Date._____

Wali Muhammad

2021-SE-39

Software Construction & Design (SCD)

Assignment

23 Design Patterns

Date._____

Organizing Design Pattern Catalog

The design Patterns are divided into 3 categories:

Creational	Structural	Behavioral
1- Factory Method	6- Adapter	13- Interpreter
2- Abstract Factory	7- Bridge	14- Template Method
3- Builder	8- Composite	15- Chain of Responsibility
4- Prototype	9- Decorator	16- Command
5- Singleton	10- Facade	17- Iterator
	11- Flyweight	18- Mediator
	12- Proxy	19- Memento
		20- Observer
		21- State
		22- Strategy
		23- Visitor

Define :-

Creational : Process of Object creation

Structural : Composition of classes or objects

Behavioral : How classes or objects interact and distribute responsibility

Date._____

Abstract Factory Design Pattern

Introduction

Provide an interface for creating families of related or dependent objects without specifing their concrete classes. Also known as Kit.

Scenario

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviours for user interfaces "widgets". An application should not hard-core its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel

Date.

better.

We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards.

WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but client aren't aware of the concrete classes they are using.

There is a concrete subclass of WidgetFactory for each look-and-feel standard. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. In other words, clients only have to commit to an interface defined by an

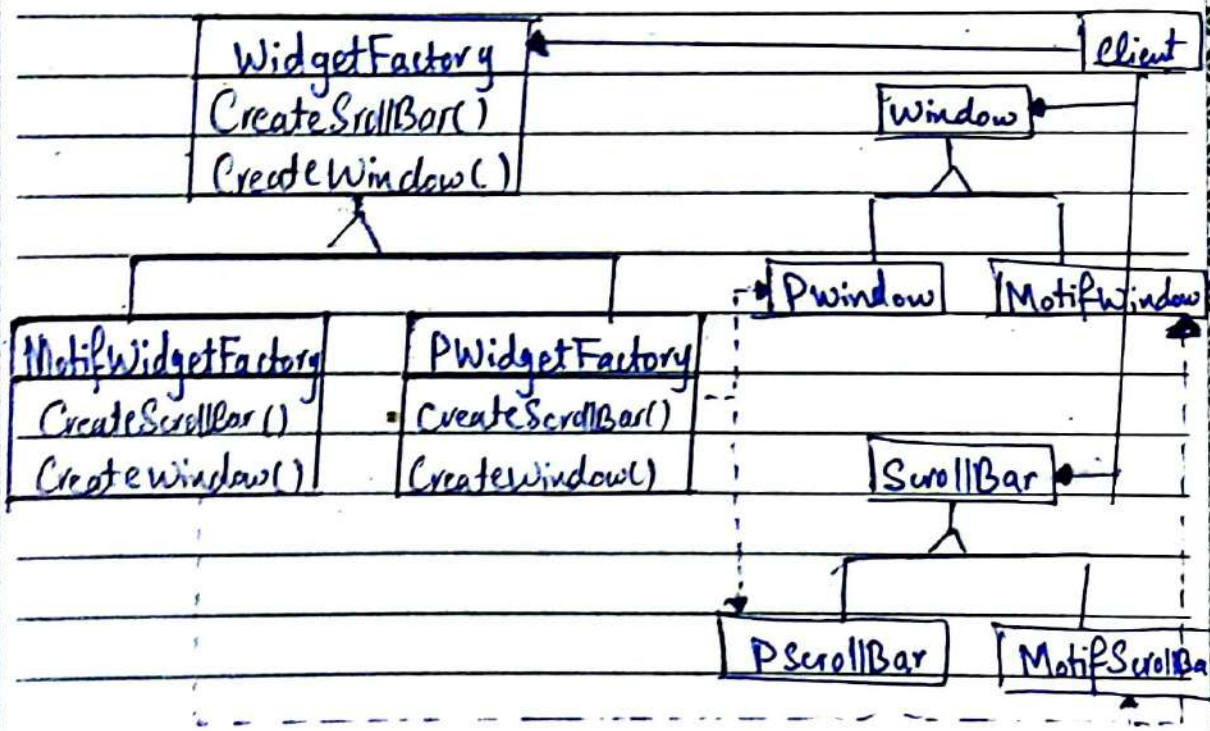
Date._____

abstract class, not a particular concrete class.

A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

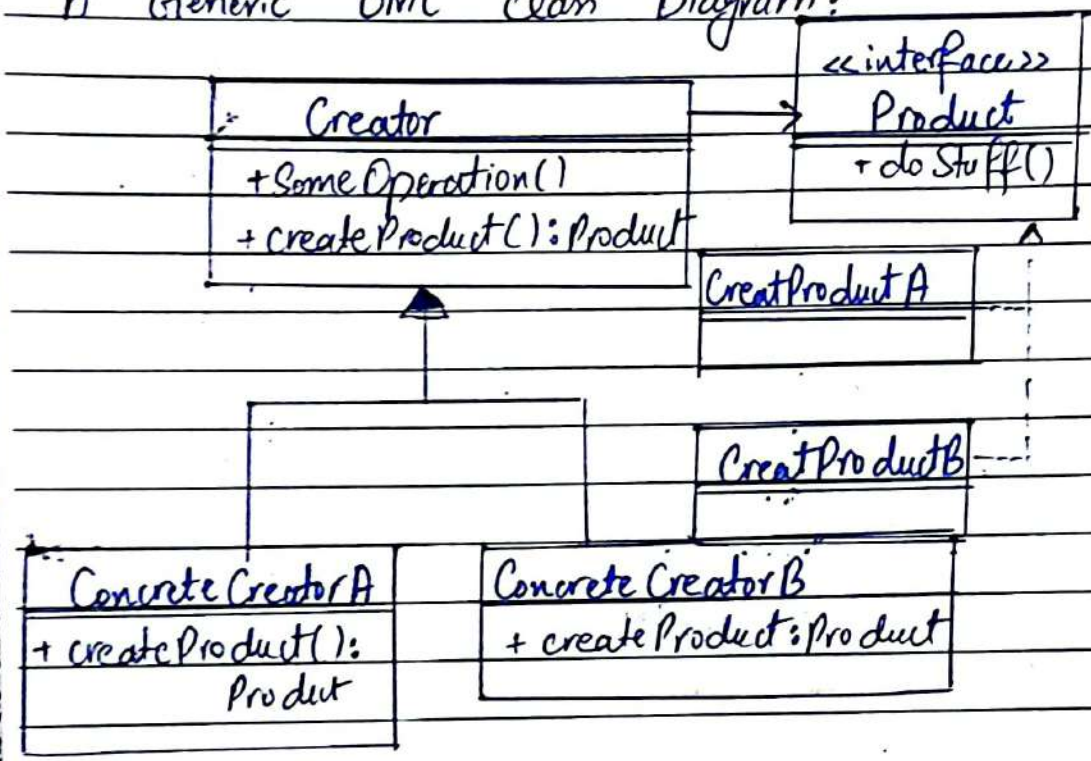
UML Class Diagram

UML Class Diagram of given Scenario:



Date._____

A Generic UML Class Diagram:



Java Code : Example 1

```
public interface MotorVehicle {
    void build();
}
public class Motorcycle implements MotorVehicle {
    @Override
    public void build() {
        System.out.println("Build Motorcycle");
    }
```

Date.

```
public class Car implements MotorVehicles {
  @Override
  public void build () {
    System.out.println ("Build Car");}.
}

public abstract class MotorVehicleFactory {
    public MotorVehicle create() {
      MotorVehicle = createMotorVehicle();
      vehicle.build();
      return vehicle;
    }
    protected abstract MotorVehicle createMotorVeclile();
}
// First, the Motorcycle factory class:
public class MotorcycleFactory extends MotorVehicleFactory
{
  @Overide
    protected MotorVehicle createMotorvehicle () {
      return new Motorcycle (); }}
// The CarFactory class
public class CarFactory extends MotorVeclicleFactory
{
  @Override
    protected MotorVeclile createMotorVeclile () {
      return new Car();
}
```

Date.

Java Code : Example 2

```
// Abstract Product interface
interface Shape {
    void draw(); }
// Concrete Products
class Circle implements Shape {
  @Override
    public void draw(){
        System.out.println("Drawing Circle");}
}
class Square implements Shape {
  @Override
    public void draw(){
        System.out.println("Square Drawing");}
}
interface ShapeFactory {        // Abstract Factory interface
    Shape createCircle();
    Shape createSquare();
}
// Concrete Factory implementing ShapeFactory
class SimpleShapeFactory implements ShapeFactory
{
  @Override
    public Shape createCircle(){
        return new Circle();}
}
```

Date._____

```
    @Override
    public Shape createSquare() {
        return new Square();
    }
}

// Client code using Abstract Factory
public class Client {
    public static void main(String[] args) {
        // Use the SimpleShapeFactory
        ShapeFactory factory = new SimpleShapeFactory();

        // Create a Circle
        Shape circle = factory.createCircle();
        circle.draw();

        // Create a Square
        Shape square = factory.createSquare();
        square.draw();
    }
}
```

Date.

Adapter Design Pattern

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

- Also known as Wrapper

Scenario

Consider a scenario in which there is an app that returns the top speed of luxury cars in miles per hour (MPH). Now we need to use some app for our client that wants the result but in kilometers per hour (KMPH).
To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:

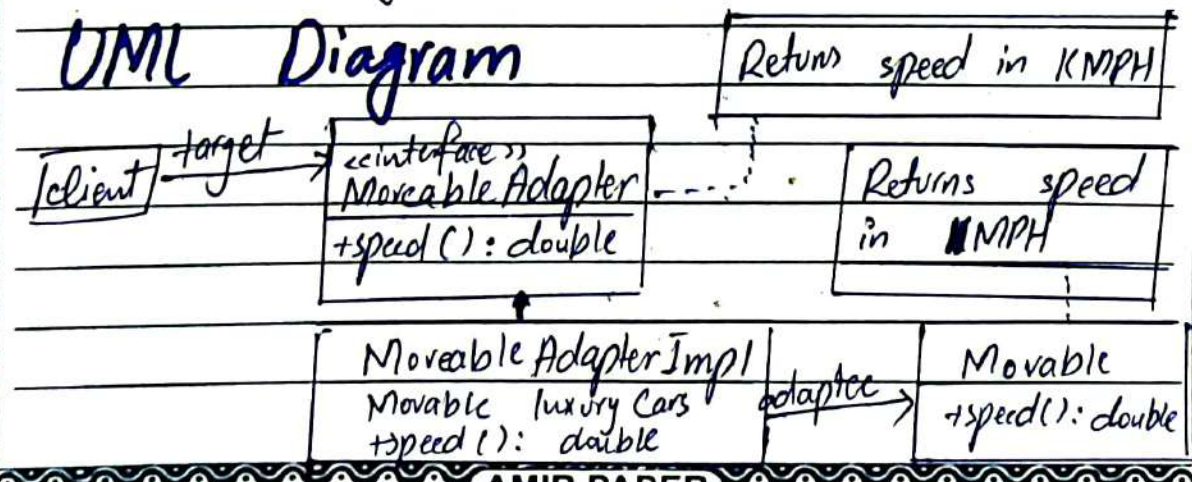
Date._____

First we will create the original interface **Movable** which is supposed to return the speed of luxury car in miles per hour (MPH): Then we will create one concrete implementation of this interface. Then we'll create an adapter **Movable Adapter** that will be based on the same Movable class. It may be slightly modified to yield different results in different scenarios.

The implementation of this interface will consist of private method **convertMPHtoKMPH()** that will be used for the conversion.

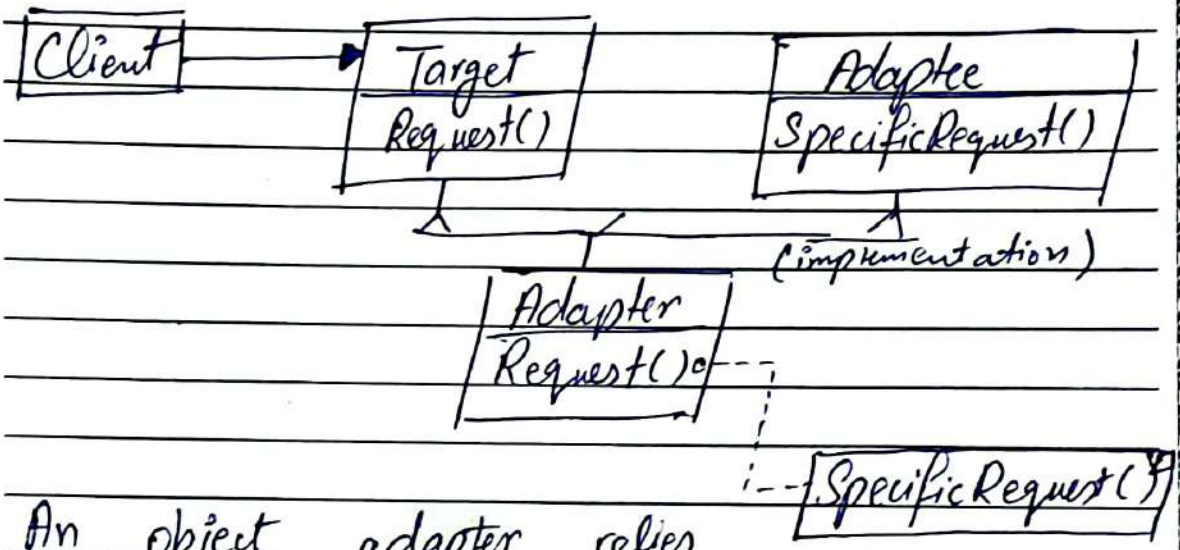
Now we'll only use the methods defined in our Adapter, and we'll get the converted speeds. In this case, the following assertion will be true.

