GOF – Design Patterns

GoF Design Pattern Types

- Creational - 5
  - Deals with the creation of an object.
- Structural - 7
  - Deals with the class structure such as Inheritance and Composition.
- Behavioral - 11
  - Provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

Creational Design Patterns

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Singleton

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for logging, drivers objects, caching, and thread pool.

Factory

- The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.
- This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

Abstract Factory

- *If you are familiar with factory design pattern in java, you will notice that we have a single Factory class. This factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.*
- In the Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class.
- Then an Abstract Factory class that will return the sub-class based on the input factory class.

Builder

- Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.
- There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.
- Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
- Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
- If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.
- We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters.

Implement Builder Design Pattern

- First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
- Java Builder class should have a public constructor with all the required attributes as parameters.
- Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
- The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Prototype

- Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.
- Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs.
- Prototype design pattern uses java cloning to copy the object.

- *It would be easy to understand prototype design pattern with an example.*
- *Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using new keyword and load all the data again from database. The better approach would be to clone the existing object into a new object and then do the data manipulation. Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision.*

Structural Design Patterns

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

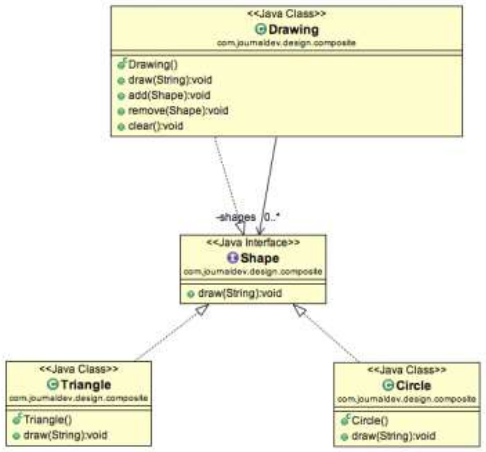
Adapter

- Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together.
- The object that joins these unrelated interface is called an Adapter.
- *One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.*

Composite

- Composite pattern is one of the Structural design pattern.
- Composite design pattern is used when we have to represent a part-whole hierarchy.

- Base Component - Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.
- Leaf - Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.
- Composite - It consists of leaf elements and implements the operations in base component.



Proxy

- Provide a surrogate or placeholder for another object to control access to it.
- The definition itself is very clear and proxy design pattern is used when we want to provide controlled access of a functionality.

Flyweight

- Flyweight design pattern is used when we need to create a lot of Objects of a class. When we need a large number of similar objects that are unique in terms of only a few parameters and most of the stuffs are common in general.

- Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects

- Suppose we have a pen which can exist with/without refill. A refill can be of any color thus a pen can be used to create drawings having N number of colors.

- Here Pen can be flyweight object with refill as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.

- All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.

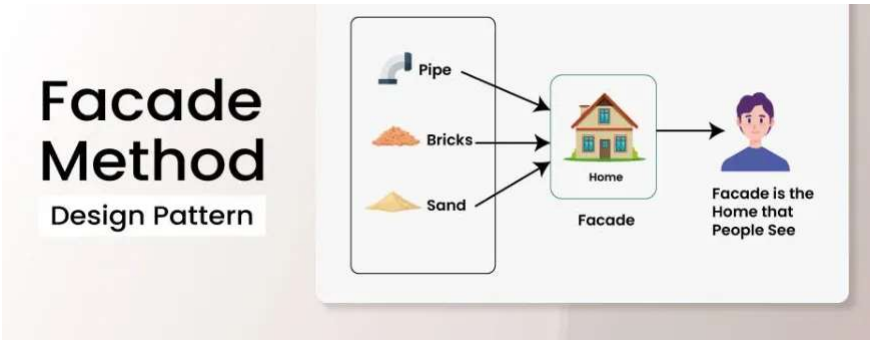
- In programming, we can see **java.lang.String** constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.

