shared between flyweight objects. It typically has a few intrinsic properties that are shared between multiple objects.
3. ConcreteFlyweight: An implementation of the Flyweight interface. It contains the intrinsic state that is shared between multiple objects. It is typically immutable and cannot be changed once it is created.
4. Client: The object that uses the flyweight objects. It typically passes the extrinsic state (i.e., state that is specific to a single object) to the flyweight objects when it needs to use them.

The Flyweight Pattern works by separating the intrinsic state of an object from its extrinsic state. The intrinsic state is shared between multiple objects, while the extrinsic state is unique to each object. The

Flyweight Factory manages a pool of flyweight objects, and when a new object is requested, it first checks if an object with the requested intrinsic state already exists in the pool. If it does, it returns that object. If not, it creates a new object and adds it to the pool.
The Flyweight Pattern is particularly useful when you need to create a large number of objects with similar or identical state. By sharing the intrinsic state between multiple objects, you can reduce the memory footprint of the system and improve performance.
We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
ShapeFactory has a *HashMap* of *Circle* having key as color of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactory*, it checks the circle object in its *HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.
FlyWeightPatternDemo, our demo class, will use *ShapeFactory* to get a *Shape* object. It will pass information (*red / green / blue/ black / white*) to *ShapeFactory* to get the circle of desired color it needs.



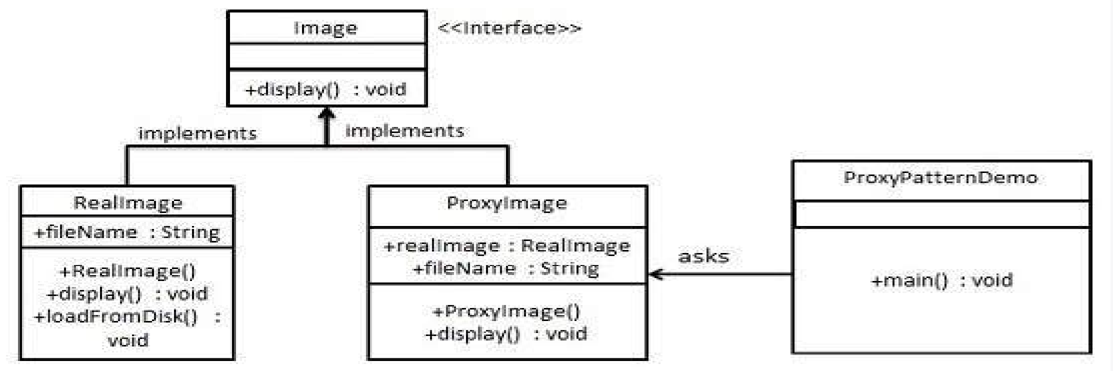
It's worth noting that the Flyweight Pattern can introduce additional complexity, as it requires the separation of intrinsic and extrinsic state, and the management of a pool of flyweight objects. Additionally, care must be taken to ensure that the Flyweight interface is well-designed and provides a clear and consistent interface to the client code.

Describe about proxy design pattern

The Proxy design pattern is a structural pattern that provides a surrogate or placeholder for another object to control access to it. It allows you to create a wrapper object that acts as a representative of the original object, allowing you to control access to the original object or add additional functionality to it.
The Proxy pattern consists of the following components:

1. Subject: An interface that defines the common methods that are shared by both the Proxy and the RealSubject. This ensures that both the Proxy and the RealSubject can be used interchangeably.
2. RealSubject: The original object that the Proxy represents. This is the object that the client wants to use, but the Proxy provides a layer of indirection to control access to the RealSubject.
3. Proxy: The wrapper object that acts as a surrogate or placeholder for the RealSubject. It implements the Subject interface and forwards requests to the RealSubject when necessary. The Proxy can add additional functionality to the RealSubject, such as caching, security checks, or logging.

The Proxy Pattern works by creating a proxy object that sits between the client code and the real subject object. The proxy object forwards requests to the real subject object, but can add additional behavior or control access to the object. This allows the proxy object to act as a filter or gatekeeper for the real subject object.
The Proxy Pattern is particularly useful when you need to control access to an object, or when you need to add additional behavior to an object. It can also be used to defer the creation of an object until it is actually needed, which can improve performance and reduce memory usage.
We are going to create an *Image* interface and concrete classes implementing the *Image* interface. *ProxyImage* is a a proxy class to reduce memory footprint of *RealImage* object loading. *ProxyPatternDemo*, our demo class, will use *ProxyImage* to get an *Image* object to load and display as it needs.



It's worth noting that the Proxy Pattern can introduce additional overhead, as it requires an additional layer of abstraction to be added to the system. Additionally, care must be taken to ensure that the Proxy interface is well-designed and provides a clear and consistent interface to the client code.

Explain about behavioral design patterns ((CIMS)^{2}OTV)

Behavioral design patterns are software design patterns that focus on the communication and interaction between different objects in a software system. They Describe how objects should interact and behave in specific situations, with the goal of making the system more flexible, efficient, and maintainable.
Some common behavioral design patterns include:

overview of all the 11 Behavioral Design Patterns:

1. **Chain of Responsibility**: This pattern allows a request to be passed through a chain of objects until it is handled by one of the objects in the chain.
2. **Command**: This pattern encapsulates a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.
3. **Interpreter**: This pattern provides a way to interpret sentences or expressions in a language.
4. **Iterator**: This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
5. **Mediator**: This pattern defines an object that encapsulates how a set of objects interact, thereby reducing the dependencies between the objects.
6. **Memento**: This pattern provides a way to capture and restore an object's internal state without violating encapsulation.
7. **State**: This pattern allows an object to alter its behavior when its internal state changes. It is useful when an object's behavior depends on its state, and when the state changes, the behavior must also change.
8. **Strategy:** This pattern allows you to define a family of algorithms, encapsulate each one, and make them interchangeable at runtime.
9. **Observer**: This pattern defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
10. **Template Method**: This pattern defines the skeleton of an algorithm in a base class, allowing subclasses to provide concrete implementations for specific steps.
11. **Visitor**: This pattern allows you to separate the algorithm from an object's structure on which it operates, by allowing you to define a new operation without changing the classes of the objects on which it operates.

Each of these design patterns provides a solution to a specific software design problem, and when used properly, they can promote flexibility, reusability, and maintainability in software systems. Overall, behavioral design patterns can help to improve the quality, maintainability, and extensibility of a software system, by promoting clear and modular communication between objects.