UML Diagram

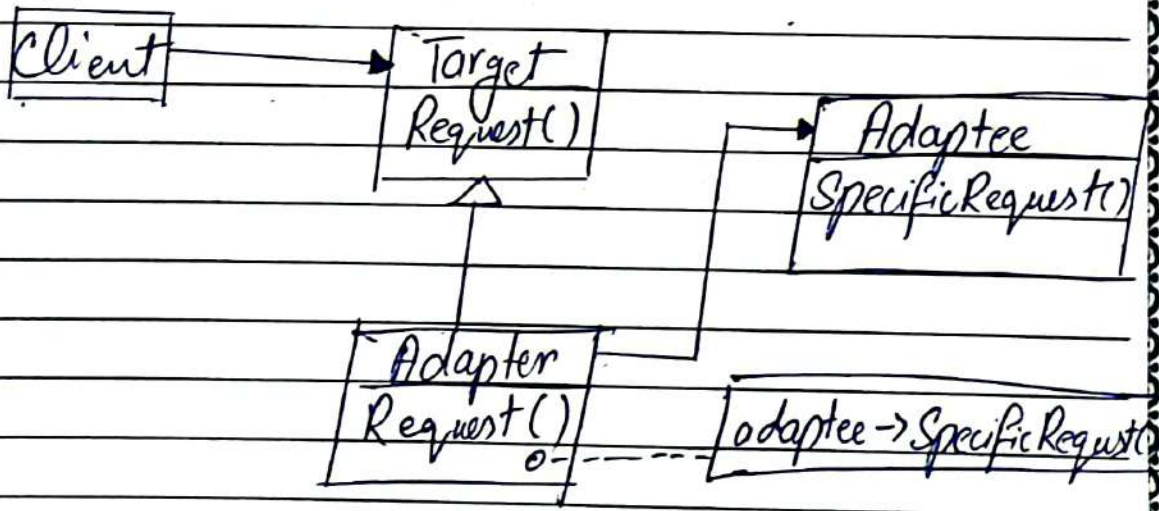


Date.

UML class diagram of Generic Scenario:



An object adapter relies on object composition:



Date.

Java Code : Example 1

```
// Java code example of scenario
public interface Movable {
    //return speed in MPH
    double getSpeed();
}
public class Bmw implements Movable {
    @Override
    public double getSpeed(){
        return 268;
    }
}
public interface MovableAdapter {
    double getSpeed(); //return speed in KMPH
}
public class MovableAdapterImpl implements
MovableAdapter {
    private Movable luxuryCars;
    @Override
    public double getSpeed(){
        return convertMPHtoKMPH(luxuryCars.
            getSpeed());
    }
    private double convertMPHtoKMPH(double mph){
        return mph * 1.60934;
    }
```

Date._____

```
@Test
public void whenConvertingMPHTOKMPH() {
    Movable bmw = new BMW();
    MovableAdapter bmwAdapter = new MovableAdapterImpl(bmw);
    assertEquals(bmwAdapter.getSpeed(), 431.30312, 0.0001);
}
```

Java Code : Example 2

```
//FileCommiter.java
import ~~ant~~ java.io.File;
public class FileCommiter {
    String diskLocation;
    public FileCommiter(){
        //default constructor
    }
    public FileCommiter(String diskLocation){
        this.diskLocation = diskLocation;
    }
    public void saveFile(File file){
        //Logic for saving the file at the
        diskLocation goes here
        System.out.println("File saved at: "+
                                diskLocation);
    }
}
```

Date._____

```
// File Emailer.java
import java.io.File;
public class FileEmailer {
    public String emailAddress;
    public FileEmailer(String emailAddress) {
        this.emailAddress = emailAddress;
    }
    public void emailFile(File file) {
        System.out.println("File emailed to: " +
                           emailAddress);
    }
}

// Client.java
import java.io.Files;

public class Client {
    public static void main(String args[]) {
        File file = new File(args[0]);
        FileCommiter fileCommiter = new FileCommiter("C://
        fileCommiter.saveFile(file);

        fileCommiter = new FileEmailerAdapter(new
            FileEmailer("abc@gmail.com"));
        fileCommiter.saveFile(file); }
}
```

Date.

Bridge Design Pattern

Decouple an abstraction from its implementation so that the two can vary independently.

- Also known as Handle / Body.

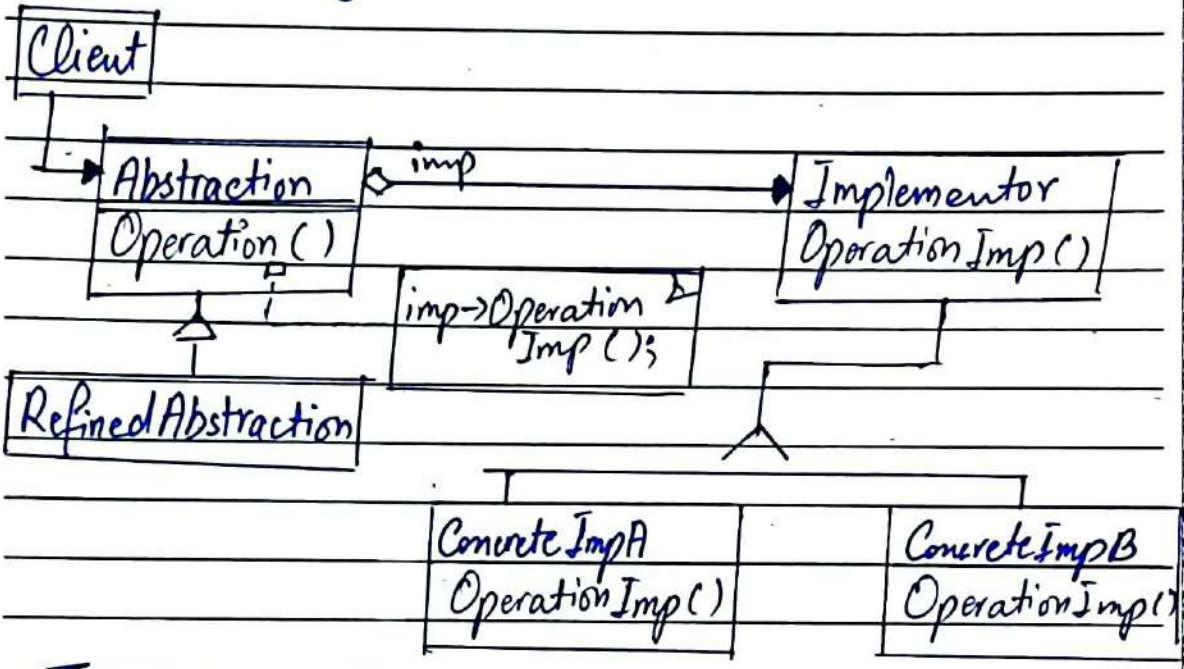
Scenario

Let say we have a Shape interface and two concrete classes that implement it: Circle and Square. We also have a Color interface and two concrete classes that implement it: Red and Blue. We want to be able to draw each shape in a different color. We can use Bride Pattern to solve this Problem.

To get Started we will first define the shape interface and Color interface. Then we will create concrete classes that implement the shap and Color interface and at the end we will test our implementation.

Date._____

UML Diagram



Java Code : Example 1

```
// Shape interface
public interface Shape {
    void draw();
}
// Color interface
public interface Color {
    void applyColor();
}
```

Date.

```
public class Circle implements Shape {
    private final Color color;
    public Circle (Color color) {
        this.color = color;
    }
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
        color.applyColor();
    }
}
public class Square implements Shape {
    private final Color color;
    public Square (Color color) {
        this.color = color;
    }
    @Override
    public void draw() {
        System.out.print("Drawing Square");
        color.applyColor();
    }
}
public class Red implements Color {
    @Override
    public void applyColor() {
        System.out.println("Red");
    }
}
```

Date.

```
public class Blue implements Color {
    @Override
    public void applyColor() {
        System.out.println("Blue");
    }
}
@Test
public class BridgePattern {
    public static void main(String[] args) {
        Shape circle = new Circle(new Red());
        circle.draw();
        Shape square = new Square(new
        square.draw();                Blue());
    }
}
```

Java Code : Example 2

```
// Vehicle Interface
public interface Vehicle {
    void manufacture();
}
// Workshop Interface
public interface Workshop {
    void work();
}
```

Date.

```
public class Car implements Vechile {
    private final Workshop workshop;
    public Car(Workshop workshop) {
        this.workshop = workshop;
    }
    @Override
    public void manufacture() {
        System.out.println("Manufacturing Car in");
        workshop.work();
    }
}

public class Bike implements Vechile {
    private final Workshop workshop;
    public Bike(Workshop workshop) {
        this.workshop = workshop;
    }
    @Override
    public void manufacture() {
        System.out.println("Manufacturing Bike in");
        workshop.work();
    }
}
```

Date.

```
public class PaintWorkshop implements Workshop {
    @Override
    public void work() {
        System.out.println("Paint Workshop");
    }
}
public class RepairWorkshop implements Workshop {
    @Override
    public void work() {
        System.out.println("Repair Workshop");
    }
}

public class BridgePattern {
    public static void main(String[] args) {
        Vechile car = new Car(new PaintWorkshop());
        car.manufacture();
        Vechile bike = new Bike(new RepairWorkshop());
        bike.manufacture();
    }
}

/* Output
Manufacturing Car in Paint Workshop
Manufacturing Bike in Repair Workshop
*/
```

Date.

Builder Design Pattern

Seperate the construction of a complex object from its representation so that the same construction process can create different representation.

Scenario:

A Scenario for builder design pattern can be, suppose a software application for a restraunt and you need to implement a system to create customized meals for customers. Each meal can have various component. However, not all customers want the same combination or preferences.

Some generic conditions for using builder design pattern can be:

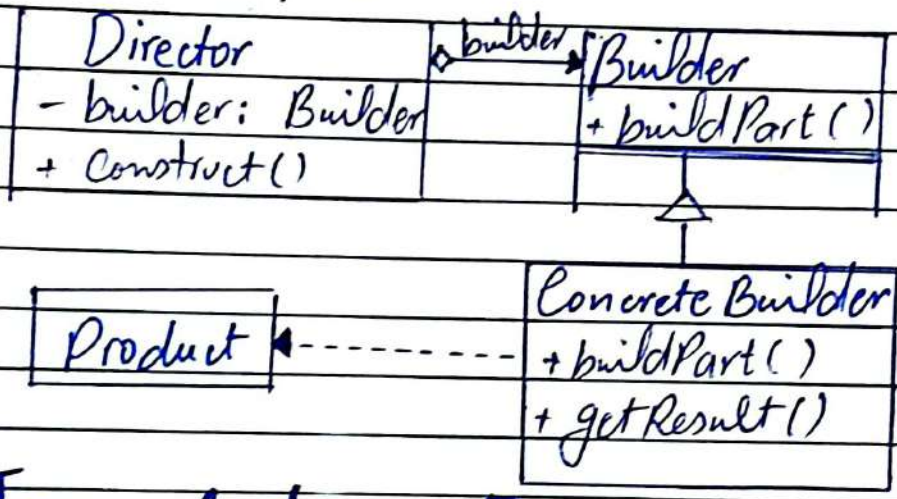
(1) Algorithm for creating the product needs to be independent from the sub-parts creation & assembly

(2) Different representation of the same

Date.

basic product.

UML Diagram



Java Code : Example 1

```
// Java code for scenario
public interface MealBuilder
    void buildMainCourse(String mainCourse);
    void buildSideDish(String sideDish);
    void buildDrink(String Drink);
    void buildDessert(String dessert);
    Meal getMeal();
}
public class MealDirector implements MealBuilder
    private Meal meal = new Meal();
    @Override
```

Date._____

```
public void buildSideDish (String sideDish){
    meal.setSideDish(sideDish);
}
@Override
public void buildMainCourse(String mainCourse){
    meal.setMainCourse(mainCourse);
}
// Same goes with remaining 2
// Product class with getters and setters
public class Meal{
    private String mainCourse, sideDish, drink,
        dessert;
    // Getter and Setters ... }
public class MealClient
    public static void main (String[] args){
        MealBuilder mealBuilder = new MealDirector();

        mealBuilder.buildMainCourse("Hamburger");
        mealBuilder.buildSidedish("Fries");
        mealBuilder:buildDrink("Coca-Cola");
        mealBuilder.buildDessert("Brownie");

        Meal nonVegetarianMeal = mealBuilder.get-
                                 Meal();
    }
}
```

Date.

Java Code : Example 2

```
public class Computer{
    private final String HDD, RAM; //required
    private final boolean GPU; //optional

    private Computer(ComputerBuilder builder){
        this.HDD = builder.HDD;
        this.RAM = builder.RAM;
        this.GPU = builder.GPU;}

    public static class ComputerBuilder{

        // Same goes here with variables
        public ComputerBuilder(String hdd,ram){
            this.HDD = hdd;
            this.RAM = ram;}
        public ComputerBuilder setGPU(boolean isGPU){
            this.isGPU = isGPU;
            return this;

            public Computer build(){
                return new Computer(this);}}

@Test
Computer computer = new Computer.ComputerBuilder
                .setGPU(true).build(); ("500","8"
```

Date.

Chain of Responsibility

Avoid coupling the sender of a request to its reciever by giving more than one object a chance to handle the request. Chain of recieving objects and pass the request along the chain until an object handles it.

