Design pattern

Design patterns is a general, reusable solution to a commonly occuring problem within a given context in Software design.

Types of Design patt.

1) Creational pattern.

These patterns deal with the creation of object in a manner suitable to the situation.

2) structural patterns

These patterns provide a way to assemble objects into larger structure while considering issues as flexibility and reusability.

3) Behavioral patterns

These patterns identify common communication pattern between objects and provide reusable solution to interaction problems.

Benefits of design pattern.

-> Improve code Quality:

Design patterns can help developers with more robust, maitainable and reusable code.

-> Reduced Development Time:

By providing pre-defined solutions to common problems, design patterns can save developers time and efforts.

-> Enhanced Communication

Design pattern provide a shared vocabulary for developers to discuss and understand software Design.

Date Month Year

2. b) c. patterns

1) strategy

The strategy design pattern is a behavioral Software Design pattern that enable selecting an algorithm at run time.

Instead of implementing a single algorithm directly, code recieves run time instructions as to which in a Family of algorithms to use.

purpos of str

- Encapsolation of algorithm:

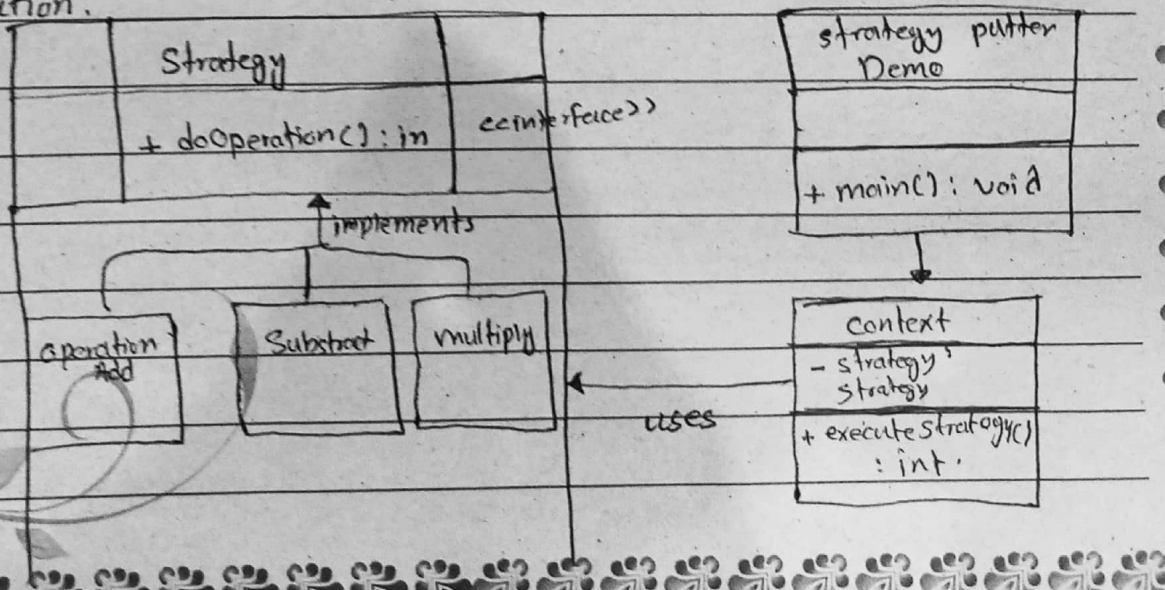
  By encapsolating algorithms into seperate classes, the strategy pattern makes it easier to manage and modify different algorithms.

- change algorithms at run time:

  The strategy pattern allow algorithms to be changed at run time, providing flexibility to adopt to different situation.

- promote code reuse:

  By Seperating algorithms from the main program logic, the strategy pattern promotes code reuse and reduce duplication.



Date Month Year

Step 1

Create an interface

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

Step 2

Create concret classess implementing the same interface.

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

OperationSubtract.java.

```
public class OperationSubstract implements Strategy {
    @Override
    public int doOperation(int num1, num2) {
        return num1 - num2;
    }
}
```

Operationmultiply.java

```
public class Operationmultiply implements Strategy {
    @Override
    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

Date Month Year

step 3

Create context class