Explain about command design pattern

Command design pattern, also known as the Command Interpreter pattern. The Command pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.
The Command pattern consists of the following main components:

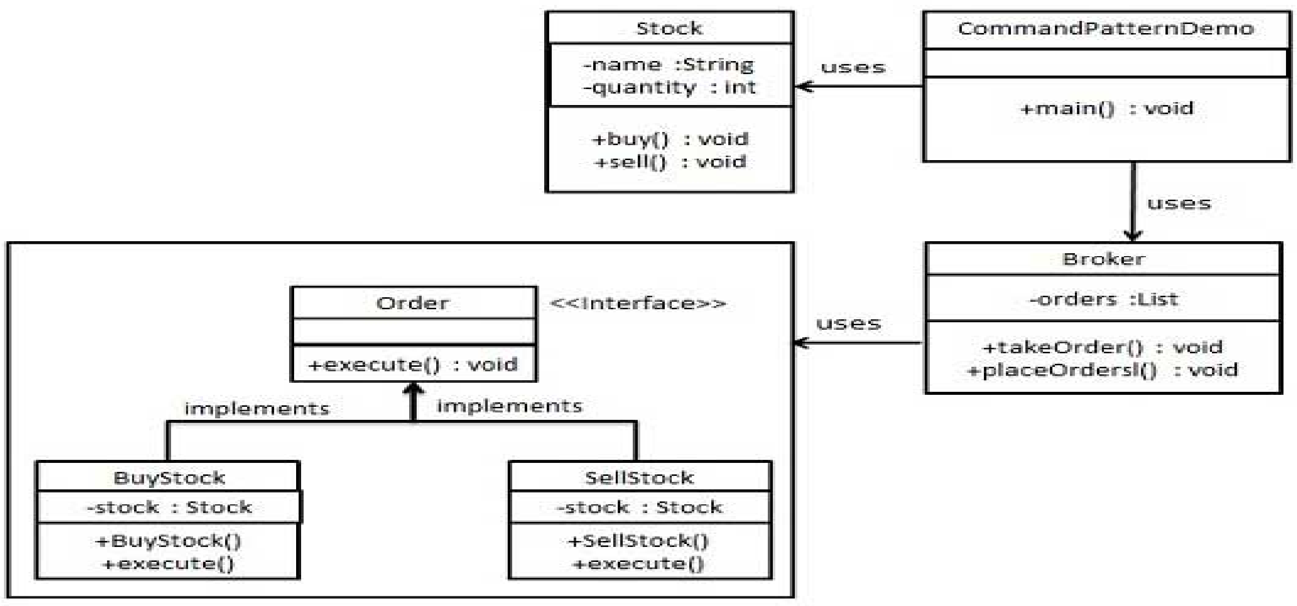
1. Command: This is the abstract base class or interface that defines the common methods or interface that all concrete command objects must implement.
2. Concrete Command: These are the concrete implementations of the command base class or interface. Each concrete command encapsulates a specific set of actions or operations to be performed.
3. Invoker: This is the class that is responsible for executing the commands. It maintains a reference to the command object and calls its execute method when necessary.
4. Receiver: This is the class that performs the actual actions or operations requested by the command.

The Command pattern is useful when you need to separate the request for an action from its execution, or when you want to provide a way to undo or redo actions. It also allows for a flexible and extensible system by allowing new commands to be added without affecting the existing code.
The key advantages of using the Command pattern include:

1. Flexibility: The Command pattern makes it easy to add new commands or modify the existing ones without affecting the client code.
2. Reusability: The Command pattern promotes code reuse by encapsulating the request as an object that can be used in multiple contexts.
3. Undo and Redo: The Command pattern provides an easy way to implement undo and redo operations by keeping a history of executed commands.

Common examples of the Command pattern include implementing undo and redo operations in a text editor, or in a game engine, where commands can be used to control the game entities and their actions.

We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.
Broker object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern.



Overall, the Command pattern is a useful pattern that promotes flexibility, reusability, and undoable operations in software systems. It provides a way to separate the request for an action from its execution, making it easier to modify or extend the system in a flexible and extensible manner.

Explain about Chain of Responsibility

The Chain of Responsibility is a behavioral design pattern that allows a request to be passed through a chain of objects until it is handled by one of the objects in the chain. The chain of objects represents a series of potential handlers for the request, and each handler is responsible for either handling the request or passing it on to the next handler in the chain.

The Chain of Responsibility pattern consists of the following main components:

1. Handler: This is the abstract base class or interface that defines the common methods or interface that all concrete handlers must implement.
2. Concrete Handler: These are the concrete implementations of the handler base class or interface. Each concrete handler encapsulates a unique set of rules and criteria for handling a specific type of request.
3. Client: This is the class that initiates the request and passes it to the first handler in the chain.

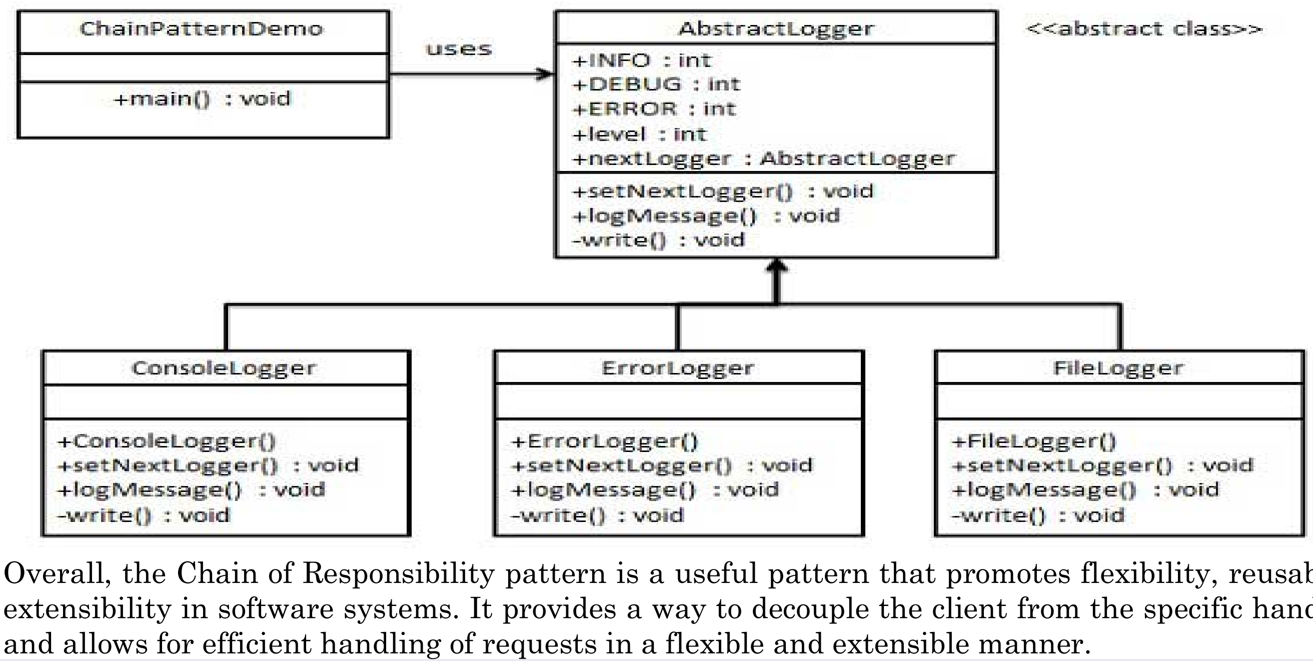
The Chain of Responsibility pattern is useful when there are multiple objects that can handle a request, and the client does not know which object can handle the request. The pattern provides a way to decouple the client from the specific handling logic by allowing the objects to decide whether to handle the request or pass it on to the next object in the chain.