Scenario / Usage

Here are some generic usages of chain of responsibility design pattern:

① The handler of the request is to be determind dynamically

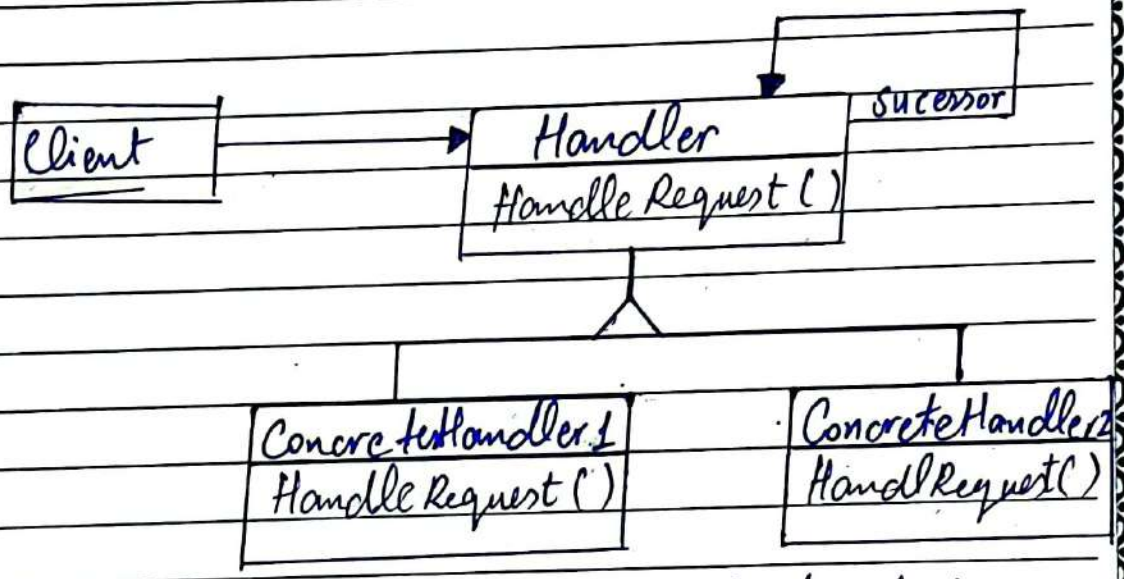
② The potential handlers for the request can be added when the system is running.

For example: In a hierarchical steup setup (like employee hierarchy in an organization) of handlers, the objects representing individual personnel at various levels in the hierarchy may

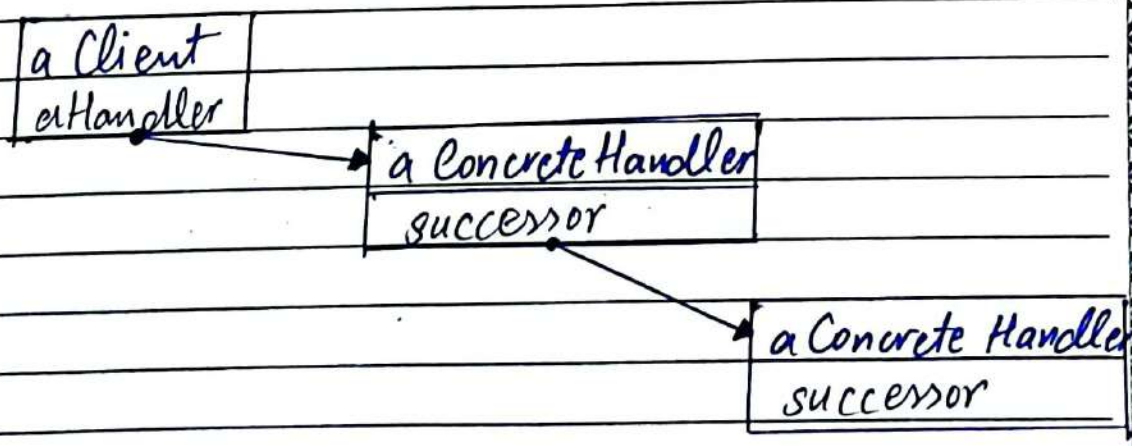
Date.

not be avaliable at system design time.

UML Diagram



A typical object structure might look like this:



Date.

Java Code : Example 1

```
public interface Logger {
    void setNext(Logger nextLogger);
    void logMessage(int level, String message);
}
public class ConsolLogger implements Logger {
    private Logger nextLogger;
    @Override
    public void setNext(Logger nextLogger) {
        this.Logger = nextLogger;
    }
    @Override
    public void logMessage(int level, String merage){
        if (level <=1) {
            System.out.println("Consol Logger: "+ merage);
        } else {
            nextLogger.logMerage(level, message);
        }
    }
}
public class FileLogger implements Logger {
    private Logger nextLogger;
    @Override
    public void setNext(Loggar nextLogger) {
        this.nextLogger = next.Logger;}
```

Date._____

```
@Override
public void logMessage(int level, String message){
    if(level <= 2){
        System.out.println("File Loggar:" + message);
    }else{
        next.Loggar.logMessage(level, message);
    }
}
}
public class ErrorLogger implements Logger{
    private Logger nextLogger;
    @Override
    // Same goes here but the condition
    // in logMessage is changed to
    // if(level <= 3), rest is the same
}

Logger consoleLogger = new Console logger();
Logger FileLogger = new FileLogger();
Logger errorLogger = new ErrorLogger();
ConsolLogger.setNext(FileLogger);
FileLogger.setNext(error Logger);

consoleLogger.logMessage(1, "This is consol msg");
ConsoleLogger.logMessage(2, "This is file msg");
consoleLogger.LogMessage(3, "This is error msg");
```

Date.

Java Code : Example 2

```
public interface Handler {
    void setNext(Handler nextHandler);
    void handleRequest(Request request);
}
public class AuthenticationHandler implements
    Handler {
    private Handler nextHandler;
    @Override
        public void setNext(Handler nextHandler){
            this.nextHandler = nextHandler;
        }
    @Override
    public void handleRequest(Request request) {
        if(request.getType().equals("Authentication")){
            System.out.println("Handling Authentication Request");
        }else {
            nextHandler.handleRequest(request);
        }
    }
}
public class AuthorizationHandler implements
    Handler {
    private Handler nextHandler;
```

Date.

```
@Override
public void handleRequest(Request request){
    if(request.getType().equals("Authorization"))
    {
        System.out.println("Handling Authorization Request");
    } else {
        nextHandler.handleRequest(request);
    }
}
@Override
private public void setNext(Handler nextHandler){
        this.nextHandler = nextHandler;
    }
} public class Request {
    private final String type;
    public Request(String type) { this.type = type; }
    public String getType() { return type; }
}

public class ChainofResponsibility {
    public static void main(String[] args) {
        Handler authenticationHandler = new A();
        Handler authorizationHandler = new A();
// new A(); => means constructor is called of that
        authenticationHandler.setNext(authorizationHandler);
authenticationHandler.handleRequest(new Request("Authe"));
authenticationHandler.handleRequest(new Request("Autho"));
}}
```

Date._____

Command Design Pattern

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

Also known as Action, Transaction

Scenario

Imagine that you are working on a new text-editor app. Your current task is to create a toolbar with bunch of buttons 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 dialogues.

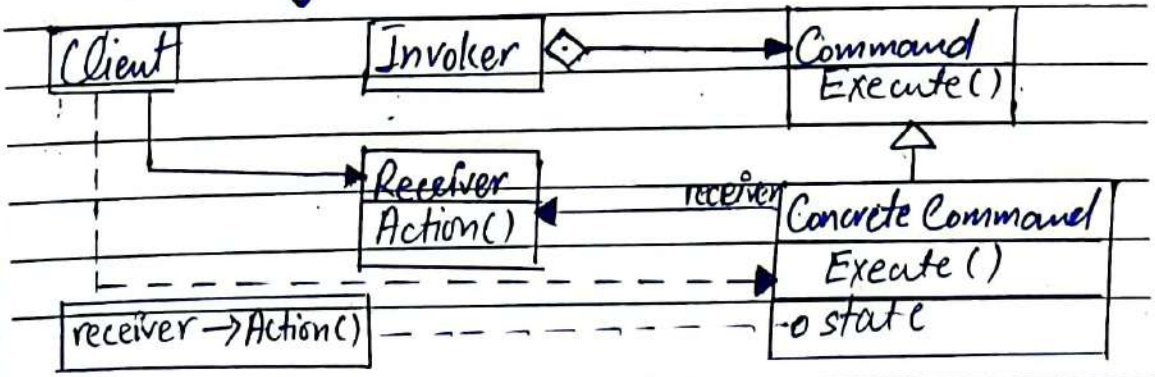
While all of these buttons look similar they are all supposed to do different things. ~~Where would~~ The simples solution is to create tons of subclasses for

Date._____

each place where the button is used. There subclasses would contain the code that would have to be executed on a button click. But this approach is deeply flawed.

This can be achieve more easily and effectively by using Command Pattern, after we use commad pattern we no longer need all those button and subclasses rather you put a single field into the base Button clas that stores a reference to command object and make the button execute that command on a click. The elements related to the same operations will be linked to the same commands, preventing any code duplication.

UML Diagram



Date.

Java Code: Example 1

```
public interface Command {
    void execute();
}
public class OpenTextFileCommand implements
    Command {
    private final Textfile textfile;
    public OpenTextFileCommand(Textfile textfile){
        this.textFile = textfile;
    }
    @Override
    public void execute(){
        textFile.open();
    }
}
public class TextFile {
    public void open(){
        System.out.println("Opening file ... ");
    }
    public void save(){
        System.out.println("Savinging file ... ");
    }
}
// U can implement as many as u can
public class TextFileApplication {
    private final Command openCommand;
```

Date.

```
    private final Command saveCommad;
    public TextFileApplication (Command open,
                                Command save)
        this.open = open;
        this.save = save;
    }
    public void clickOpen () { openCommand.execute(); }
    public void clickSave () { saveCommand.execute(); }
}
public class Client {
    public static void main (String [] args) {
        TextFile textfile = new TextFile();
        Command open = new OpenTextFileCommand(
        Command save = new OpenTextFileCommand(
                                    (textFile);

        TextFileApplication textFileApplication = new
        TextFileApplication (open, save);
        textFileApplication.clickOpen();
        textFileApplication.clickSave();
    }
}
```

Java Code: Example 2

```
// Java code for Light Switch
public interface Command {
    void execute();
}
```

Date._____

```
public class TurnOn implements Command {
    private final Light light;
    public TurnOn (Light light) {this.light = light;}
    @Override
    public void execute() { light.turnOn(); }
}
public class TurnOff implements Command {
// Same goes for this one, it just turns off light
}
class public class Light {
    public void turnOn() {
        System.out.println("Light turned on.");}
    public void turnoff() {
        System.out.println("Light turned off.");}
}public class RemoteControl {
    private final Command turnOn;
    private final Command turnoff;
    public RemoteControl (Command turnOn, Command turnoff) {
        this.turnOn = turnOn;
        this.turnoff = turnoff;
    }
    public void presOn() { turnOn.execute; }
    public void presOff() { turnOff.execute; }

// Client is same as before, only the light switch objects are created to turn on or off.
```

Date.

Composite Design Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Scenario / Usage

Given below are two conditions which together must be application to justify the use of Composite pattern while creating a product -

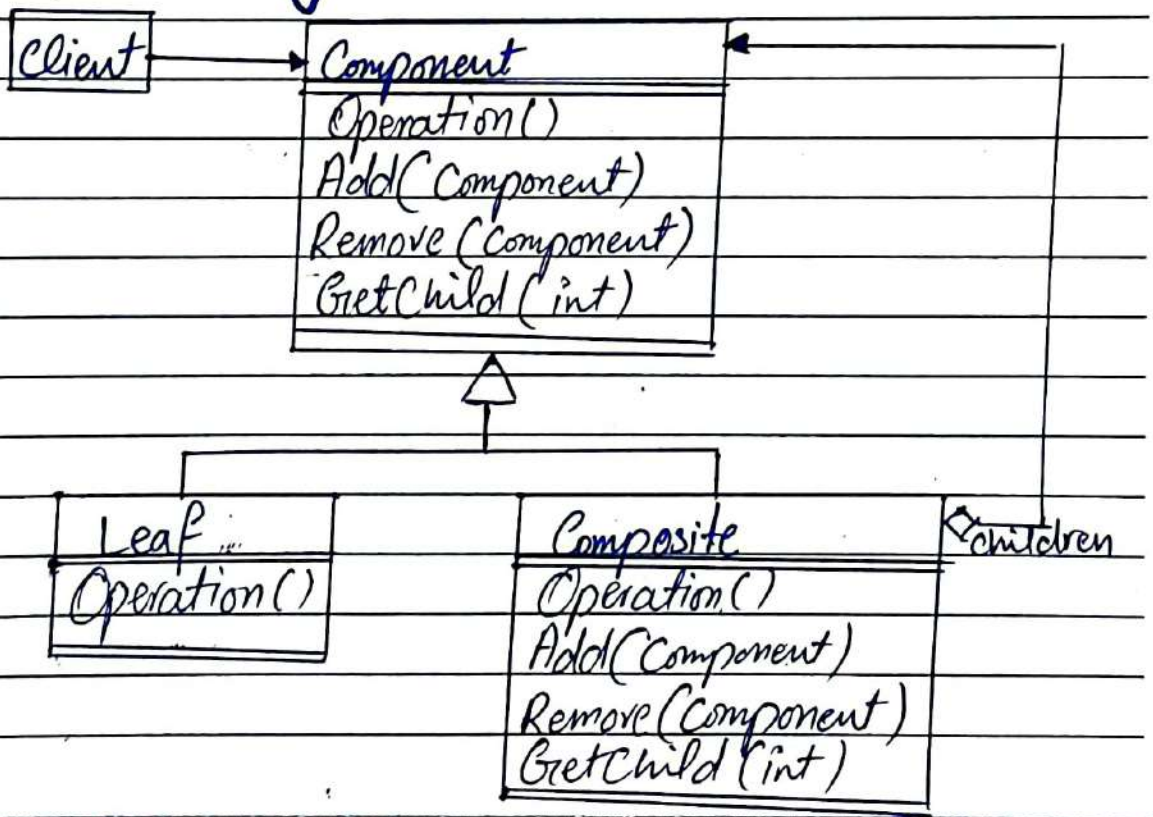
1. Part - whole hierarchy exists.
2. Same treatment required for part - whole objects : The system treats the objects representing the part and whole in the same way as it processes them.

Considering a hierarchy such as a binary tree where lowest level nodes point to the root of the sub-tree. At the next level multiple such sub-tree now become

Date._____

parts of the next level sub-tree's root. So, as we move up whole objects come together as part of the next level whole. This is hierarchical part-whole relationship. Since, the role the same individual node plays changes from part to whole - we inherently need to treat the node which plays both the same way aiding in their recursive traversal.