```
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public int executeStrategy(int n1, int n2) {
        return strategy.doOperation(n1, n2);
    }
}
```

step 4

use the context to see change in behaviour when its changes its strategy.

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

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

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

2 Decorater Design patte

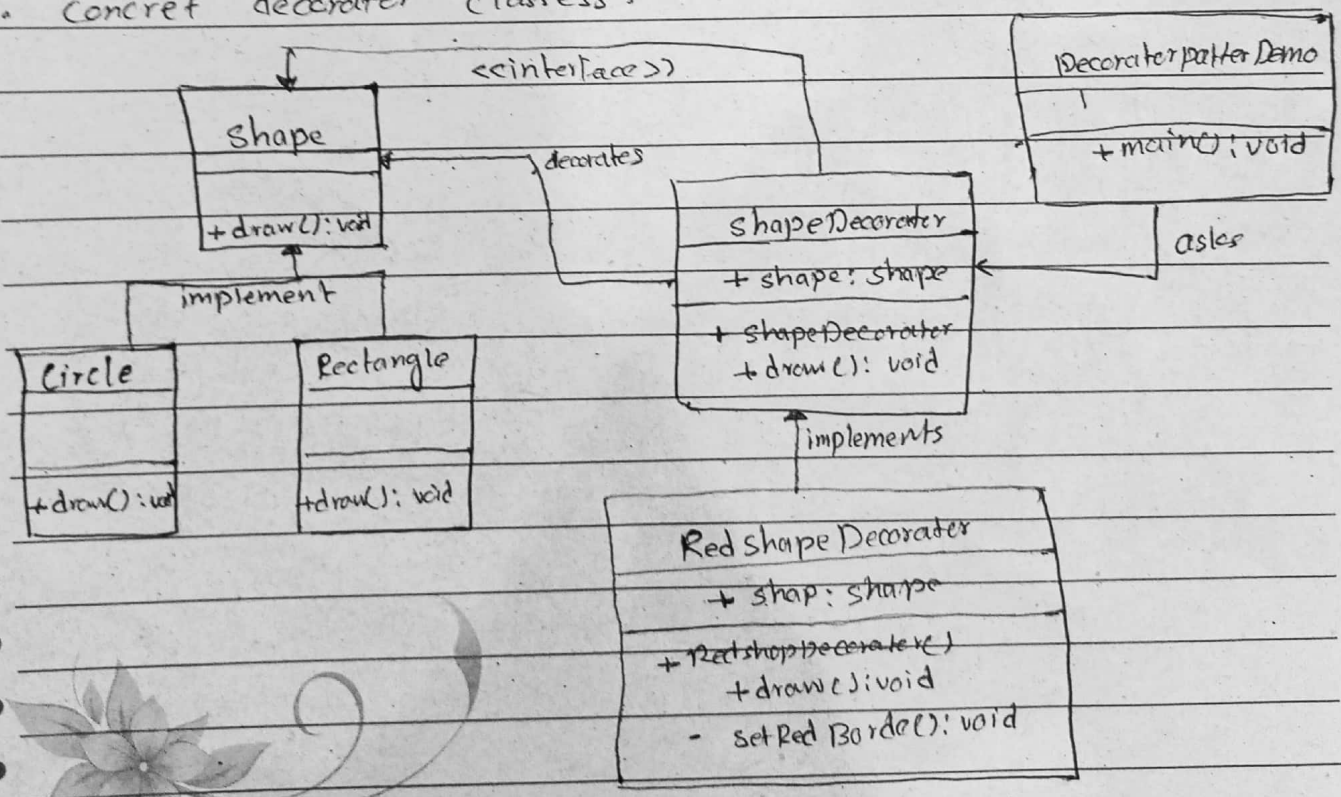
Decorater Design pattern is a structural design pattern that allow you to attach additional responsibilities to an object dynamically.

Purpose

- Add new funtionalities.
- promote code reuses
- Enhance flexibility

Componet of Decor pat

- component interface
- Concret components classes
- Decorter interface
- concret decorater classess.



Date Month Year

Step 1

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

Step 2

```
Public Class Rectangle implements shape {
    @override
    public void draw() {
        system.out.print("shap: Rectangle");
    }
}
```

Step 2

```
public class Circle implement shape {
    @override
    public void draw() {
        System.out.print("circle";
    }
}
```

Step 4

create abstract decorator class implementing the shap interface.

```
public abstract class ShapeDecorator implements shape {
    protected shap decoratedshape;
    public shapeDecorater(shap decoratershap) {
        this.decoratedshape = decoratedshape;
    }
    public void draw() {
        decoratedshape.draw();
    }
}
```

Create concrete decorator class extending the shapedecorator class.

```
public class RedshapeDecorator extends shapeDecorator {
    public RedshapDecorator (shape decoratedshape {
        super (decoratedshape);
    }

    @Override
    public void draw () {
        decoratedshap.draw();
        setRedBorder (decoratedshape);
    }
    private void setRedBorder (shap decoratedshape){
        System.out.print ("Bord color red");
    }
}
```

Step 5:

Use the RedshapDecorator to decorate shap objectr.

```
public class DecoratorpatternDemo {
    public static void main (strig[] args) {
        shap Circle = new Circle ();
        shape redCircle = new RedshapDecorator (new Circle())
        shap redRectangle = new RedshapDecorator (new Retang()) //
        systen.out ("Cirt with normal Border");
        Circle.draw();

        syster.out.print ("circle with of red bordr");
        redcircle.drawc();