The key advantages of using the Chain of Responsibility pattern include:

1. Flexibility: The Chain of Responsibility pattern makes it easy to add new handlers to the chain or modify the existing handlers without affecting the client code.
2. Reusability: The Chain of Responsibility pattern promotes code reuse by encapsulating the handling logic into a separate class that can be used in multiple contexts.
3. Extensibility: The Chain of Responsibility pattern makes it easy to add new types of requests or handlers to the system without modifying the existing code.

Common examples of the Chain of Responsibility pattern include handling requests in an e-commerce website, where a request can be passed through a series of objects to determine if the user is eligible for a discount or a coupon. Another example is in a helpdesk application, where a request can be passed through a chain of support staff until it is resolved.

We have created an abstract class *AbstractLogger* with a level of logging. Then we have created three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Overall, the Chain of Responsibility pattern is a useful pattern that promotes flexibility, reusability, and extensibility in software systems. It provides a way to decouple the client from the specific handling logic and allows for efficient handling of requests in a flexible and extensible manner.

Explain about Interpreter pattern

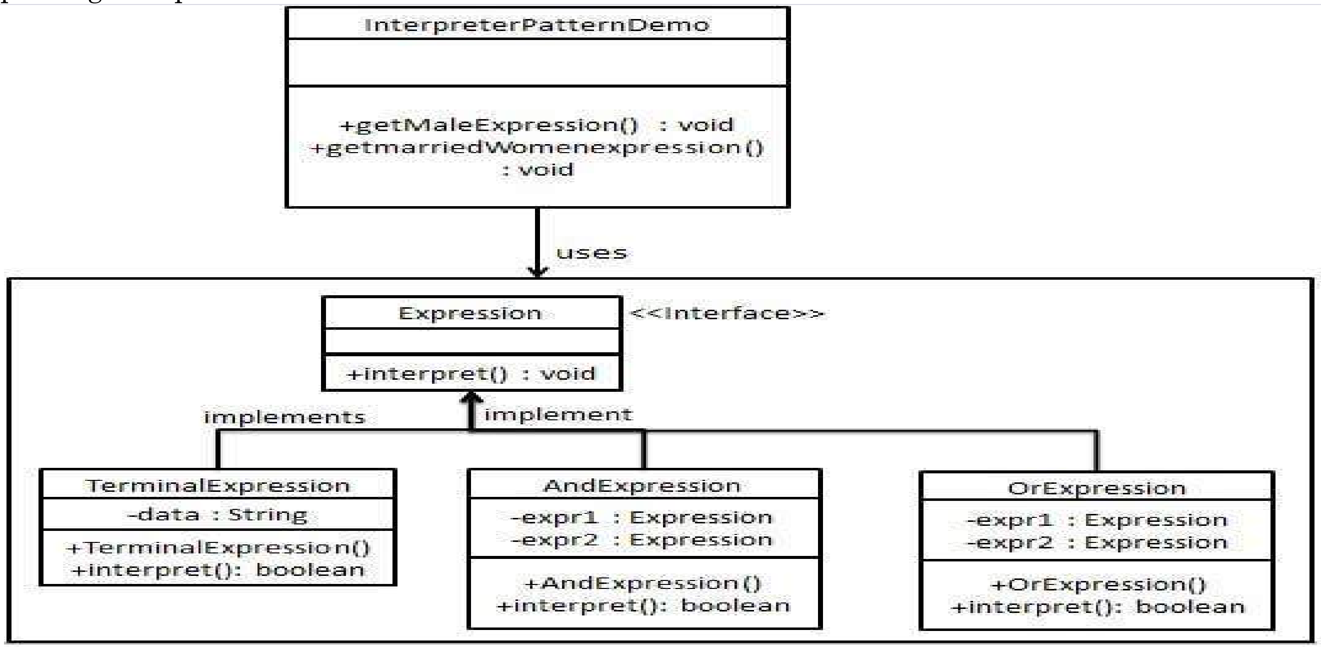
The Interpreter pattern is a behavioral design pattern that provides a way to interpret or evaluate a language or grammar. It defines a language or syntax and provides a way to interpret or evaluate sentences or expressions written in that language.
The Interpreter pattern consists of the following main components:

1. Context: This is the class that contains the information that the interpreter uses to interpret the language or syntax.
2. Abstract Expression: This is the base class or interface that defines the common methods or interface that all concrete expression objects must implement.
3. Terminal Expression: These are the concrete implementations of the expression base class or interface. Each terminal expression represents a terminal symbol in the language or syntax, such as a variable, a number, or a keyword.
4. Non-Terminal Expression: These are the concrete implementations of the expression base class or interface. Each non-terminal expression represents a non-terminal symbol in the language or syntax, such as a sequence of expressions or a repeated expression.

