SKILL LYNC WEBINARS

Design Patterns

Anku Kumar Choudhary

AGENDA

- What are design patterns
- Design pattern evolution and Gangs of four
- Types of Design pattern
  - Creational
  - Structural
  - Behavioural

Prerequisites

1. Good understanding of any programming language
2. Object oriented programming

Expectation from session

- Recognize and apply design patterns
- Refactor existing designs to use design patterns
- Reason about applicability and usability of design patterns

What are design patterns

- Design Patterns are reusable solutions to common programming problems.
- Design Patterns got popularized with the 1994 book Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, John Vlissides, Ralph Johnson and Richard Helm (who are commonly known as a Gang of Four, hence the GoF acronym).
- The original book was written using C++ and Smalltalk as examples, but since then, design patterns have been adapted to every programming language imaginable: C#, Java, PHP and even programming languages that aren't strictly object-oriented, such as JavaScript.
- The appeal of design patterns is immortal: we see them in libraries, some of them are intrinsic in programming languages, and you probably use them on a daily basis even if you don't realize they are there.

What will we cover?

- **SOLID** Design Principles:
  1. Single Responsibility Principle
  2. Open-Closed Principle
  3. Liskov Substitution Principle
  4. Interface Segregation Principle
  5. Dependency Inversion Principle

SKILL LYNC WEBINARS

Creational Design Patterns:

- Builder
- Factories
  - Factory Method
  - Abstract Factory
- Prototype
- Singleton

Structrural Design Patterns:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Design Patterns:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

SKILL LYNC WEBINARS

SOLID PRINCIPLES

Single Responsibility Principle

There Should never be more than one reason for a class to change

- SRP provides focused, single functionality
- Addresses a specific concern
- With SRP code can be modified with minimal changes
- SRP leads to loosely coupled classes
- SRP makes code clean which speeds up testing and increases maintainability, extensibility and reusability of code
- DEMO ::

Interface Segregation Principle

- “Clients should not be forced to depend upon interfaces that they do not use.”
- One fat interface need to be split to many smaller and relevant interfaces so that client can know about the interfaces that are relevant to them
- The ISP was first used and formulated by Robert C. Martin while consulting for xerox
- Demo : Example and Exercise ::

Open Closed Principle

- “Software entities such as classes, modules, functions, etc. should be open for extension, but closed for modification”
- Any new functionality should be implemented by adding new classes, attributes and methods, instead of changing the existing ones.
- Bertrand Meyer originated the term OCP Robert C. Martin considered this as most important principle

Implementation guidelines

- The easiest way to apply OCP is to implement the new functionality on new derived classes
- Allow clients to access the original classes with abstract interface

What happens if we don’t follow OCP

- End up testing the entire functionality
- QA team need to test the entire flow
- Costly process for the organization
- Breaks the single responsibility as well
- Maintenance overheads increases on the class due to huge chuck of unorganized code
- DEMO

Liskov Substitution Principle(LSP)

- “S is a subtype of T, then objects of type T may be replaced with objects of type S”
- Derived types must be completely substitutable for their base types
- Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (Strong) behavioral subtyping
- Introduced by Barbra Liskov

Implementation Guidelines

- No new exception can be thrown by the subtype
- Clients should not know which specific subtype they are calling
- New derived classes just extend without replacing the functionality of old classes
- DEMO ::

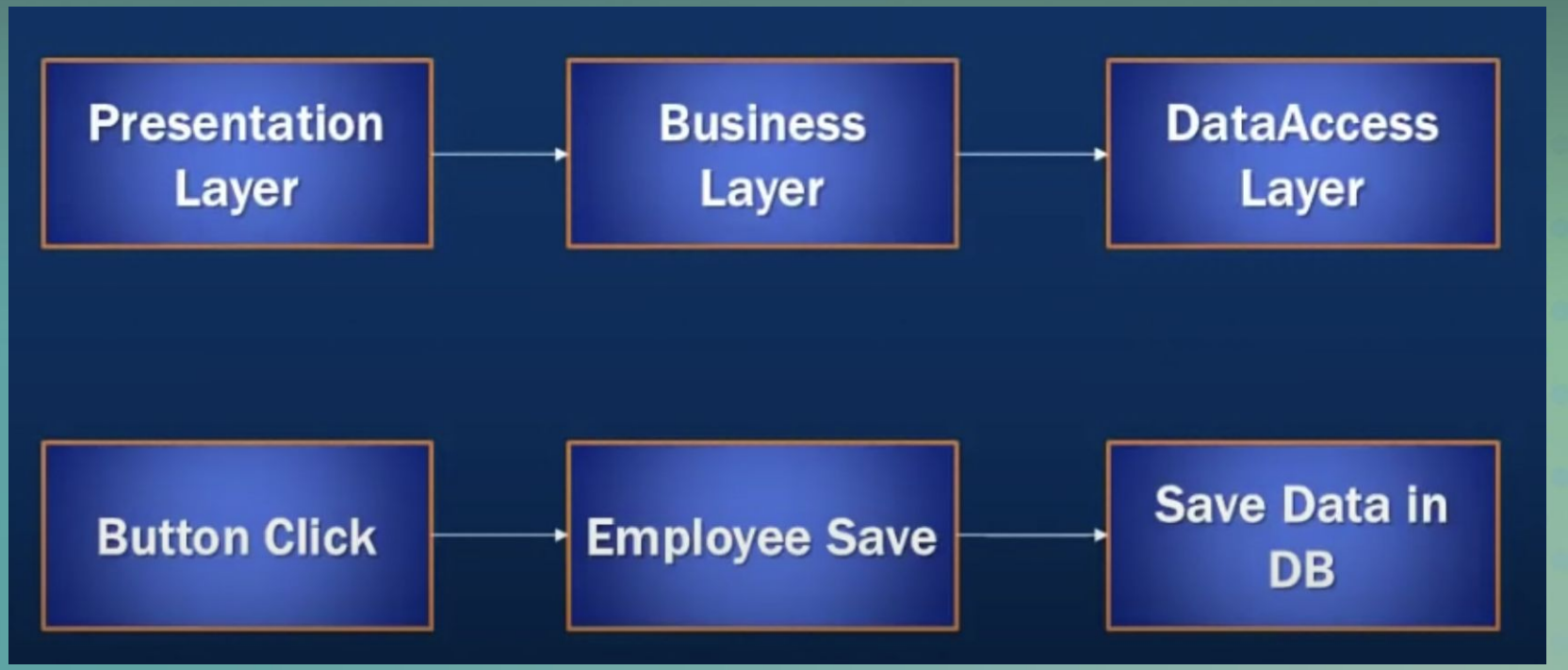
Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions

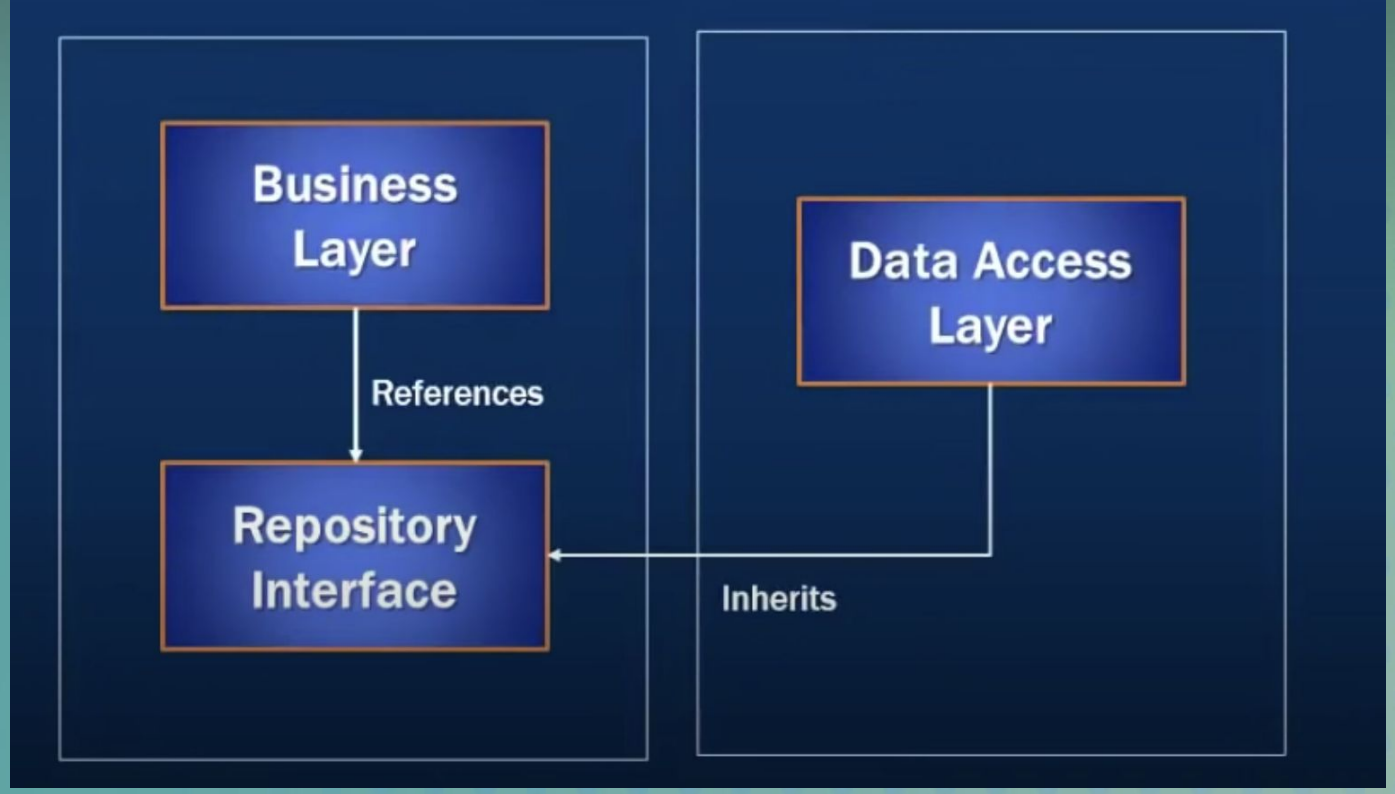
Abstraction should not depend on details. Details should depend on abstraction

Dependency Inversion Principle

- The interaction between high level and low level modules should be thought of as an abstract interaction between them
- Introduced by Robert C Martin
- Applying the dependency inversion principle can also be seen as an example of the adapter pattern (To be discussed in detail)

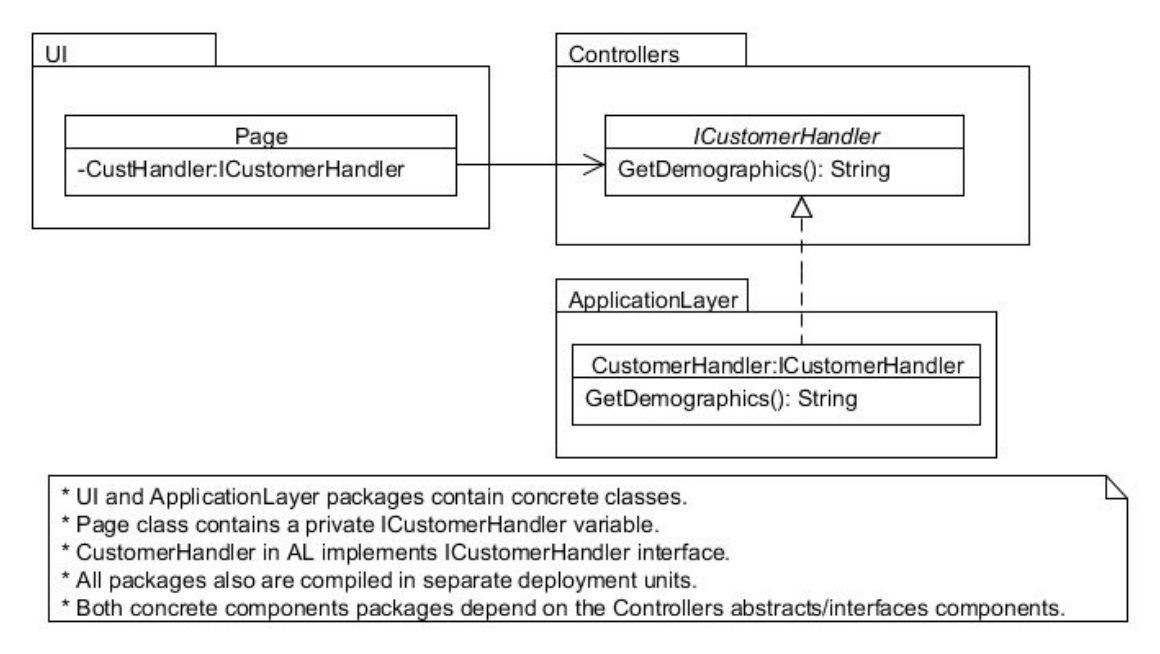


- In conventional application architecture, lower-level components (e.g., DataAccessLayer) are designed to be consumed by higher-level components (e.g., BusinessLayer or Presentation Layer) which enable increasingly complex systems to be built.
- In this composition, higher-level components depend directly upon lower-level components to achieve some task.
- This dependency upon lower-level components limits the reuse opportunities of the higher-level components.



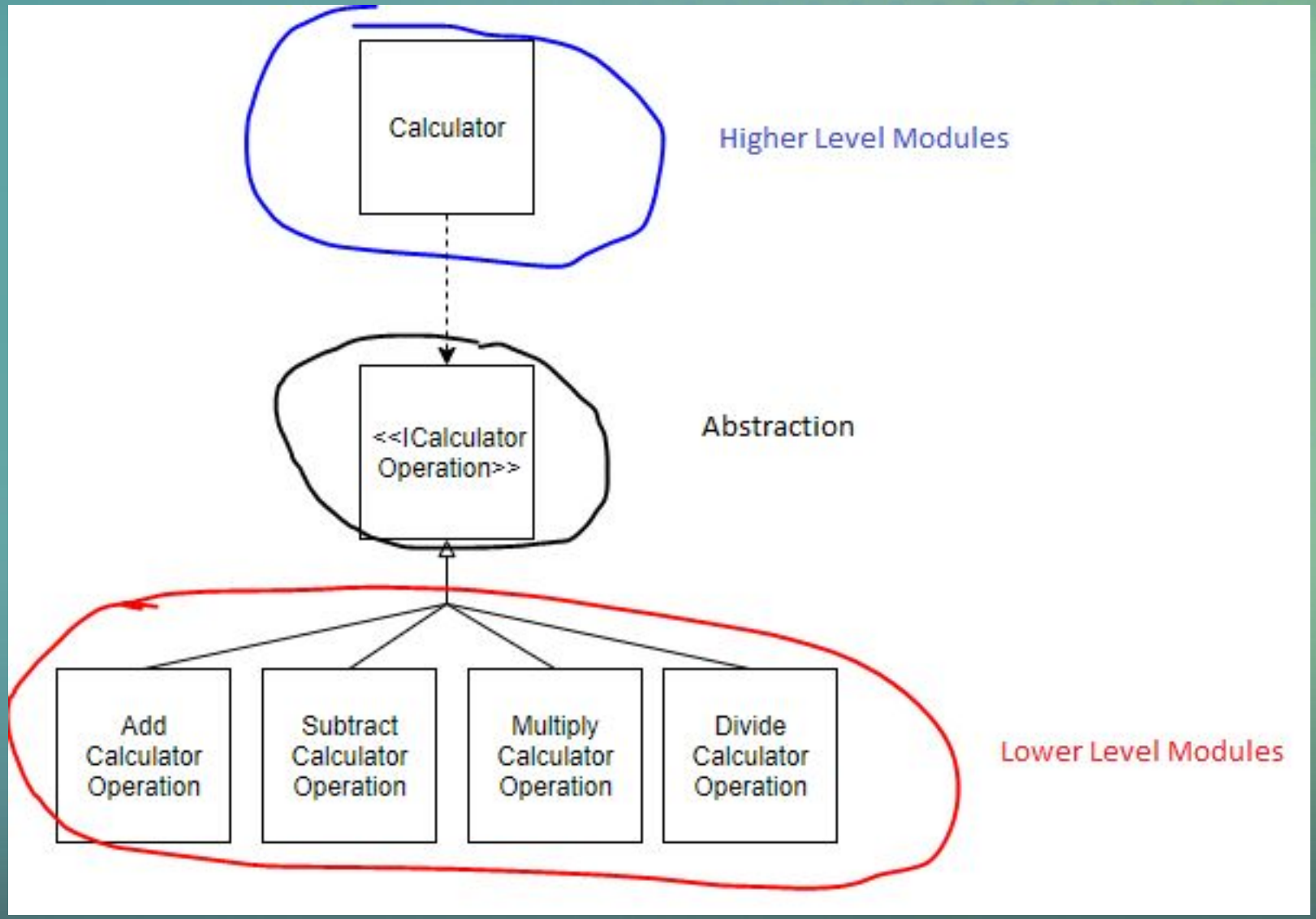
- With the addition of an abstract layer, both high and lower-level layers reduce the traditional dependencies from top to bottom.
- Nevertheless, the "inversion" concept does not mean that lower-level layers depend on higher-level layers directly.
- Both layers should depend on abstractions (interfaces) that expose the behavior needed by higher-level layers.

Model–view–controller (MVC)



- UI and ApplicationLayer packages contain mainly concrete classes.
- Controllers contains abstracts/interface types.
- UI has an instance of ICustomerHandler. All packages are physically separated.
- In the ApplicationLayer there is a concrete implementation of CustomerHandler that Page class will use.
- Since the UI doesn't reference the ApplicationLayer or any other concrete package implementing ICustomerHandler, the concrete implementation of CustomerHandler can be replaced without changing the UI class.

Exercise for DIP



SKILL LYNC WEBINARS

DESIGN PATTERNS

CREATIONAL DESIGN PATTERN

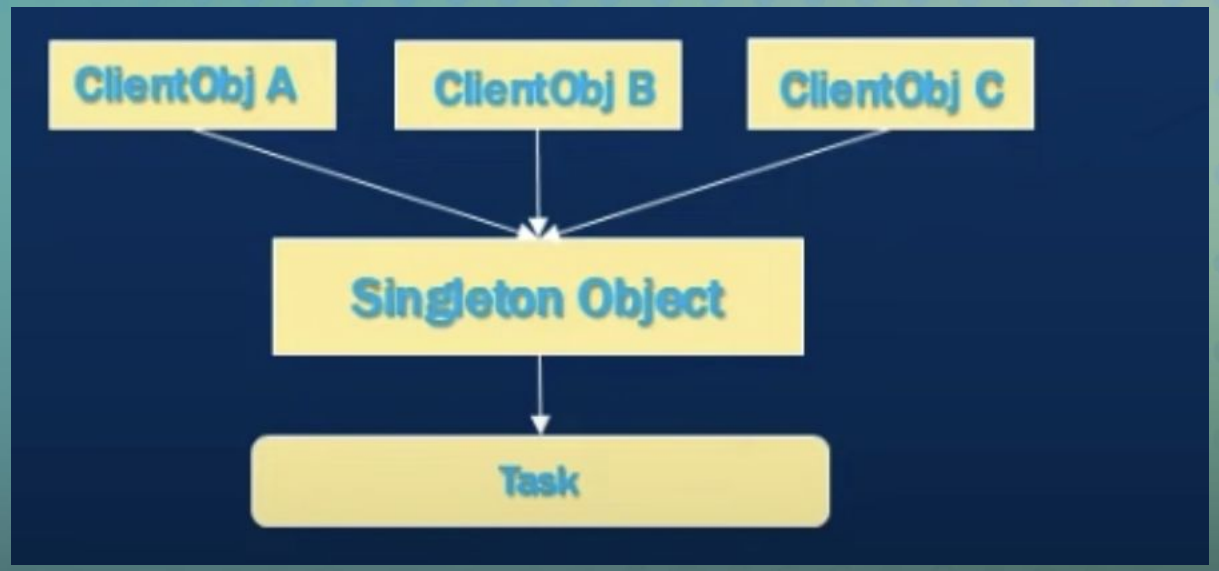
Creational Design Patterns:

- Builder
- Factories
  - Factory Method
  - Abstract Factory
- Prototype
- Singleton

- Creational design patterns deals with object creation mechanisms
- Creating objects in a manner suitable to the situation.
- The basic form of object creation could result in design problems or in added complexity to the design.

Singleton Design Pattern

- Creational Design pattern
- Only one instance of the class should exist
- Other classes should be able to get instance of singleton class
- Used in Logging, Cache, Session, Drivers



Singleton Design Patter - Implementation

- Declaring all constructors of the class to be private
- providing Public static method that returns a reference to the instance
- The instance is stored as private static variable.