        systen.out.pr ("rectangle of redBord")
        redRectangle.draw();
    }
```

Date Month Year

3. Factory Design pattern

The factory method design pattern is a creational design pattern that provides an interface for creating an object in a super class, but allow subclass to alter the type of objects that will be created.

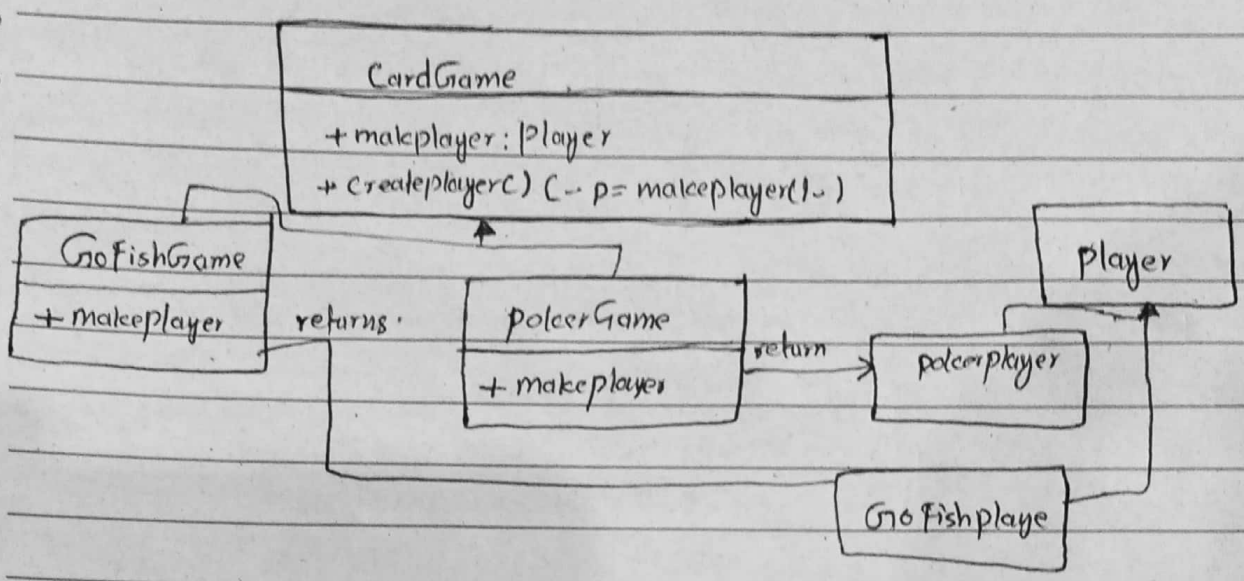
This is done by creating objects by calling a factory method - either specified in an interface and implemented by child classes, or implemented in a base class and optionally overriden by derived classes rather than by calling a constructor.

Usage

1) Create object based on configuration or input:
When you need to create objects based on different configuration settings or input values, The Factory method design pattern can be use to encapsolate the logic for creating the appropriate object type.

2) providing a consistant interface for creating objects:
When you have multiple ways of creating objects, the factory method pattern can be use to provide a consistant interface for creating objects, making it easier for client code to interact with the different object creation mechanisms.

3) Extending the functionality of object creation:
When you want to extend the functionality of object creation without modifying existing code, The factory method pattern can be use to override the factory method in subclasses to add additional logic or different object creation strategies.



Examples 1 creating shapes

```
interface
shape {
    void
    draw();
}
class Circle implement-
class player {
    private string playerType;
    public player (string playerType) {
        this.player = playerType;
    }
    public void print() {
        System.out.println(playerType);
    }
}
class pokerplayer extends player {
    public pokerplayer() {
        super("poker player");
    }
}
Class Gofishplayer extends player {
    public GoFishplayer() {
        super("Gofish playe");
    }
}
```

Date Month Year


```
abstract class CardGame {
    private in numplayer;
    private List<player>players = new ArrayList<>();
    Private boolean playerInit;
    public CardGame(int numplayers) {
        this.numplayers = numplayers;
        this.playerInit = false;
    }
    public void Createplayers() {
        for(int i=0; i<numplayers; i++) {
            players.add(makeplayers());
        }
    }
    public abstract player makeplayers();
    public void print() {
        if(!playerInit) {
            Createplayers();
            playlerInit = true;
        }
        For(player player: players) {
            player.print();
        }
    }
    Class PokerGame extends CardGame {
        public PokerGame() {
            Super(5)
        }
        public player makeplayers() {
            return new pokerplayer();
        }
    }
    Class GofishGame extends CardGame {
        public GofishGame() {
            Super(4);
        }
    }
```