The Interpreter pattern is useful when you need to define a language or syntax and provide a way to interpret or evaluate expressions written in that language. It is often used in compilers, interpreters, and query languages.
The key advantages of using the Interpreter pattern include:

1. Flexibility: The Interpreter pattern allows you to define new expressions and add them to the language or syntax without affecting the existing code.
2. Reusability: The Interpreter pattern promotes code reuse by encapsulating the language or syntax in a separate object that can be used in multiple contexts.
3. Extensibility: The Interpreter pattern allows you to add new operations to the language or syntax by defining new non-terminal expressions.

Common examples of the Interpreter pattern include interpreting mathematical expressions, regular expressions, and database queries.
We are going to create an interface *Expression* and concrete classes implementing the *Expression* interface. A class *TerminalExpression* is defined which acts as a main interpreter of context in question. Other classes *OrExpression*, *AndExpression* are used to create combinational expressions.
InterpreterPatternDemo, our demo class, will use *Expression* class to create rules and demonstrate parsing of expressions.



Overall, the Interpreter pattern is a useful pattern that provides a way to define and interpret a language or syntax. It allows you to define new expressions, add them to the language or syntax, and interpret expressions written in that language.

Explain about Iterator design pattern

The Iterator pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The pattern defines an interface for iterating over an aggregate object, and provides a way to traverse the elements of the aggregate without exposing its implementation details.
The Iterator pattern consists of the following main components:

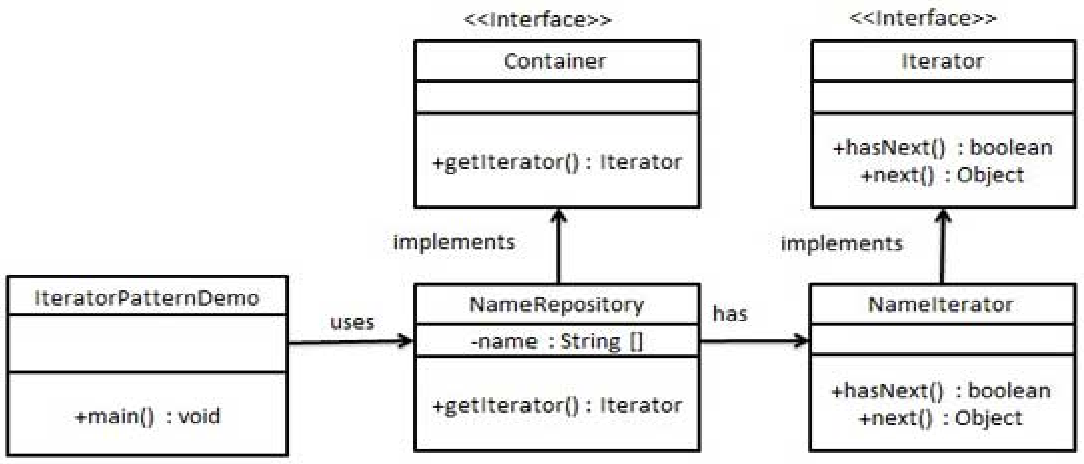
1. Iterator: This is an interface that defines the methods that the concrete iterators must implement. It typically includes methods for checking if there are more elements, getting the next element, and resetting the iterator.
2. Concrete Iterator: These are the classes that implement the Iterator interface and provide a way to traverse the elements of the aggregate object.
3. Aggregate: This is an interface that defines the methods for creating an iterator object. It typically includes a method for creating a new iterator object.
4. Concrete Aggregate: These are the classes that implement the Aggregate interface and provide a way to create a new iterator object.

The Iterator pattern is useful when you need to traverse the elements of an aggregate object without exposing its implementation details. It is often used in conjunction with other patterns, such as the Composite pattern and the Visitor pattern.
The key advantages of using the Iterator pattern include:

1. Encapsulation: The Iterator pattern encapsulates the traversal algorithm in a separate object, which allows you to change the traversal algorithm without affecting the aggregate object.
2. Reusability: The Iterator pattern promotes code reuse by allowing you to reuse the same iterator object to traverse different aggregate objects.
3. Separation of Concerns: The Iterator pattern separates the concerns of iterating over the elements of an aggregate object from the concerns of the aggregate object itself.

Common examples of the Iterator pattern include iterating over the elements of a list, a tree, or a database query result set.

We're going to create a *Iterator* interface which narrates navigation method and a *Container* interface which retruns the iterator . Concrete classes implementing the *Container* interface will be responsible to implement *Iterator* interface and use it
IteratorPatternDemo, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.



Overall, the Iterator pattern is a useful pattern that provides a way to traverse the elements of an aggregate object without exposing its implementation details. It encapsulates the traversal algorithm in a separate object, which allows you to change the traversal algorithm without affecting the aggregate object.

Explain about Mediator design pattern

The Mediator pattern is a behavioral design pattern that provides a way to reduce the coupling between objects by mediating their communication through a central mediator object. The pattern defines a mediator object that encapsulates the communication between objects and provides a way for them to communicate without knowing about each other.
The Mediator pattern consists of the following main components:

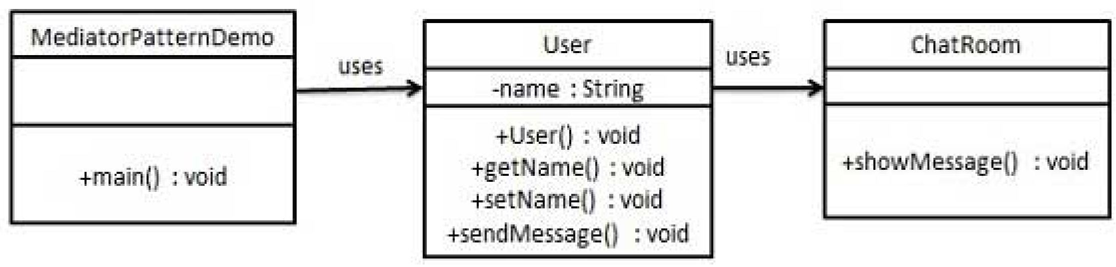
1. Mediator: This is an interface or abstract class that defines the methods for communicating between the objects. It typically includes methods for registering and notifying the objects.
2. Concrete Mediator: This is the class that implements the Mediator interface or abstract class and provides the concrete implementation of the communication between the objects.
3. Colleague: This is an interface or abstract class that defines the methods for communicating with the mediator object. It typically includes methods for sending and receiving messages.
4. Concrete Colleague: These are the classes that implement the Colleague interface or abstract class and provide the concrete implementation of the communication with the mediator object.