Follow -Ups

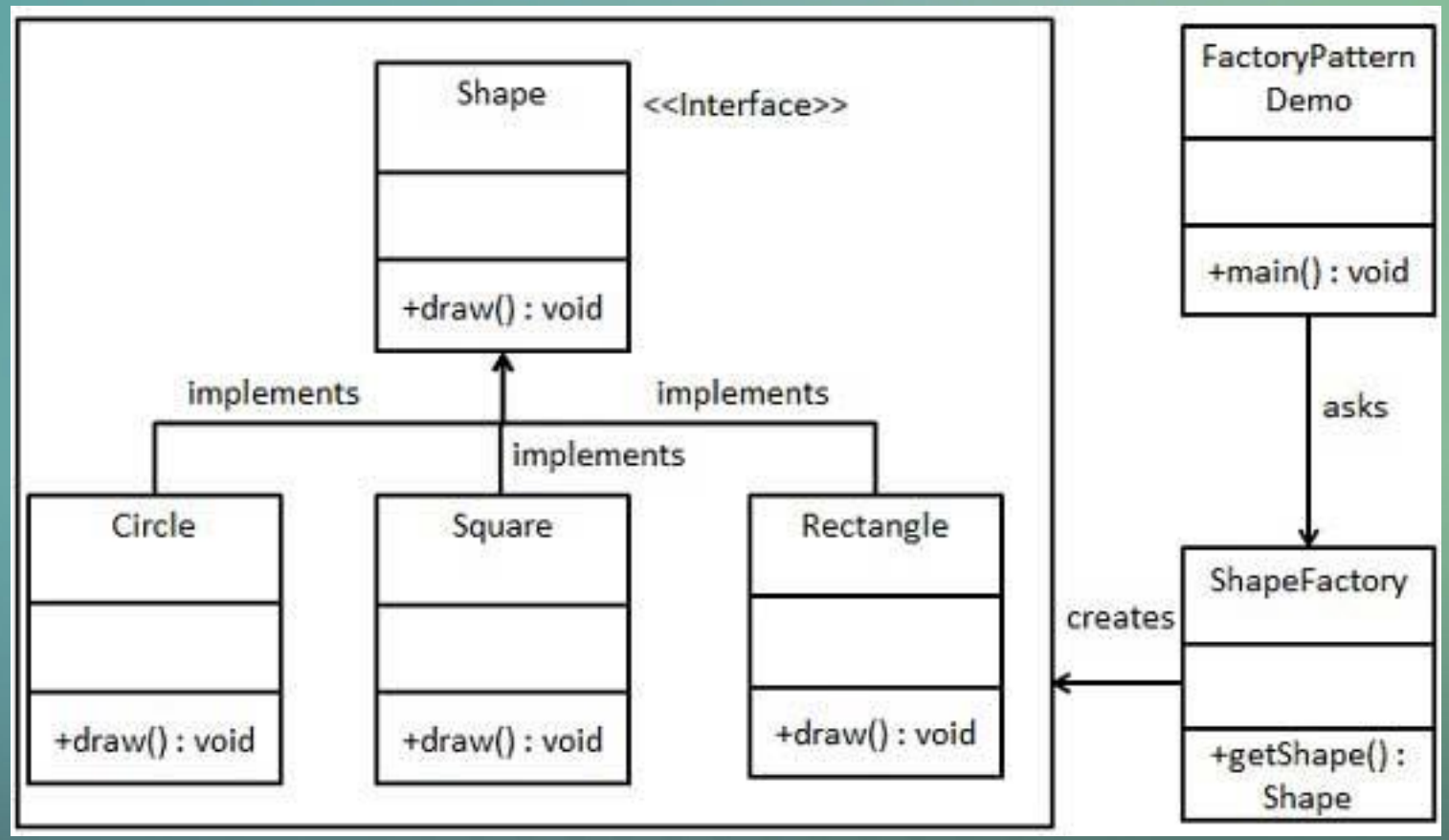
- Why singleton class needs to be sealed?
- Thread Safety in singleton classes
  - Ensure double Lock for object creation
    - https://en.wikipedia.org/wiki/Double-checked_locking
  - Explained in demo

Factory Design Pattern

- Factory pattern is one of the most used design patterns.
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implement Factory Design Pattern

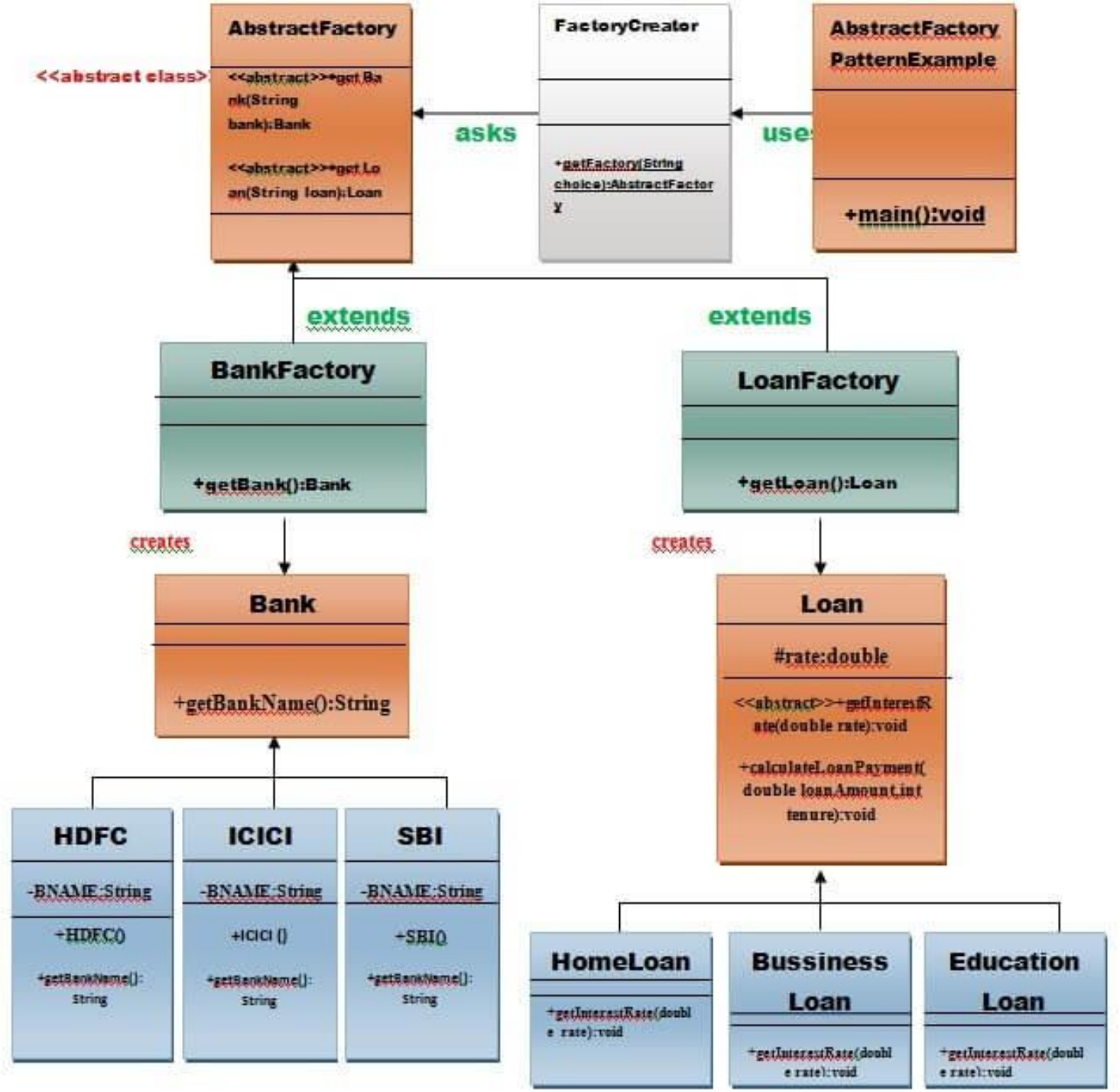
UML diagram



Abstract Factory design pattern

- Abstract Factory design pattern is one of the Creational pattern this pattern is almost similar to Factory Pattern
- Abstract factory pattern is considered as another layer of abstraction over factory pattern.
- Abstract Factory patterns work around a super-factory which creates other factories.
- Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern.
- At runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

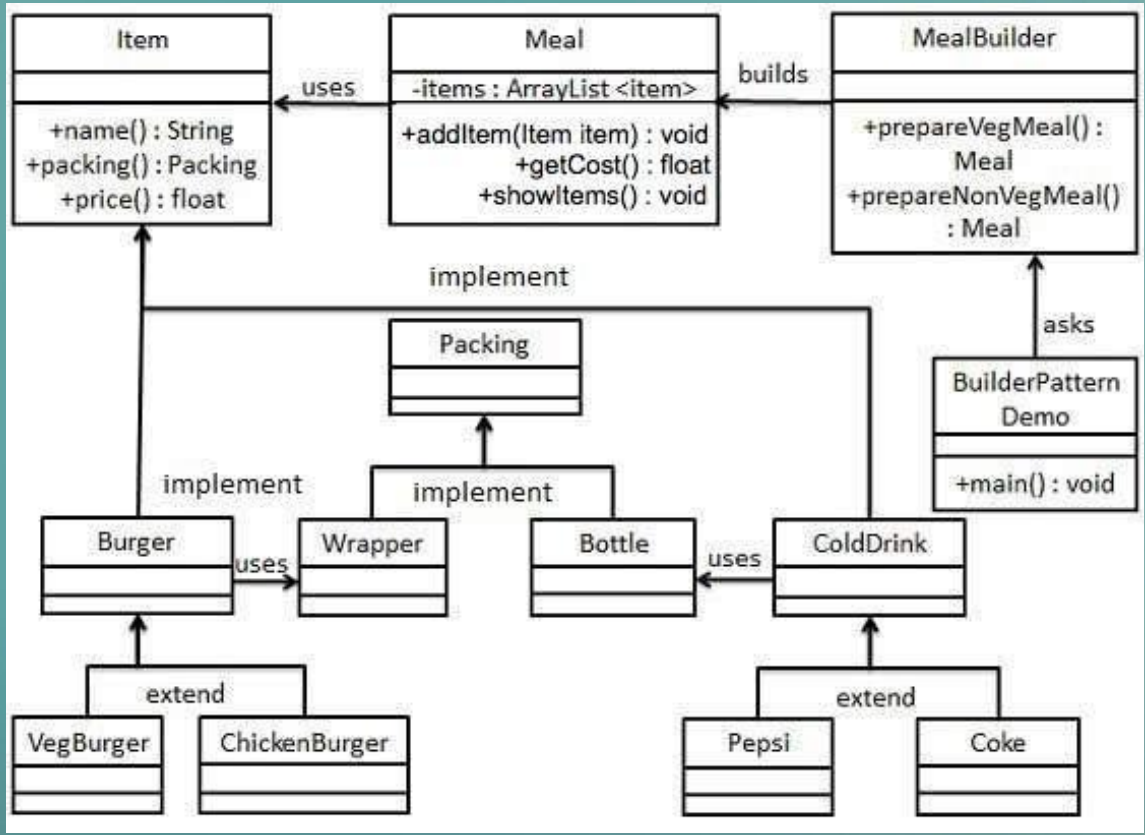
UML Diagram : Write code



Builder Design Pattern

- Builder Pattern builds a complex object from simple objects using step-by-step approach”
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- The main advantages of Builder Pattern are as follows:
  - It provides clear separation between the construction and representation of an object.
  - It provides better control over construction process.
  - It supports to change the internal representation of objects.

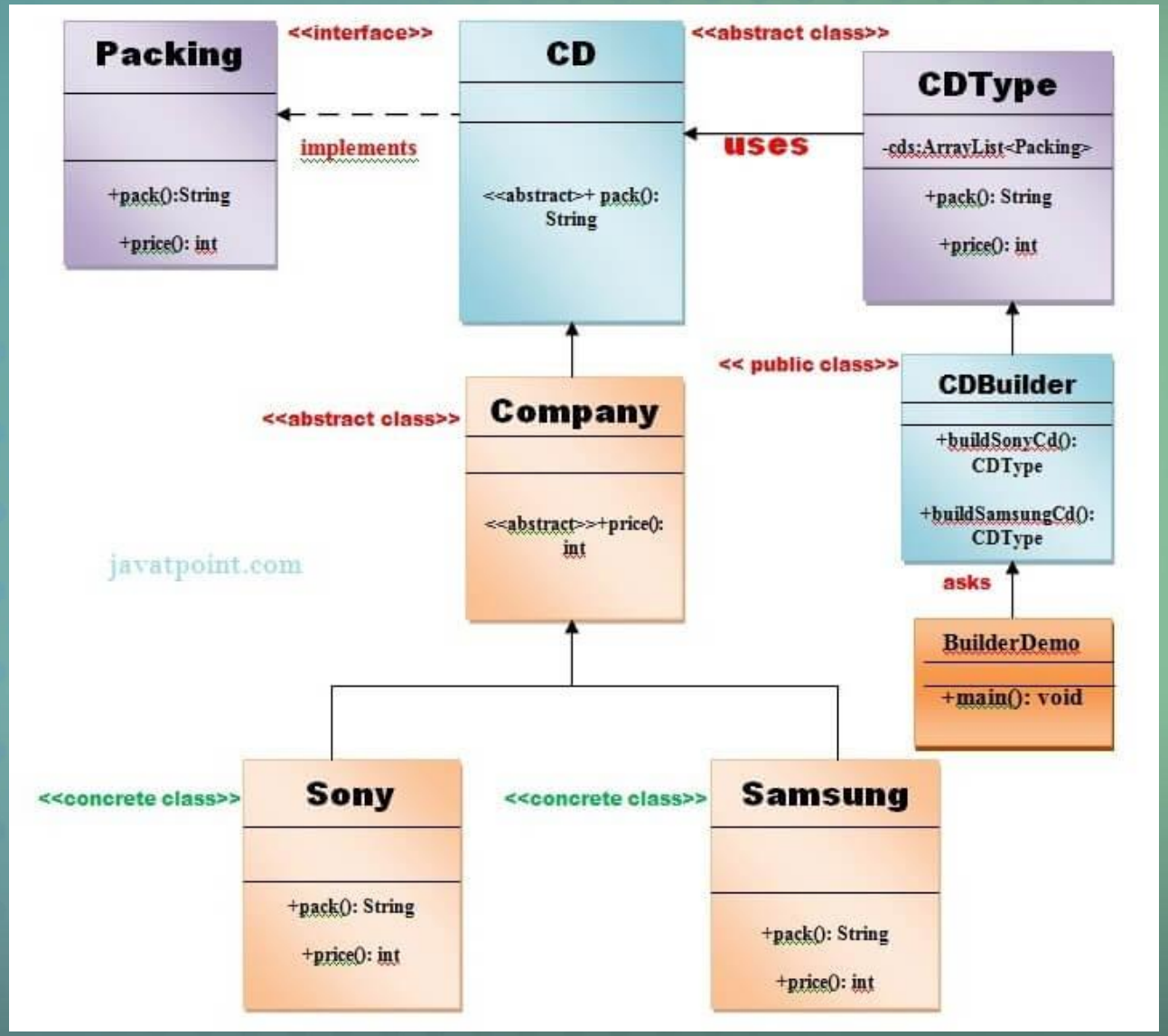
UML Diagram ::



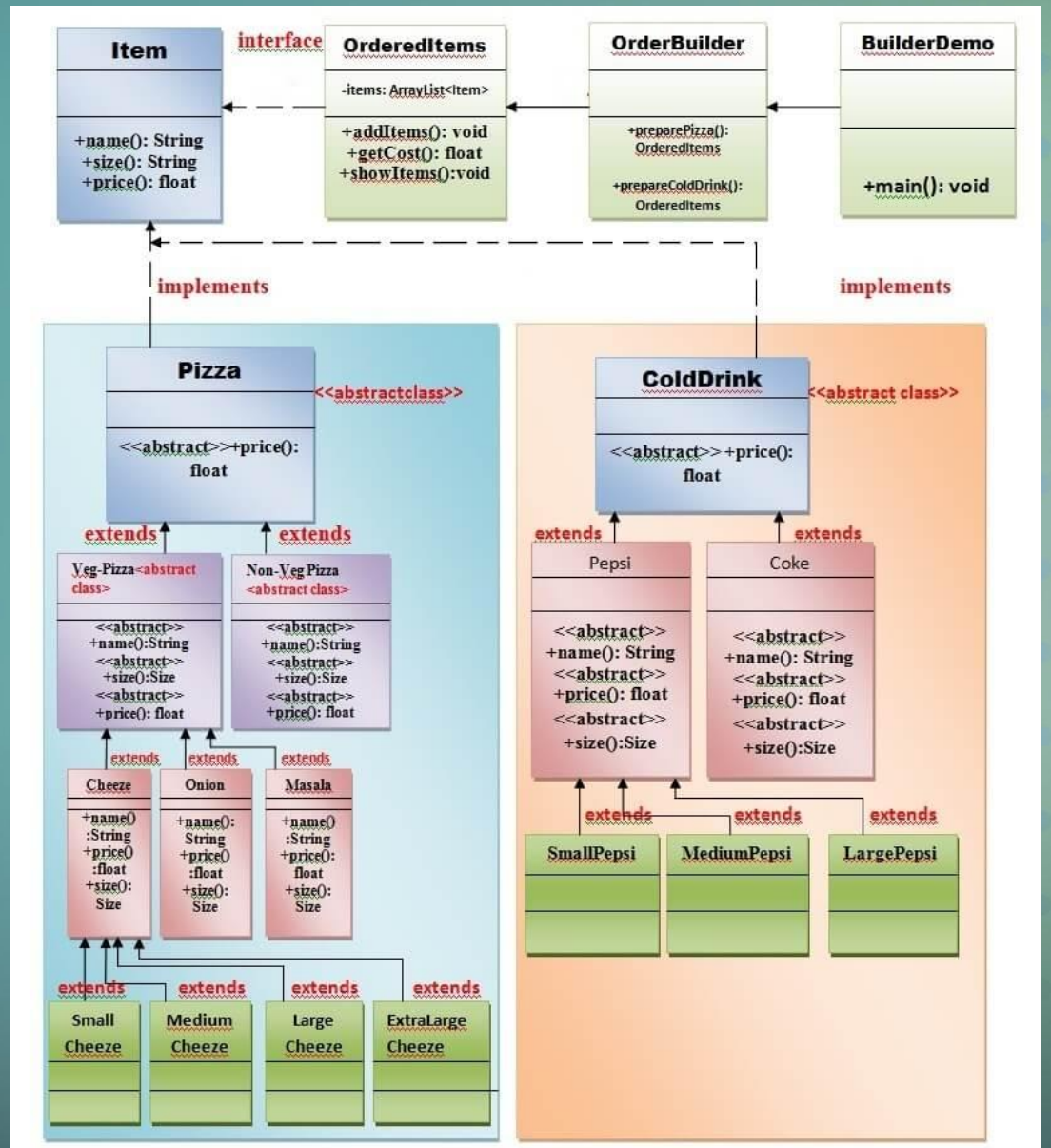
Consider business case of fast-food restaurant a typical meal could be a Burger and cold drink. Burger could be either a veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or Pespi and will be packed in a bottle.

NOTE : UML Diagram isn't perfect, there may be better design possible, please re-iterate before implementing

Question 1 : Practice



Question 2 : Practice



Prototype Design Pattern

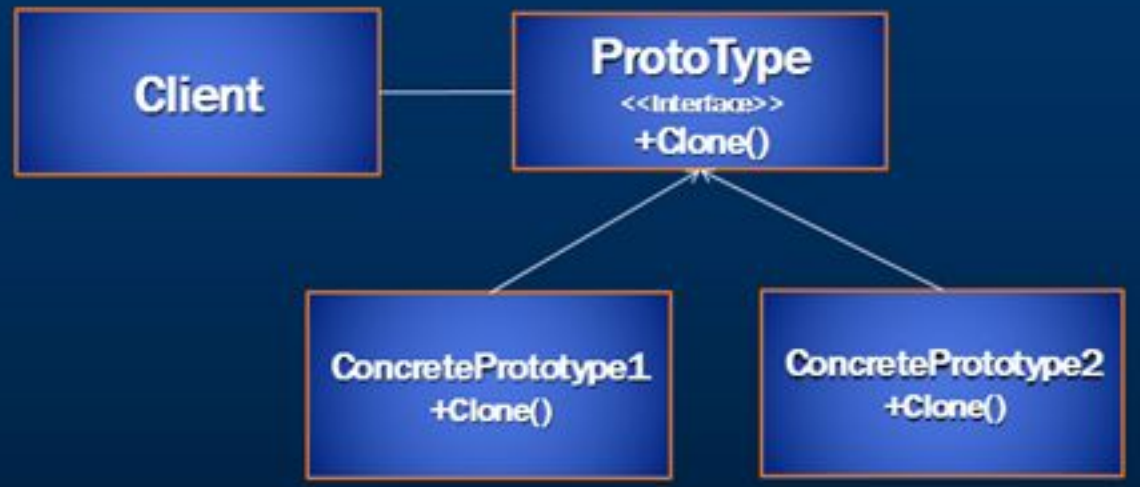
- "Prototype Design Pattern Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype”
- To simplify, instead of creating object from scratch every time, you can make copies of an original instance and modify it as required.
- Prototype is unique among the other creational patterns as it doesn't require a class but only an end object.