```
pulblic play makeplayers(){
    return new GoFishplayer();
}}

public class Main {
  public static void main(String[] args){
    CardGame PoleerGame = new PolcerGame();
    CardGame gofishGame = new GoFishGame();
    System.out.println("--- poker ---");
    polceGame.print();
    System.out.println();
    system.out.println("--- Go fish ---");
    gofishGame.print();
  }
}
```

Observers Desgn part

Observers pattern is use when there is one to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.
observer pattern fails under behavioral pattern Category.

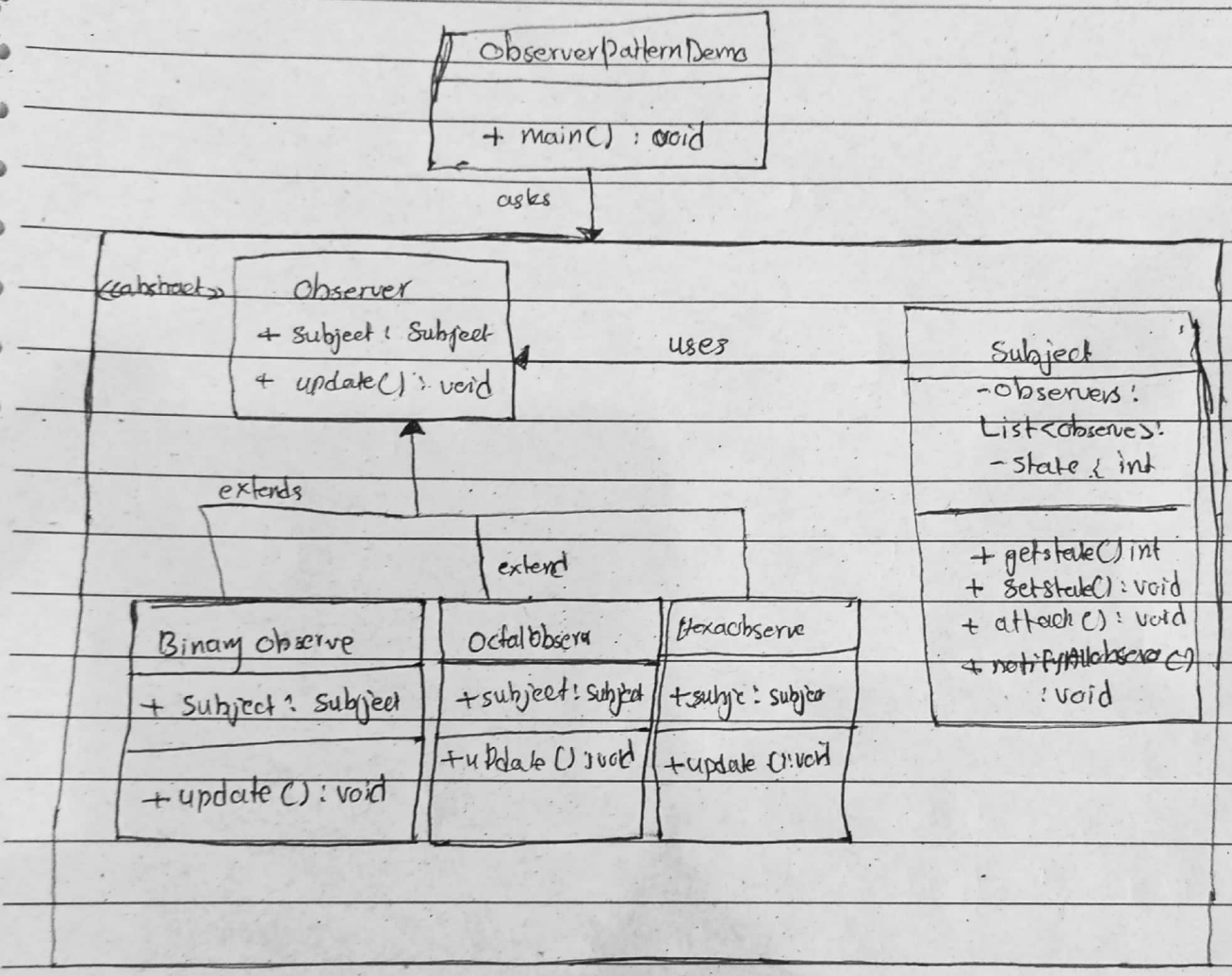
Usage

1) monitoring stock price: a stock market application can use the observer pattern to notify investors when the price of the stocks they are
2) Tracking whether updates: A whether forcasting service can use the observer pattern to notify subscribers when updates are available.
3) ~~Tracking~~ monitoring system performance: it use observer patten to notify administrators wher system performance matrix exceed certain thresholds.

Date Month Year

④

UML Diagram




```
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(Observers observers : observers){
        observer.update();
    }
}
```

Step 2 : Create observer class:

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

Step 3 : Create concrete observer class

```
public class BinaryObserver extend Observer {
    puble BinaryObserver(Subject subje) {
        this.subject = subject;
        this.subject.attach(this);
    }
    @Override
    public void update(){
        System.out.print("Binary string :" + integer.toBinaryString(subject.getState());
    }
}
```

```
// octalObserver.java
public class OctalObserver extend Obser {
    public ~~octalObserver(Subject subject)~~ {
        this.subject = subject;
        this.subject.attach(this);
    }
    @Override
    public void update() {
        System.out.print("octal string:" Integer.toOctalStrig(subject.getstate()));
    }
```

Date Month Year

```
// Hexaobserver.java

public class Hexaobserver extends Observer{
    public Hexaobserve (subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }
    @Override
    public void update(){
        System.out.print("Hex String: "+Integer.toHexString(subject.getState()).
        toUpperCase());
    }
}

// use Subject and concret observe object

public class observerpatternDemo{
    public static void main(String[] args){
        Subject subject = new Subject();

        new Hexaobserve(subject);
        new octalobserve(subject);
        new Binaryobserve(subject);
        System.out.print("First state change: 15");
        subject.setState(15);

