Design Patterns

Design Principles

Single Responsibility Principle (SRP)

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

It tells us that a class should have a functional togetherness and focus only one responsibility. Therefore, a class should only be changed for only one abstraction reasons.

In a way, single responsibility principle is an application of Dijkstra's Seperation of Concerns princible. According to Dijkstra, intelligent thinking can only be achieved by focusing an aspect of a subject one at a time.

Levels of SRP; Package: Structures released together should be in the package. Class: A class should abstract only one aspect and has the related data&logics. Method: A method should do a reusable and unbreakable job that directly connected with the abstraction specified by the method's class.

Classes can get more complex in time, in that case, classes can be divided into parts to preserve single responsibility principle. For example, a service class' responsibilities can be seperated in helpers classes. Usually, a class should have methods at most between 15-20, after this interval, classes should be divided. Also, an aggregate class should not implement more than 3-5 interfaces.

Open-Closed Principle (OCP)

Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification.

Software systems always change in terms of features, users etc. These changes must be handled with extensions, not with modifications. In that sense, software modules should be closed for modification. New requirements on softwares should be handled by applying extension.

At developer level, source codes should not be modified. Existed codes should live as they were, developers should only add new codes to change.

Open for extension simply means seperation of modificable part and unmodificable part. Modificable part should be modified by overriding with extended classes, which provides reusability and maintainability for whole system. Therefore, cost and time spent in production get lower.

Liskov Substitution Principle (LSP)

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

A subclass should not change main aspects of its superclass. For examle, assuming a Rectangle super class has attributes of shortEdge and longEdge, so if a Square class extends the Rectangle class which is in real life not an illegal thing to say, Square class cannot be compatible with an API that used to communicate with the Rectangle class because square has no different edges. In this case, Square class changes the main aspects of its super class and it is not desirable.

It can be figured out by thinking about writing tests about super class. For above example, a rectangle's area can be tested by setting short and long edges which would be asserted wrong for square object because expected value would be different for square and rectangle.

```
rectangle.setShortEdge(5);
rectangle.setLongEdge(10);
assert rectangle.getArea() == 5*10;
```

As it can be seen that expected area is 50, but for a square, since there is no short and long edges, expected area would be 25 or 100 and this situation leads bigger problems for clients.

Main purpose of LSP, designed program should be verified by users of the program. The most important aspect is the way of using the system by clients.

Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces that they do not use.

ISP is a version of SRP that aims high togetherness and get rid of fat or polluted interfaces to get nicer and more focussed interfaces.

For example, an interface has 5 methods and 3 of them are used by a client, other 2 of them are used by another client. These kinds of interfaces are called fat interface and they should be divided into parts.

Anti-ISP status has 2 typical indicators:

1. Different clients call methods of same interfaces.
2. Subtypes of an interface have hard time to run some methods.

In these kinds of cases, interfaces should be divided into parts and subtypes should takeover the methods they need so that they can do their jobs easily.

To give another example, assuming a logger interface is designed to serve both logging a database and a file. If an operation does not need to log into database, the class of the operation should implement the the database writing methods as for example UnsupportedOperation. The solution for this example would be defining a Log interface that extended by DBLogger and FileLogger interfaces and in the future if there occurs a need of console logging feature, a ConsoleLogger interface can be added to the equation.

Dependency Inversion Principle (DIP)

High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions.

Demeter Law

Creational Design Patterns

Factory Pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

For example, assuming a company's employee management system has an object for HR and an object for employee. HR adds new employee according to employee's types with if-else statements as below.

```
public void addNewEmployee(int no, String name, int year, String
department, String type, String departmentManaged,
            double bonus) {
        Employee employee = null;
        switch (type) {
        case "Employee":
            employee = new Employee(no, name, year, department, type);
            employees.add(employee);
            break;

        case "Manager":
            employee = new Employee(no, name, year, department, type,
departmentManaged);
            employees.add(employee);
            break;

        case "Director":
            employee = new Employee(no, name, year, department, type,
departmentManaged, bonus);
            employees.add(employee);
            break;
        }
    }
```

In the future, if employee's business logic gets complex, addNewEmployee method would become an if-else hell.