Why Prototype Design Pattern is required?

- Creating an object is an expensive operation and in many cases it is more efficient to copy an object.
- For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Prototype Representation

Prototype Design Pattern

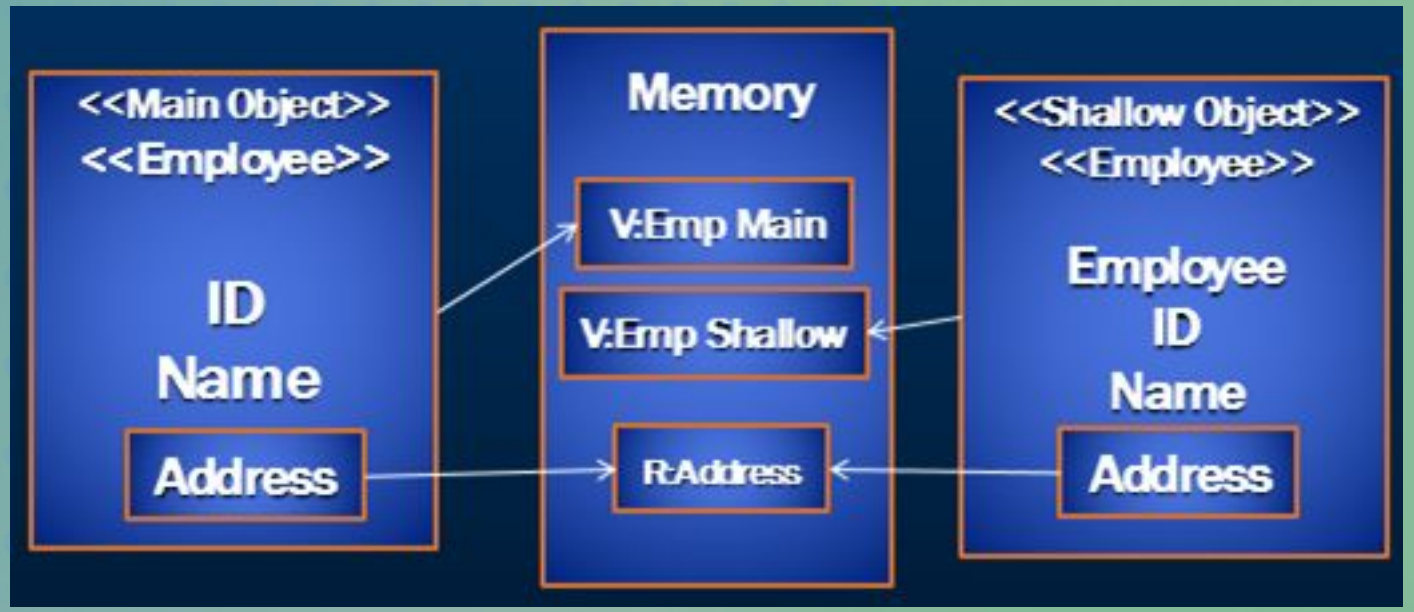


Shallow And Deep Copy

- **Shallow and Deep Copy :**
  The idea of using copy is to create a new object of the same type without knowing the exact type of the object we are invoking.

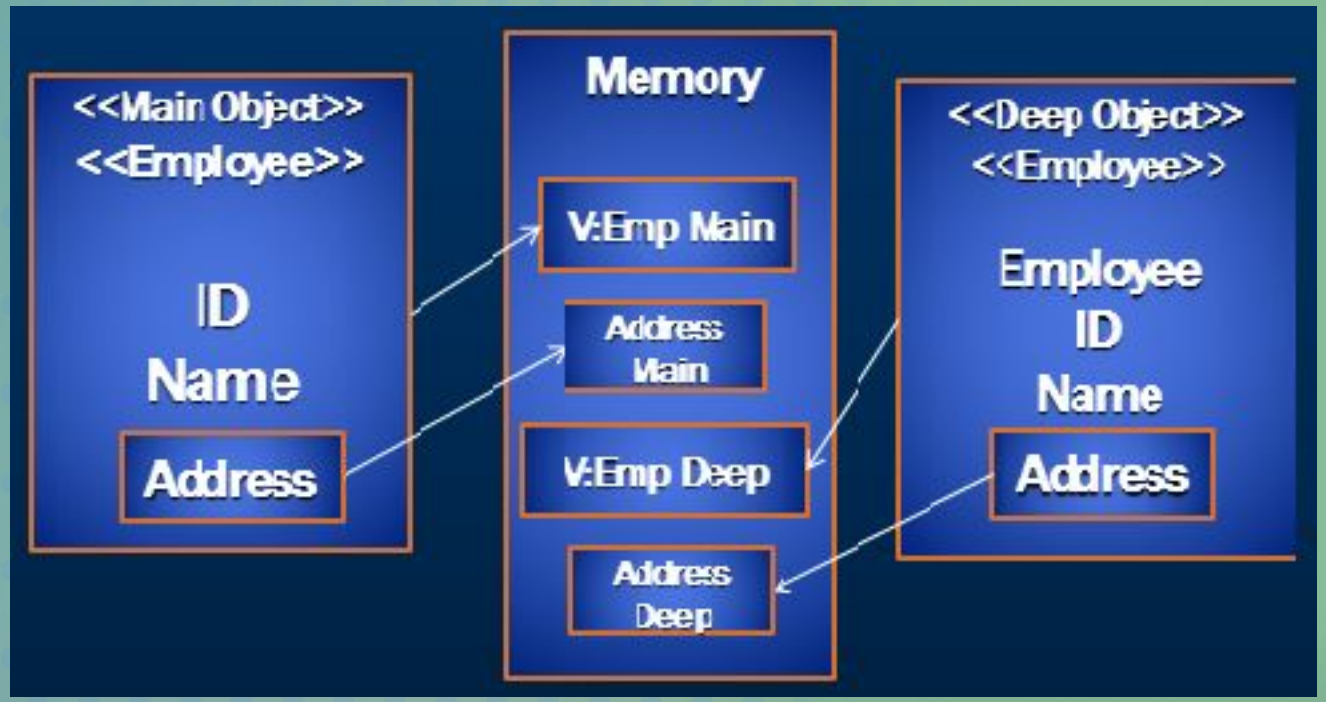
Shallow Copy

- Copies an object's value type fields into the target object and the object's reference types are copied as references into the target object.

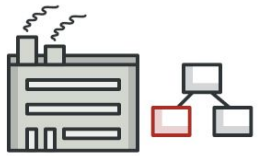


Deep Copy

- Deep Copy copies an object's value and reference types into a complete new copy of the target objects.

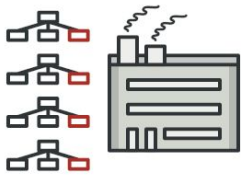


Summary of Creational Design Patterns



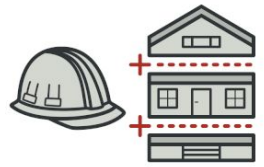
Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



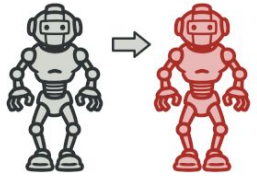
Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.



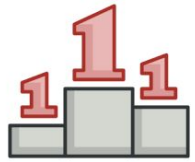
Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.



Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Structrural Design Patterns:

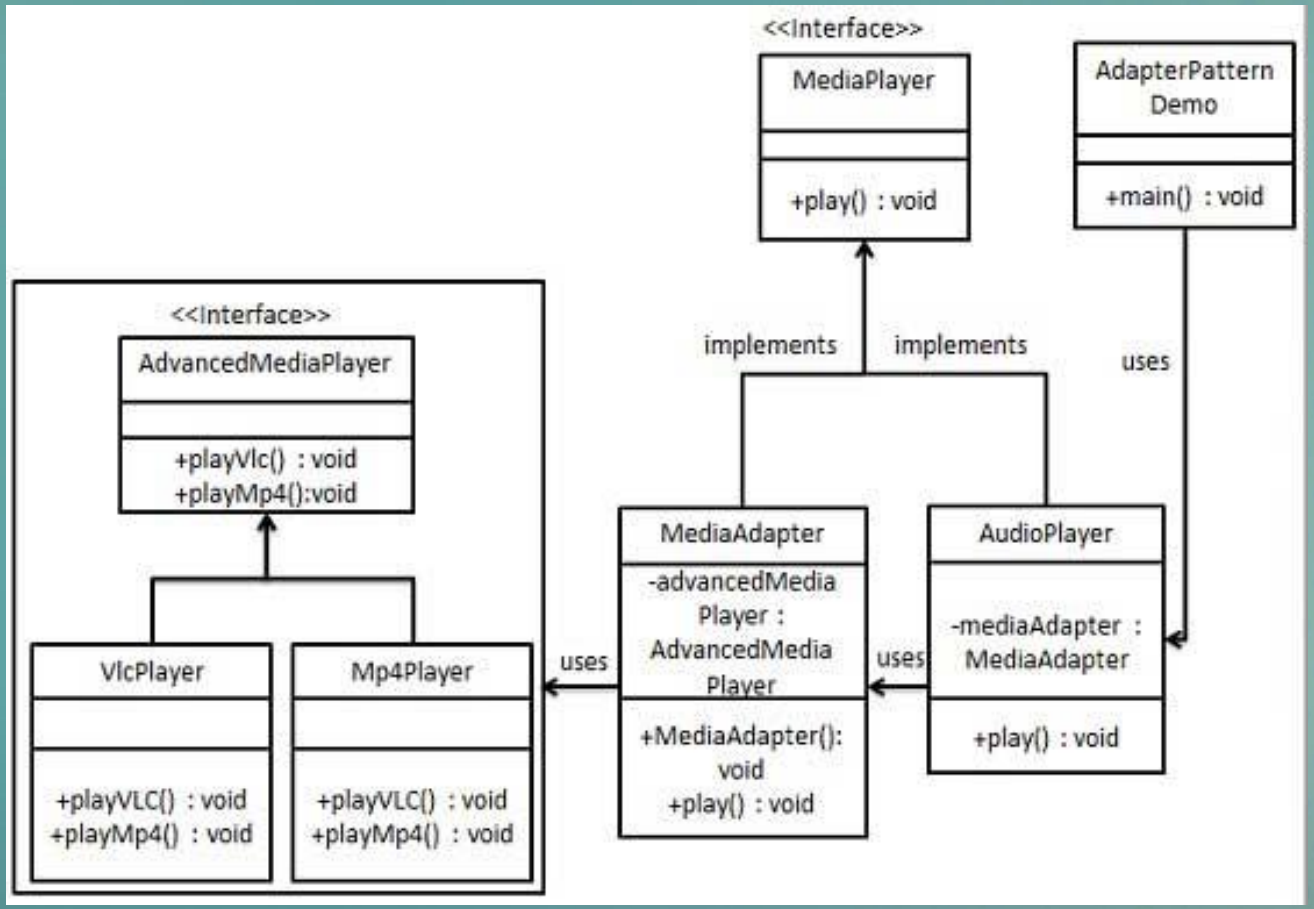
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Adapter Pattern

- Adapter pattern works as a bridge between two incompatible interfaces.
- This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Implement Code from UML



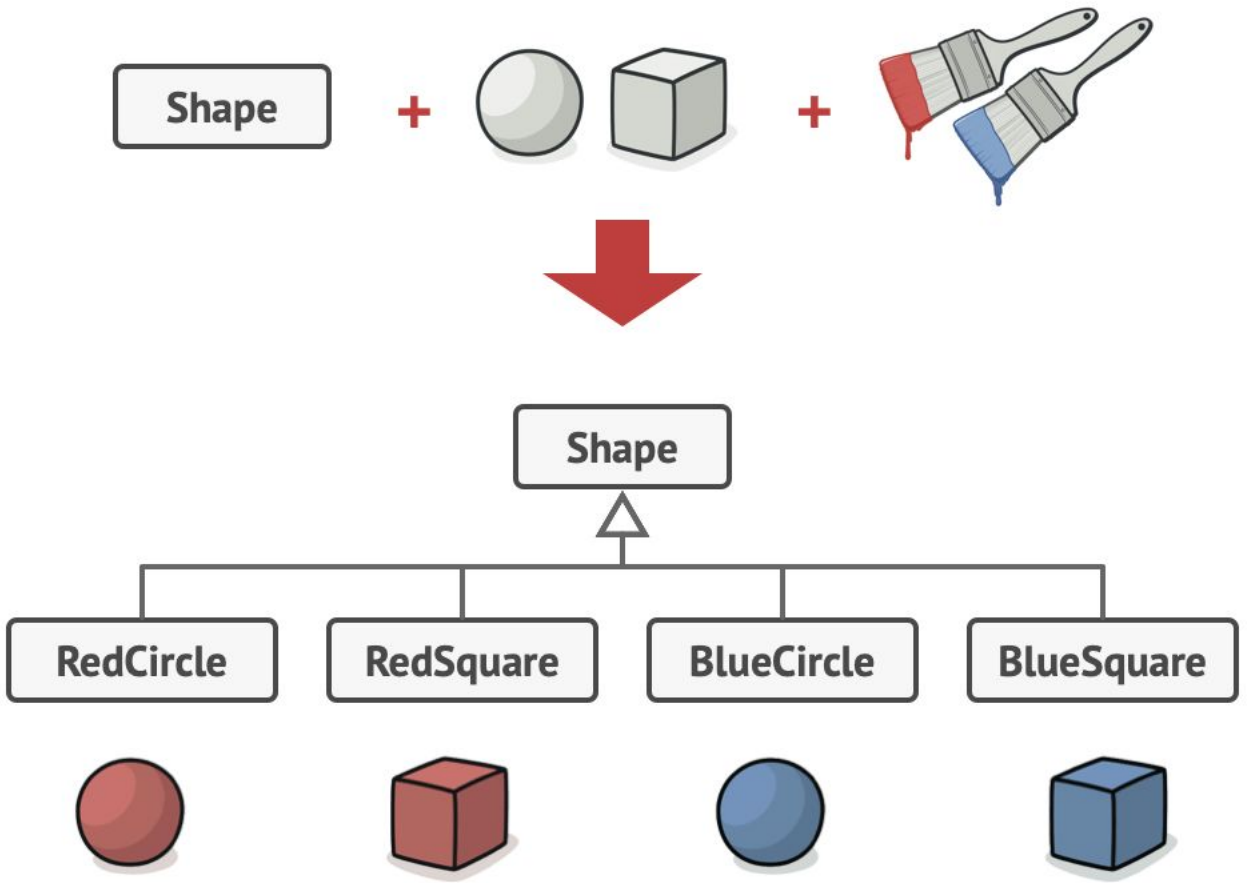
- We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.
- We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can 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.

NOTE : UML Diagram isn't perfect, there may be better design possible, please re-iterate before implementing

Bridge Pattern

- Bridge is a structural design pattern that lets us split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

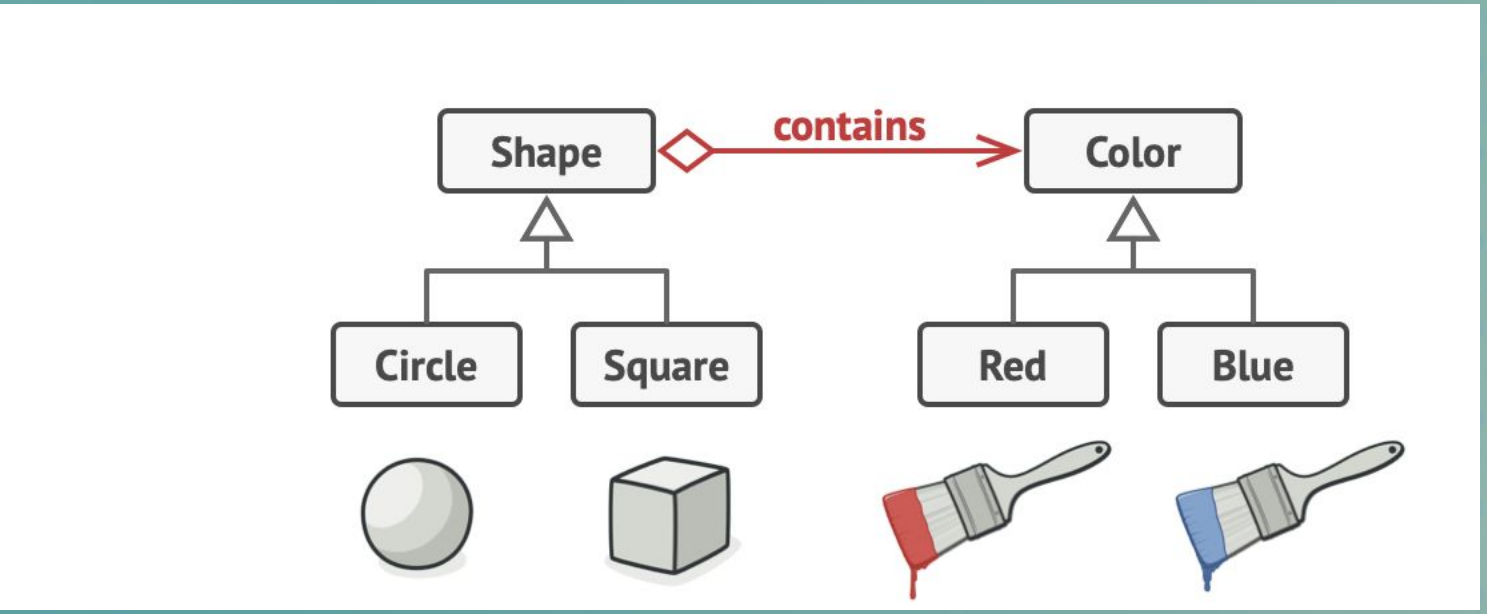
Problem



Number of class combinations grows in geometric progression.

- Given geometric Shape class with a pair of subclasses: Circle and Square.
- We want to extend this class hierarchy to incorporate colors
- We can create Red and Blue shape subclasses.
- However, as we already have two subclasses, We'll need to create four class combinations such as BlueCircle, RedSquare etc
- Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape we would need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

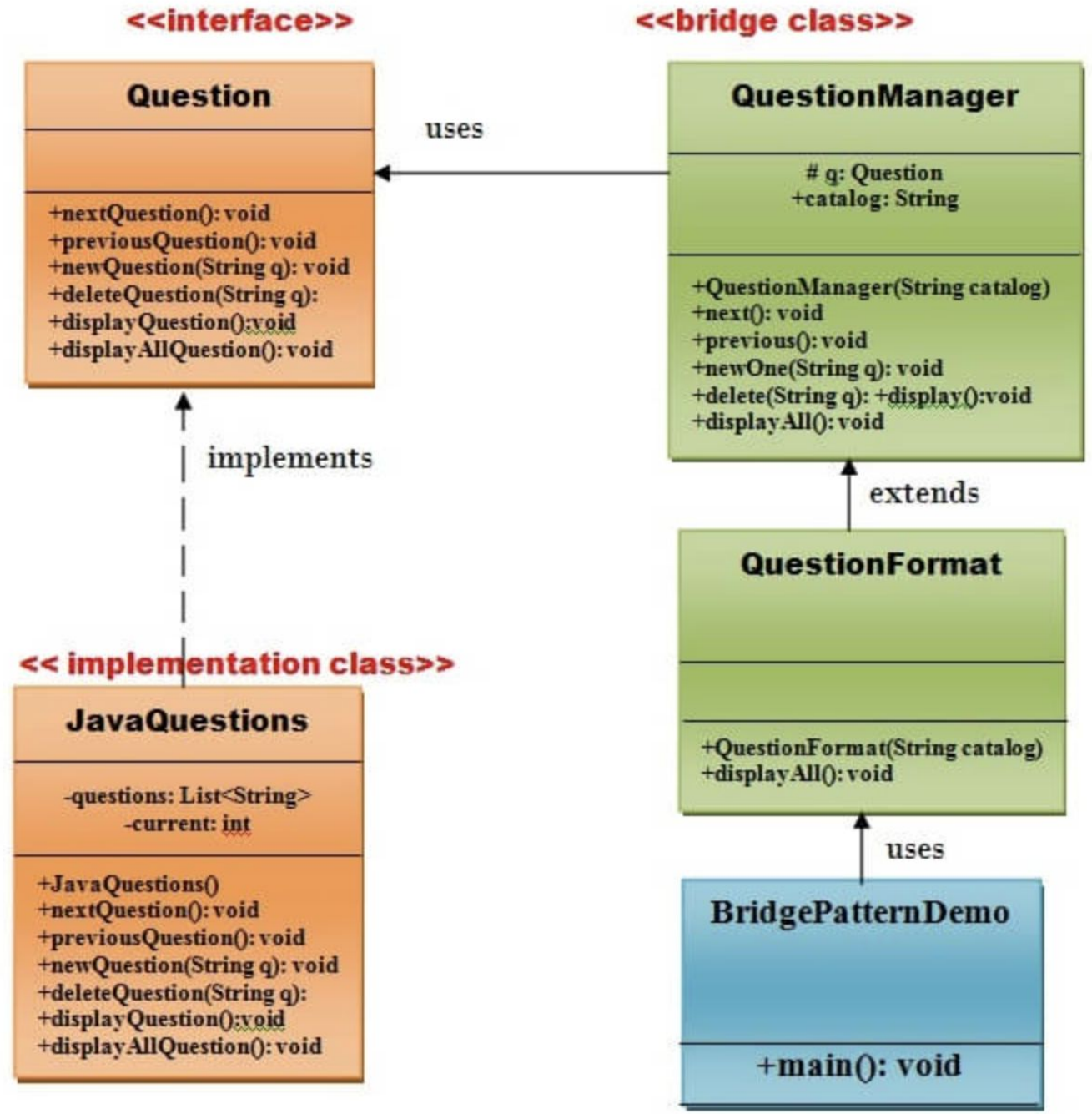
Solution



- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that we extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.