        System.out.println("second state change. 10");
        Subject.setState(10);
    }
}
```

Date Month Year

chain Responsibility:

The chain of Responsibility design pattern is a behavioral design pattern that allows a request to be passed a long a chain of objects until it is handled.
Each object in the chain can decide whether to handle the request itself or to pass it on to the next object in the chain.

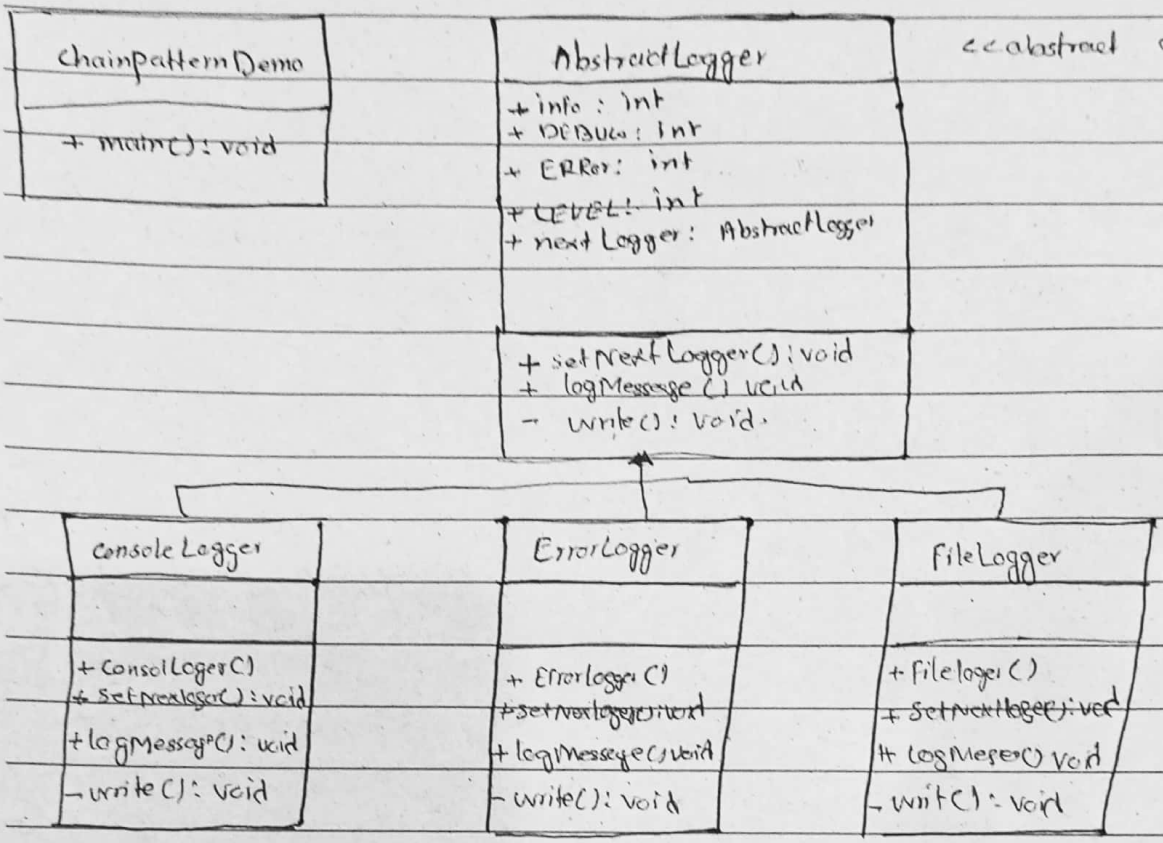
There are some benefits of chain responsibility design patter:

- Loose Couplering.
  The sender of request does not need to know whill object will handle it.
- Modular code:
  The code is easier to maintain.
- Reusability:
  The pattern can be easily reused in different part of the system

usage

-> when you want to decouple the sender of a request from its reciever.
The sender of a request does not need to know which object will hand the request.
-> when you want to handle request in different ways depending on the type of request.
-> when you want to execute a sequence of tasks in a specific order.
This pattern allows you to define a chain of hand

Date Month Year



Step1: Create an abstract logger class

```
public abstract class AbstractLogger {
    public static int Info = 1;
    public static int Debug = 2;
    public static int ERROR = 3
    protected int level;

    protected AbstractLogger nextLogger;
    public void setNextLogger ( AbstractLogger nextLogger) {
        this.nextLogger = nextLogger;
    }
    public void logMessage (int level, String message) {
        if (this.level <= level) {
            write (message)
        }
        if (nextLogg.logMessage(level, messg);
    }
}
```

Date Month Year

```
    abstract protected void write (String message);
}
```

Step2: Create concrete classes extending the logger.

```
public class ConsoleLogger extends AbstractLogger {
    public consoleLogger (int level) {
        this.level = level;
    }
    @Override
    protected void write (String message) {
        System.out.println ("standard console logger" + message)
    }
}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {
    public ErrorLogger (int level) {
        this.level = level;
    }
    @Override
    protected void write (String message) {
        System.out.println ("Error console" + message);
    }
}
```

FileLogger.java.

```
public class FileLogger extends AbstractLogger {
    public fileLogger (int level) {
        this.level = level;
    }
    @Override
    protected void write (String message) {
        System.out.print ("File:logger" + message);
    }
}
```

Step3:

Create different types of loggers. Assign them error levels and set next logger in each logger.