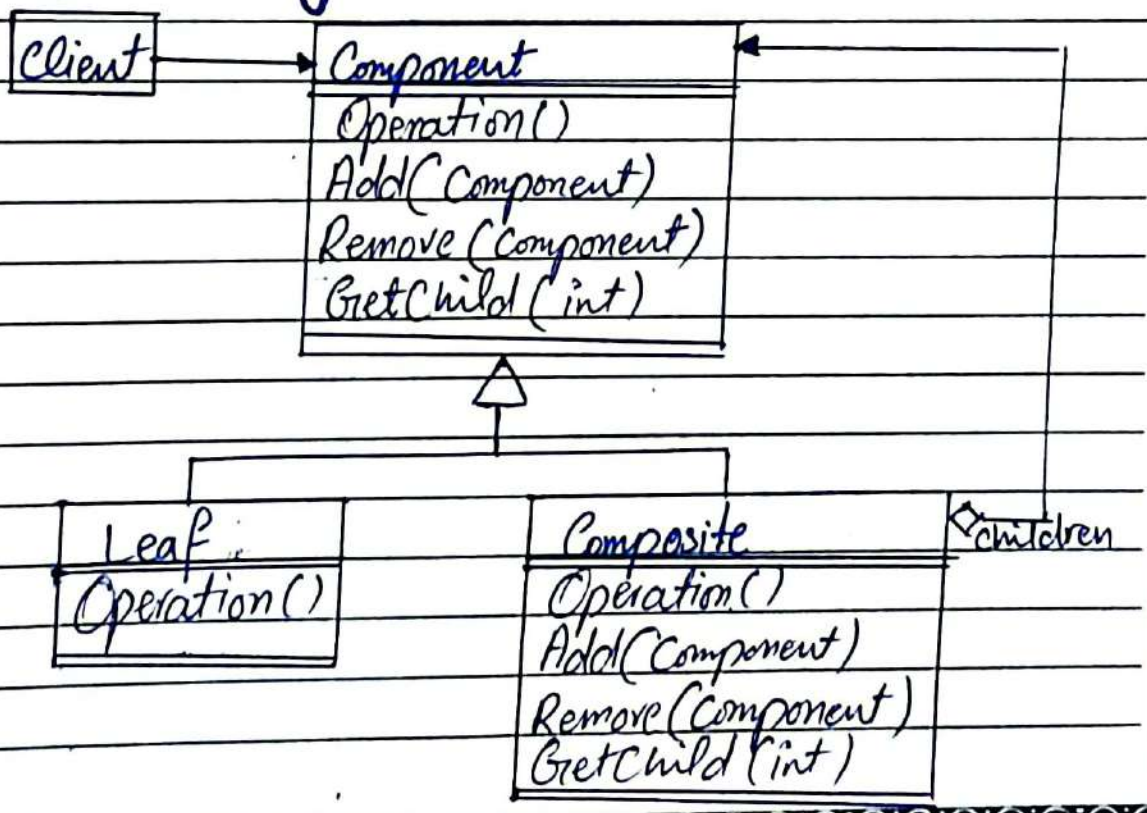
UML Diagram



Date.

parts of the next level sub-tree's root. So, as we move up whole objects come together as part of the next level whole. This is hierarchical part-whole relationship. Since, the role the same individual node plays changes from part to whole - we inherently need to treat the node which plays both the same way aiding in their recursive traversal.

UML Diagram



Date.

Java code : Example 1

```
public interface Department {
    void printDepartmentName();
}

public class FinancialDepartment implements
    Department {
    private Integer id;
    private String name;
    public void printDepName() {
        System.out.println(getClass().getSimpleName());
    }
    // Rest of the getters, setters...
}
public class ~~leaf class~~, SalesDepartment
    implements Department {
    private Integer id;
    private String name;
    public void printDepName() {
        System.out.println(getClass().getSimple
                           Name());
    }
    // Rest goes the same here
}
public class HeadDepartment implements
    Department {
    private Integer id;
    private String name;
    private List<Department> childDepartments;
```

Date._____

```
public HeadDepartment(Integer id, String name){
    // assign values to attributes
    this.childDepartments = new ArrayList<>();
}
public void printDepartmentName(){
    childDepartments.forEach(Department::
                                printDepName);
}
public void addDepartment(Department dep){
    childDepartments.add(dep);
}
public void removeDepartment(Department dep){
    childDepartments.remove(dep);
}
}
public class Composite {
    public static void main(String[] args){
        Department salesdep = new SaleDepartment
        (1, "Sales department");
        Department financialdep = new
        FinancialDepartment(2, "Financial Department");
        HedDepartment headdep = new HedDepartment
        (3, "Head department");
        headDepartment.addDepartment(salesDepartment);
        headDepartment.addDepartment(financialDepartment);
        headDepartment.printDepartment();
    }
```

Date.

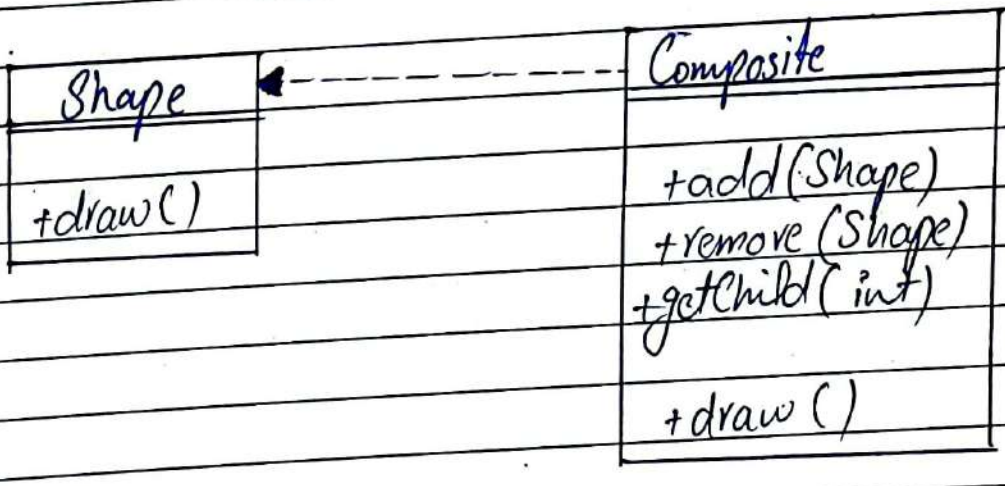
Java Code : Example 2

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

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Square");
    }
}

public class CompositeShape implements shape {
    private final List<Shap> childShape = new ArrayList<>();
    @Override
    public void draw() {
        childShapes.forEach(Shap::draw);
    }

    public void addShape(Shape shape) {
        childShapes.add(shape);
    }

    public void removeShape(Shape shape) {
    }
}
```

Date.

```
// Composite Class
public static void main(String[] args) {
    CompositeShape compositshape = new CompositeShape();
    Shape circle = new Circle();
    Shape square = new Square();

    compositeShape.addShape(circle);
    compositShape.addShape(square);

    compositeShape.draw();

}
```

UML Diagram of Example 2



Date.

Decorator Design Pattern

Attach additional responsibilities to an object dynamically. Decorator provide a flexible alternative to subclassing for extending functionality.

- Also known as Wrapper.

Scenario

Let's say we have a Shape interface and two concrete classes that implement it: Circle and Rectangle. We also have an abstract decorator class ShapeDecorator that implements the Shape interface and has a Shape object as its instance variable. RedShapeDecorator is a concrete class implementing ShapeDecorator.

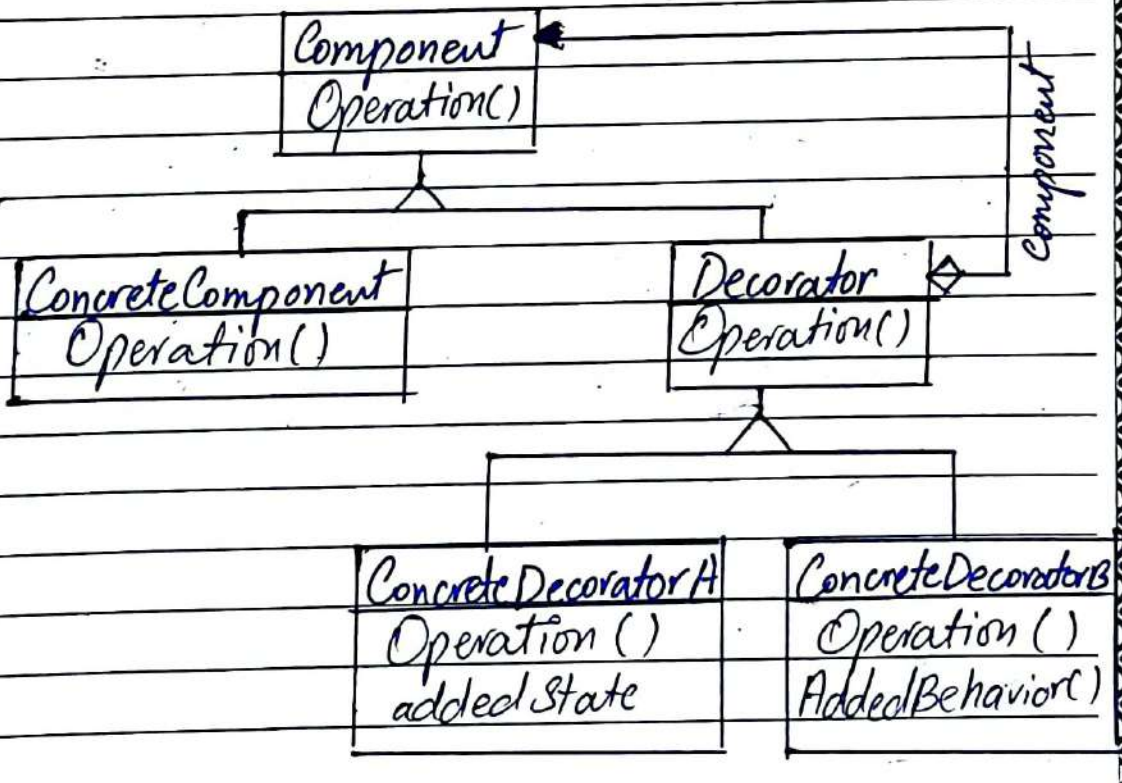
The Shape interface declares the interface for objects ShapeDecorator that can be decorated. The ShapeDecorator class implements the Shape interface and has a Shape object as its instance variable

Date.

The RedShapeDecorator class is concrete decorator class that extends the ShapeDecorator class and adds a red bordar to the shape.

The Decorator pattern can be used to add behavior to an individual object, either statically or dynamically, without affecting the behaviour of other objects from the same class.

UML Diagram



Date.

Java Code : Example 1

```
// Christmas Tree Decorator
public interface ChristmasTree {
    String decorator();
}
public class ChristmasTreeImpl implements
        ChristmasTree {
    @Override
    public String decorator() { return "Christmas"; }
}
public abstract class TreeDecorator implements
        ChristmasTree {
    private ChristmasTree tree;
    // standard constructor
    @Override
    public String decorator() {
        return tree.decorator();
    }
}
public class BubbleLights extends TreeDecorator {
    public BubbleLights(ChristmasTree tree) {
        super(tree); }
    public String decorate() {
        return super.decorator() +
            decorateWithBubbleLights();
    }
```

Date._____

```
    private String decorateWithBubbleLights() {
        return "with Bubble Lights";
    }
}

public void whenDecoratorInjectedAtRuntime() {
    ChristmasTree tree1 = new Garland(new
                        ChristmasTreeImpl());
    assertEquals(tree1.decorate(), "Christmas
                        tree with Garland");
    ChristmasTree tree2 = new BubbleLights(
        new Garland(new ChristmasTreeImpl()));
    assertEquals(tree2.decorate(), "Christmas
    tree with Garland with Bubble Lights");
}
```

Java Code : Example 2

```
// Java code of Scenario
public interface Shape {
    void draw();
}

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

Date.

```
public abstract class ShapeDecorator implements Shape{
    protected final Shape shape;
    public ShapeDecorator(Shape shape){
        this.shape = shape;}
    @Override
    public void draw(){ shape.draw();}}
public class RedShapeDecorator extends ShapeDecorator{
    public RedShapeDecorator(Shape shape){ super(shape);}
    @Override
    public void draw(){ shape.draw();
        setRedBorder(shape);}
    private void setRedBorder(Shape shape){
        System.out.println("Border color : Red");}}
public class DecoratorPattern {
    public static void main(String[] args){
        Shape circle = new Circle();
        Shape redCircle = new RedShapeDecorator(
                                new Circle());
        System.out.println("~~Circle~~ Normal Circle");
        circle.draw();

        System.out.println("Circle with red border");
        redCircle.draw();
    }}
```

Date._____

Facade Design 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.

Scenario / Usage

The basic usage of facade pattern is to solve a problem that is "Simplify access to a set of interfaces in a subsystem."

Facade solves this problem by providing a higher-level interface one level above all the sub-level interfaces. This higher-level interface then deal with the details and intricacies of the sub-system interface and at the same time provide a simplified interfaces to the clients of the subsystem.