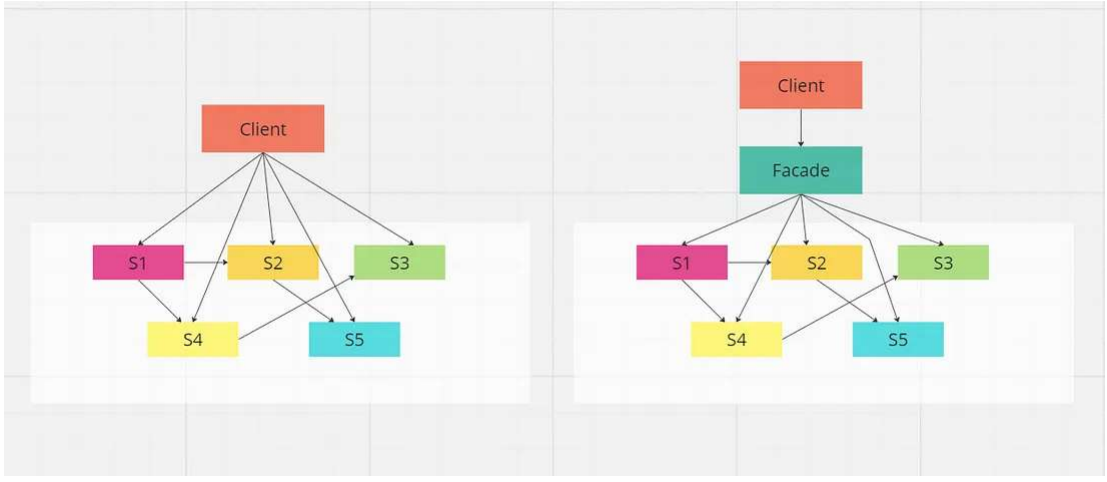
Key Features

- Flyweight Factory: The flyweight factory manages and creates flyweight objects. It ensures that flyweight objects are shared and reused across multiple clients.
- Flyweight Interface: The flyweight interface defines the methods that flyweight objects must implement. It includes operations that can be shared and executed by multiple objects.
- Concrete Flyweight: The concrete flyweight class represents the shared flyweight object. It stores the intrinsic state and provides an implementation for shared operations.
- Client: The client represents the context in which flyweight objects are used. It stores the extrinsic state and interacts with the flyweight objects through the flyweight factory.

Facade

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

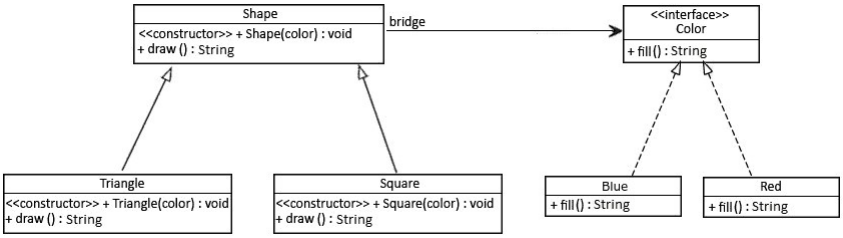

Facade
Method
Design Pattern
Pipe
Bricks
Sand
Home
Facade
Facade is the
Home that
People See


Client
S1
S2
S3
S4
S5
Client
Facade
S1
S2
S3
S4
S5

Bridge

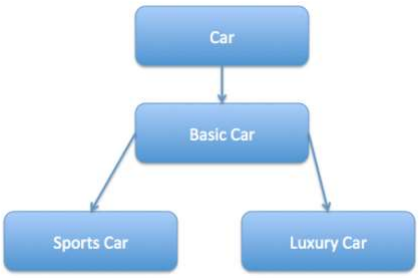
- We use abstraction to decouple client code from implementations.
- Decouple an abstraction from its implementation so that the two can vary independently.

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



Decorator

- Allows adding new functionality to an existing object without altering its original class.
- Involves wrapping the original object in a decorator class, which has the same interface as the object it decorates.



Behavioral Design Patterns

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

Template Method

- Template method defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

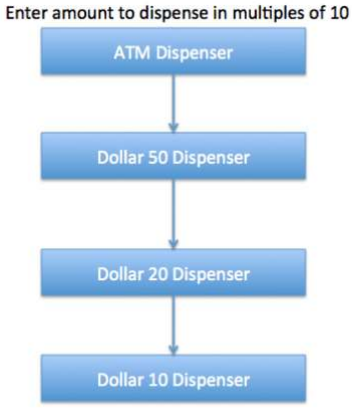
Mediator

- Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other.
- Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other.
- If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend.
- Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.
- Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it's own logic to provide way of communication.