The Mediator pattern is useful when you have a set of objects that need to communicate with each other but you want to reduce the coupling between them. It is often used in complex systems where the communication between the objects can become tangled and difficult to maintain.
The key advantages of using the Mediator pattern include:

1. Decoupling: The Mediator pattern reduces the coupling between objects by mediating their communication through a central mediator object.
2. Flexibility: The Mediator pattern allows you to change the way the objects communicate with each other without affecting their implementation.
3. Simplification: The Mediator pattern simplifies the communication between the objects by encapsulating it in a separate mediator object.

Common examples of the Mediator pattern include a chat room application, a traffic control system, and a flight control system.

We are demonstrating mediator pattern by example of a chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes *ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages.
MediatorPatternDemo, our demo class, will use *User* objects to show communication between them.



Overall, the Mediator pattern is a useful pattern that provides a way to reduce the coupling between objects by mediating their communication through a central mediator object. It encapsulates the communication between the objects in a separate mediator object, which simplifies the communication and allows you to change it without affecting the implementation of the objects.

Explain about Memento design pattern

The Memento pattern is a behavioral design pattern that provides a way to capture and restore the state of an object without violating its encapsulation. The pattern defines a memento object that encapsulates the state of an object at a particular point in time, and provides a way to restore that state later.

The Memento pattern consists of the following main components:

1. Originator: This is the class that creates and maintains the state of the object. It typically includes methods for saving and restoring the state of the object.
2. Memento: This is the class that encapsulates the state of the object at a particular point in time. It typically includes methods for getting and setting the state of the object.
3. Caretaker: This is the class that manages the memento objects. It typically includes methods for saving and retrieving memento objects.

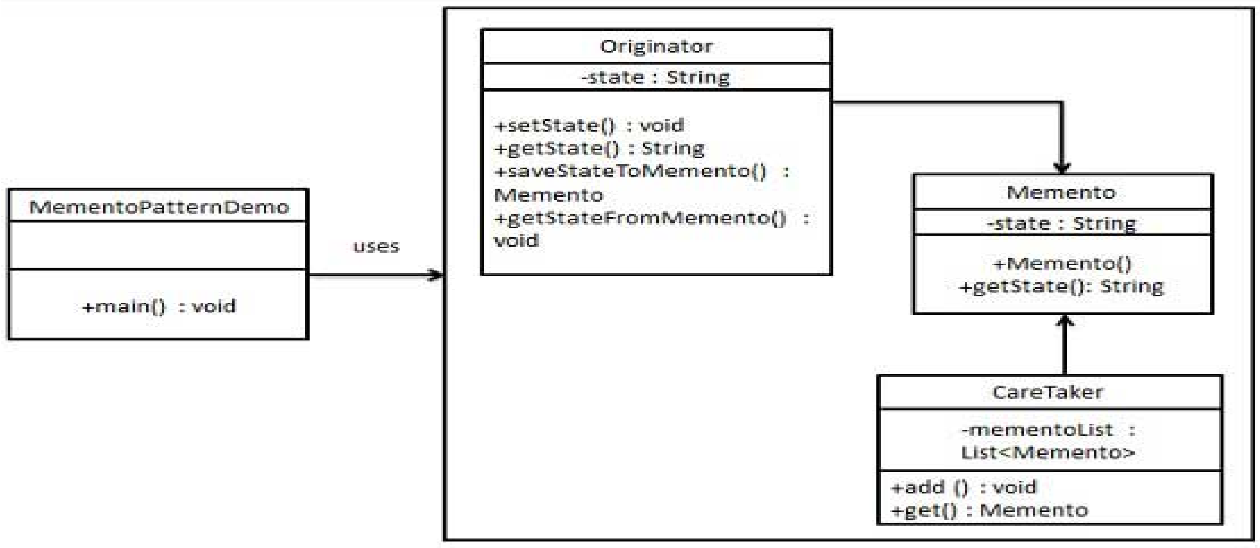
The Memento pattern is useful when you need to capture the state of an object and restore it later, without violating its encapsulation. It is often used in undo/redo functionality in applications, where the user can undo or redo their actions.

The key advantages of using the Memento pattern include:

1. Encapsulation: The Memento pattern encapsulates the state of an object in a separate memento object, which allows you to capture and restore the state without violating its encapsulation.
2. Flexibility: The Memento pattern allows you to capture and restore the state of an object at any point in time, which provides flexibility and support for undo/redo functionality.
3. Simplicity: The Memento pattern simplifies the process of saving and restoring the state of an object by encapsulating it in a separate memento object.

Common examples of the Memento pattern include a text editor that allows users to undo or redo their actions, a game that allows players to save and restore their progress, and a browser that allows users to restore their tabs after a crash.

Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. We have created classes *Memento*, *Originator* and *CareTaker*.
MementoPatternDemo, our demo class, will use *CareTaker* and *Originator* objects to show restoration of object states.



Overall, the Memento pattern is a useful pattern that provides a way to capture and restore the state of an object without violating its encapsulation. It encapsulates the state of the object in a separate memento object, which provides flexibility and support for undo/redo functionality, and simplifies the process of saving and restoring the state of an object.

Explain about strategy design patterns

The Strategy design pattern is a behavioral design pattern that allows the selection of an algorithm at runtime. It encapsulates a group of related algorithms and allows the client to select and use one of them without tightly coupling the algorithm's implementation to the client code.

The Strategy pattern consists of three main components:

1. Context: This is the class that interacts with the Strategy pattern and is responsible for configuring and selecting the appropriate strategy to use for a given task. It maintains a reference to the current strategy object, and it delegates the algorithmic logic to the selected strategy.