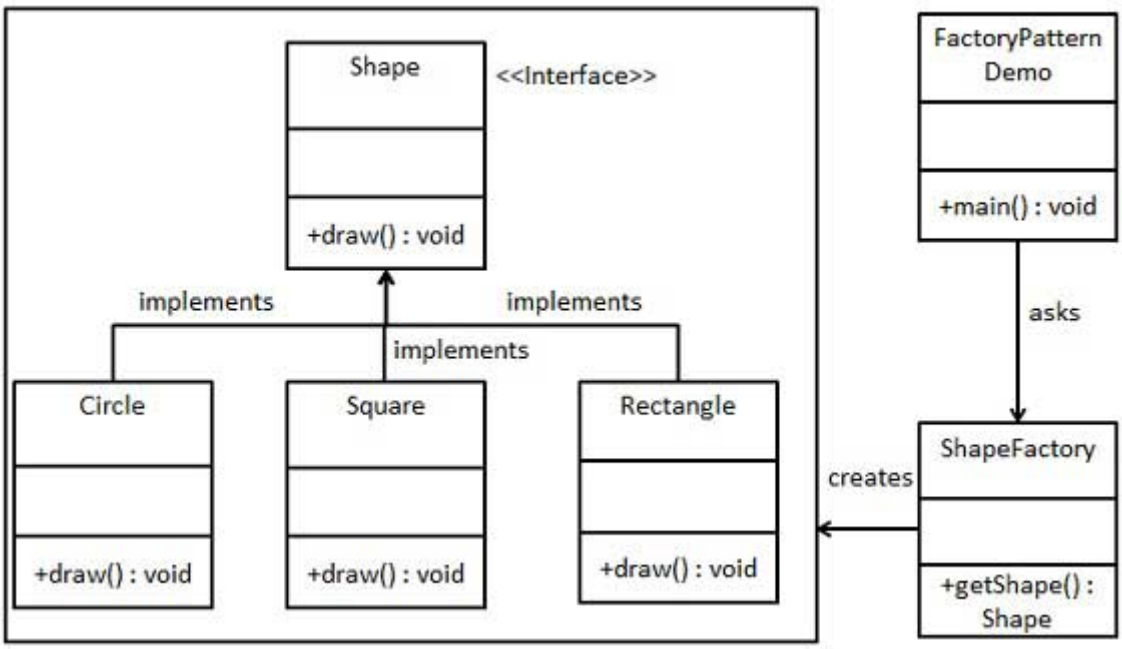
Factory method solves the problem that where to create objects by abstracting the creations of objects and defining creations in factory methods rather than in client codes. Factory method is responsible for creation of objects with interfaces and these operations happen in subclasses. Each subclass creates a different objects.

Instead of above structure, creation of objects can be handled by an interface called Factory with a single method of create(), and the implementation classes of EmployeeFactory, ManagerFactory and DirectorFactory.

```
public interface Factory {
    Employee create();
}
```

But still, create method has no parameters and this can cause some problems for some system. Even tough passing parameters instead of collecting them in the method when possible is a very bad practice, sometimes it can be must to pass parameters. There is trade-off here, fewer parameters passed, more the factory method can be benefit.

The main goal of factory method is creating just one object. By this way, if-else hell can be removed from the equation. So, creating a class for factory, not an interface, that creates all kinds of objects related to job conflicts the principles SRP and OCP. Because this means handling more than one task in one place and opening for modification.




```
public interface Shape {
   void draw();
}
```

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
```

```
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

ShapeFactory object creates the appropriate shape according to the parameter passed to getShape() method.