```
public class chainpatternDemo {
    private static AbstractLogger getchainofloggers() {
        AbstractLogger errorlogger = new Errorlogger(AbstractLogger.ERROR);
        AbstractLogger filelogger = new filelogger(Abstractlogger.DEBUG);
        AbstractLogger consolelogger = new consolelogger(Abstractlogge.Info);

        errorLogger.setNextlogger(Filelogger);
        Filelogger.setNextLogger(Consollogger);
        return errorlogger;
    }

public static void main(String[] args) {
    AbstractLogger loggerchain = getchainofloggers();
    Loggerchain.logMessage(AbstractLogger.Info,
        "this is an information");
    loggerchain.logMessage(Abstractlogger, DEBUG, "this debug level");

    loggerchain.logmessage(AbstractLogger.ERROR,
        "This is an error info");
    }
}
```

Singleton Design pattern

The Singleton design pattern is a creational design pattern that ensures a class has only one instance, and provide a global point of access to that instance.

This design pattern is useful when you need to control access to a shared resource, such as database or a file..

usages

The Singleton design pattern is often used in the following situations.

- logging: A logging class can be used to log messages to a file or console.

Date Month Year

- Configuration:

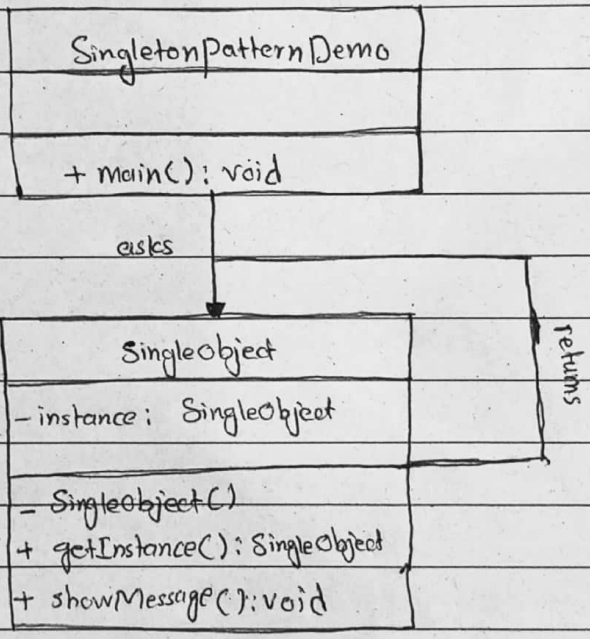
A configuration class can be used to store and access application-wide configuration settings.

- Caching:

A caching class can be used to store and retrive frequently accessed data.

- Thread pools:

A thread pool class can be use to manage a pool of threads for handling asynchronous tasks.



Date Month Year

Step 1: Create a Singleton class.

```
public class Singleobject {
    private static Singleobject instance = new Singleobject();
    private Singleobject() {}
    public static Singleobject getInstance() {
        return instance;
    }
    public void showMessage() {
        System.out.println("Hello World");
    }
}
```

Step 2:

Get the only object from Singleton class.

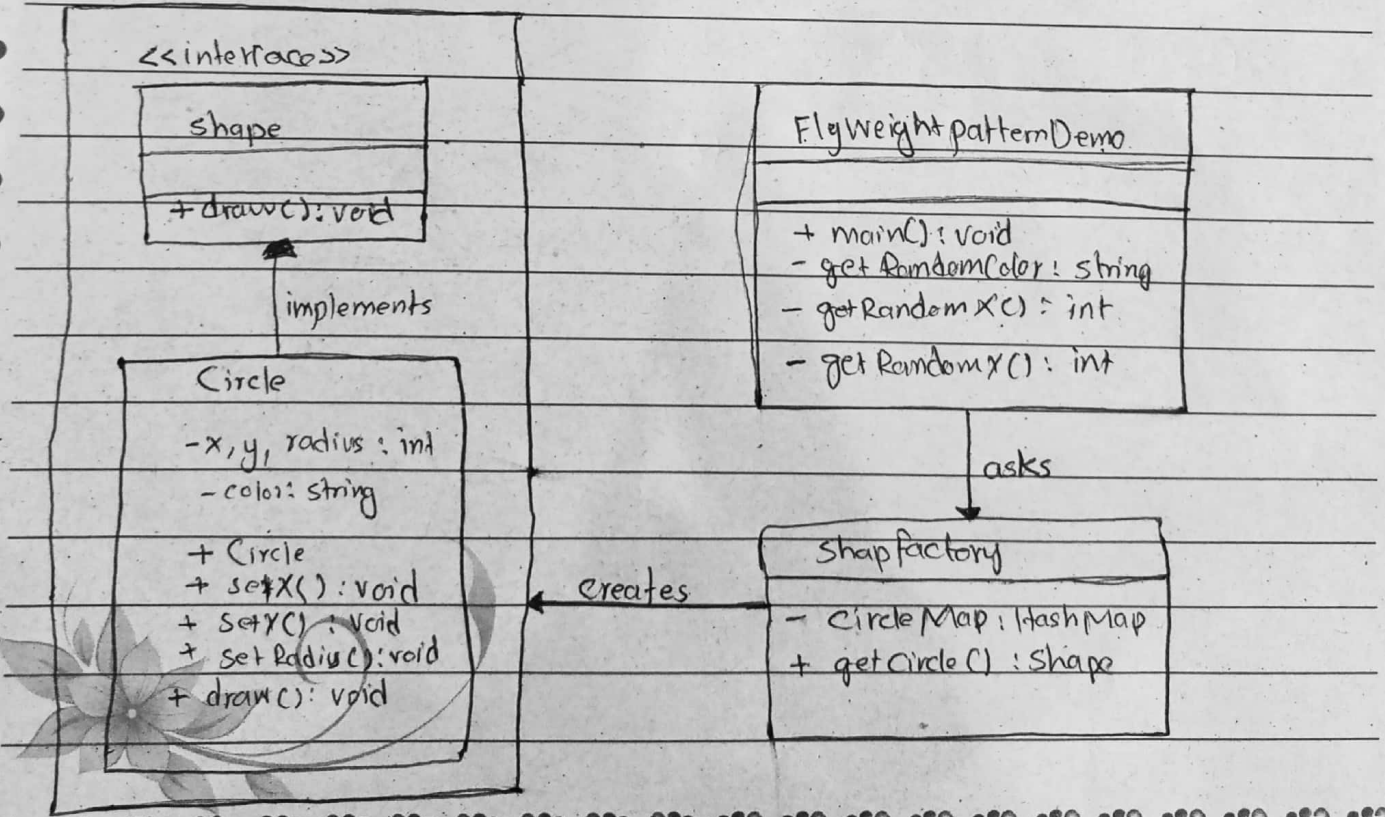
```
public class SingletonpatternDemo {
    public static void main(String[] args) {
        Singleobject object = Singleobject.getInstance();
        object.showMessage();
    }
}
```

Flyweight

The Flyweight Design pattern is a structural design pattern that allow you to fit more objects into available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object

Usage

The flyweight design pattern is useful when you need to create a large number of similar objects, and the memory footprint of each object is significant.
This pattern can be use to reduce memory usage and improve performance, especially in application that deal with a lot of data, such text editors, graphic applications and network applications



Date Month Year

Step 1: Create an interface.

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

Step 2: Create Concrete class implementing the same interface.