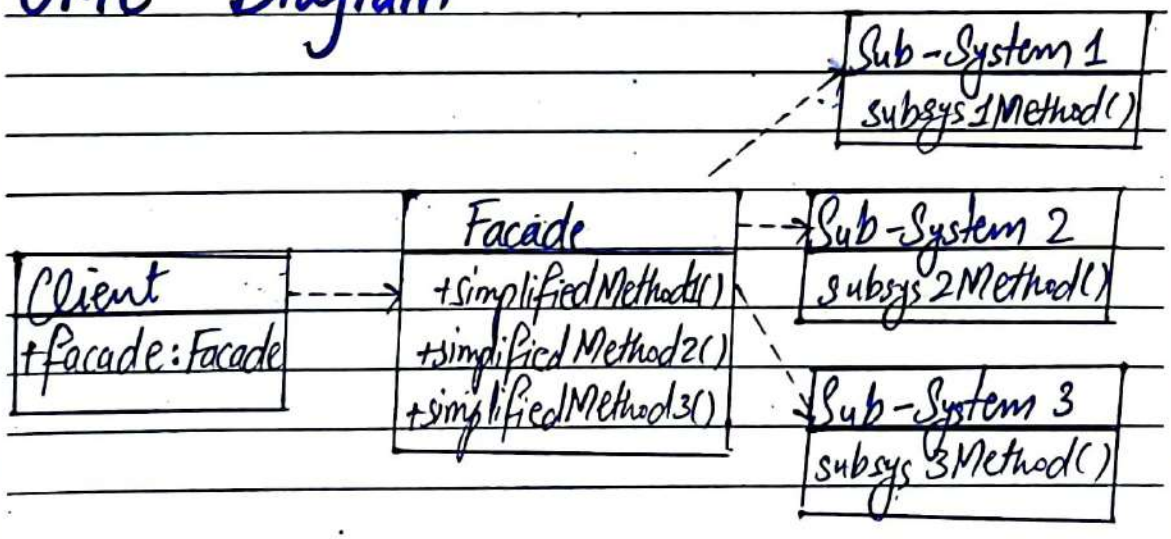
Facade pattern improves decoupling between the clients and subsystems as

Date._____

the higher-level interface can absorb any changes which might happen in the lower subsystems interfaces and provide a consistent interface to the clients. This also has a direct benefit of system portability as the client only design and code for the higher-level interface is defined by the Facade layer.

Layered subsystems can have a facade for each layer. Thus communication between layers of the subsystem is much simplified as it happens through the subsystem facades rather than directly deal with the complexities of the subsystem's classes.

UML Diagram



Date.

Java Code : Example 1

```
// Java code for lagacy system
airFlowController.takeAir
public class CarEnginFacad {
    private static int DEFAULT_COOLING_TEMP = 90;
    private static int MAX_ALLOWED_TEMP = 50;
    private FuelInjector fuelInjector = new FuelInjector();
    private AirFlowController airFlowController =
                    new AirFlowController();
    private // Rest of the existing members
    public void start() {
        fuelInjector.on();
        airFlowController.takeAir();
        FuelInjector.on();
        Fuel Injector.inject();
        starter.start();
    }

    public void stop() {
        fuelInjector.off();
        catalytic Converter.off();
        cooling Controller.cool(MAX_ALLOWED_TEMP);
        cooling Controller.stop();
        airFlow Controller.off();
    }
}
```

Date.

```
public class FacadePattern
    public static void main(String[] args){
        CarEnginFacade car = new CarEnginFacade();

        Car car = new Car();
        CarEnginFacade carFacade = new CarEnginFacade
        carEnginFacade.start(); // Starts the car(car);
        carEnginFacade.stop(); // Stops the car
    }
}
```

Java code : Example 2

```
public class CPU{
    public void freeze(){
        System.out.println("CPU is forzen.");}
    public void jump(long position){
        System.out.println("Jumping 2 position " +
                                position");
    }public void execute(){
        system.out.println("CPU is executing");}
}
public class Memory{
    public void load(long position, byte[] data){
        System.out.println("Loading data at position"
                                + position);
    }
}
```

Date.

```
public class HardDrive
    public byte [] read(long lba, int size){
        System.out.println("Reading" + size + "bytes
                "from LBA" + lba);
        return new byte[size]; }
}
public class ComputerFacade {
    private final CPU cpu;
    private final Memory memory;
    private final HardDrive hardDrive;
    // Default Constructor
    public void start(){
        cpu.freeze();
        memory.load(0, hardDrive.read(0, 1024));
        cpu.jump(0);
        cpu.execute();
    }
}

public static void main(String[] args){
    CPU cpu = new CPU();
    Memory memory = new Memory();
    HardDrive hardDrive = new HardDrive();
    ComputerFacade cf = new ComputerFacade(cpu,
                                memory, hardDrive);
    computer.start();
}
```

AMIR PAPER

Date.

Factory Method

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

- Also known as Virtual Constructor.

Scenario / Usage

Factory method design pattern creates objects in such a way that it lets the sub-class decide how to implement the object creation logic.

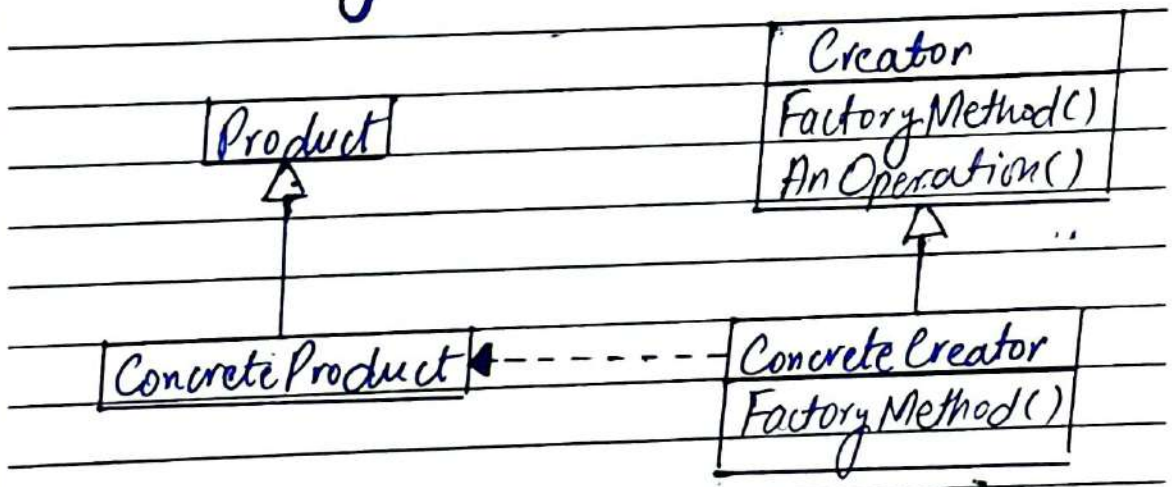
In the factory pattern, there is a base factory interface / base class which defines a common method for creating objects of subclasses. The actual logic for creation of different type of objects is present in the implementation / subclasses which determin how to implement the creation logic by overriding the pre-defined instances creation method.

Date.

Use the Factory Method pattern when:

1. A class can't anticipate the class of objects it must create.
2. A class wants its subclasses to specify the objects it creates.
3. Classes delegate responsibilities to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

UML Diagram



Java Code : Example 1

```
// ~~Image~~Shape Drawer using Factory Method
public interface Shape {
    void draw();
}
```

Date._____

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
public void class Square implement Shape {
@Override
    public void draw() {
        System.out.println("Drawing Square");
    }
public class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType.equals("Circle")) { return new Circle(); }
        else if (shapeType.equals("Square")) {
            return new Square(); }
        else
            return null;
    }
public class FactoryPattern {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
        Shape circle = shapeFactory.getShape("Circle");
        Shape Square = shapeFactory.getShape("Square");
        circle.draw();
        square.draw();
    }
}
```

Date.

Java Code : Example 2

```
public interface Document {    //Document.java
    void displayContents();
}
public class PDFDocument implements Document {
    @Override
    public void displayContents() {
        System.out.println("Displaying PDF");}
}
public class XMLDocument implements Document {
    @Override
    public void displayContents() {
        System.out.println("Displaying XML");}
}
public class DocumentFactory {
    public Document getInstance(String Document) {
        switch (doc Instance Identifier) {
        case "PDF":
            return new PDFDocument();
        case "XML":
            return new XMLDocument();
        }
    return null;}
}
```

Date.

```
// Client.java
public class Client {
    public static void main (String args[])
    {
        XMLDocument xmlDoc = (XMLDocument)
        new DocumentFactory().getInstance ("XML");

        PDFDocument pdfDoc = (PDFDocument)
        new DocumentFactory().getInstance ("PDF");

        // Display contents;
        xmlDoc.displayContents();
        pdfDoc.displayContents();
    }
}
```

Date._____

Flyweight Design Pattern

Use sharing to support large number of fine-grained objects efficiently.

Usage

The goal of the flyweight pattern is to reduce memory usage by sharing as much as data as possible, hence its a good basis for lossless algorithms for compression.

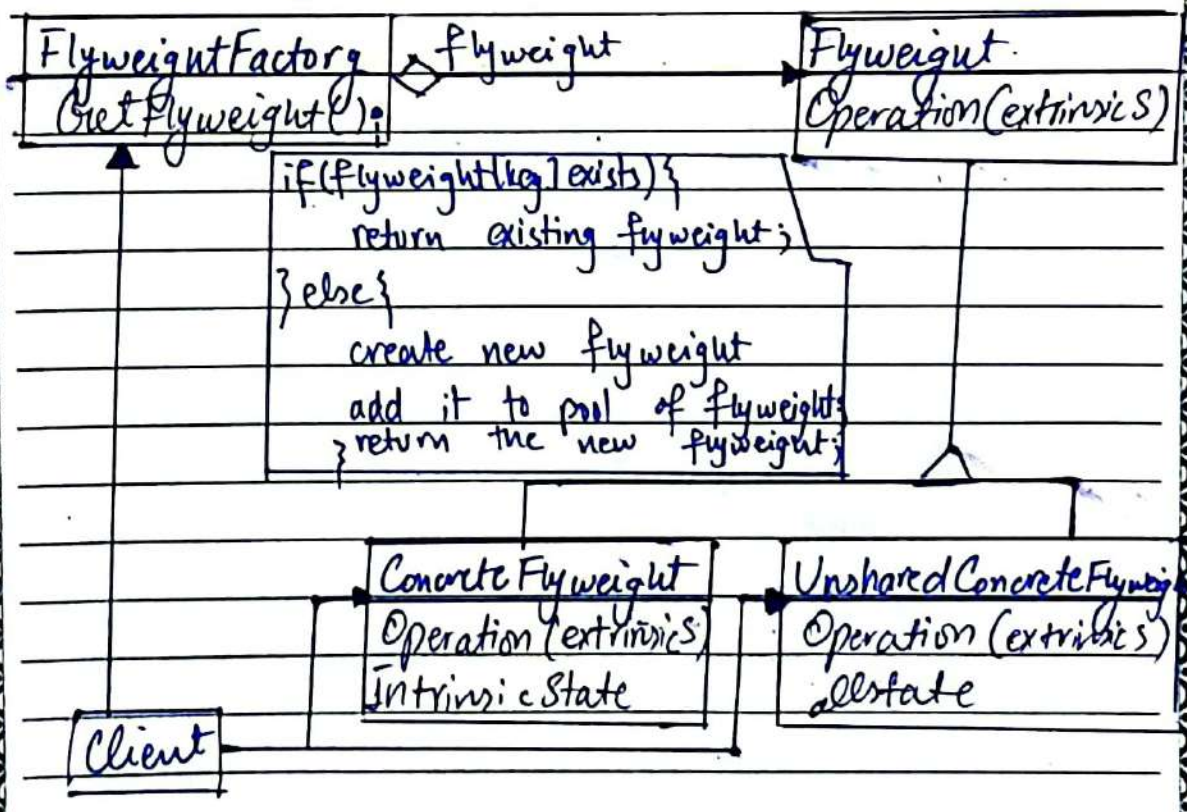
A classic example of this usage is in a word processor. Here, each character is flyweight object which shapes the data needed for the rendering. As a result, only the position of the character inside the document acts as a pointer with its extrinsic state being the context-dependent information.

Its very important that the flyweight objects are immutable: any operation on the state must be performed by factory.

Date.

The key concept here is the distinction between intrinsic and extrinsic state. Instrinsic state is stored in the flyweight; thereby making it sharcable. Extrinsic state depends on and varies with the flyweight's context and therefore making it shareable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared.

UML Diagram



Date.

Java Code : Example 1

```
public interface Shape {
    void draw(Graphics g, int x, int y, int width, int
                                    height, Color color);
}
public class Line implements Shape {
    public Line {
        System.out.println("Drawing Line");
        try {
            Thread.sleep(2000);
        }catch (InterruptException e){
            e.printStackTrace();
        }
    @Override
    public void draw (// same arguments) {
        line.setColor(color);
        line.drawLine (x1, y1, x2, y2);
    }
public class Oval implements Shape {
    private boolean fill;
    public Oval (boolean f){ this.fill = fill;
        System.out.println("Creating Oval with fill "+fill
    try {
        Thread.sleep(2000);
    }catch (InterruptedException e){
        e.printStackTrace();
    }
}
```

Date.