Abstract Factory Pattern

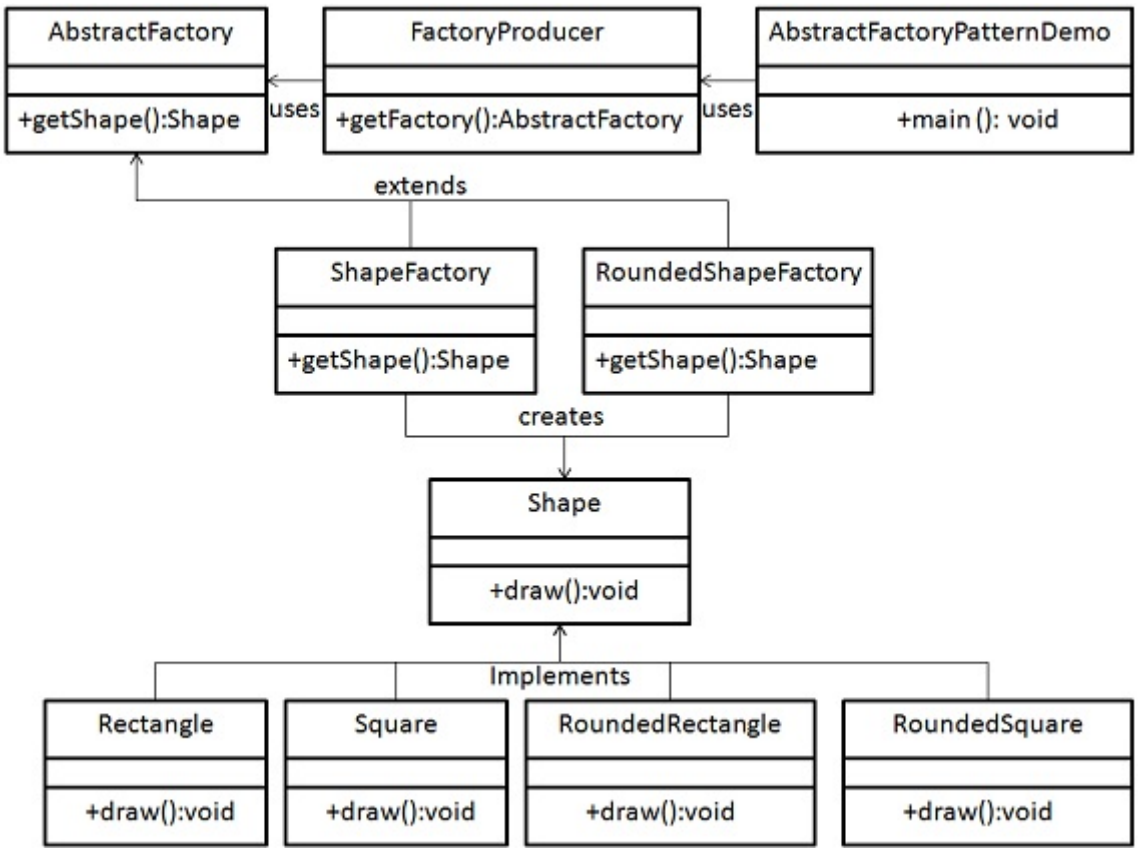
Provide an interface for creating families of related of dependent objects without specifying their concrete classes.

Abstract factory is used for creating a group of objects instead of just one. Factory method abstracts only one object, abstract factory more than one. In that sense, abstract factory has more than one factory method.

The group of objects must be related and dependent with each other. A family of related product objects is designed to be used together, and this constraint must be enforced.

For example, for a windowing system, a family of objects could be Button, Label, List, Scrollbar etc.

There are some constraints while using abstract factory, such as if different kinds of objects are needed to create by different platforms, abstract factory interfaces must be divided. For example, if there are two different university with different kinds of features such as one of them has virtual class but the other does not, these universities' abstract factory interfaces must be divided. Similar problem can occur with different parameters with same objects and again it can be solved by dividing the interfaces.



Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

```
public interface Shape {
   void draw();
}
```