Create UML Diagram and develop code

Code from UML Diagram ::

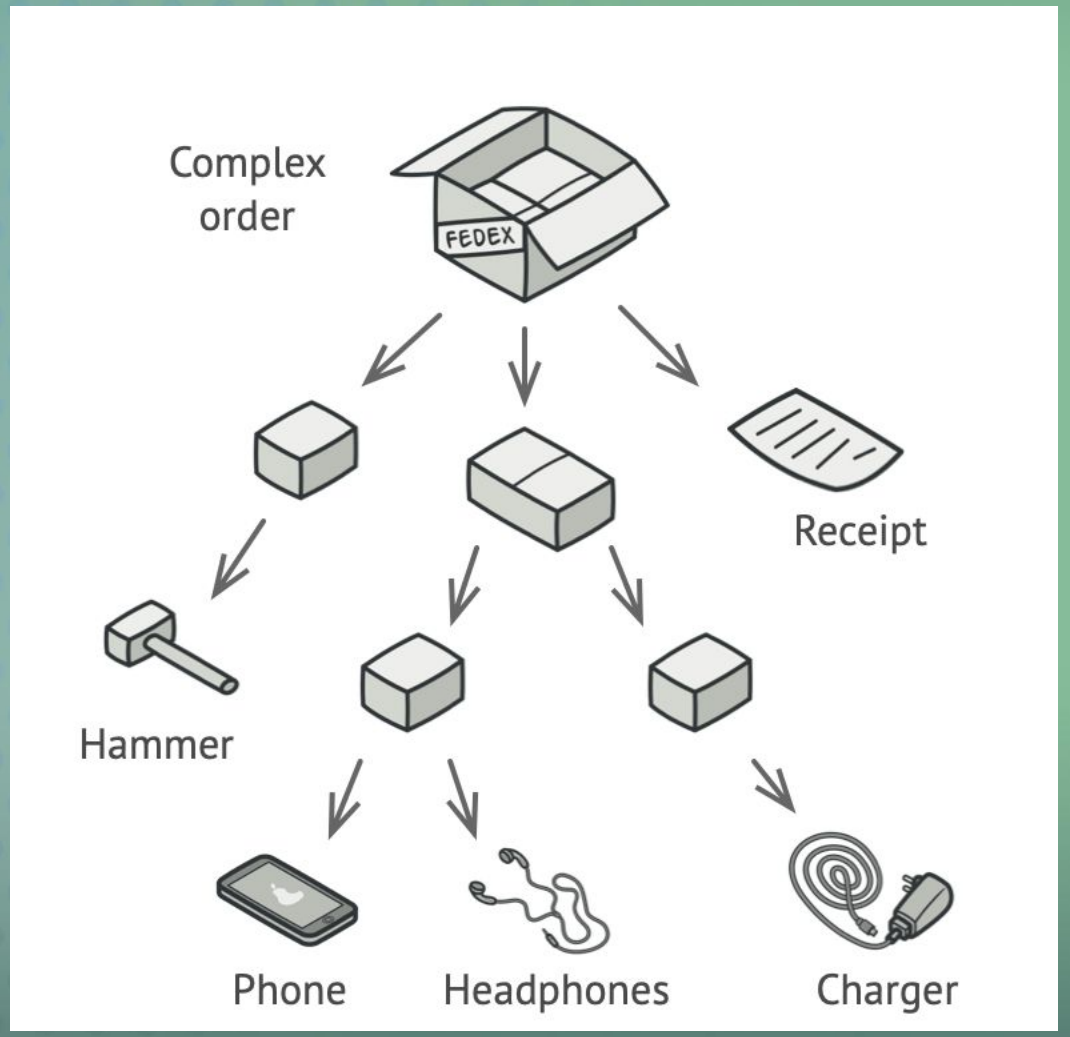


Composite Pattern

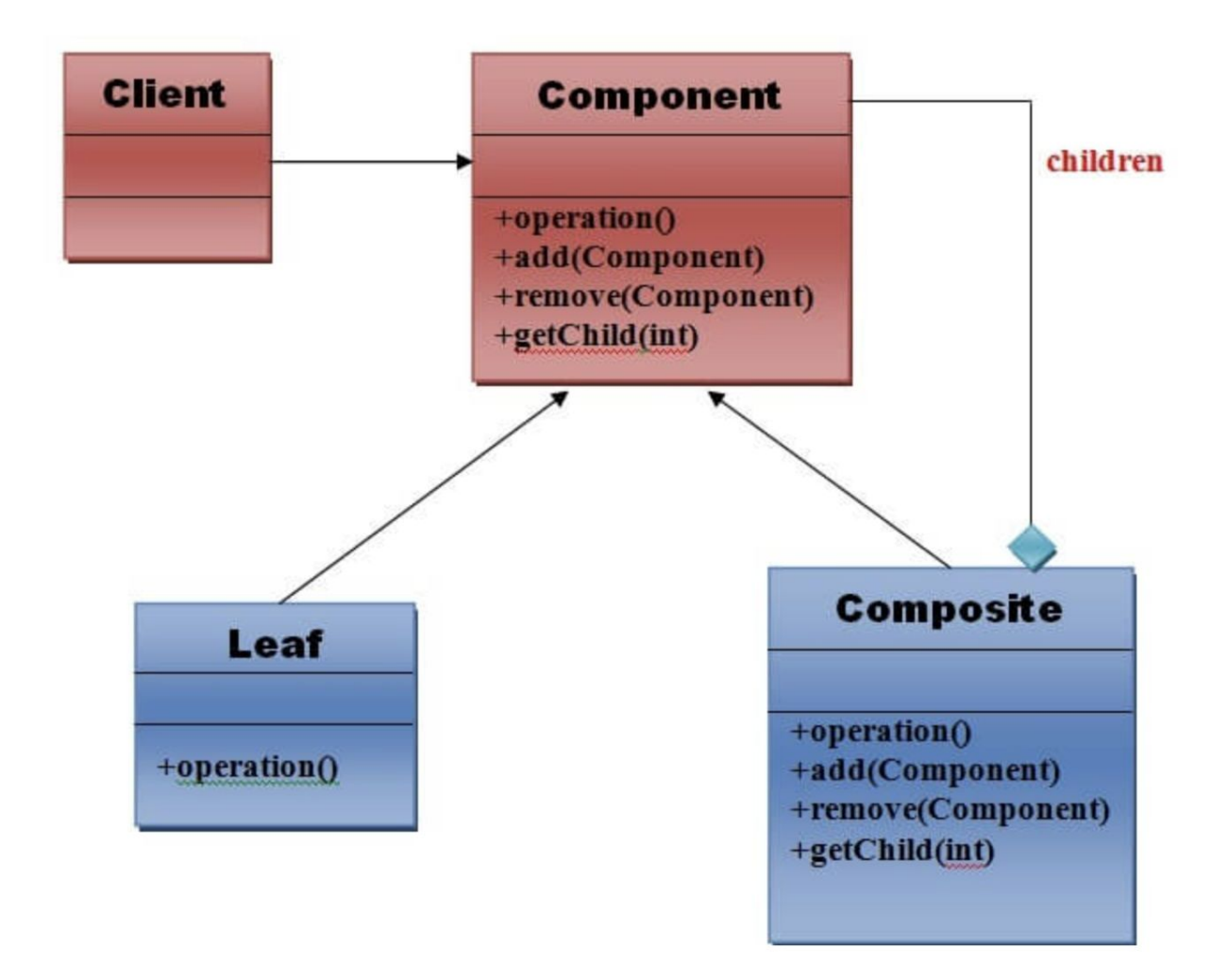
- Composite pattern is used where we need to treat a group of objects in similar way as a single object.
- Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy.
- This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

Problem Statement :

- Using the Composite pattern makes sense only when the core model of our application can be represented as a tree.
- For example, imagine we have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- We decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would We determine the total price of such an order?

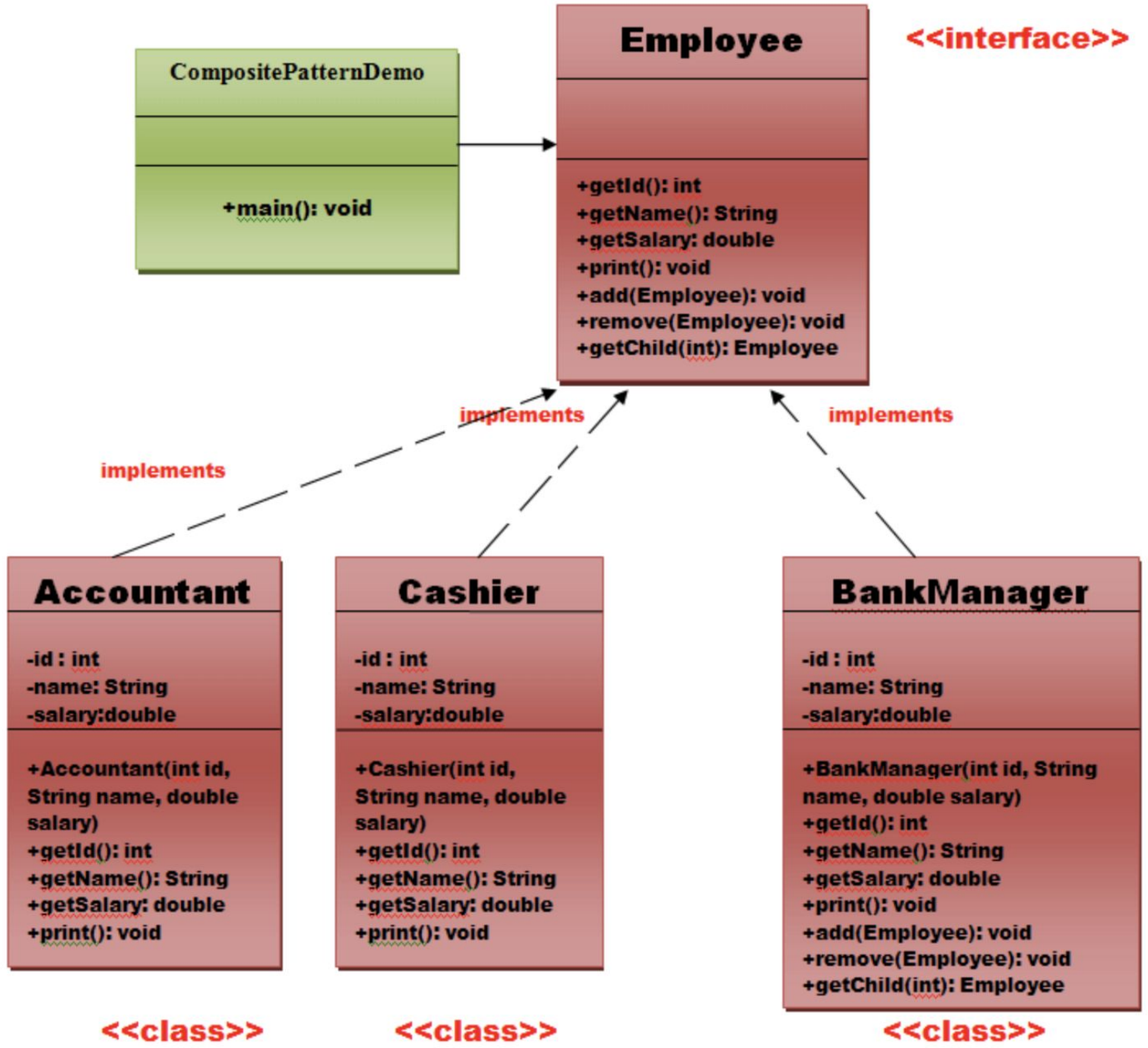


UML for Composite pattern



1. Component
   - Declares interface for objects in composition.
   - Implements default behavior for the interface common to all classes as appropriate.
   - Declares an interface for accessing and managing its child components.
2. Leaf
   - Represents leaf objects in composition. A leaf has no children.
   - Defines behavior for primitive objects in the composition.
3. Composite
   - Defines behavior for components having children.
   - Stores child component.
   - Implements child related operations in the component interface.
4. Client
   - Manipulates objects in the composition through the component interface.

Code from UML Diagram ::

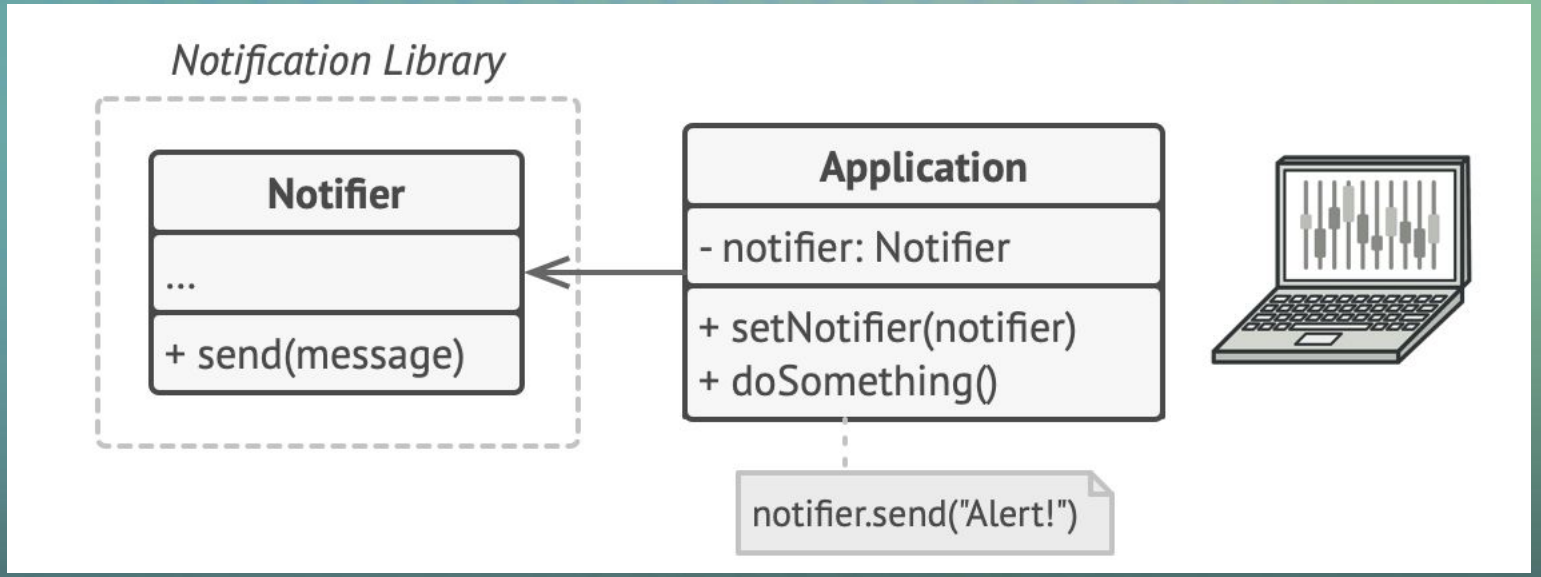


Decorator Design Pattern

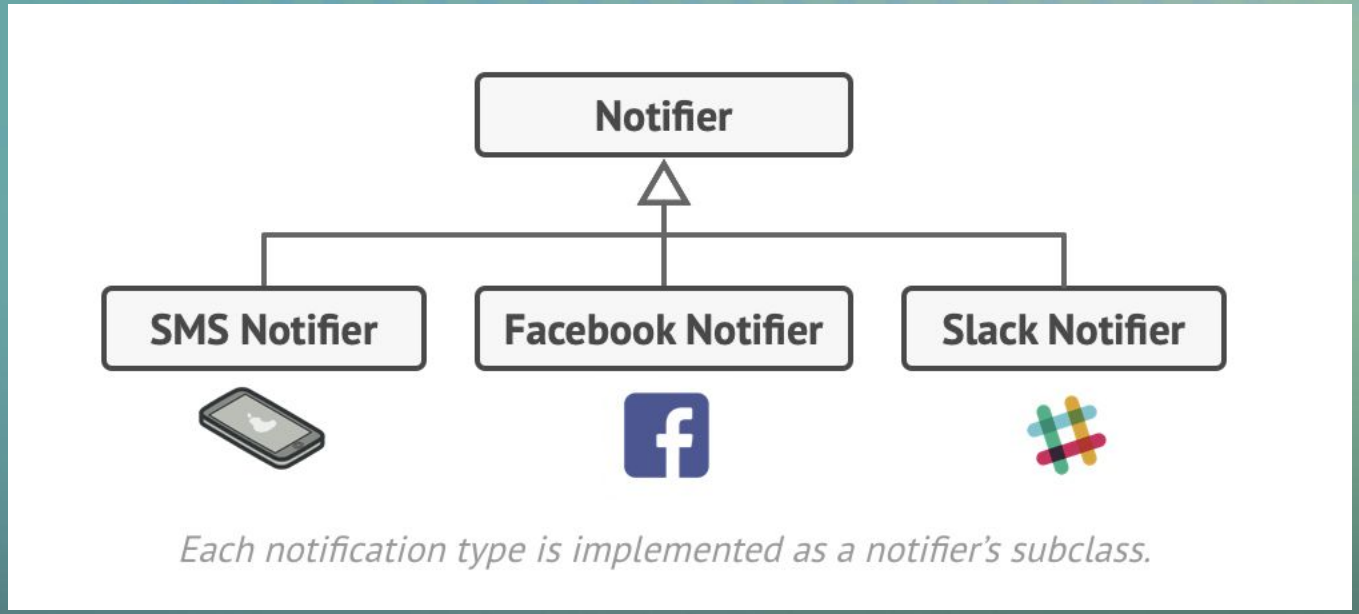
- Decorator pattern allows user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as
- This pattern acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping existing class methods signature intact.

Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.
- The initial version of the library was based on the Notifier class that had only a few fields, a constructor and a single send method.



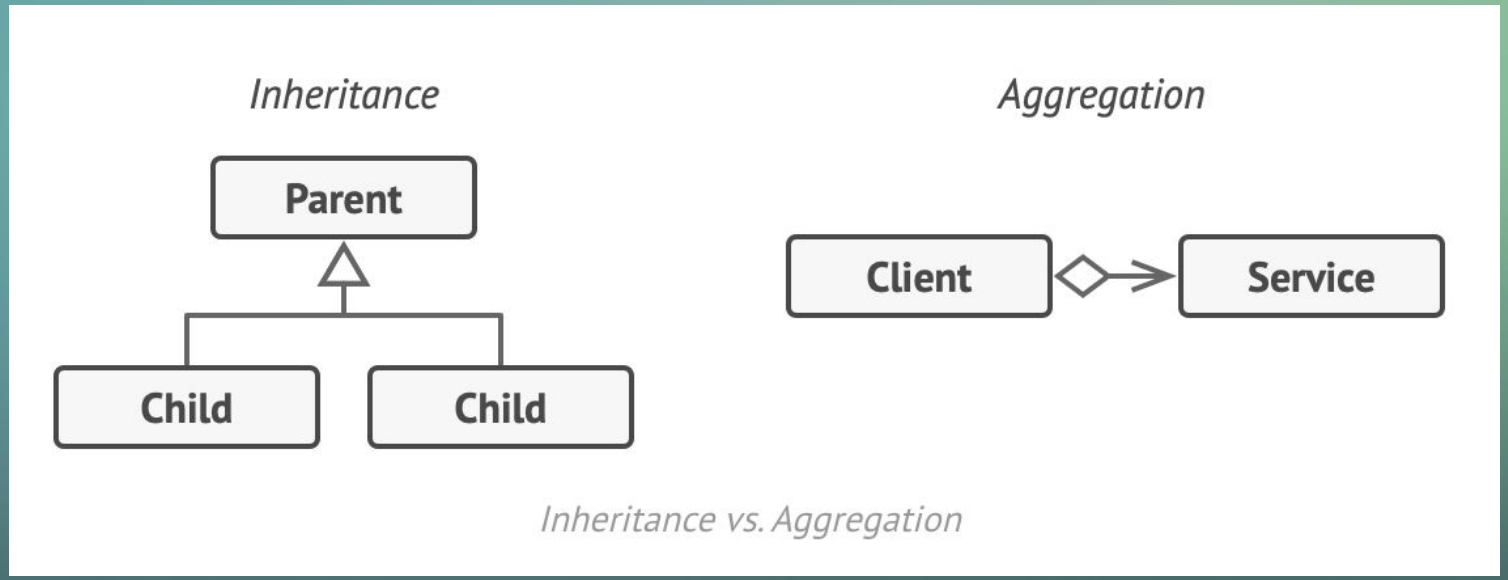
At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.