Chain of Responsibility

- Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them.
- Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

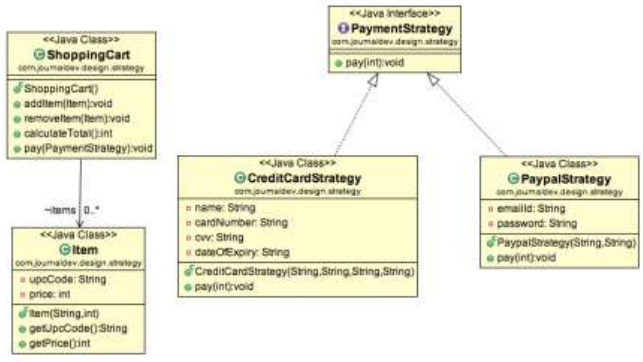


Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Java Message Service (JMS) uses Observer design pattern along with Mediator pattern to allow applications to subscribe and publish data to other applications.
- Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

Strategy

- Strategy pattern is also known as Policy Pattern.
- We define multiple algorithms and let client application pass the algorithm to be used as a parameter.
- One of the best example of strategy pattern is Collections.sort() method that takes Comparator parameter.



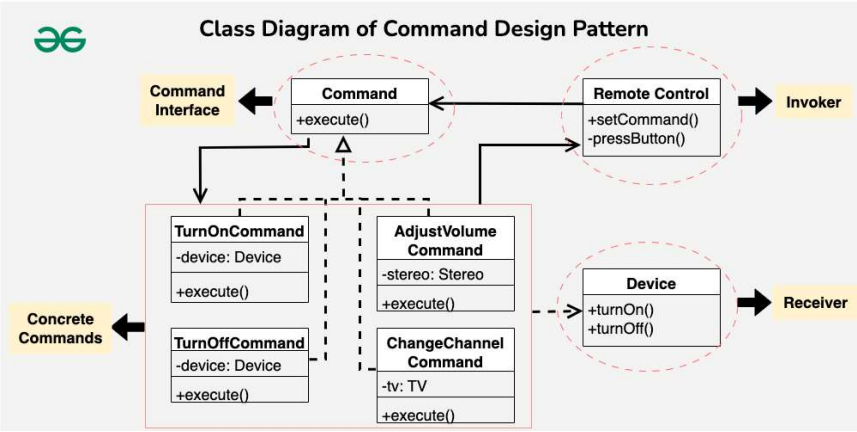
Command

- Command design pattern is used to implement loose coupling in a request-response model.
- In command pattern, the request is send to the invoker and invoker pass it to the encapsulated command object.
- Command object passes the request to the appropriate method of Receiver to perform the specific action.

Components of the Command Design Pattern

- Command Interface
- Concrete Command Classes
- Invoker (Remote Control)
- Receiver (Devices)

- *Imagine you are tasked with designing a remote control system for various electronic devices in a smart home. The devices include a TV, a stereo, and potentially other appliances. The goal is to create a flexible remote control that can handle different types of commands for each device, such as turning devices on/off, adjusting settings, or changing channels.*



State

- If we have to change the behavior of an object based on its state, we can have a state variable in the Object.
- Then use if-else condition block to perform different actions based on the state.
- State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations.

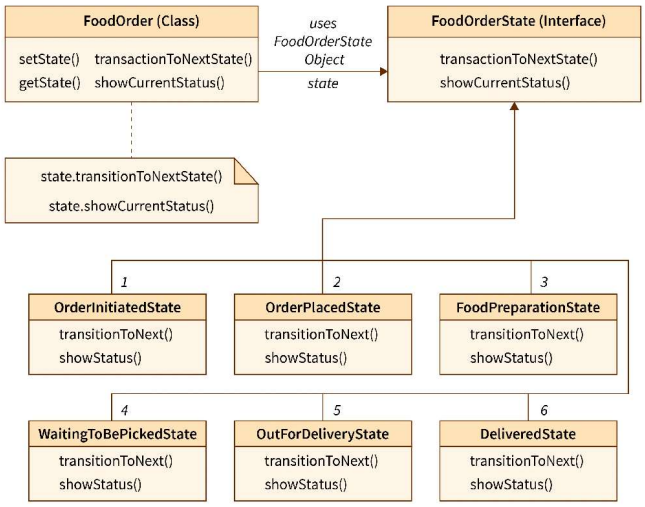
- *Consider a scenario where a user wants to order food online and get it delivered and the service running at the backend needs to handle the various stages that can be there during the entire process from order initiation and order placing to the final delivery of the order.*

```
main():

    while(true):
        if(received order request):
            print(order initiated)

            if(order placed):
                print(order placed)

                while(not cook available):
                    print(waiting for the initiation of the cooking process)

                if(food preparation started):
                    print(food preparation started)

                    while(not food prepared):
                        print(food being prepared)

                    print(food is prepared)
.....
```