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}

public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}

public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
```

```
            return new RoundedRectangle();
         }else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new RoundedSquare();
         }
         return null;
      }
   }
```

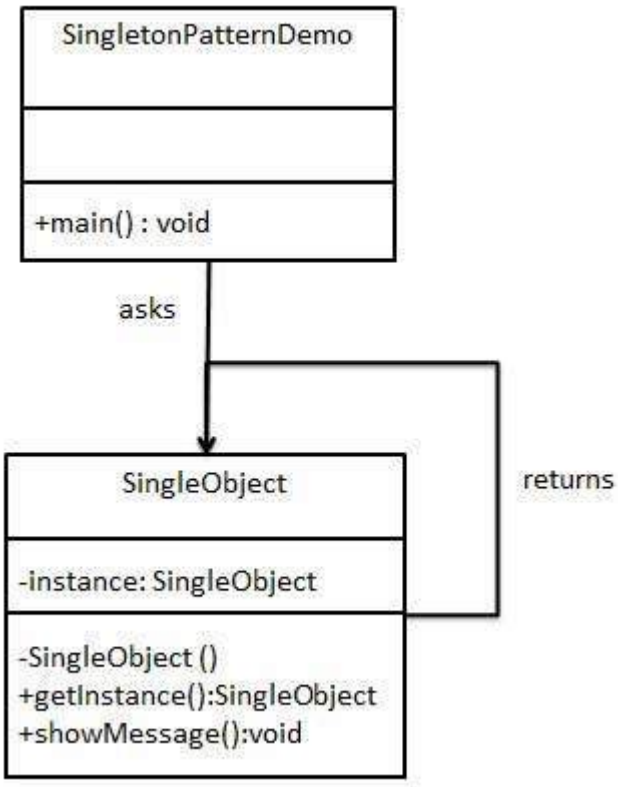
```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

Singleton Pattern

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.




```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
```

```
        //SingleObject object = new SingleObject();

        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
     }
  }
```

Adapter Pattern

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Decorator Pattern

Attact additional responsible to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Often times, new responsibilities are given to objects instead of whole class and inheritence is used to give these responsibilities. Main problem of this approach, inheritence has a compile-time structure. Each inheritence requires a new recompilation. So that, changing code & recompiling code and the number of subclasses becomes a big problem.

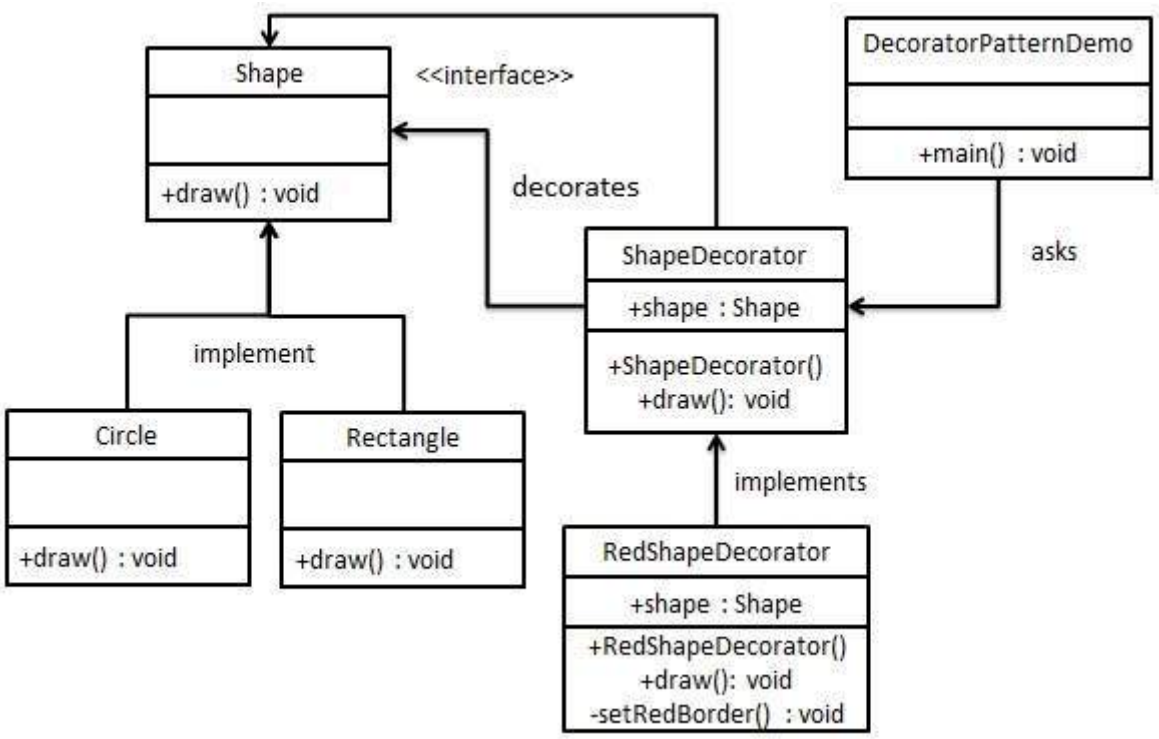
Favor object composition over class inheritance.