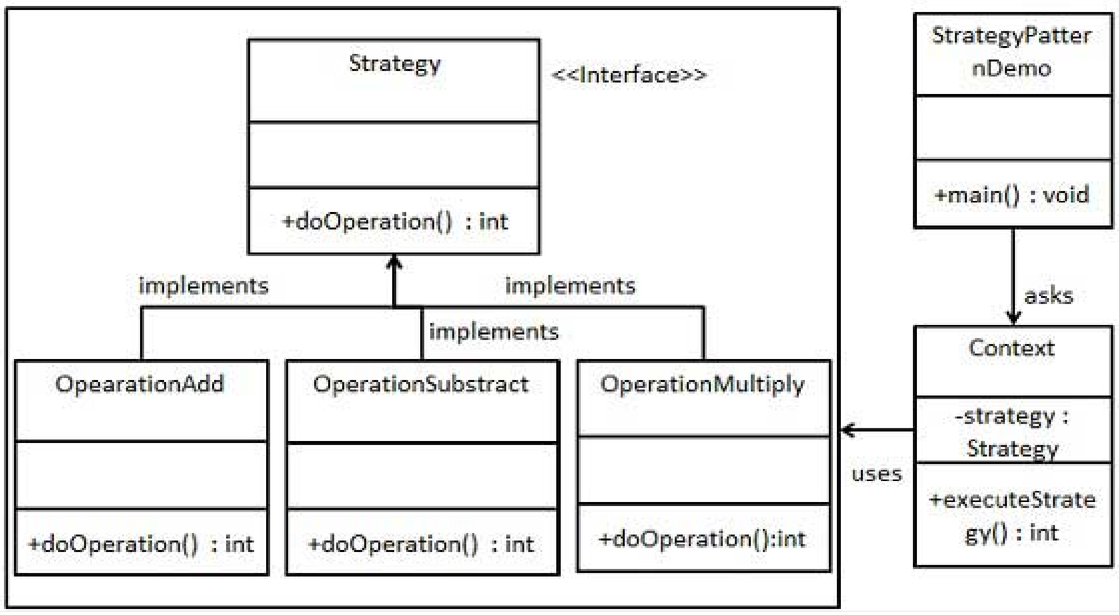
2. Strategy: This is the abstract base class or interface that defines the common methods or interface that all concrete strategies must implement.
3. Concrete Strategy: These are the concrete implementations of the abstract strategy base class or interface. Each concrete strategy encapsulates a unique algorithmic logic that can be used interchangeably by the context.

The Strategy pattern can be useful in situations where there are multiple algorithms that can be used to solve a problem, or when there is a need to swap algorithms at runtime based on changing conditions. For example, in an online shopping application, the strategy pattern can be used to calculate the shipping cost based on different shipping methods, and the client can select the appropriate strategy based on their preferences.

The key advantages of using the Strategy pattern include:

1. Flexibility: The Strategy pattern allows the client to choose the appropriate algorithm dynamically at runtime, without modifying the existing code.
2. Reusability: The Strategy pattern promotes code reuse by encapsulating the algorithmic logic into a separate class that can be used in multiple contexts.
3. Extensibility: The Strategy pattern makes it easy to add new algorithms or strategies to the system without modifying the existing code.

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy. *StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



Overall, the Strategy pattern helps to create more flexible, maintainable, and extensible software systems by allowing us to encapsulate and swap out different algorithms or behaviors at runtime.

Explain about state design pattern

The State pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. It defines a set of states for an object and provides a way to switch between those states dynamically.

The State pattern consists of the following main components:

1. Context: This is the class that defines the interface to the clients and maintains a reference to the current state object.
2. State: This is the interface or abstract class that defines the methods for handling the different states of the context object.
3. Concrete State: These are the classes that implement the State interface or abstract class and provide the concrete implementation of the behavior for the different states of the context object.

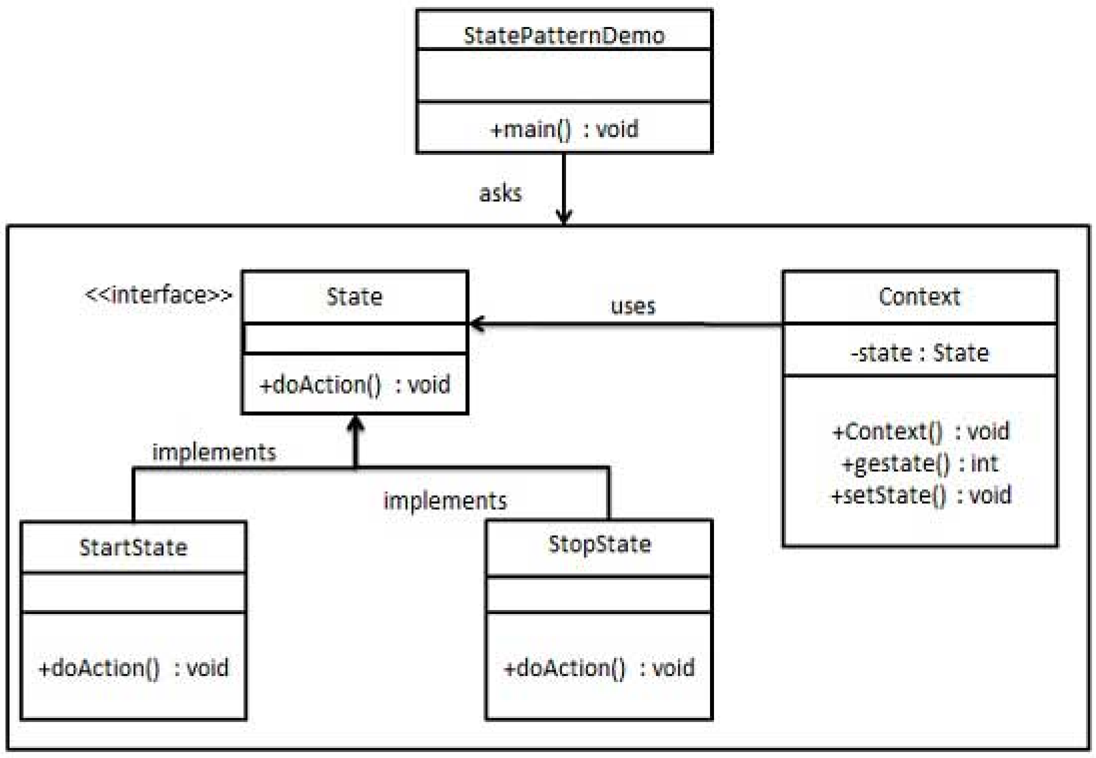
The State pattern is useful when you have an object that has a number of states and its behavior changes based on its internal state. It is often used in applications where the behavior of an object needs to change dynamically, based on user input or other external factors.

The key advantages of using the State pattern include:

1. Flexibility: The State pattern allows you to add new states to an object without affecting its existing states or behavior.
2. Encapsulation: The State pattern encapsulates the behavior of an object in separate state objects, which improves encapsulation and reduces code duplication.
3. Simplicity: The State pattern simplifies the code by removing the need for conditional statements or switch statements to handle different states.

Common examples of the State pattern include a vending machine that changes its behavior based on the state of its inventory, a game that changes its behavior based on the player's level, and a traffic light system that changes its behavior based on the time of day.