- The State Design patterns are used mainly in the following 2 scenarios:

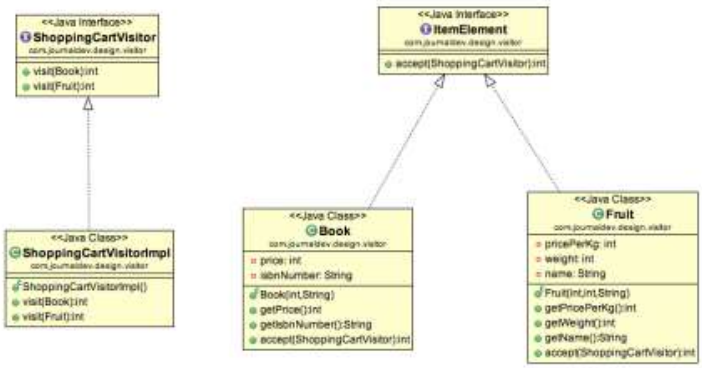
  - When we need to change the behavior of an object based on modifications in the object's internal state and handling it requires applying a lot of conditional statements.
  - When the addition of new states need not affect the behavior of existing states. In other words, the state's pattern is useful when state-specific behavior can be determined independently.

Structure of State Design Pattern



Visitor

- Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.
- For example, think of a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid.
- Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern.



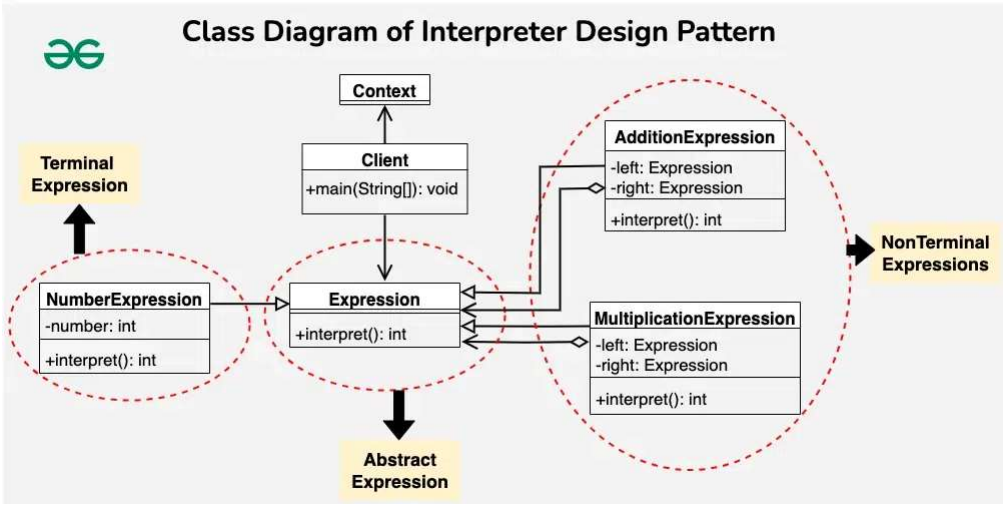
Interpreter

- The Interpreter design pattern defines a way to interpret and evaluate language grammar or expressions.
- It provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes.
- Each class represents a rule or expression in the grammar, and the pattern allows these classes to be composed hierarchically to interpret complex expressions.

Components

- AbstractExpression
  - Abstract class or interface that declares an abstract interpret() method.
- TerminalExpression
  - Terminal expressions represent the terminal symbols or leaves in the grammar.
- NonterminalExpression
  - Non-terminal expression classes are responsible for handling composite expressions, which consist of multiple sub-expressions.
- Context
  - The context may include variables, data structures, or other state information
- Client
  - The client is responsible for creating the abstract syntax tree (AST) and invoking the interpret() method on the root of the tree.
- Interpreter
  - The interpreter is responsible for coordinating the interpretation process.

- *Imagine you are traveling to a foreign country where you do not speak the native language. In such a scenario, you may need the assistance of an interpreter to help you communicate effectively with the locals.*



Iterator

- Provide a standard way to traverse through a group of Objects.
- Iterator pattern is widely used in Java Collection Framework.
- Iterator interface provides methods for traversing through a collection.

Memento

- The memento design pattern is used when we want to save the state of an object so that we can restore later on.
- As your application progresses, you may want to save checkpoints in your application and restore them to those checkpoints later.
- The intent of the Memento Design pattern is without violating encapsulation, to capture and externalize an object's internal state so that the object can be restored to this state later.