Object composition is a runtime structure. Inheritance is static, composition is dynamic.

Assuming there is a toast maker and adds different ingredients to toasts according to clients' wishes. So, instead of creating every ingredients' combinations, a Toastable interface can be created and an abstract class of Topping implements this interface to add each ingredient on top of the current toast.

This method is like a pipeline and simply adding ingredients at anytime to current state of the toast without effecting previous ingredients.

Decorator pattern often requires passing the same type of object to an abstract class's constructor. So that, previously added items are visible to currently to be added item and the current item adds itself following to previously added items.




```
public interface Shape {
   void draw();
}
```

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}

public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }
```

```
    public void draw(){
       decoratedShape.draw();
    }
 }
```

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

Facade Pattern

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

Generally, systems are complex structures, but interfaces should be simple. Because clients' dependencies should be decreased as much as possible. In order to manage complexity of a system, systems are usually

divided into subsystems. Due to this approach, subsystems becomes clients of each other. Therefore, subsystems' dependencies should be as low as possible.

Facade is providing a service of a subsystem by a simple interface. Therefore, facade provides a minimum dependency among subsystems.

Clients time to time need objects from subsystems without dealing with the details. Without facade, these details must be handled by clients resulting in fragile and complex clients. Making these details available for a client can be problematic in many ways. For example, clients should know about technological details about the system or maybe there can be a detail that confidential for clients.

Therefore, clients should be dependent to an interface, not to details. Facade reduces subsystems' complexities and converts them to a simple interface.

Computer system actually is a facade system. Clients start a computer with a simple press. Otherwise, client should start all the services of a computer manually, namely cpu, gpu, os etc. So that, client is served by only facade object which handles all details according to client's needs.

Facades can have multiple layers to reduce complexity.

In conclusion, facade provides an entry point for subsystems. Converts a system with fine-grained interfaces to a system with coarse-grained interface. Facade can be used whenever a system requires simplification and simple interface.

Similarities with others: Abstract factory can be considered as a creation facade system.

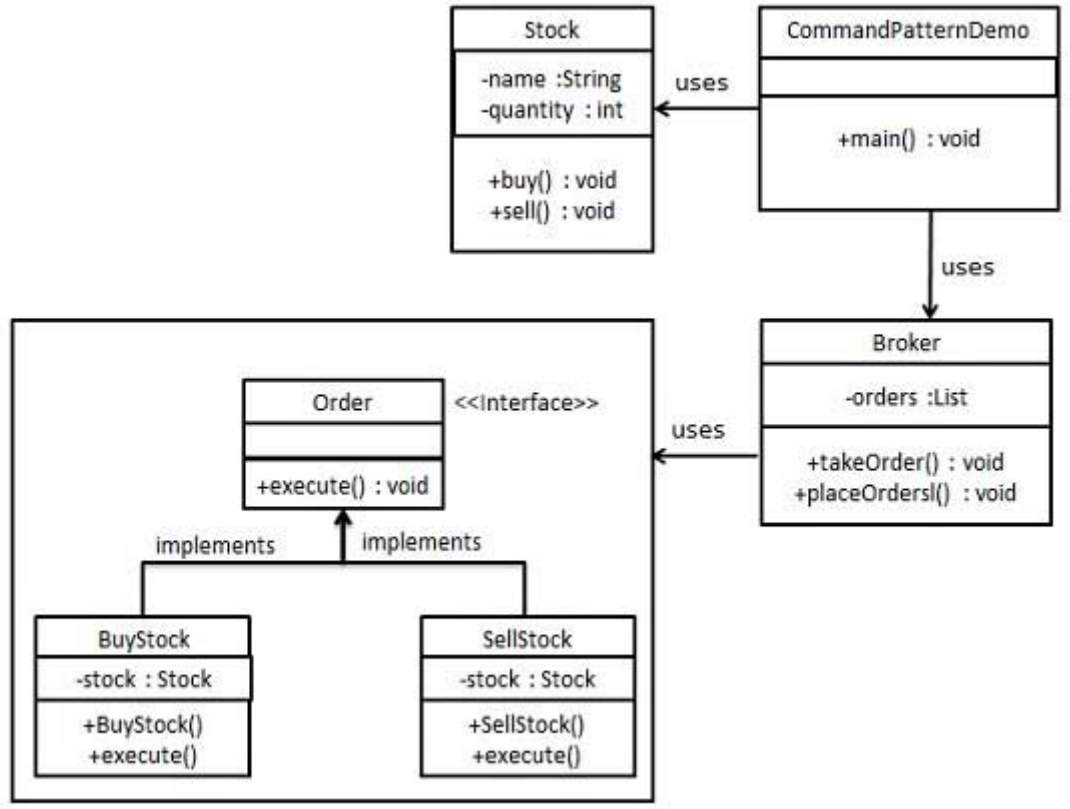
Behavioral Design Patterns

Command Pattern

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

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.

The class diagram below shows an example that an interface Order which is acting as a command, a Stock class which acts as a request, 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.