```
    @Override
    public void draw(// same as previous)
        circle.setColor(color);
        circle.drawOval(x, y, width, height);
        if (fill) {
            circle.fillOval(x, y, width, height);
        }
    }
}

// All of these ~~Fills~~ classes use these imports
import java.awt.Color;
import java.awt.Graphics;
```

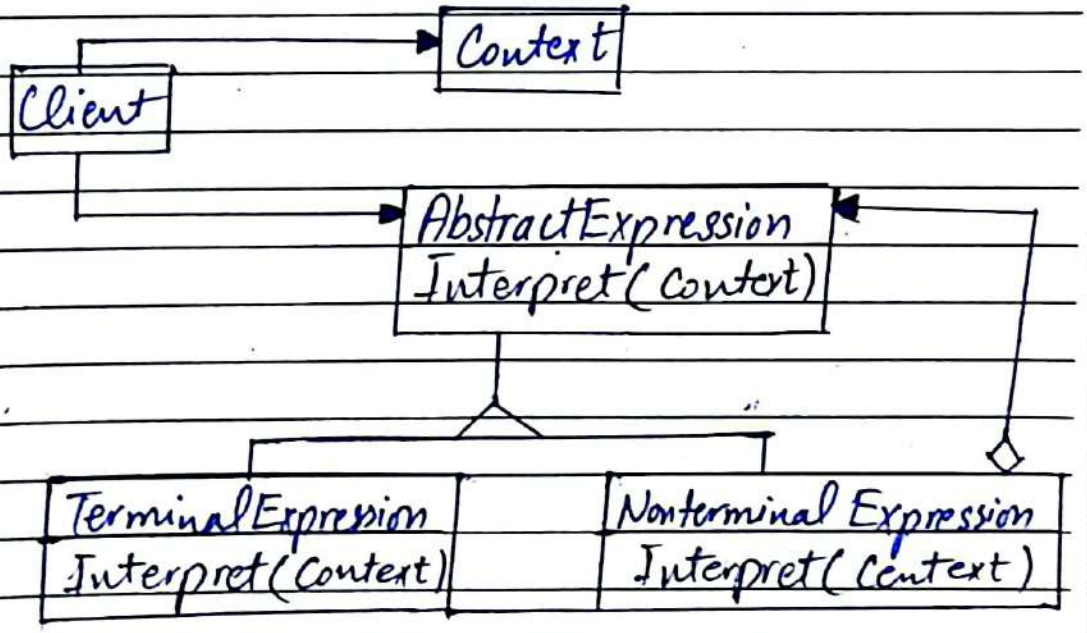
Java Code : Example 2

```
public interface Vehicle {
    public void start(); stop();
    public Color getColor();
}

public class Car implements Vehicle {
    private Engine engine;
    private Color color;
}

private static Map<Color, Vehicle>
    vehicleCache = new HashMap<>();
```

Date._____

```
public static Vechile createVechile(Color color){
    Vechile newVechile = vechiles Cache.
                computeIfAbsent(color, newColor ->{
    Engine newEngine = new Engine();
    return new(newEngine, new Color);
});
return newVechile;
}
```

UML Diagram of Interpreter Design Pattern



Date._____

Interpreter Design Pattern

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentence in the language.

Scenario / Usage

Consider a scenario searching for strings that match a pattern is a common Problem. Regular expressions are a standard language for specifying Patterns of strings. Rather than building custom algorithm to match each pattern against strings, search algorithm could interpret an regular algorithms that specifies a set of strings to match.

The interpreter pattern describes how to define a grammer for simple language represent sentences in a language, and interpret these sentences.

Date._____

Java Code : Example 1

```
import java.util.regex.Pattern
import java.util.Map;
public interface Expression {
    int interpret(Map<String, Expression> variables);
}

public class NumberExpression implements Expression{
    private final int number;
    public NumberExpression(int number) { // Default code }
    @Override
    public int interpret(Map<String, Expression var)
        return number;
    }
}

public class AditionExpression implements
    Expression {
    private final Expression leftExpression;
    private final Expression rightExpression;
    public AditionalExpression(leftExpression, rightExpression
    } // Default code }
    @Override
    public int interpret(Map<String, Expression> variables)
    { // Same code as before }
```

Date._____

```
public abstract class ExpressionEvaluator {
    private final Map<String, Expression> variables =
                                    new HashMap<>();
    public void setVariable(String name, If
        Expression expression) {
        variables.put(name, expression);
    }
    public int evaluate(Expression expression) {
        return expression.interpret(variables);
    }
}
```

Java Code : Example 2

```
// In this example only 2 new classes
// are introduced by the same method as
// before so no need to write whole code
public class NumberExpression { };
public class AdditionExpression { };
public abstract class ExpressionEvaluator {
    private final
    // Same code as previous
}
```

-> This is the extended version of Example
=> Example 2 has the same code as Example 1 only the classes names was change that are mentioned above.

Date.

Iterator Design Pattern

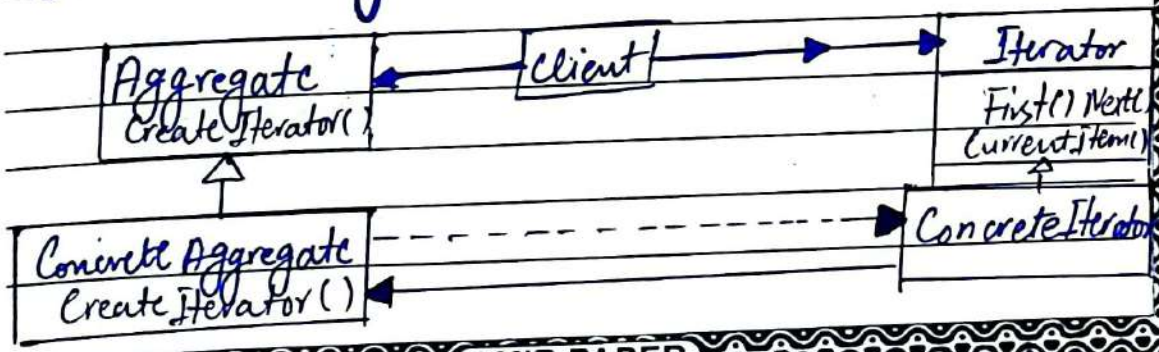
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Scenarios / Usage

Given below are the scenarios in which an iterator pattern is ideal ~~for~~ using:

1. Collection's internal implementation is to be hidden
2. Multiple way of traversing the collection are possible.
3. Iteration logic is decoupled from the collection's internal structure.

UML Diagram



Date.

Java Code : Example 1

```
java.util.Iterator
public class NameRepository implements Iterable<String>{
    private final String[] name = {"John", "Bob", "Alice"}
    @Override
    public Iterator<String> iterator(){
        return new NameIterator(); }
    private class NameIterator implements Iterator<String>{
    private int index;
    @Override
    public boolean hasNext(){
        return index < names.length;
    }
    @Override
    public String next(){
        if (hasNext()){
            return names[index++];
        }
        return null;
    }
}
} //Now we can use NameRepository class as
NameRepository nameRepo = new NameRepository();
for (String name: nameRepository){
    System.out.println(name);
}
```

Date._____

Java Code : Example 2

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Book {
    privat final String title, author, ISBN;
    public Book(String title, String author, String ISBN){
        // Default Constrator code }
    // Rest of the gettor settors
}
public class BookRepository implements Iterable{
    private final List<Book> book = new ~~Array~~{
        ArrayList<>();
    public void addBook(Book book){
        books.add(book);
    }
    @Override
    public Iterator<Book> iterator(){
        return new BookIterator();
    }
    private class BookIterator implements
Iterator<Book>{
    private int index;
    @Override
```

Date.

```
public boolean hasNext (){
    return index < booksize ();
}
@Override
public Book next (){
    if (hasNext ()){
        return book.get(index++);
    }
    return null;
}
}
// Now we can use BookRepository class
BookRepository bookrepo = new BookRepository ();
bookRepo.addBook(new Book("The Great Gatsby",
"F. Scott Fitzgerald" "978-0-684-80146-3"));
bookRepo.addBook(new Book("1984", "George
Orwell", "978-0-14-103614-4"));

for (Book book : booRepository){
    System.out.println(book.getTitle() + " by "
    + book.getAuther() + "(ISBN: + book.getISBN()
                        + ")");
}
```

Date._____

Mediator Design Pattern

Define an object that encapsulate how a set of object interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

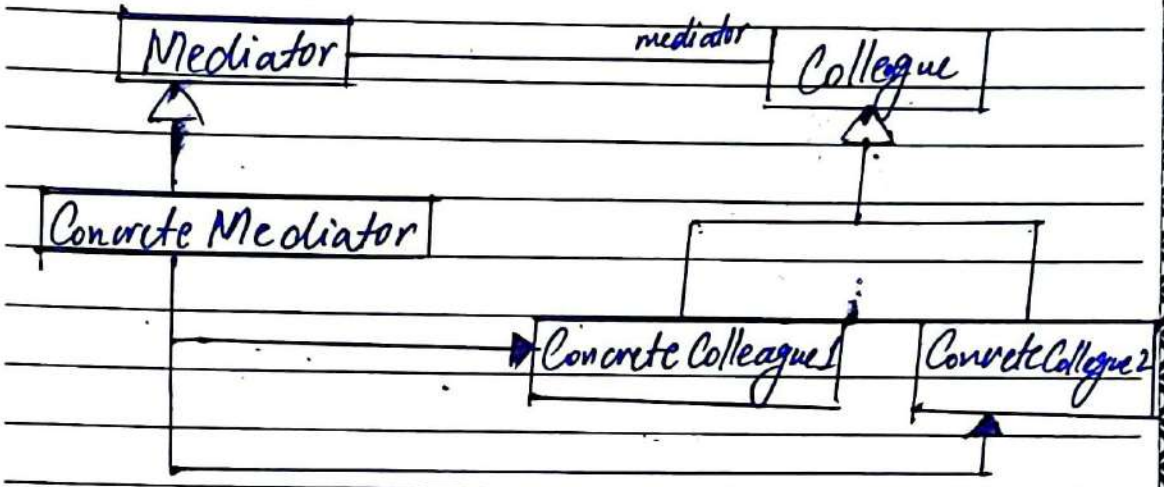
Scenario / Usage

The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other.
The usage of mediator are:

(1) When you want ~~the~~ loose coupling
(2) When you want interation of the classes/objects independently.
(3) Components depends only one one class instedd of being coupled to dozens of their colleagues.

Date.

UML Diagram



Java Code : Example 1

```
import java.util.ArrayList;
import java.util.List;
public class ChatRoom {
    private static final List<User> users = new
    public static void addUser(User user){ ArrayList<>();
        users.add(user);
    }
    public static void showMessage(User user,
    String message){
        for(User u : users){
            if(u != user){
                System.out.println(user.getName()+
    " sent Message :" +message)
            }
        }
    }
}
```

Date.

```
// Assuming that user class is already created
// we will be able to use ChatRoom class

User John = new User("John");
User mary = new User("Mary");
User Bob = new User("Bob");
ChatRoom.addUser(John);
ChatRoom.addUser(mary);
ChatRoom.addUser(Bob);

John.sendMessage("Hi everyone");
mary.sendMessage("Hello John!");
bob.sendMessage("what's up?");
```

Java Code: Example 2

```
// same import statement
interface AirTrafficControl {
    void registerFlight(Flight flight);
    void sendMessage(Flight sender, String message);
}

class ConcreteAirTrafficControl implements AirTrafficControl {
    private List<Flight> flights;
    public ConcreteAirTrafficControl() {
        this.flights = new ArrayList<>();
    }
```

Date.

```
@Override
public void registerFlight (Flight flight){
    flights.add(flight);
}
@Override
public void sendMessage (Flight sender, String msg){
    for (Flight flight : flights){
        if (flight != sender){
            flight.recieveMessage(menage);}
    }}}

abstract class Flight {
    protected AirTrafficControl airTrafficControl;
    public Flight(AirTrafficControl airTrafficContro){
        // Default Constructor }
    abstract void sendMessage(String menage);
    abstract void recieveMessage(String menage);
}
class Airplane extend Flight {
    // Simple class with some override methods
}
public class MediatorAirTrafficControl{
    public static void main(String[] args){
    ConcreteAirTrafficControl atc = new ConcreteAirTrafficContro
    atc.registerFlight(airplane1); atc.register(airplane2);
    Airplane airplane1 = new Airplane("A123");
    Airplane airplane2 = new Airplane("A23");}
```

Date.

Memento Design Pattern

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Scenario / Usage

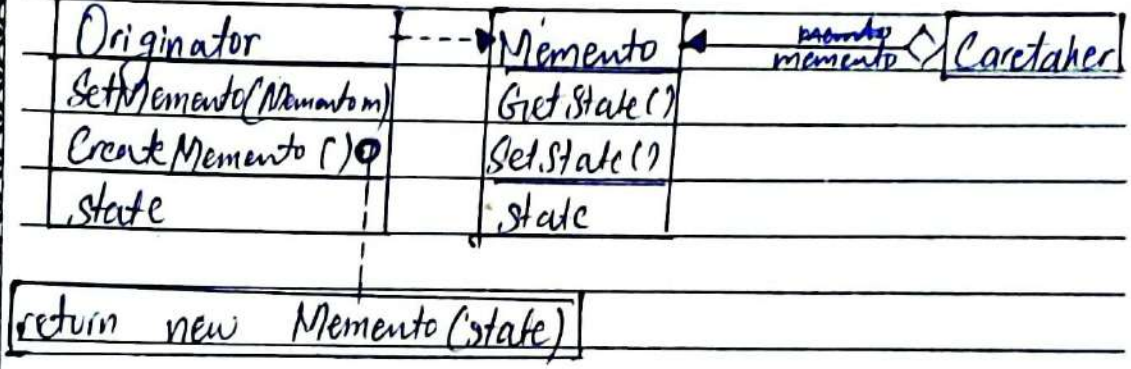
Memento design pattern can be chossen when the following 2 scenarios occurs:

① The state of an object in the system needs to be captured / saved so that object can be restored to that state later.

② Direct ~~class~~ access to objects state is not an option as that would break the encapsulation of object by exposing its internal structure.

Date.

UML Diagram



Java Code : Example 1

```
public class TextEditor {
    private String text;
    public void setText(String text) {
        this.text = text; }
    public Memento save() {
        return new Memento(text);
    }
    public void restore(Memento memento) {
        text = memento.getText(); }
}
public class Memento {
    private final String text;
    Public Memento(String text) { this.text = text; }
    public String getText() { return text; }
}
```

Date.

```
TextEditor textEditor = new TextEditor();
textEditor.setText("Hello!");
Memento memento = textEditor.save();
textEditor.setText("Goodbye");
System.out.println(textEditor.getText());
textEditor.restore(memento);
System.out.println("textEditor.getText());
```

Java Code : Example 2

```
class Memento {
    private String state;
    public Memento(String s) { this.state = s; }
    public String getState() { return state; }
}

class Originator { private String state;
    // Default Constructor and function getter
    public void getStateFromMemento(Memento memento)
    { state = memento.getState(); }
}

// Main class function
    Originator org = new Originator();
    org.setState("State #1");
    Memento saveState = org.saveStateFromMemento();
    org.setState("State #2");
    System.out.println("Current state: " + org.getState());
}
```

Date.