Each notification type is implemented as a notifier's subclass.

Solution

- Extending a class is the first thing that comes to mind when we need to alter an object's behavior. However, inheritance has several serious caveats that we need to be aware of.
  - Inheritance is static. We can't alter the behavior of an existing object at runtime. We can only replace the whole object with another one that's created from a different subclass.
  - Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.
- One of the ways to overcome these caveats is by using Aggregation or Composition instead of Inheritance.



Inheritance vs. Aggregation

Beverage Decorator ::::

Problem Statement :

Design a Coffee ordering system ->

This Coffee Ordering System has multiple types of Coffees, like, House Blend, Espresso. Along with the coffee there are condiments as well, like, Mocha, Milk, Soy and Whipped Milk.

This means that there can be Espresso with Streamed Milk and Mocha or House Blend with soy and whip. Actually, the number of Beverages that this system can make is pretty high which is equal to the combination of each of the coffee type and condiments available.

The problem is to add all these types of beverages into the Coffee ordering system so that it can take orders for all the available beverages.

Approach 1

Well, the very first solution that comes to the mind is to create classes of all these different types of beverages.

We'll create an abstract class Beverage and the classes HouseBlend, DarkRoast, HouseBlendWithMochaAndSteamedMilk will implement it.
The number of classes that can be created depending upon the combination of the Coffee and the condiments is pretty huge..

Code Snippets for approach1


```
// Super class of all the beverages.
abstract class Beverage {

private String description;

    Beverage() {
        description = "Sample Beverage.";
    }

    public String getDescription() {
        return this.description;
    }

    // this method is abstract because the different types of beverages will have different cost
    public abstract double cost();
}
```

```
class HouseBlend extends Beverage {

    private String description;

    HouseBlend() {
        this.description = "House Blend";
    }

    public double cost() {
        return 10.5;
    }
}
```

```
class DarkRoast extends Beverage {

    private String description;

    DarkRoast() {
        this.description = "Dark Roast";
    }

    public double cost() {
        return 5.6;
    }
}
```

```
class HouseBlendWithMochaAndStreamedMilk extends Beverage {

    private String description;

    HouseBlendWithMochaAndStreamedMilk() {
        this.description = "House Blend With Mocha And Streamed Milk";
    }

    @Override
    public String getDescription() {
        return this.description;
    }

    @Override
    public double cost() {
        return 11.45;
    }
}
```

Issues with this approach

- There are lot of classes which means that there has been a Class Explosion.
- If we require to add a new coffee type or a new condiment type then we'll have to create multiple classes representing all the different types of combinations.
- This system is a Maintenance Nightmare.

Approach 2 : Bringing the number of classes down

It can be observed that there are 4 condiments and it’s just their availability status that is contributing to this class explosion. We can try to bring down the number of classes available by adding the condiment availability into the super class, Beverage.

Code Snippet for approach2


```
class Beverage {
    private boolean milk;
    private boolean soy;
    private boolean mocha;
    private boolean whip;

    void setMilk() {
        this.milk = true;
    }
    void setSoy() {
        this.soy = true;
    }
    void setMocha() {
        this.mocha = true;
    }
    void setWhip() {
        this.whip = true;
    }
    double cost() {
        return 0D + (this.milk ? 1.5 : 0) +
                (this.soy ? 0.5 : 0) + (this.mocha ? 2 : 0) +
                (this.whip ? 2.5 : 0);
    }
}
```

```
class HouseBlend extends Beverage {

    double cost() {
        return 10.5 + super.cost();
    }
}
```

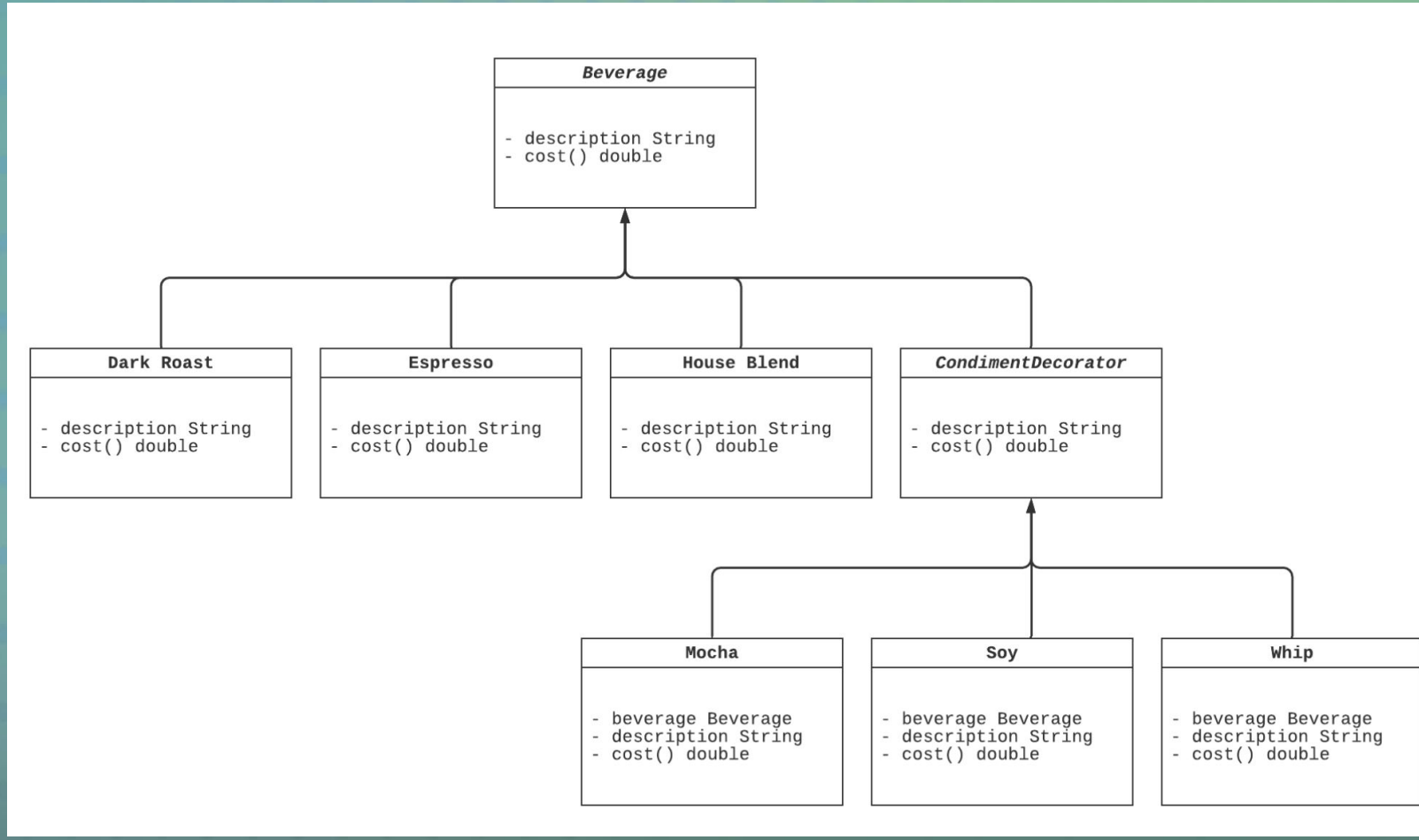
```
class DarkRoast extends Beverage {

    double cost() {
        return 5.6 + super.cost();
    }
}
```

Issues with this approach ::

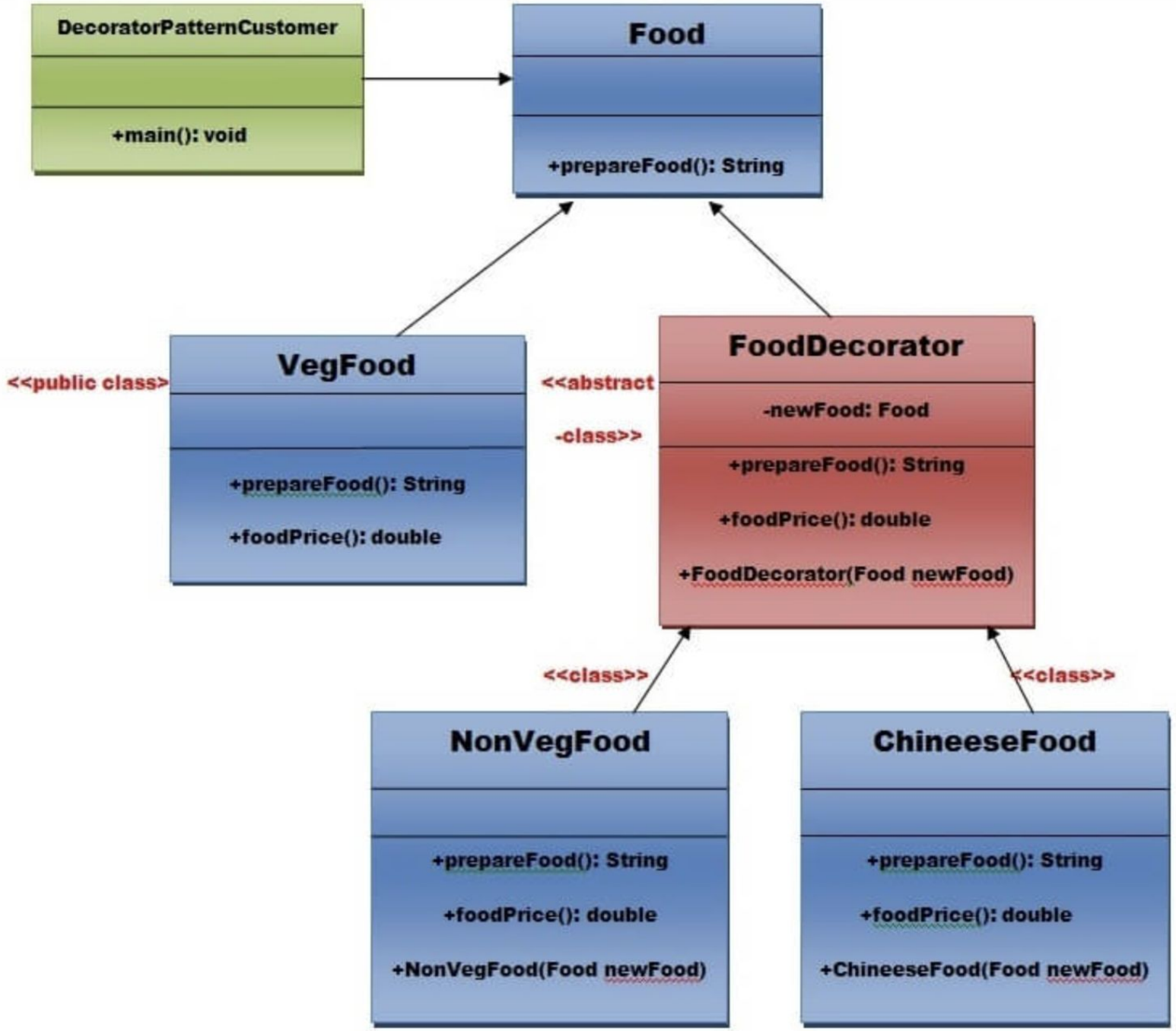
- While we've reduced the number of classes required but we'll need to keep on updating the Beverage class for changes like, cost of Milk or cost of Mocha. This makes Beverage class always open for modification.
- What if there's a double mocha? There's no way we can add mocha two times.
- The method setWhip() doesn't seem appropriate for IceTea.
- Whenever a new type of condiment will be added then the Beverage class will require a change, And it violates the principle:??

Solving via Decorator Design Pattern

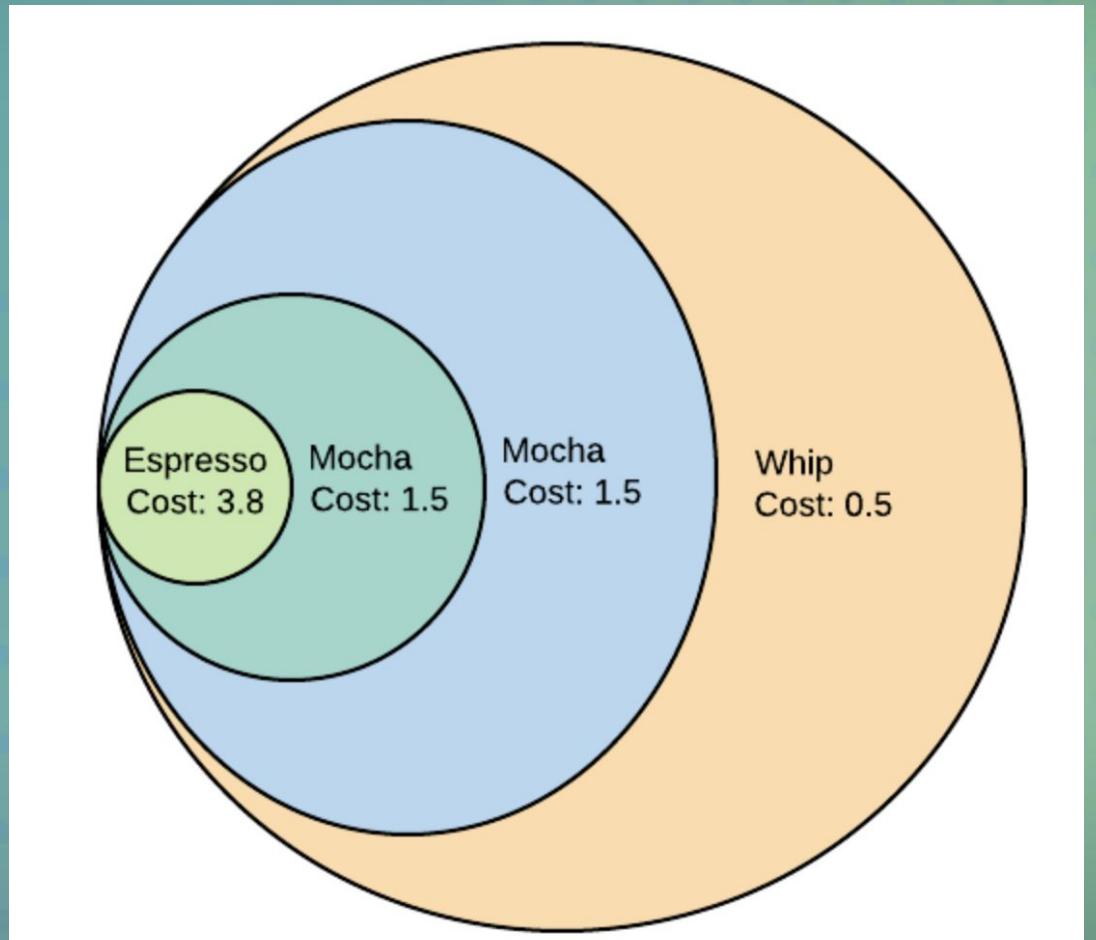
Develop code for
UML Diagram ::



UML for Decorator Pattern Code ::



Demo for Espresso ::

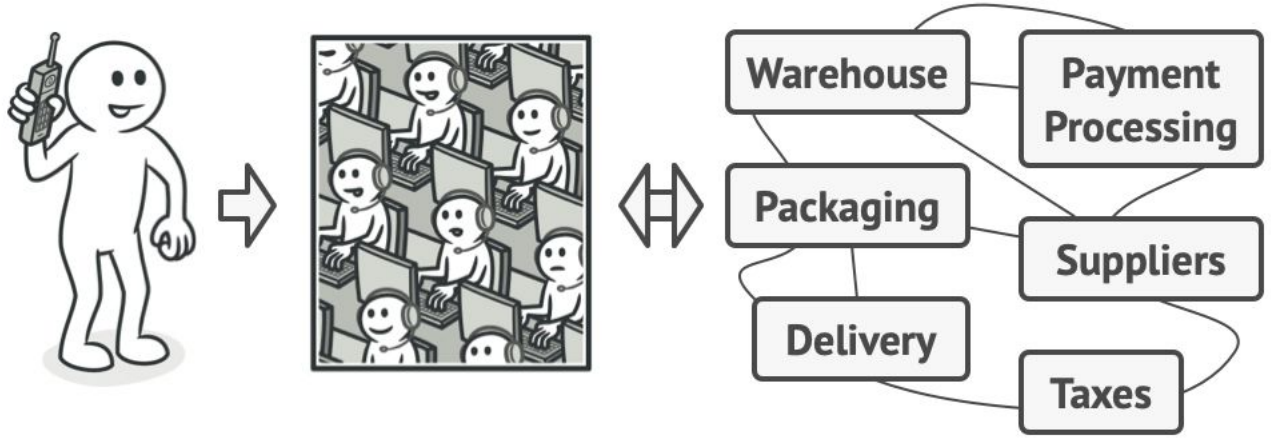


Espresso With Double
Mocha and Whip

Façade Pattern

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.
- Practically, every **Abstract Factory** is a type of **Facade.**

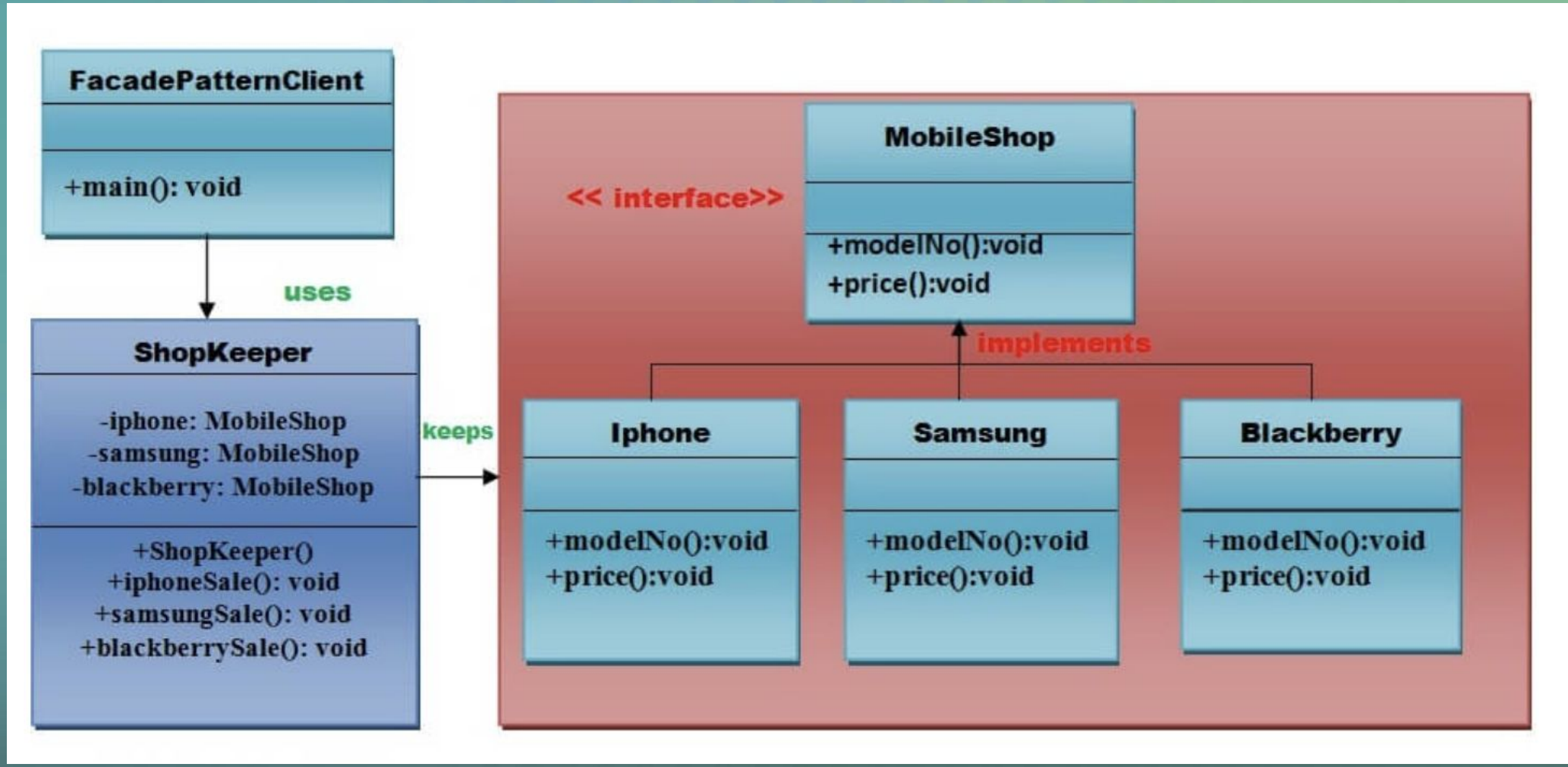
Real-World Analogy



Placing orders by phone.