```
public interface Order {
   void execute();
}
```

```
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

```
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }
```

```
   public void execute() {
      abcStock.buy();
   }
}

public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

```
public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){

      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

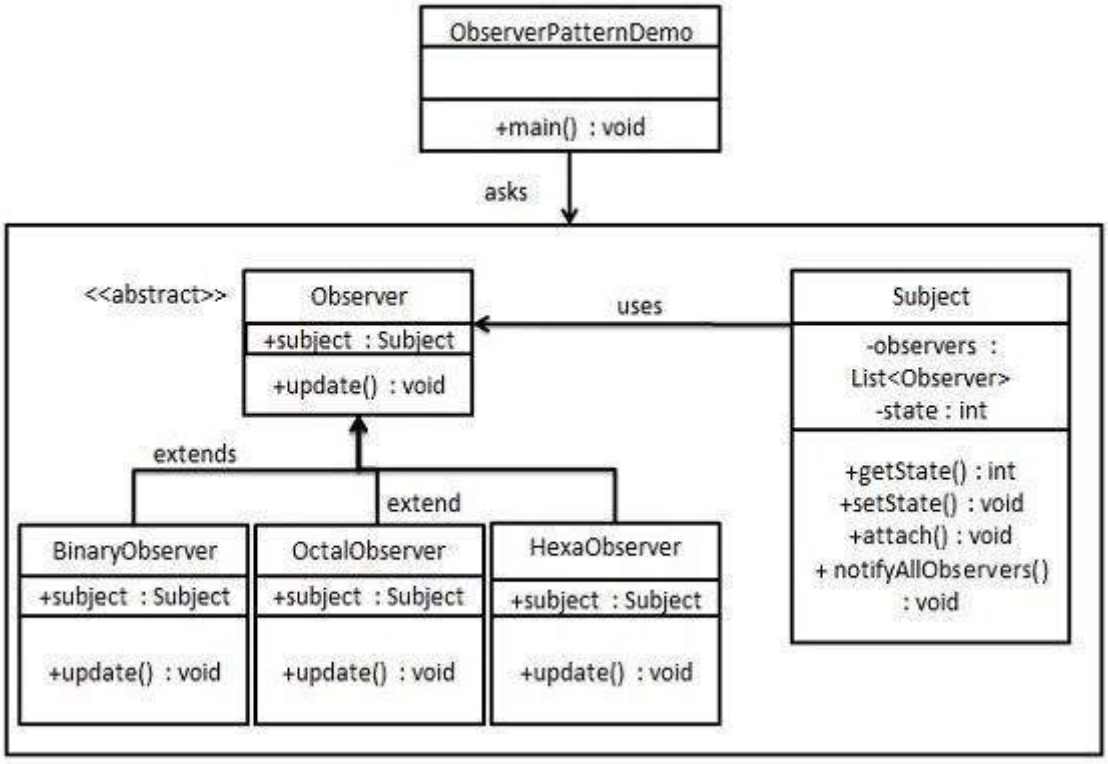
```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

Observer Pattern

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

From time to time it is desirable to be aware of changes in the state of some objects. Querying for these objects periodically is a way of achieving it. However, this approach requires unnecessary operation in the system and causes uncertain delays.

Observer pattern provides a subscription solution that notifies listeners of the objects. Observable, subscribed object, can be observed by observer, listener, by receiving event notifications. Therefore, on observables there is an interface that allows subscription, and on observers there is an interface that provides receiving event notifications. This pattern is also known as Publisher-Subscriber, Producer-Consumer or Event-Notification.



For class diagram above, Subject class, sets the state with setState method and calls notifyAllObservers method to update their content.

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
```

```
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}

public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString(
subject.getState() ) );
   }
}
```