```
public class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color) {
        this.color = color;
    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public void setRadius(int radius) {
        this.radius = radius;
    }
    @Override
    public void draw() {
        System.out.println("Circle : Draw() [Color : " + color + ", x : " + x + ", y : " + y + ", radius
    }
}
```

Step 3

Create a factory to generate object of concret class based on given information.

ShapeFactory.java

Date Month Year

```
import java.util.HashMap;
public class shapefactory {
    private static final Hashmap CircleMap = new HashMap();
    public static shape getCircle(string color) {
        Circle circle (circle) CircleMap.get(color);
        if(Circle == null) {
            Circle = new Circle(color);
            CircleMap.put(color, Circle);
            System.out.print("creating circle of color:"+color);
        }
        return circle;
    }
}
```

Step 4

use the factory to get objects of concret class by passing an information such as color

```
public class FlyweightpatternDemo {
    private static final String colors[] = {"red", "Green" "Blue"}
    public static void main(string[] args) {
        for(int i=0; i<20; ++i) {
            Circle circle = (Circle) shapefactory.getCircle(getRandomColor());
            Circle.setX(getRandomx());
            Circle.setY(getRandomx());
            Circle.setRadius(100);
            Circle.draw();
        }
    }
```

```
private static String getRandomColor() {
    return colors[(int)(Math.random() * colors.length)];
}
private static int getRandomX() {
    return (int)(Math.random() * 600);
}
private static int getRandomY() {
    return (int)(Math.random() * 600);
}
}
```

Adapter Design

The adapter design pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate.
it acts as a bridge between two objects by transalating the methods of one object into methods that the other objeet can understand.

This allow you to use, and it can also make your code more reusable.

usage of design patter

The adapter design pattern is useful when we want to:

- use an existing class, but its interface is not compatible with the rest of your code.
- Reuse several existing subclasses that lacke some common functionality that can not be added to the superclass.
- wrap an existing class with a new interface.
- Impedance match an old component to a new system.