Observer Design Pattern

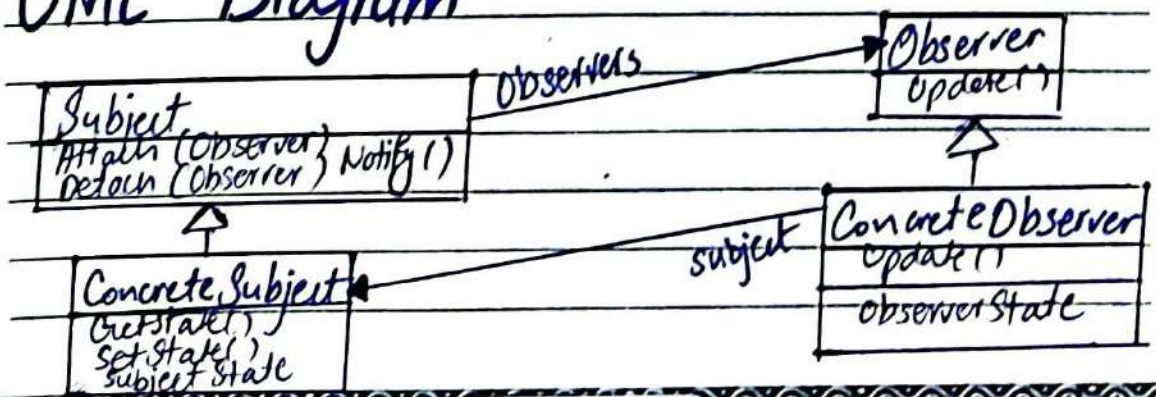
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Scenario / Usage

Scenario in which Observer Pattern can be used

1. Change of an object requires changes to multiple others
2. Notification capability needed without tight rules regarding how this notification is used

UML Diagram



Date.

Java Code : Example 1

```
import java.utils.Observable;
import java.utils.Observer;

class MyObservable extends Observable {
    private String state;
    public void setState(String state){ this.state = state;
        setChange();
        notifyObservers(state); }
}
class MyObserver implements Observer {
    public void update(Obserable obj, Object arg){
        System.out.println("State changed: "+ args); }
}
public class Main {
    public static void main(String[] args){
        MyObserable obs = new MyObserable();
        MyObserver ob = new MyObserver();

        obs~~erable~~.addObserver(ob);
        obs.setState("State#1");
}
```

Java Code : Example 2

```
import java.util.ArrayLists;
import java.util.List;
```

Date._____

```
clan MyObserable {
    private List<Observer> observers = new ArrayList<>();
    private String state;
    public void addObserver(Observer observer){
        observers.add(observers);
    }
    public void setState(String state){ this.state = state;
        for(Observer observer){
            observer.update(state); } }
public interface Observer {
    void update(String state); }
clan MyObserver implements Observer {
    public void update(String state){
        System.out.println("State changed: " + state);
    }
}
public clan Main {
    public static void main(String[] args){
        MyObservable observable = new MyObserable();
        MyObserver observer = new MyObserver();

        overable.addObserver(observer);
        observable.setState("State #1");
    }
}
```

Date.

Prototype Design Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

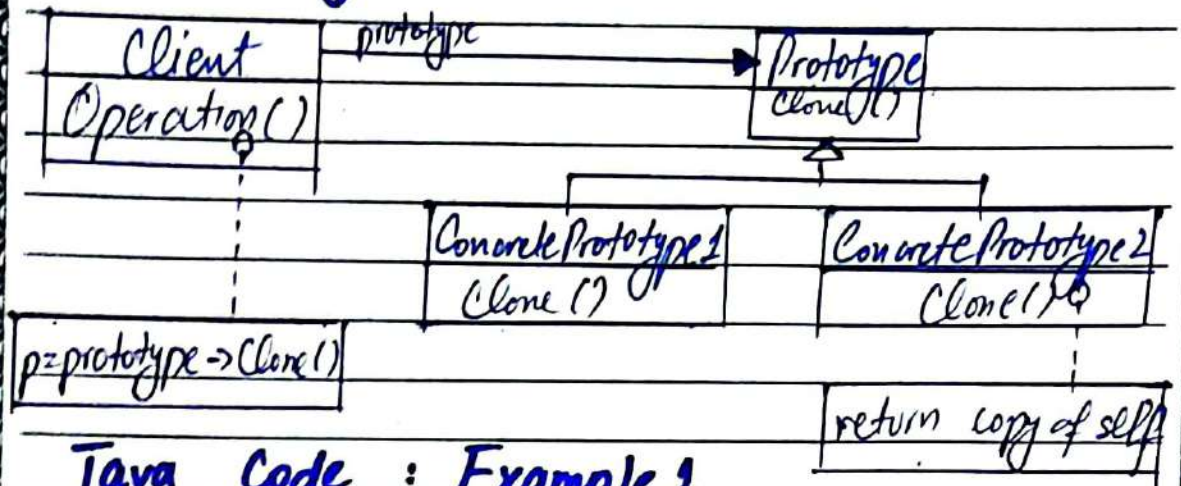
Scenario / Usage

Scenario in which Prototype design can be used:

1. Extract class to be instantiated is specified dynamically.
2. When Abstract Factory - like class hierarchy is not required
3. Object creation using "new" keyword is costlier
4. Object States can be pre-defined

Date._____

UML Diagram



Java Code : Example 1

```
abstract class Prototype implements Cloneable {
    abstract Prototype clone() throws CloneNotSupported;
}
class ConcretePrototype class extends Prototype {
    private String field;
    //Default Constructor
}
@Override
public Prototype clone() throws CloneNotSupported {
    return new ConcretePrototype(field);
}
@Override
public String toString() { return field; }
}
```

Date.

```
public class Main {
    public static void main (String [] args) {
        try {
            ConcreteProtoType pt = new ConcreteProtoType();
            CinrcleProtoType clone = (CastedProtoType)
                                    prototype.clone();
            System.out.println("Prototype " + prototype);
                                y      {" Clone " + clone};
        } catch (CloneNotSupportedE) {
            e.printStackTrace();
        }
    }
}
```

Java Code : Example 2

```
interface Shape extends Cloncable {
    Shape clone();
    void draw(); }
// class Circle implements Shape {
// Default Constructor and getter, setters }
// Main class
    Circle originalCirde = new Circle();
    Circle clonedCircle = (Circle)originalCircle.clone();
    originalCircle.draw();
    cloneCircle.draw(); }
}
```

Date.

Proxy Design Pattern

Provide a surrogate or placeholder for another object to control access to it.

Usage

① Remote Proxy: clients can access objects on a remote location as if they are co-located with them

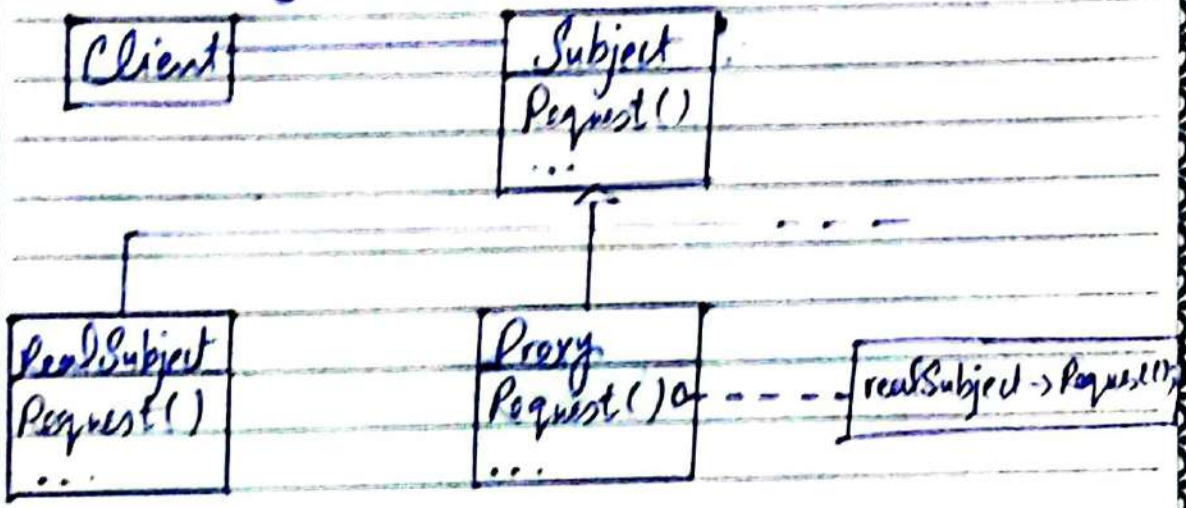
② Virtual Proxy: creates on instance of an expensive Object only on demand.

③ Protection Proxy: A protection proxy regulates access to the original object It is similar to authorization.

④ Smart Proxy: A smart reference proxy does additional actions when is accessed which typically include things like loading a persistent object into memory when its first referenced, laching of objects to avoid inconsistancies in data held by the objects

Date.

UML Diagram



Java Code : Example 1

```
public interface RealObject { void performOpe(); }
class ConcreteObject implements RealObject {
    public void performOpe() { performOpe
        System.out.println("Performing Operation...");
    }
}
class ProxyObject implements RealObject {
    private RealObject realObject;
    //Default Constructor
    public void performOpe() {
        System.out.println("Proxy Before operation");
        realObject.performOpe()
        System.out.println("Proxy After operation");
    }
}
```

Date.

```
// Main Class
    RealObject realObject = new CreateObject();
    RealObject proxyObject = new ProxyObject();

    proxyObject.performOpe();
}
}
```

Java Code : Example 2

```
interface Image { void display(); }
class RealImage implements Image {
    // All code as before only the names are
    // changed of methods and variables
}
class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;
    // Rest are the same, same logic different names
}
public class Proxy {
    public static void main(String[] args) {
        ~~Image.display();~~
        Image image = new ProxyImage("img.jpg");
        image.display();
        image.display();
    }
}
```

Date.

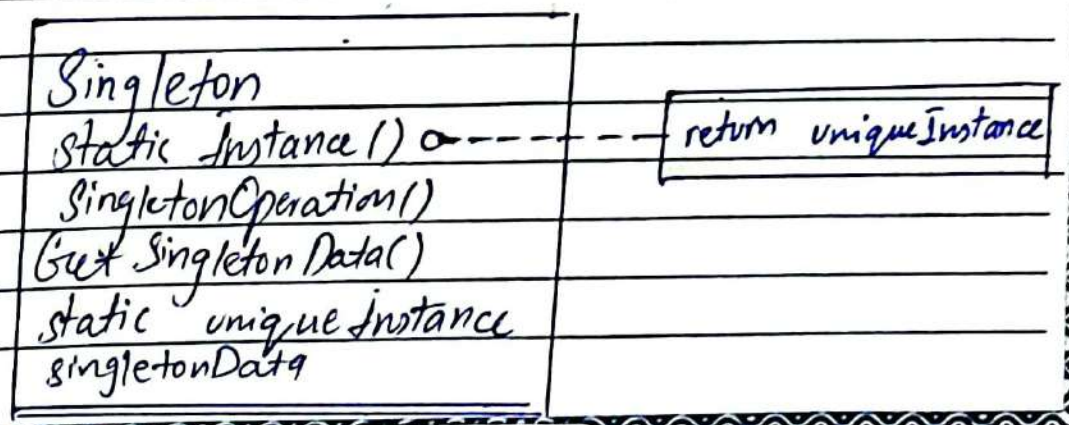
Singleton Design Pattern

(Allow an object to alter its). Ensure a class only has one instance, and provide a global point of access to it.

Usage / Scenario

The usage of singleton Pattern is:
① A class just have a single instance
② Provide a global access point to that instance

UML Diagram



Date.

Java Code : Example 1

```
class Sigleton {
    private static Singleton instance;
    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton s = Singleton.getInstance();
        System.out.println(s);
    }
}
```

Java code : Example 2

```
public class Singleton {
    private Singleton() { }
    private static class SingletonHelper {
        private static final Singleton INSTANCE
            = new Singleton();
    }
```

Date._____

```
public static Singleton getInstance()
{
    return SingletonHelper.INSTANCE;
}
public void showMessage(){
    System.out.println("Hello");}
}
public class Main {
public static void main(String[] args){

    Singleton singleton1 = singelton.getInstance();
    Singleton singleton2 = singlton.getInstance();

    System.out.println(singleton1 == singleton2);
    Singelton1.showMenage();}

}}
```

Date._____

State Design Pattern

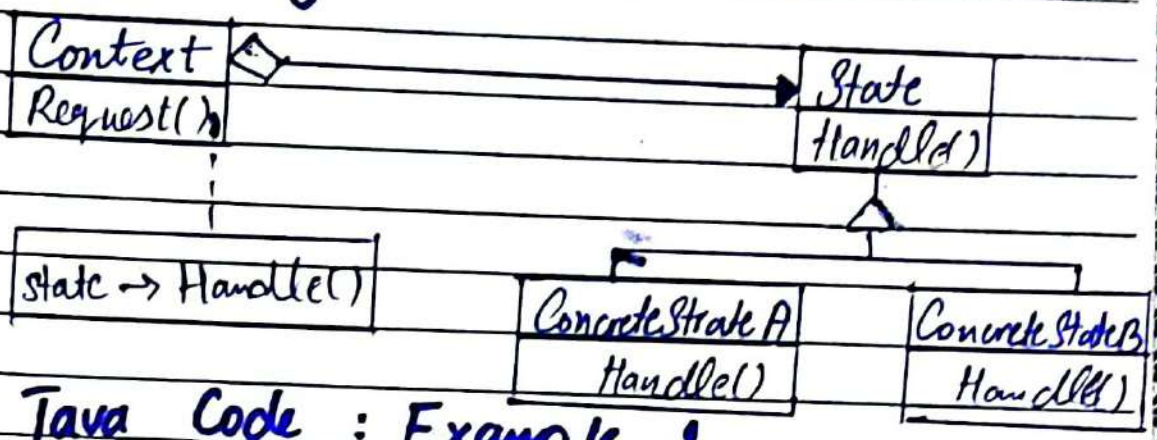
Allow an object to alter its behaviour when its internal state changes. The object will apear to change its class.