We are going to create a *State* interface defining an action and concrete state classes implementing the *State* interface. *Context* is a class which carries a State.
StatePatternDemo, our demo class, will use *Context* and state objects to demonstrate change in Context behavior based on type of state it is in.



Overall, the State pattern is a useful pattern that allows an object to alter its behavior when its internal state changes. It encapsulates the behavior of an object in separate state objects, which improves encapsulation and reduces code duplication, and simplifies the code by removing the need for conditional statements or switch statements to handle different states.

Explain about observer design patterns

The Observer design pattern is a behavioral pattern that allows for one-to-many communication between objects, where changes made to one object are propagated to all dependent objects. The pattern is used when we have a set of objects that need to be notified when another object changes its state.

The Observer pattern consists of the following main components:

1. Subject: This is the object that is observed, and its state changes are propagated to all registered observers.
2. Observer: This is the interface or abstract class that defines the common methods or interface that all concrete observers must implement.
3. Concrete Observer: These are the concrete implementations of the observer interface that are notified when the state of the subject changes.

The Observer pattern promotes loose coupling between objects, where the subject and observers are decoupled, and the observers do not need to know about the implementation details of the subject. This makes it easy to add new observers or remove existing ones without affecting the subject or other observers.

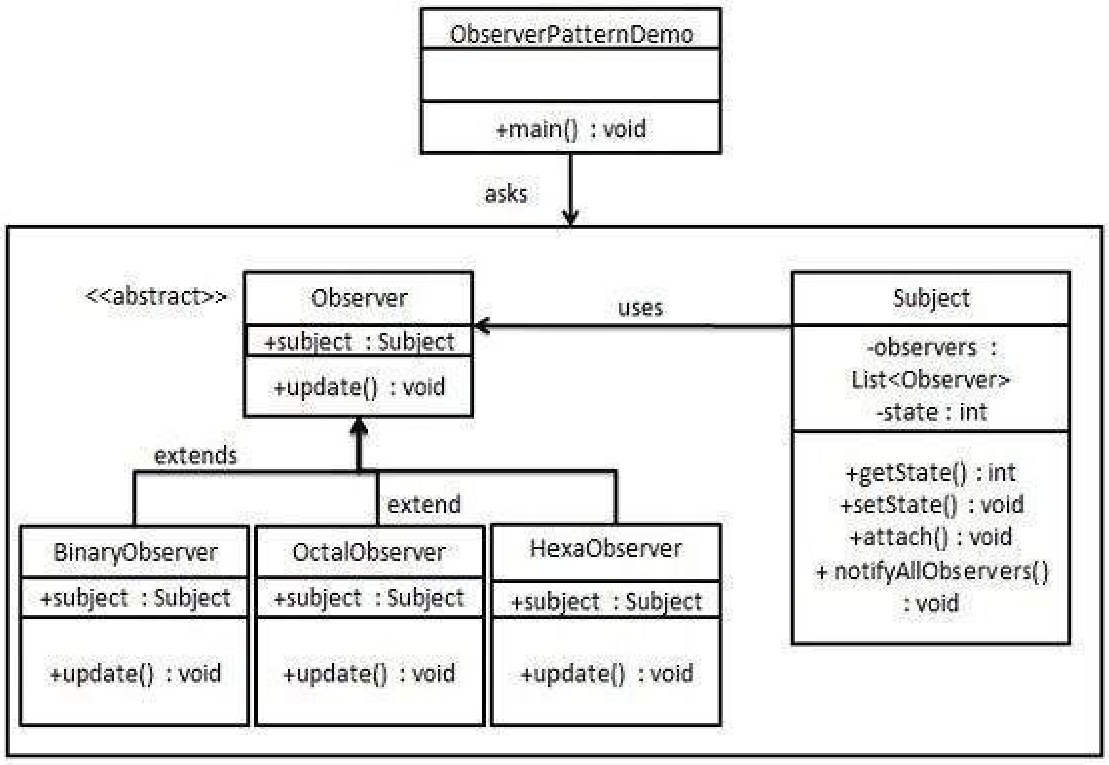
The key advantages of using the Observer pattern include:

1. Flexibility: The Observer pattern makes it easy to add new observers or remove existing ones without affecting the subject or other observers.
2. Reusability: The Observer pattern promotes code reuse by decoupling the subject and observers and making them more modular.
3. Maintainability: The Observer pattern makes it easier to maintain the code by separating concerns and making it more modular.

Common examples of the Observer pattern include the Model-View-Controller (MVC) architecture used in graphical user interfaces, where the model is the subject, and the views are the observers. Another example is a stock price monitoring system, where multiple users can subscribe to changes in the stock prices, and the subject notifies all subscribers when the prices change.

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.

ObserverPatternDemo, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



Overall, the Observer pattern is a useful pattern that promotes loose coupling, reusability, and maintainability in software systems. It allows for efficient communication between objects while preserving their independence, making it a valuable tool for designing modular and flexible software systems.

Explain about Template Method design pattern

The Template Method pattern is a behavioral design pattern that defines the basic structure of an algorithm and allows subclasses to override certain steps of the algorithm without changing its structure. It encapsulates a common set of steps in an algorithm in a base class and allows subclasses to implement specific steps to customize the behavior of the algorithm.

The Template Method pattern consists of the following main components:

1. Abstract Class: This is the class that defines the common template method and the basic steps of the algorithm. It also includes abstract methods that subclasses must implement to customize the algorithm.
2. Concrete Class: This is the class that extends the abstract class and provides the concrete implementation of the abstract methods to customize the algorithm.

The Template Method pattern is useful when you have a series of steps in an algorithm that are common to multiple subclasses, but certain steps need to be customized. It is often used in applications where the overall process is the same, but the details of each step may vary.