By below code, subject object is passed through observers and called its attach() method to add theirselfs. Whenever the setState() method is called from client, all the observers will be notified by subject class.

```
Subject subject = new Subject();

new HexaObserver(subject);
new OctalObserver(subject);
new BinaryObserver(subject);

subject.setState(15);
subject.setState(10);
```

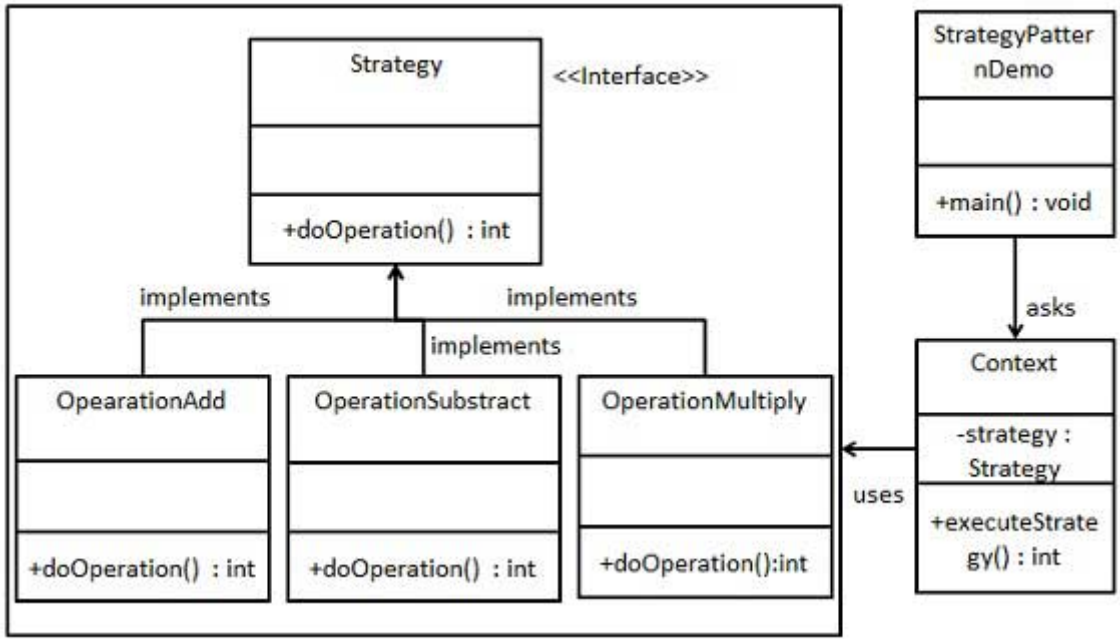
Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independent from clients that use it.

Often times, in software development there are more than one way to resolve an issue. Business logic of software gets bigger and these solutions are sometimes already defined algorithms or sometimes basicly different methods. For example, there could be more than one way of validating a data and these validations should not be handled with if-else statements. This solution is never suitable for long run, ignores future changes which conflicts with SRP and OCP.

To solve these problems, putting different algorithms abstracted from client in a way of interchangeable. In this way, algorithms would be seperated from each other so they will not be dependent on each other and they will be abstracted. Strategy pattern's solution constructing an interface that implemented by sibling classes. Each sibling does a different algorithm and a context object in the program decides which sibling to use. Therefore, new strategies can be added to system easily.

If there is more than one way of resolve an issue, strategy pattern should be used and an abstraction of algorithms should be provided to system.




```java
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

```java
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```