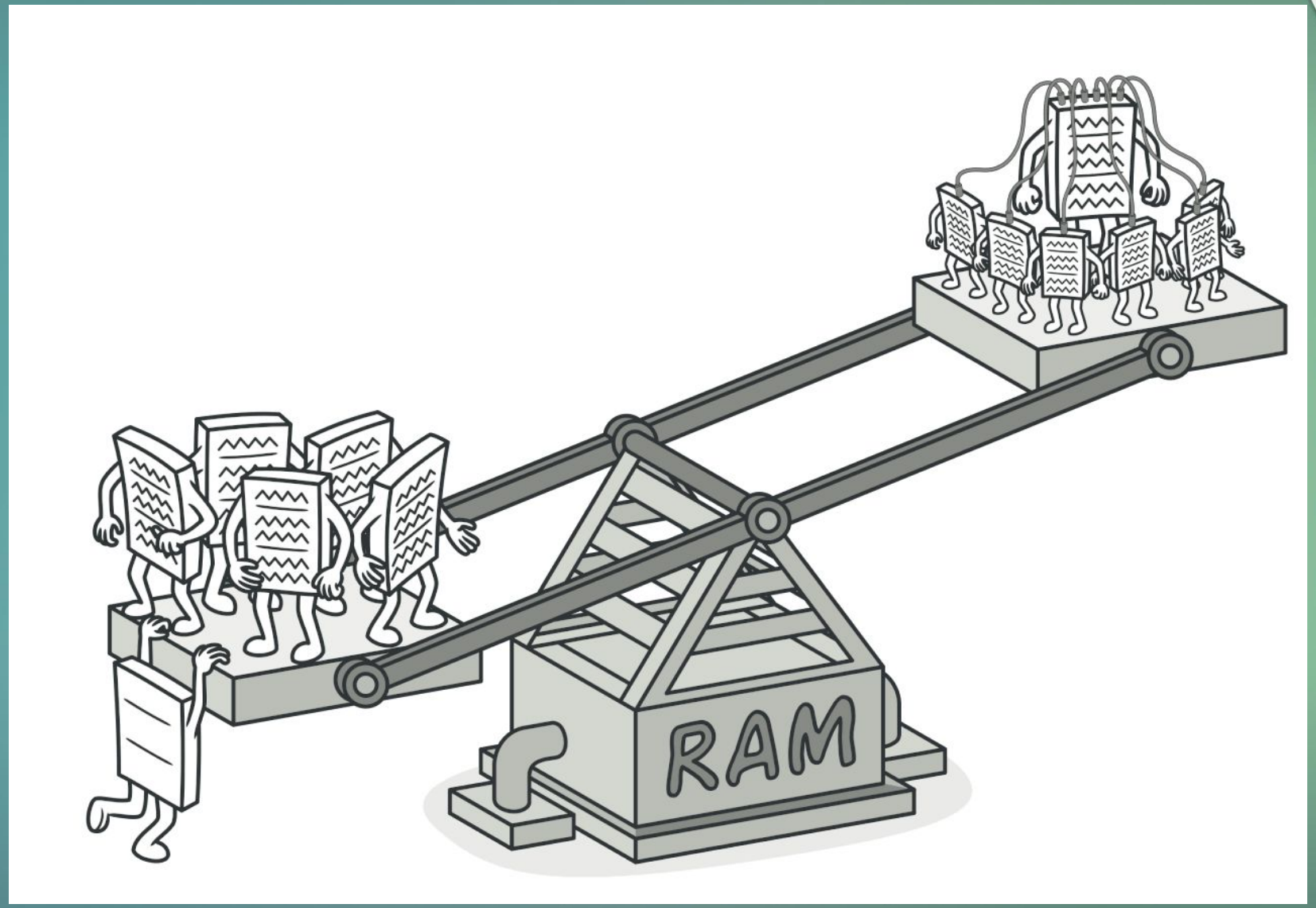
When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.

UML Diagram :: Code



Flyweight Pattern

- Flyweight pattern is primarily used to reduce the number of objects(objects of order ~10^5) created and to decrease memory footprint and increase performance.
- Flyweight comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.
- Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.



Properties

- Intrinsic Properties :
  Properties which are same for set of objects
- Extrinsic Properties :
  Properties which are different for a set of objects

Implementation of Developer and tester Employee

- Interface : Which contain common method : Employee
- Object : Individual class : Developer, tester
- Intrinsic Property (Developer: Fix the issue, Tester : Test the issue)
- Extrinsic Property : Skills
- We use Factory to use return object : EmployeeFactory
- Client : Client class

Proxy Pattern

- Proxy means ‘in place of, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains Proxy Design Pattern.
- Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.
- A real-world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash and provides a means of accessing that cash when required. And that’s exactly what the Proxy pattern does – “Controls and manage access to the object they are protecting“.

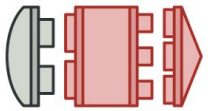
When to use this pattern?

- Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.
- When we want to limit access to certain user i.e Delete Operation in database can only be performed by admin
- A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Pros and Cons of Proxy design pattern

PROS	Cons
One of the advantages of Proxy pattern is security.	
Proxy pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.	This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.
The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.	

Summary of various structural design patterns

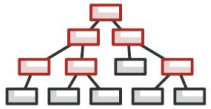


Adapter

Allows objects with incompatible interfaces to collaborate.

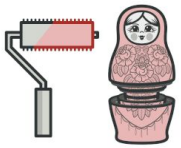
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

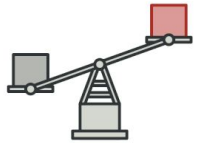


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

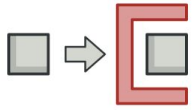
Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Behavioral Design Patterns:

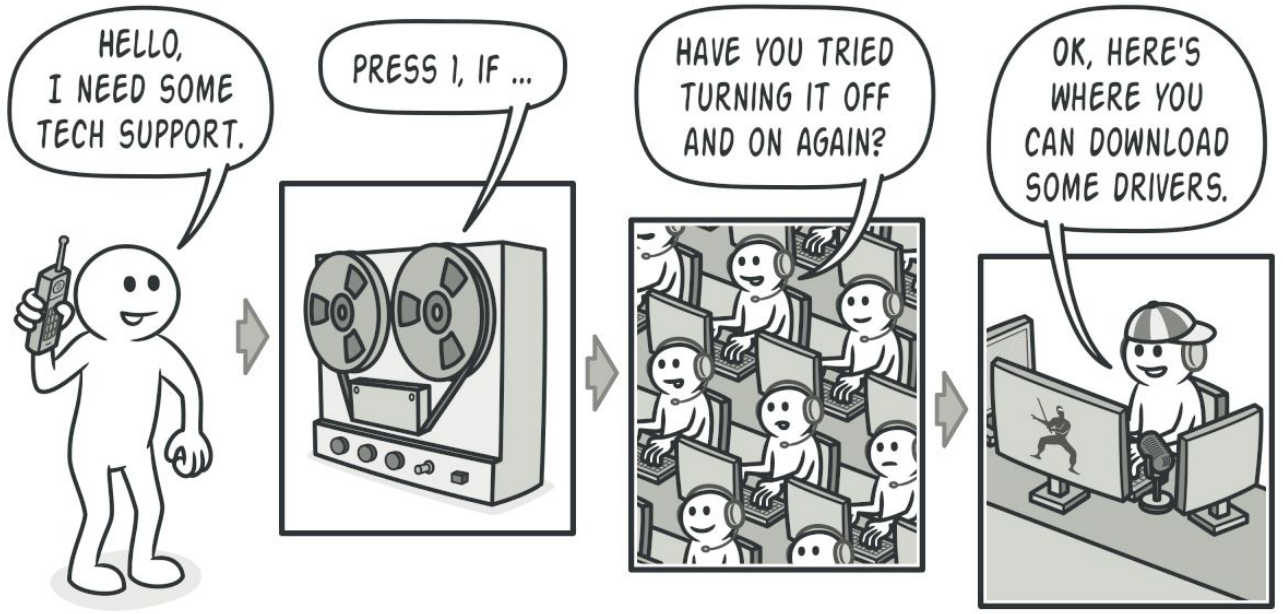
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

In software engineering, behavioral design patterns are design patterns that identify common communication patterns among objects. By doing so, these patterns increase flexibility in carrying out communication.

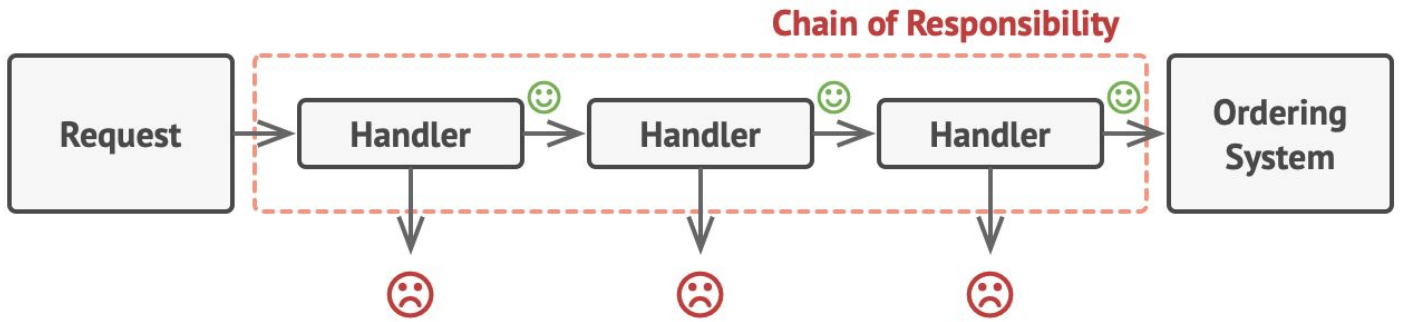
Chain of Responsibility

- As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request.
- This pattern decouples sender and receiver of a request based on type of request.
- This pattern comes under behavioral patterns.
- In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Real World Analogy

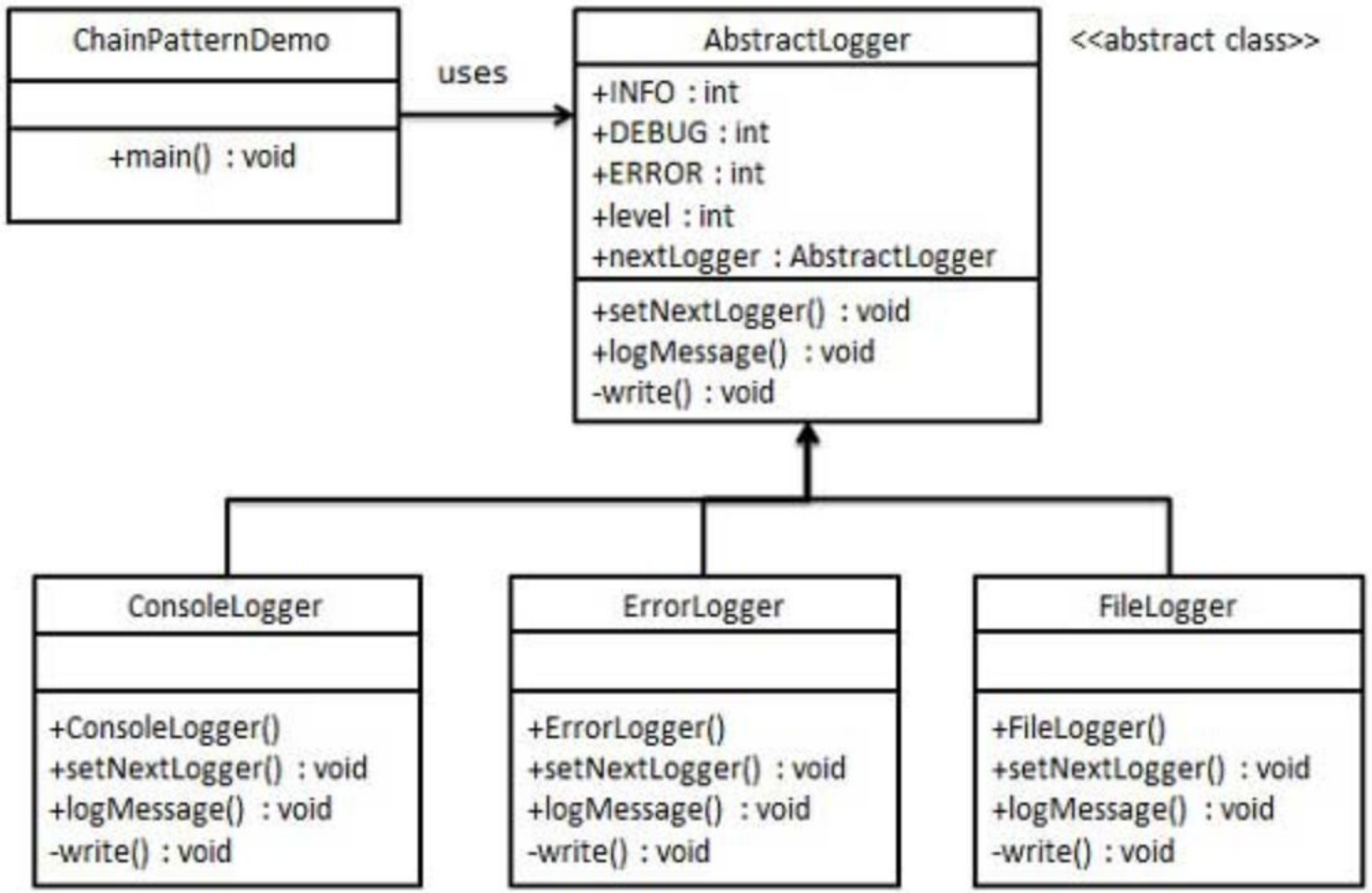


A call to tech support can go through multiple operators.



Handlers are lined up one by one, forming a chain.

UML Diagram to code::



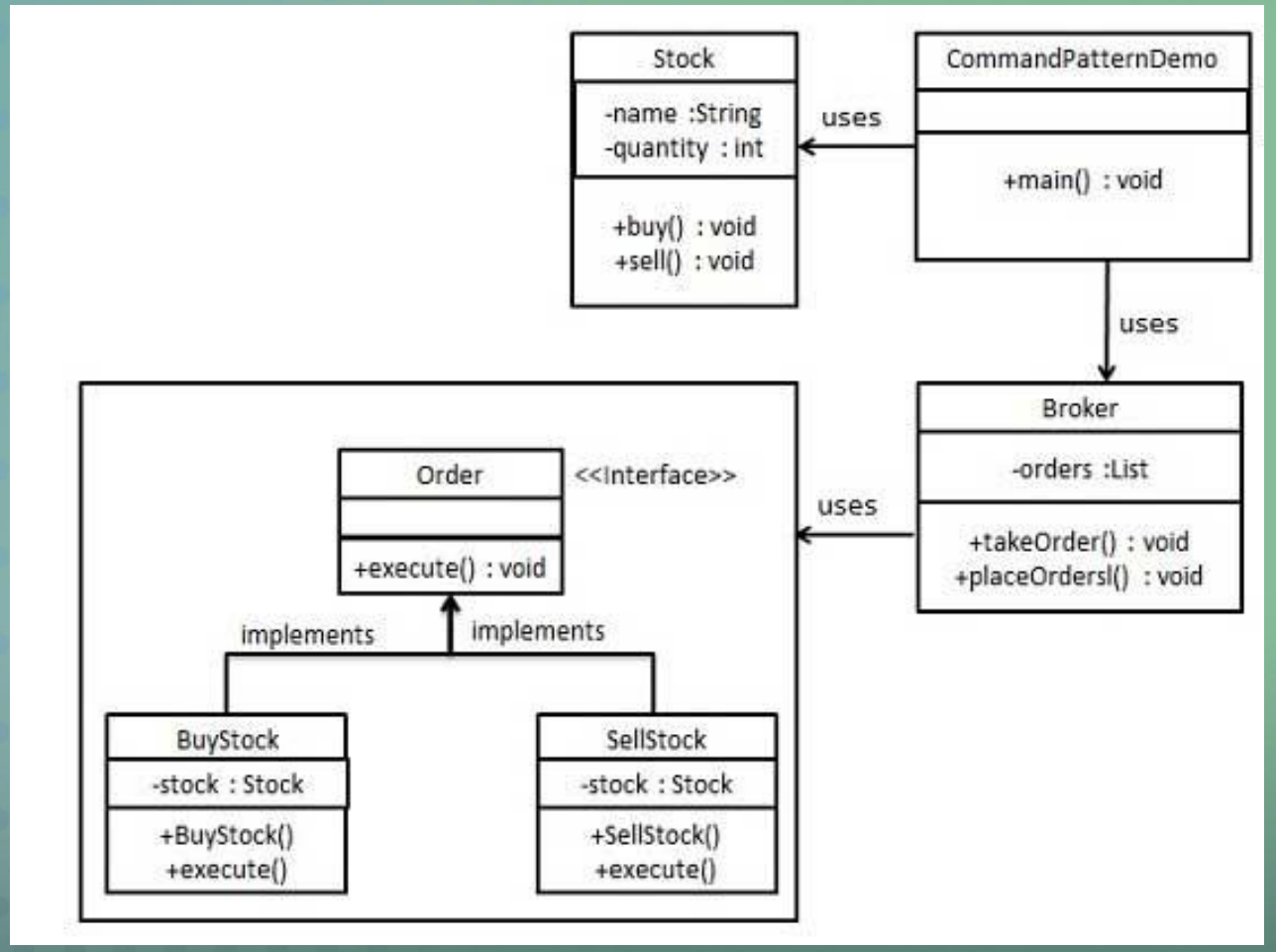
We have created an abstract class AbstractLogger with a level of logging. Then we have created three types of loggers extending the AbstractLogger. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.