Scenario

If there are state-based scenario in a system where the different state of an object are implemented via multiple sets of conditional statements. In such scenario, instead of writing if-else blocks for each states and juggling with the various state dependent values, it becomes much easier to encapsulate each of these set of statements into seperate classes of their own. The implementation of each can thus vary independently of the other states.

Date.

UML Diagram



Java Code : Example 1

```
public interface State { void handleRequest(); }
class ConcreteStateA implements State {
    public void handleRequest() {
        System.out.println("Handle request in
                            state A");
    }
}
class ConcreteStateB implements State {
    // Same as stateA
}

public class Context {
    private State state;
    // Default Constructor, getter and
       setters
    public void request() {
        state.handleRequest(); }
}
```

Date.

```
public class Main
    public static void main(String[] args) {
        State stateA = new ConcreteStateA();
        State stateB = new ConcreteStateB();
        Context context = new Context(stateB);
        context.request();
        context.setState(state);
        context.request();
    }
}
```

Java Code : Example 2

```
public class Content {
    private State state;
    public Content() { state = new ConcreteStateA(); }
    public void setState(State state) {
        this.state = state;
    }
    public void request() {
        request.handle(this);
    }
}
interface State { void handle(Context context); }
```

Date._____

```
class ConcreteStateA implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling request in StateA");
        context.setState(new ConcreteStateB());
    }
}
class ConcreteStateB implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling request in StateB");
        context.setState(new ConcreteStateA());
    }
}
public class Main {
    public static void main(String[] args) {
        Context context = new Context();
        context.request(); //Request in StateA
        context.request(); //Request in StateB
    }
}
```

Date._____

Strategy Design Pattern

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

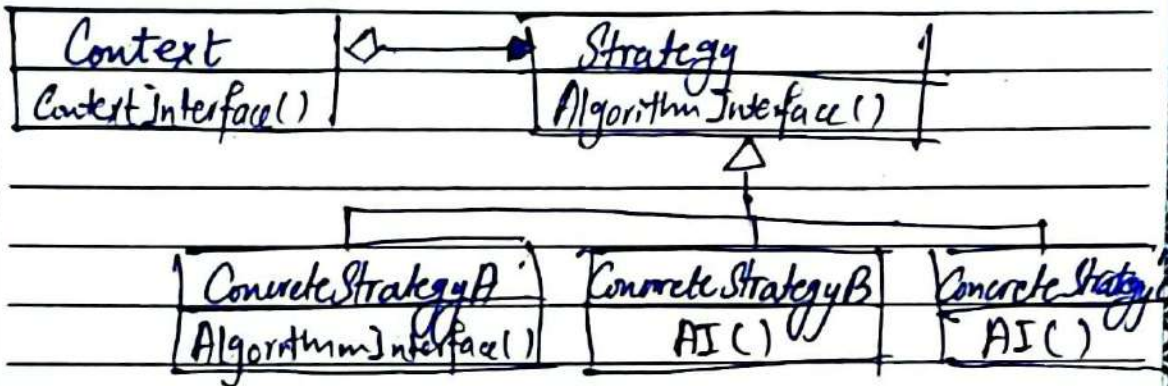
Usage

The usage of strategy design pattern is when there is a family of interchangeable algorithms which one algorithm is to be used depending on the other program execution context. Strategy Pattern achieves this by clearly defining the family of algorithms, encapsulating each one and finally making them interchangeable. Each of the algorithm in the family can be vary independently from the clients that use it.

Date.

UML Diagram

AJ -> Algorithm Interface



Java Code : Example 1

```
public interface Strategy {
    void executeStrategy();
}
class ConcreteStrategyA implements Strategy {
    public void executeStrategyA() {
        System.out.println("Executing Strategy A");
    }
}
class ConcreteStrategyB implements Strategy {
    // Same code as above with A -> B
}
class Context {
    private Strategy strategy;
    // Default Constructor, setter function
```

Date.

```
public void ereateStrategy(){
    strategy.ereateStrategy();
  }
}
public class Main {
  public class static void main(String[] args){
    Strategy strategyA = new ConcreteStrategyA();
    Strategy strategyB = new ConcreteStrategyB();

    Context context = new Contat(strategyA);
    context.execute.Strategy();
    contert.setStrategy(strategyB);
    context.execulte.Strategy();
  }
}
```

Java Code : Example 2

```
interface PaymentStrategy { void pay(int amount);
}
class CreditCardP implements PaymentStrategy {
  private String cardNumber;
  public CreditCardP(String cardNumber){
    this.cardNumber = cardNumber;
  }
  @Override
```

Date.

```
public void pay(int amount){
    System.out.println("Paid " + amount);
}
class PayPalP implements PaymentStrategy {
    private String email;

// Same code as CreditCardP but names are
// changed only
}
class ShoppingCart {
    private PaymentStrategy ps;
    public void setPaymentStrategy(PaymentStrategy ps)
    { this.ps = ps; }
    public void checkout(int amount){
        paymentStrategy.pay(amount);
    }
}
public class Main {
    public static void main(String[] args){
        ShoppingCart cart = new ShoppingCart();
        cart.setPayment(new CreditCardP("123456"
        cart.checkout(100);
        cart.setPayment(new PayPalP("john@gmail.com
        cart.checkout(50);
    }
}
```

AMIR PAPER

Date.

Template Method

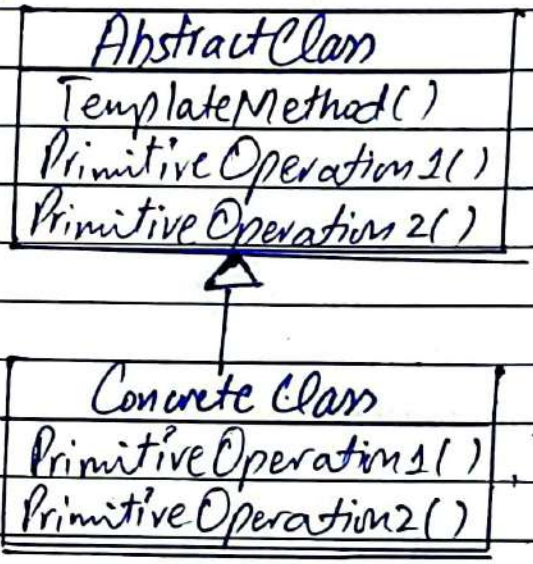
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefines certain steps of an algorithm without changing the algorithm's structure.

Usage

Template Method is used when the skeleton of an algorithm can be defined in terms of a series of operations, with few of these operations being redefined by sub-classes without changing the algorithm.

Date.

UML Diagram



Java Code : Example 1

```
abstract class AbstractClass {
    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
}
class ConcreteClass extends Abstraclass {
    void primitiveOpertion1() {
        System.out.println("Primitive operation 1");
    }
```

Date.

```
    void primitiveOperation2() {
        System.out.println("Primitive operation 2");
    }
}

public class Main {
    public static void main(String[] args) {
        AbstractClass abstractClass = new AbstractClass();
        abstractClass.templateMethod();
    }
}
```

Java Code : Example 2

```
abstract class Recipe {
    public final void cook() {
        prepareIngredients();
        cookIngredients();
        serve();
    }
    protected abstract void prepareIngredients();
    protected abstract void cookIngredients();
    private void serve() {
        System.out.println("Serve the dish");
    }
}
```

Date._____

```
class PastaRecipe extends Recipe {
    @Override
    protected void prepareIngredients() {
        System.out.println("Prepare pasta");
    }
    @Override
    protected void cookIngredients() {
        System.out.println("Boil pasta");
    }
class CakeRecipe extends Recipe {
    // Same code logic - only names are
    changed
}
public class Main {
    public static void main(String[] args) {
        Recipe pastaRecipe = new PastaRecipe();
        Recipe cakeRecipe = new CakeRecipe();

        System.out.println("Making pasta");
        pastaRecipe.cook();
        System.out.println("Making Cake");
        cakeRecipe.cook();
    }
}
```

Date.

Visitor Design Pattern

Represent an operation to be performed on the element of an object structure. Visitor let you defines a new operation without changing the classes of the elements on which it operates.

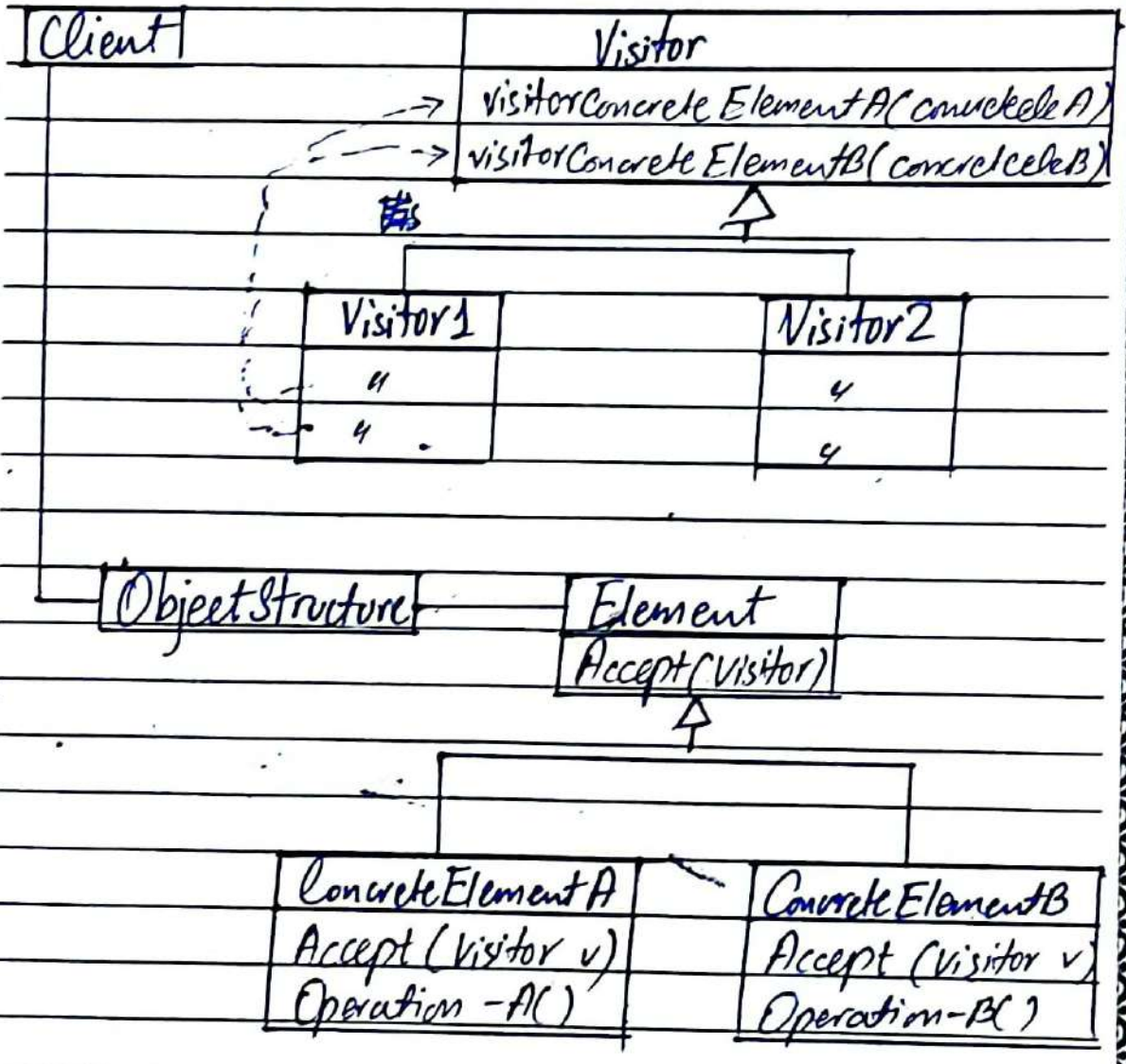
Usage

2 typical usage of visitor patters:
① Element Hierarchy is for elements which are to be worked upon.
② Visitor Hierarchy is for the visitors which operate on the Element Hierarchy.

=> Visitor & Element hierarchy(s) are independent of each other in definition and compilation. This allows us to add classes in Visitor hierarchy to apply operations in the element as they are traversed.

Date.

UML Diagram



Java Code : Example 1

```
interface Element {
    void accept(Visitor visitor);
}
```

Date._____

```
class ConcreteElement implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
interface Visitor {
    void visit(ConcreteElement element);
}
class ConcreteVisitor implements Visitor {
    public void visit(ConcreteElement element) {
        System.out.println("Visiting element");
    }
}
public class Main {
    public static void main(String[] args) {
        Element element = new ConcreteElement();
        Visitor visitor = new ConcreteVisitor();

        element.accept(visitor);
    }
}
```

Java Code : Example 2

```
interface Visitable {
    void accept(Visitor visitors);
}
```

Date.

```
class ConcreteElementA implements Visitable {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    void operationA() {
        System.out.println("Operation A on
    ConcreteElementA");
    }
}
class ConcreteElementB implements Visitable {
    @Override
    public void accept(Visitor visitor) {
        visitor.visitor(this);
    }
    void operationB();
        System.out.println("Operation B on
    ConcreteElementB");
    }
}
interface Visitor {
    void visit(ConcreteElementA elementA);
    void visit(ConcreteElementB elementB);
}
```

Date._____

```
class ConcreteVisitor implements Visitor {
    @Override
    public void visit(ConcreteElementA elementA){
        System.out.println("Visitor is performing
        operation on ConcreteElementA");
    }
    @Override
    public void visit(ConcreteElementB elementB){
        System.out.println("Visitior is performing
        operation on ConcreteElementB");
    }
}
public class Main {
    public static void main(String[] args){
        ConcreteElementA elementA = new
        ;
                                    ConcreteElement();
        ConcreteElementB elementB = new
                                    ConcretElement();
        Visitor visitor = new ConcreteVisitor();
        elementA.accept(visitor);
        elementB.accept(visitor);
    }
}
```

The End