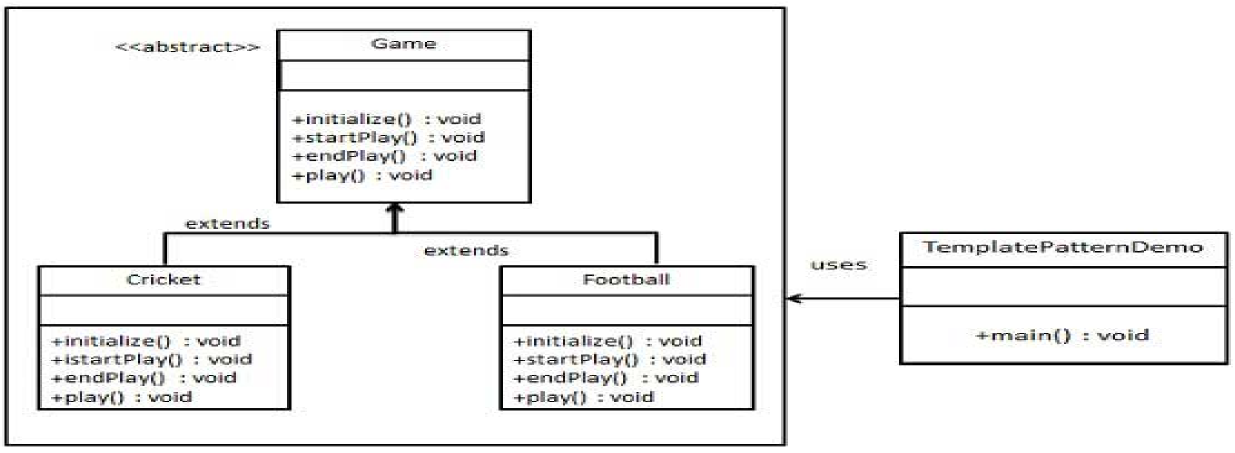
The key advantages of using the Template Method pattern include:

1. Code Reusability: The Template Method pattern promotes code reusability by encapsulating the common steps of an algorithm in a base class.
2. Flexibility: The Template Method pattern allows subclasses to override certain steps of the algorithm without changing its overall structure, which provides flexibility and customization.
3. Maintainability: The Template Method pattern simplifies code maintenance by encapsulating the common steps of an algorithm in a base class, which reduces the need for duplicate code.

Common examples of the Template Method pattern include a report generator that has a common set of steps for generating a report, but allows different types of reports to be generated, and a game that has a common set of steps for playing, but allows different game modes to be played.

We are going to create a *Game* abstract class defining operations with a template method set to be final so that it cannot be overridden. *Cricket* and *Football* are concrete classes that extend *Game* and override its methods.

TemplatePatternDemo, our demo class, will use *Game* to demonstrate use of template pattern.



Overall, the Template Method pattern is a useful pattern that defines the basic structure of an algorithm and allows subclasses to override certain steps of the algorithm without changing its overall structure. It promotes code reusability, flexibility, and maintainability by encapsulating the common steps of an algorithm in a base class.

Explain about visitor design pattern

The Visitor pattern is a behavioral design pattern that separates an algorithm from an object structure on which it operates. It allows adding new operations or algorithms to the object structure without changing the classes of the objects.

The Visitor pattern consists of the following main components:

1. Visitor: This is the interface or abstract class that defines the methods for visiting each element in the object structure.
2. Concrete Visitor: This is the class that implements the Visitor interface or abstract class and provides the concrete implementation of the methods for visiting each element in the object structure.
3. Element: This is the interface or abstract class that defines the methods for accepting visitors.
4. Concrete Element: This is the class that implements the Element interface or abstract class and provides the concrete implementation of the methods for accepting visitors.
5. Object Structure: This is the class that represents the collection of elements and provides the interface for visiting them.

The Visitor pattern is useful when you have a set of objects that need to be processed in different ways by different algorithms, and you don't want to modify the object structure classes each time a new algorithm is introduced. It is often used in applications that require multiple algorithms to operate on a collection of objects.

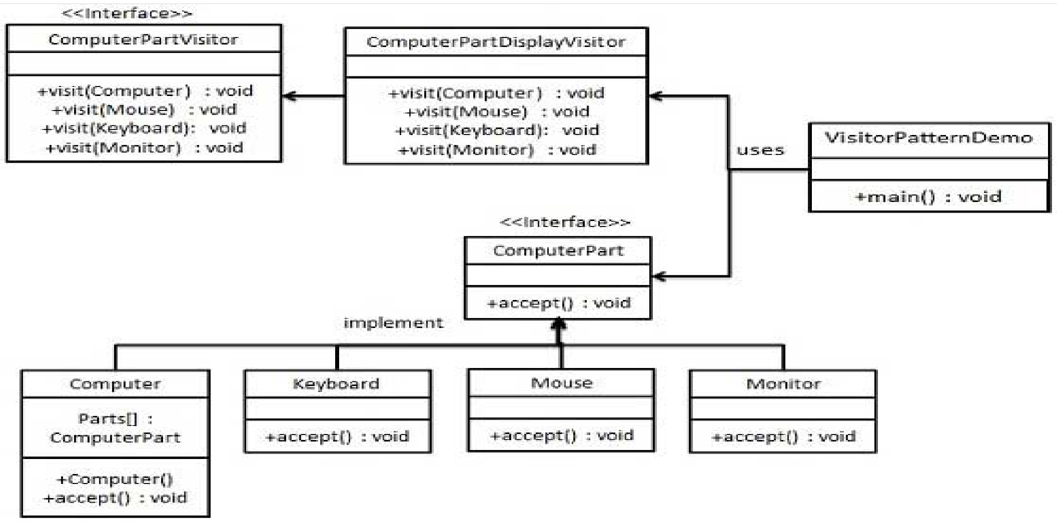
The key advantages of using the Visitor pattern include:

1. Separation of Concerns: The Visitor pattern separates the algorithm from the object structure, which improves the separation of concerns and reduces code duplication.
2. Extensibility: The Visitor pattern makes it easy to add new algorithms to an object structure without changing the classes of the objects.
3. Flexibility: The Visitor pattern allows objects to be processed in different ways by different algorithms, which provides flexibility and customization.

Common examples of the Visitor pattern include a code analyzer that operates on a set of classes to identify potential performance issues, and a report generator that operates on a set of objects to generate different types of reports.

We are going to create a *ComputerPart* interface defining accept opearation.*Keyboard*, *Mouse*, *Monitor* and *Computer* are concrete classes implementing *ComputerPart* interface. We will define another interface *ComputerPartVisitor* which will define a visitor class operations. *Computer* uses concrete visitor to do corresponding action.

VisitorPatternDemo, our demo class, will use *Computer* and *ComputerPartVisitor* classes to demonstrate use of visitor pattern.



Overall, the Visitor pattern is a useful pattern that separates an algorithm from an object structure on which it operates. It improves the separation of concerns, reduces code duplication, and allows adding new algorithms to an object structure without changing the classes of the objects.