Command Design Pattern

- Command pattern is a data driven design pattern and falls under behavioral pattern category.
- A request is wrapped under an object as command and passed to invoker object.
- Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

UML Diagram to code::

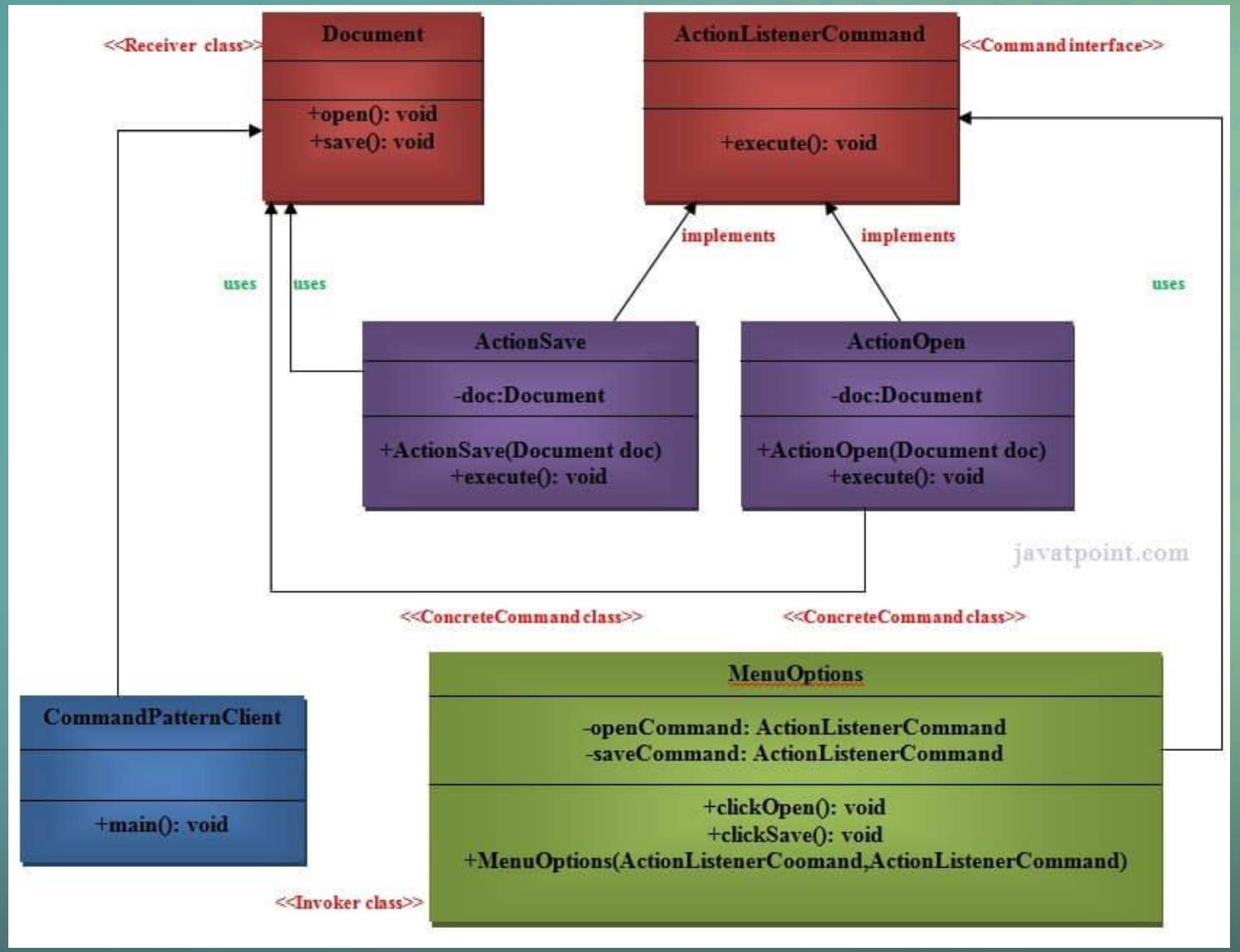
- We have created an interface order which is acting as command. We have created a stock class which acts as a request. We have concrete command class BuyStock and SellStock implementing order interface which will do actual command processing. A class broker is created which acts as an invoker object. It can take and place orders.
- Broker objects 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



These are the following participants of the Command Design pattern:

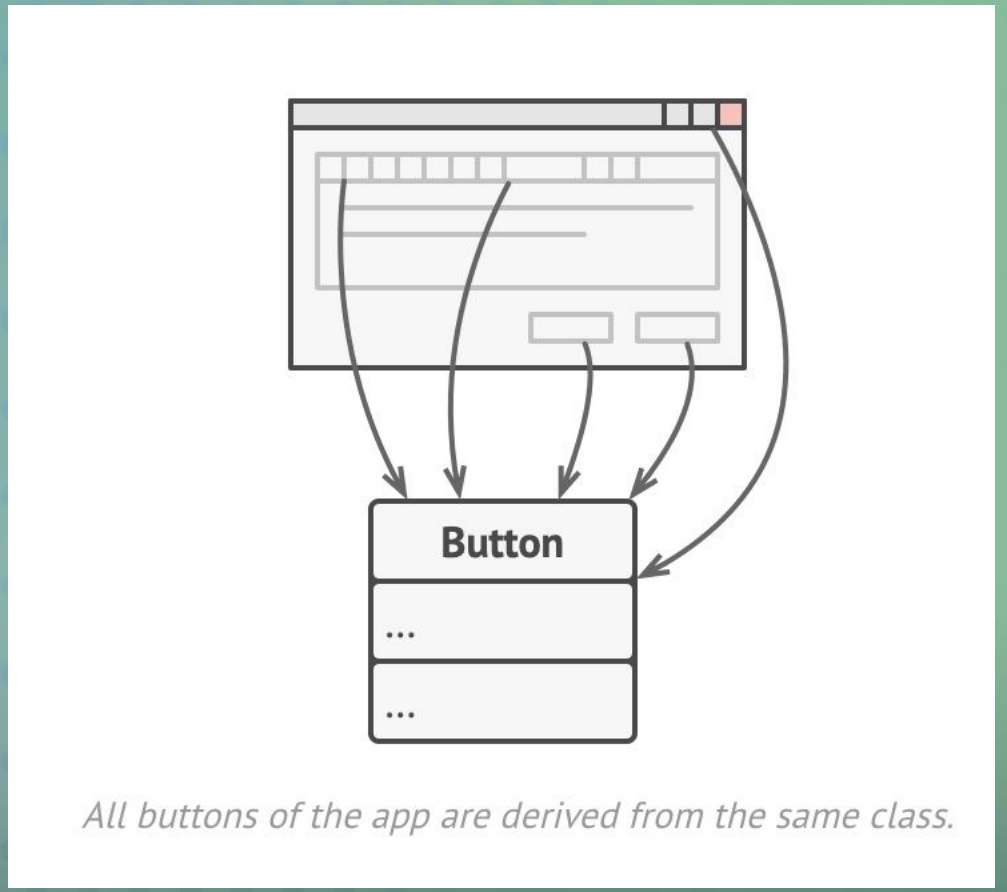
- **Command :** This is an interface for executing an operation.
- **ConcreteCommand :** This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
- **Client :** This class creates the ConcreteCommand class and associates it with the receiver.
- **Invoker :** This class asks the command to carry out the request.
- **Receiver :** This class knows to perform the operation.

UML Diagram to code ::



Problem Statement revisit::

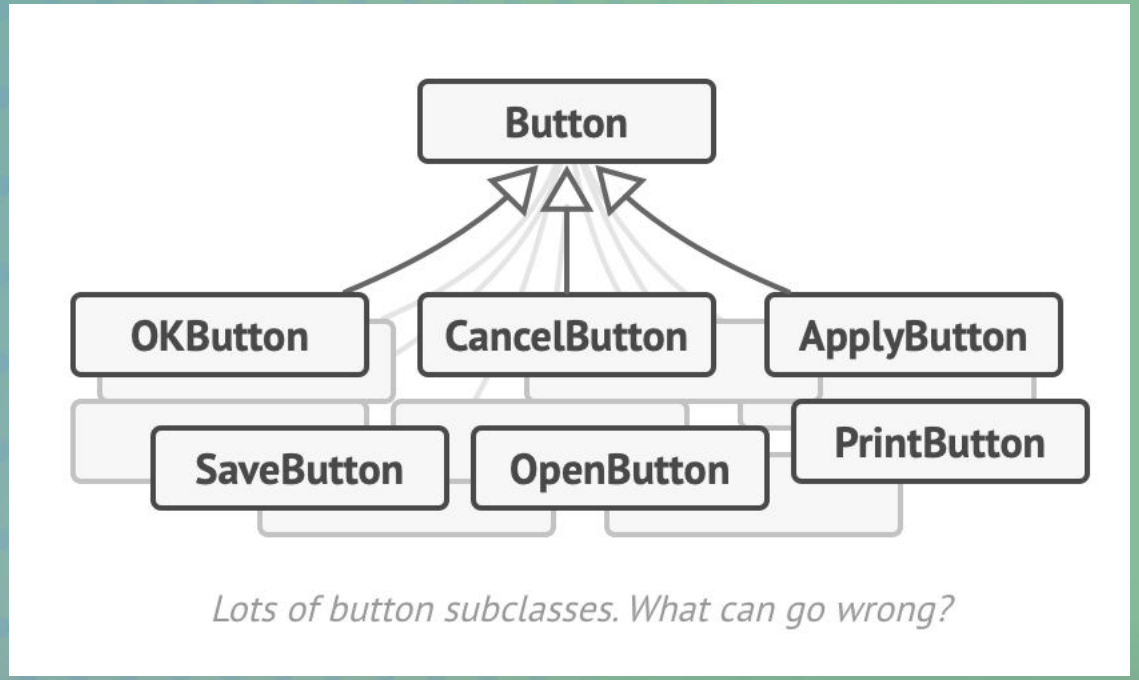
Imagine that you're working on a new text-editor app. Your current task is to create a toolbar with a bunch of buttons for various operations of the editor. You created a very neat Button class that can be used for buttons on the toolbar, as well as for generic buttons in various dialogs.



All buttons of the app are derived from the same class.

Solution Approach 1

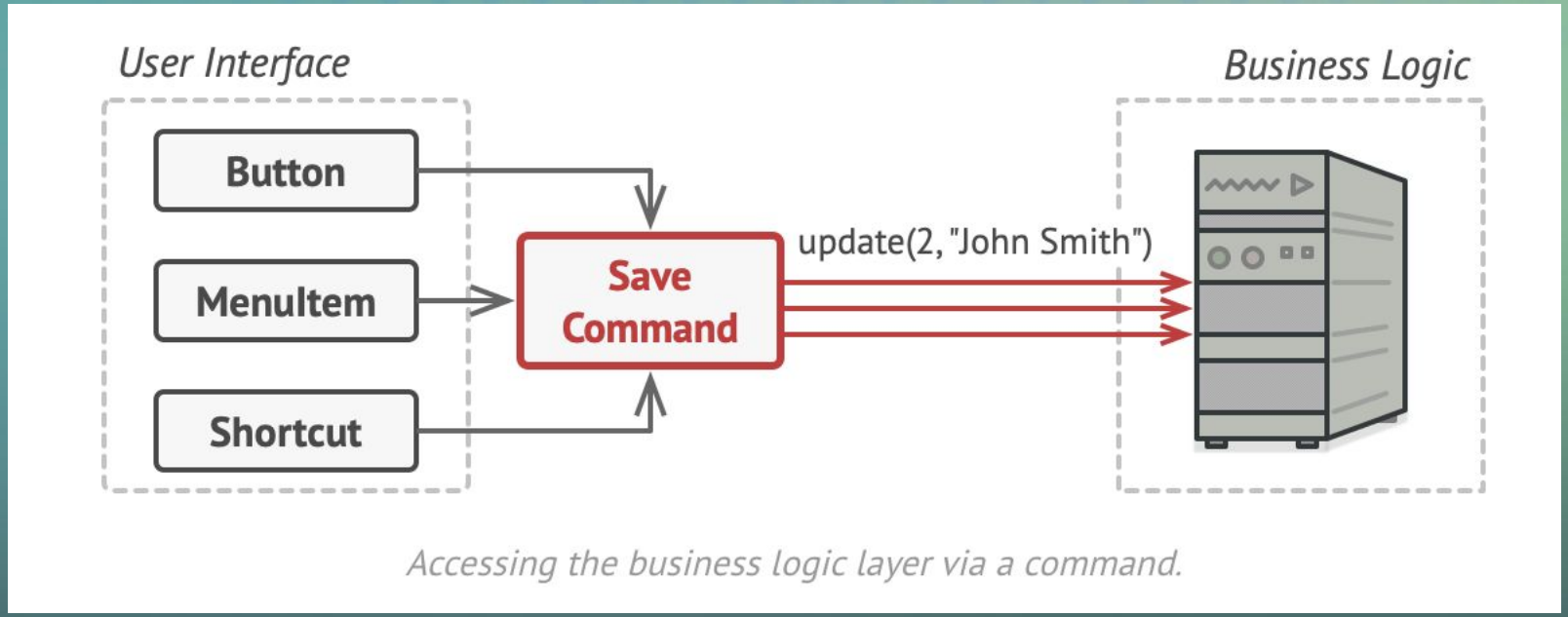
While all of these buttons look similar, they're all supposed to do different things. Where would you put the code for the various click handlers of these buttons? The simplest solution is to create tons of subclasses for each place where the button is used. These subclasses would contain the code that would have to be executed on a button click.



Lots of button subclasses. What can go wrong?

Solution Approach 2

- The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate command class with a single method that triggers this request.
- Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.



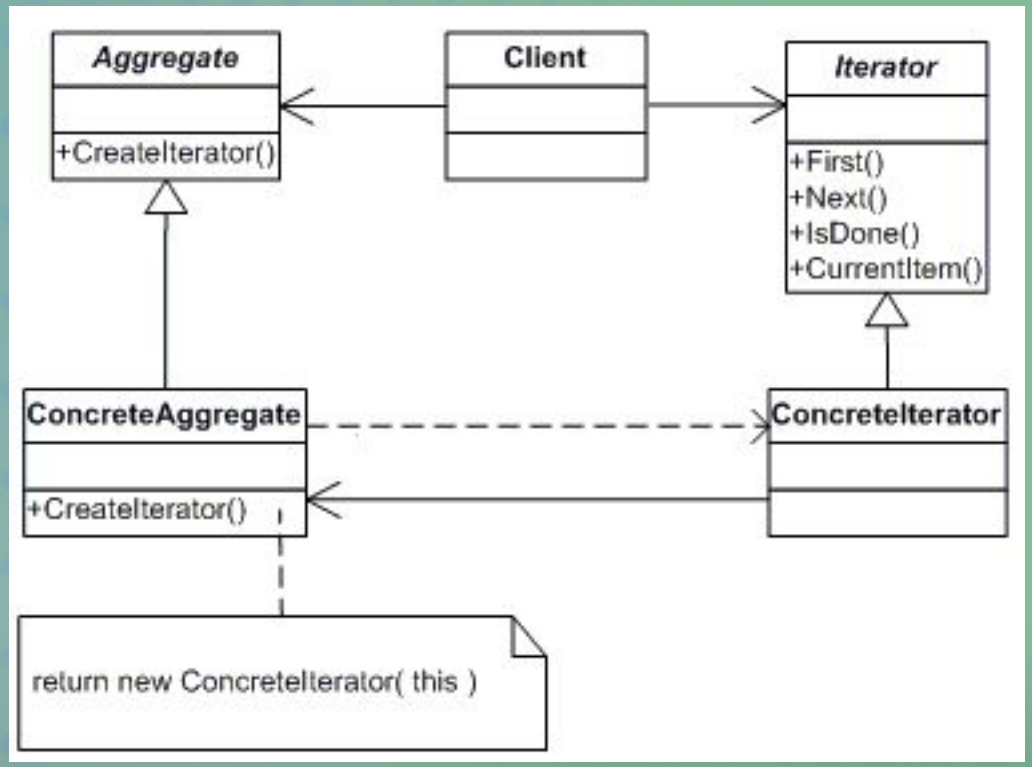
Accessing the business logic layer via a command.

Iterator Design Pattern

- Iterator pattern is very commonly used design pattern in Java and .Net programming environment etc in their collection library
- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Iterator pattern falls under behavioral pattern category.

UML Diagram to code ::

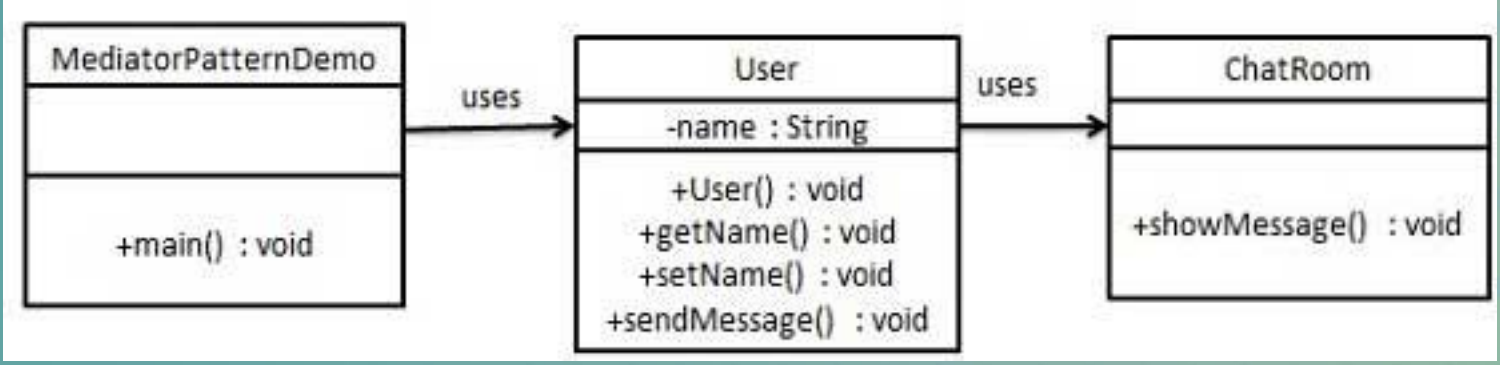
- The classes and objects participating in this pattern include:
- **Iterator (AbstractIterator)** defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)** implements the Iterator interface. keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)** defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)** implements the Iterator creation interface to return an instance of the proper ConcreteIterator



Mediator Design Pattern

- Mediator pattern is used to reduce communication complexity between multiple objects or classes.
- This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.
- Mediator pattern falls under behavioral pattern category.

UML Diagram to Code ::



Memento Pattern

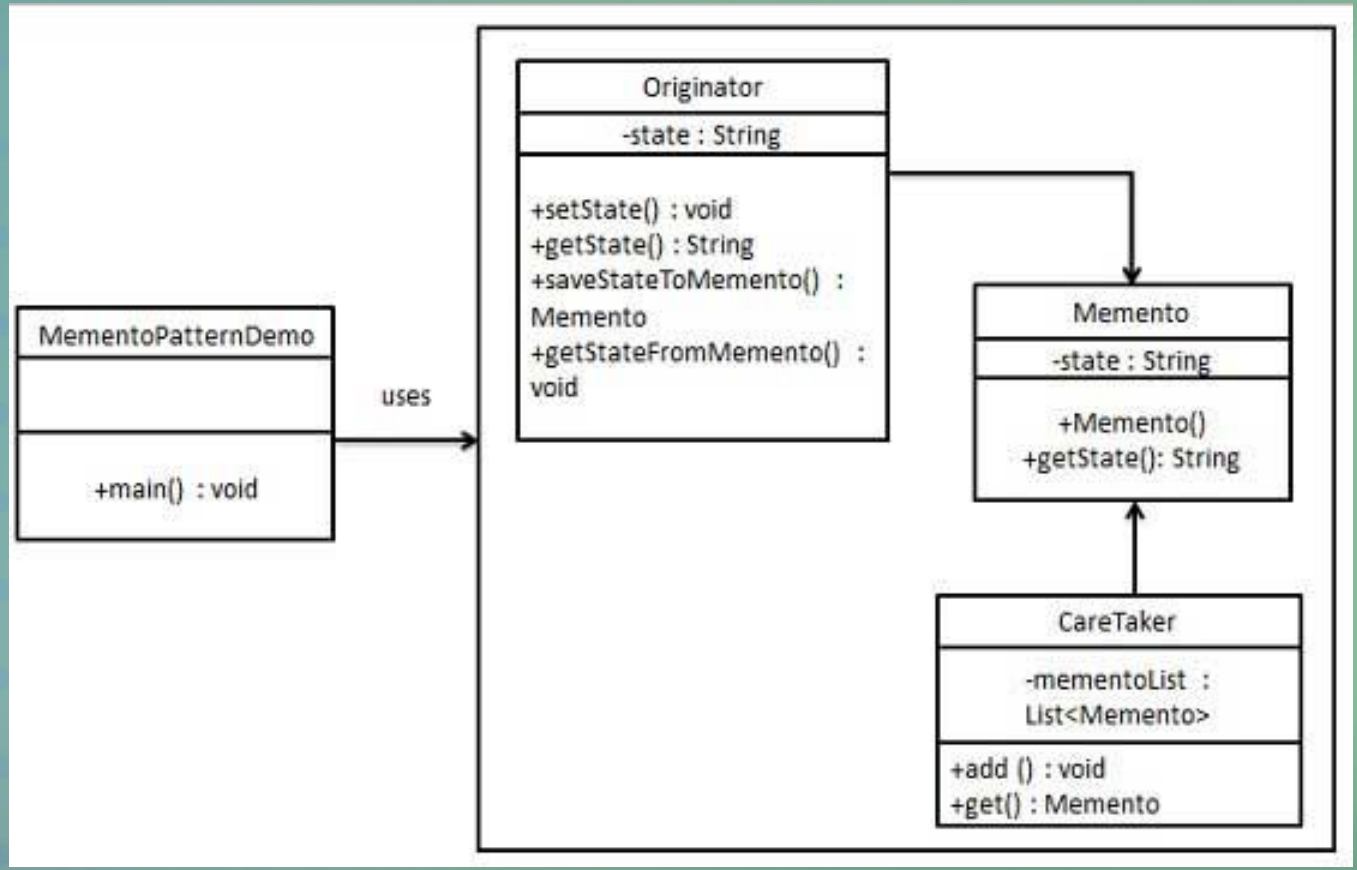
- Memento pattern is used to restore state of an object to a previous state.
- Memento pattern falls under behavioral pattern category.
- The Memento design pattern without violating encapsulation, captures and externalizes an object‘s internal state so that the object can be restored to this state later.

Implementation and UML Diagram to code ::

Memento pattern uses three actor classes.

1. Memento contains state of an object to be restored.
2. Originator creates and stores states in Memento objects
3. 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.



Observer Pattern

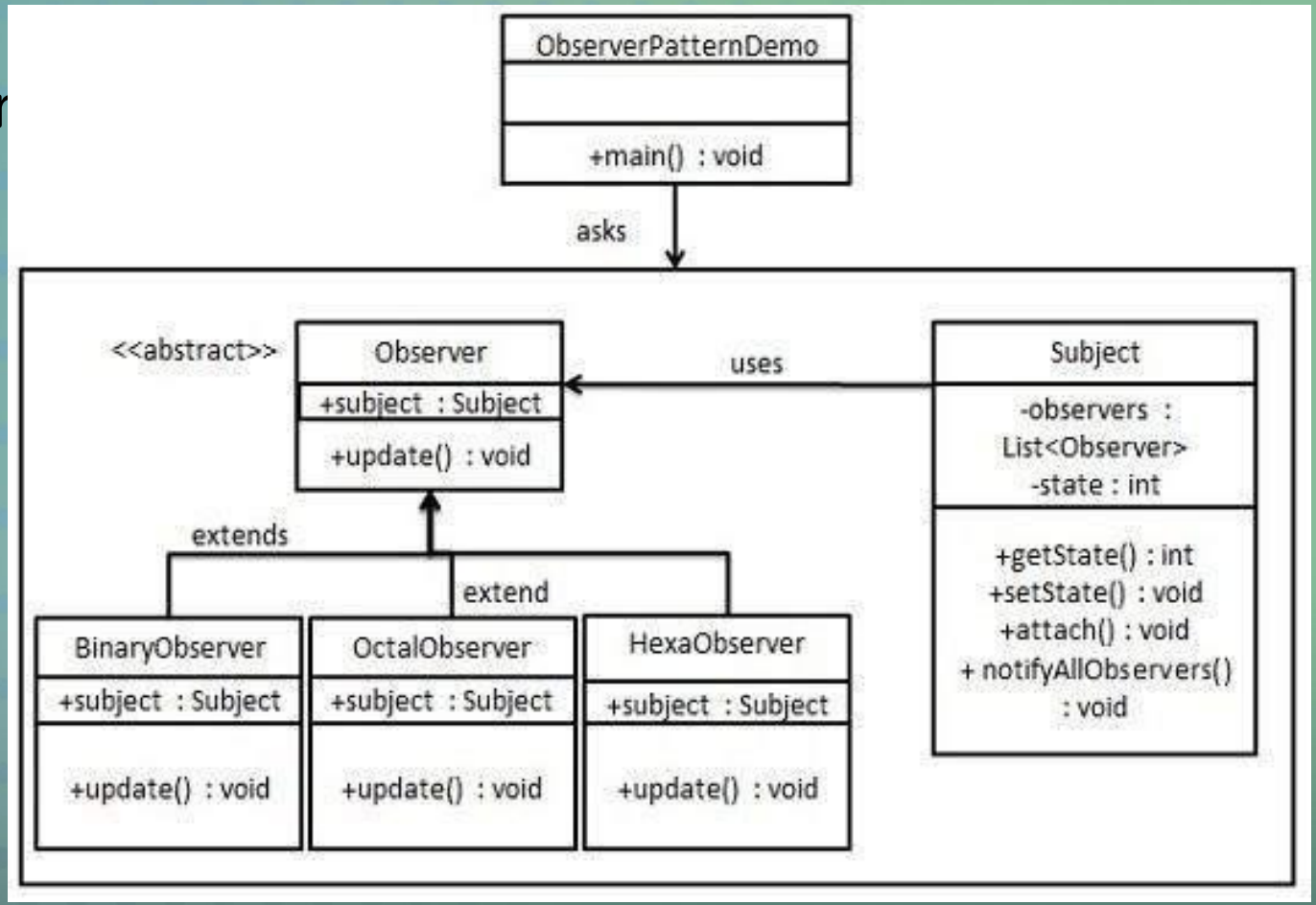
- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.
- Observer pattern falls under behavioral pattern category.

Implementation and UML diagram to code

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.



YouTube case study :

- In YouTube one channel can have multiple subscribers,
  - For subscribers to know about any new video gets added to their channel they can pull (go to channel and check) request

  Or

  Channel can push notification to all their subscribers

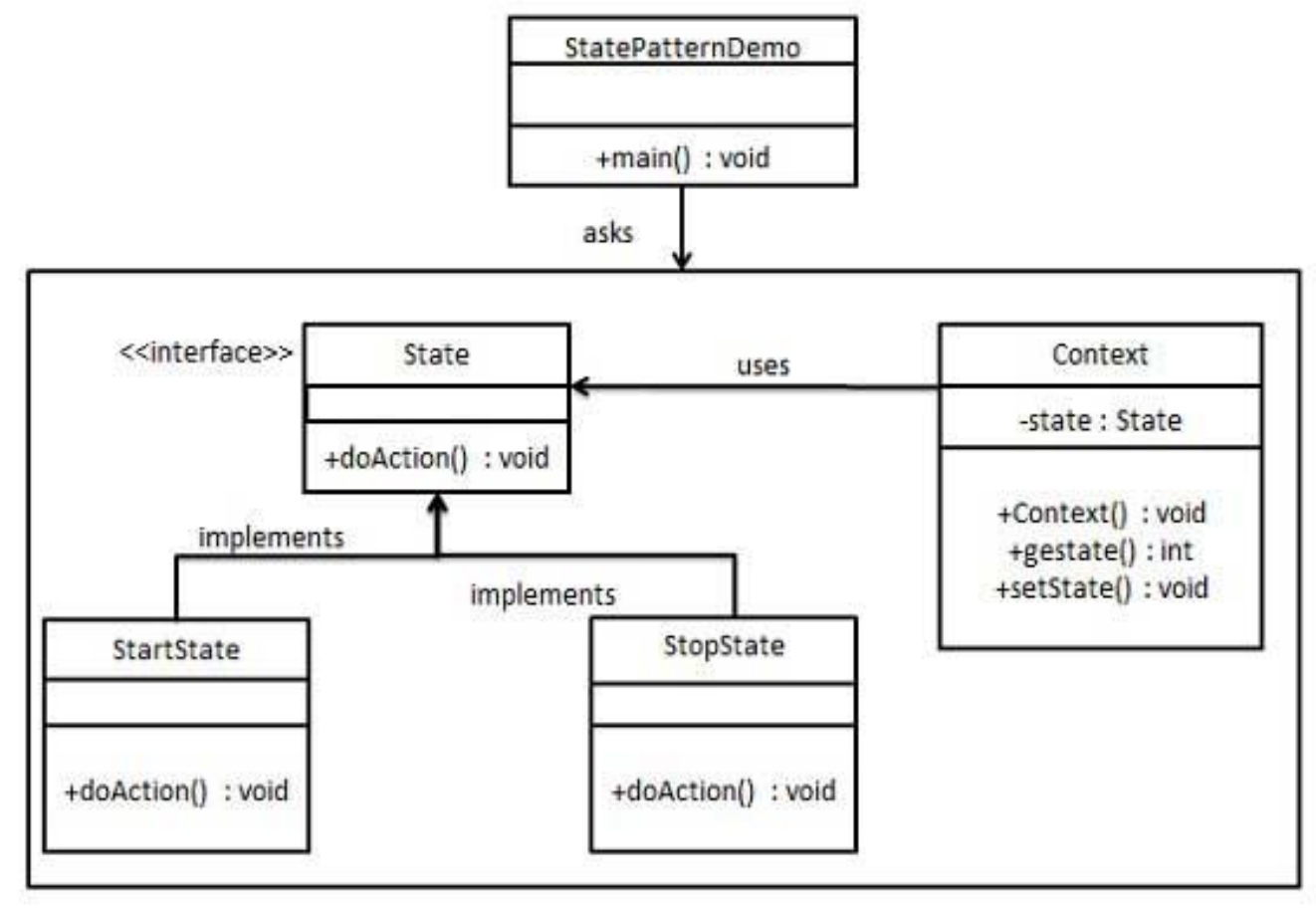
  We'll see the 2nd approach for implementation

State Design Pattern

- In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.
- In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

Implementation & UML diagram to code

- 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.

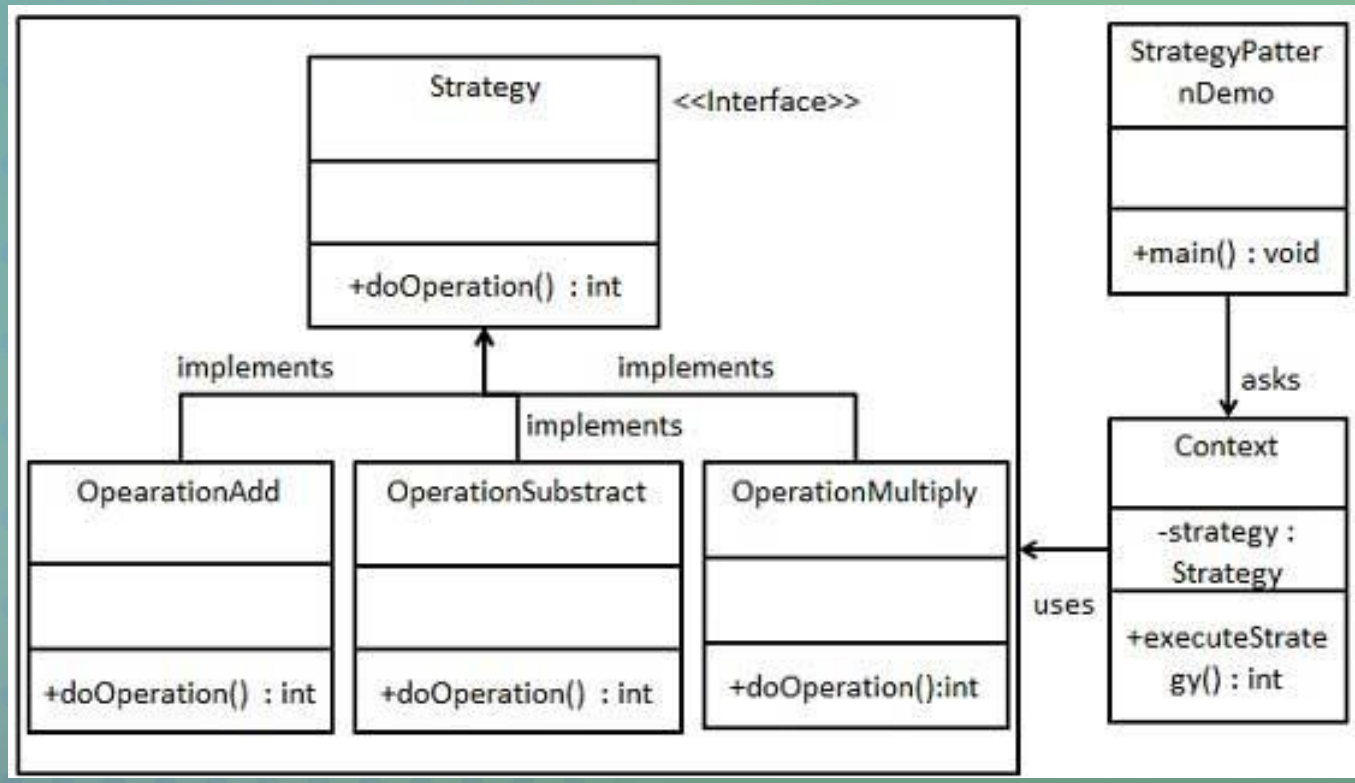


Strategy Patter

- In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Implementation & UML diagram to 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.

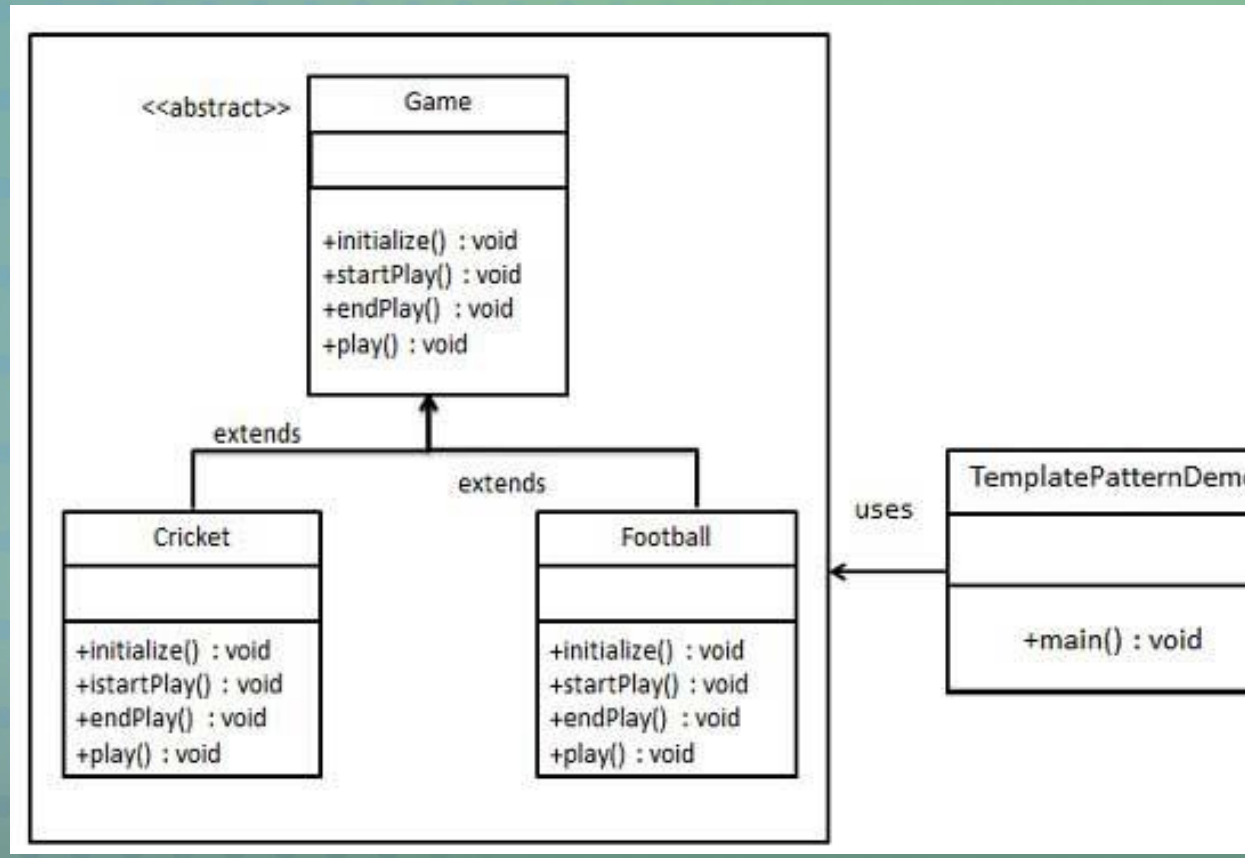


Template Pattern

- In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.
- This pattern comes under behavior pattern category.

Implementation & UML diagram to code

- 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.

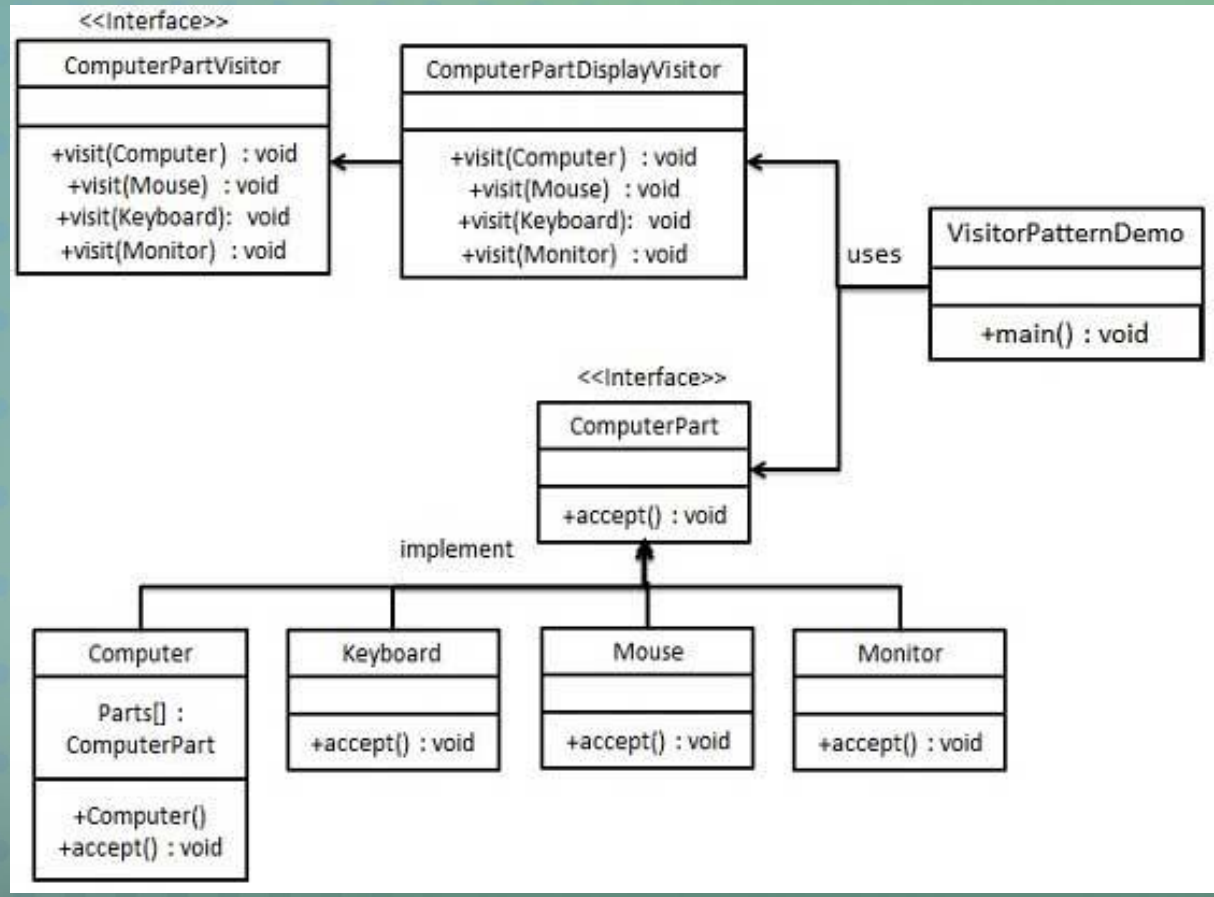


Visitor Pattern

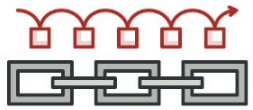
- In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class.
- By this way, execution algorithm of element can vary as and when visitor varies.
- This pattern comes under behavior pattern category.
- As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

Implementation & UML diagram to code

- We are going to create a *ComputerPart* interface defining accept opearation.*Keyboard*, *Mouse*, *Monitor* an d *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.



Summary of Behavioural Design patterns



Chain of Responsibility

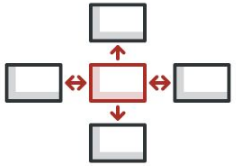
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

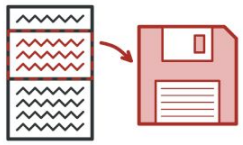
Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



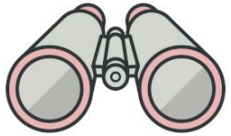
Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



Memento

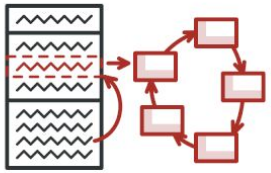
Lets you save and restore the previous state of an object without revealing the details of its implementation.



Observer

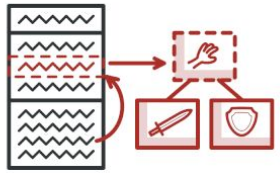
Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Summary of Behavioural Design patterns



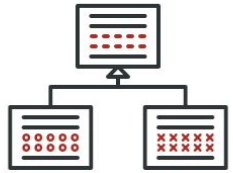
State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



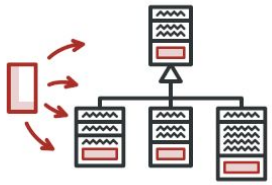
Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



Visitor

Lets you separate algorithms from the objects on which they operate.

ANSWERS TO ASSESMENT

1. **D) Single Reconstruction Principle**
2. **Design principles** are those principles that are followed while designing software systems for any platform by making use of any programming language. SOLID principles are the design principles that we follow as guidelines to develop robust, extensible and scalable software systems. These apply to all aspects of programming.
   Design Patterns are the reusable template solutions for commonly occurring problems that can be customized as per the problem requirements. These are well-implemented solutions that are tested properly and are safe to use. Factory Design Pattern, Singleton pattern, Strategy patterns are a few of the examples of design patterns.
3. Answer : A, Explanation: In proxy pattern, a class represents functionality of another class. In proxy pattern, we create object having original object to interface its functionality to outer world.
4. Prototype design pattern is used for creating duplicate objects based on the prototype of the already existing object using cloning. Doing this has a positive impact on the performance of object creation. Creating objects using the new keyword requires a lot of resources and is a heavyweight process that impacts performance. Hence, the prototype design pattern is more advantageous than the object created using a new keyword.
5. A - These design patterns are specifically concerned with communication between objects.
6. Answer : B Explanation :: true. Event handling frameworks like swing, awt use Observer Pattern.
7. A – Bridge
8. A -- Iterator Pattern
9. In Strategy pattern, a class behavior or its algorithm can be changed at run time.
10. B – Abstract Factory Method