```
private static String getRandomColor() {
    return colors[(int)(Math.random()*colors.length)];
}
private static int getRandomX() {
    return (int)(Math.random()*100);
}
private static int getRandomY() {
    return (int)(Math.random()*100);
}
}
```

Adapter Design

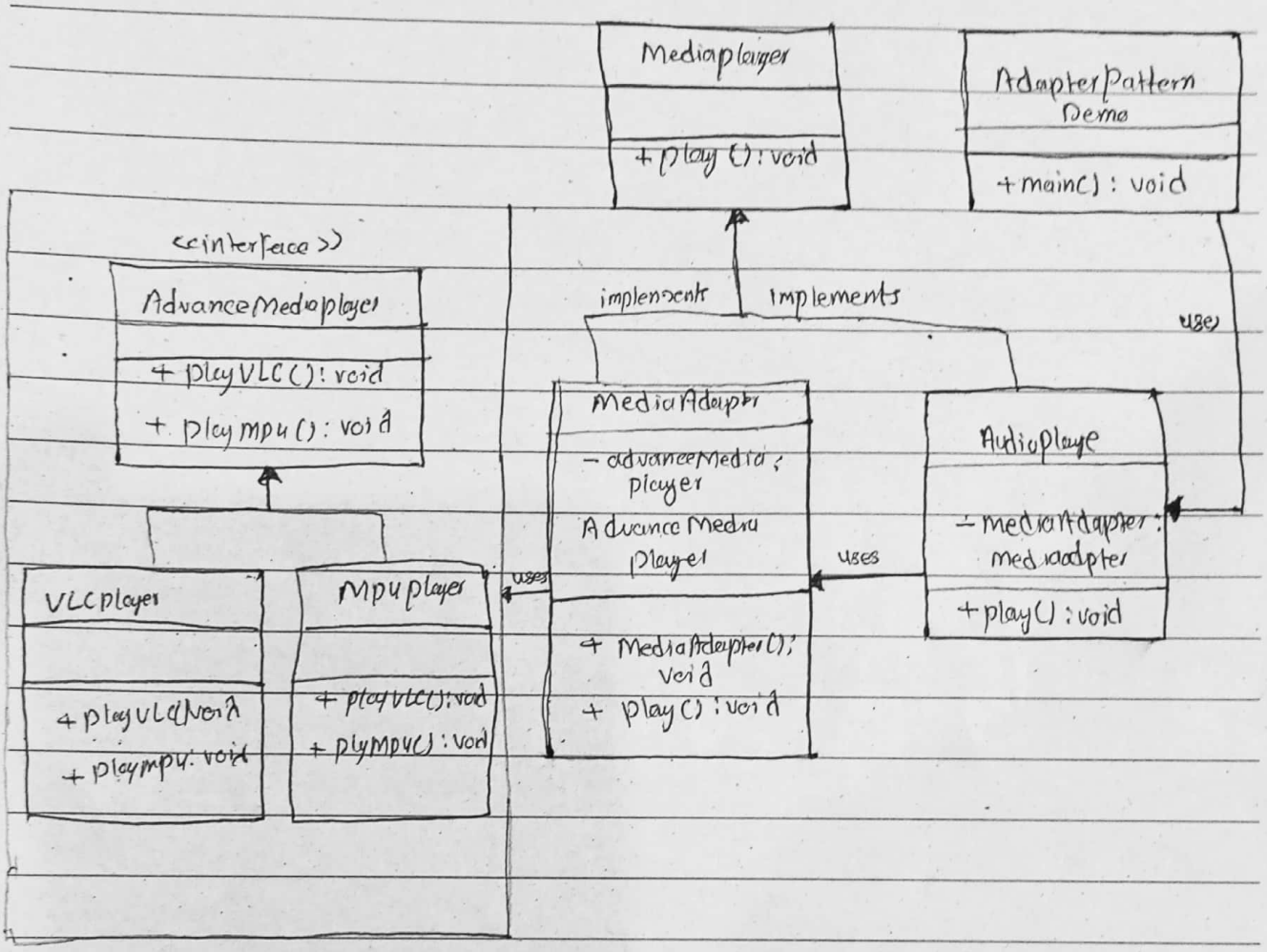
The adapter design pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate.
it acts as a bridge between two objects by transalating the methods of one object into methods that the other object can understand.

This allow you to use, and it can also make your code more reusable.

usage of design pattern

The adapter design pattern is useful when we want to:

- use an existing class, but its interface is not compatible with the rest of your code.
- Reuse several existing subclasses that lacks some common functionality that can not be added to the superclass.
- wrap an existing class with a new interface.
- Impedance match an old component to a new system.



Step 1: Create interface for media player and Advance Media player

```
public interface mediaplayer{
    public void play(string audioType, string filename);
}

public interface AdvanceMediaplayer{
    public void playVLC(string filename);
    public void playMpu(string filename);
}
```

Step2: Create Concrete classes implementing the AdvanceMediaplayer interface.

```
public class Vlcplayer implements AdvanceMediaplayer {
    @Override
    public void playVlc(String filename) {
        System.out.println("playing vlc file. Name: " + filename);
    }
    @Override
    public void playMpu(String filename) {
        // do nothing
    }
}

public class mpuplayer implements AdvanceMediaplayer {
    @Override
    public void playVlc(String filename) {
    }
    @Override
    public void playMpu(String filename) {
        system.out.println("playing mp4 file" + filename);
    }
}
```

Step 3

Create adapter class implementing the Mediaplayer interface.

```
public class MediaAdapter implements Mediaplayer {
    AdvancedMediaplayer advancedMusicplayer;
    public MediaAdapter(String audioType) {
        if(audioTyp.equalIgnoreCase("vlc")) {
```

```
advanceMusicPlayer = new VlcPlayer();
} else if (audioType.equalsIgnoreCase("mp4")) {
    advanceMusicPlayer = new Mp4Player();
}
}
@Override
public void play(String audioType, String fileName) {
    if (audioType.equalsIgnoreCase("[illegible]")) {
}
}
```

Step 4:

Create Concrete class implementing the MediaPlayer Interface.

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
    if (audioType.equalsIgnoreCase("mp3")) {
        System.out.println("playing mp3 file" + fileName);
    }
    else if (audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")) {
        mediaAdapter = new MediaAdapter(audioType);
        mediaAdapter.play(audioType, fileName);
    }
    else {
        System.out.println("Invalid media " + audioType + " format
            not supported");
    }
}
}
```

Step5:

use the audioplayer to play different types of audio formats.

```
public class AdapterpatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioplaye = new AudioPlaye();

        audioplayer.play("mp3", "beyond the horize.");
        audioplayer.play("mp4", "alone.mp4");
        audioplayer.play("vlc");
        audioplayer.play("avi");
    }
}
```

Date Month Year

25

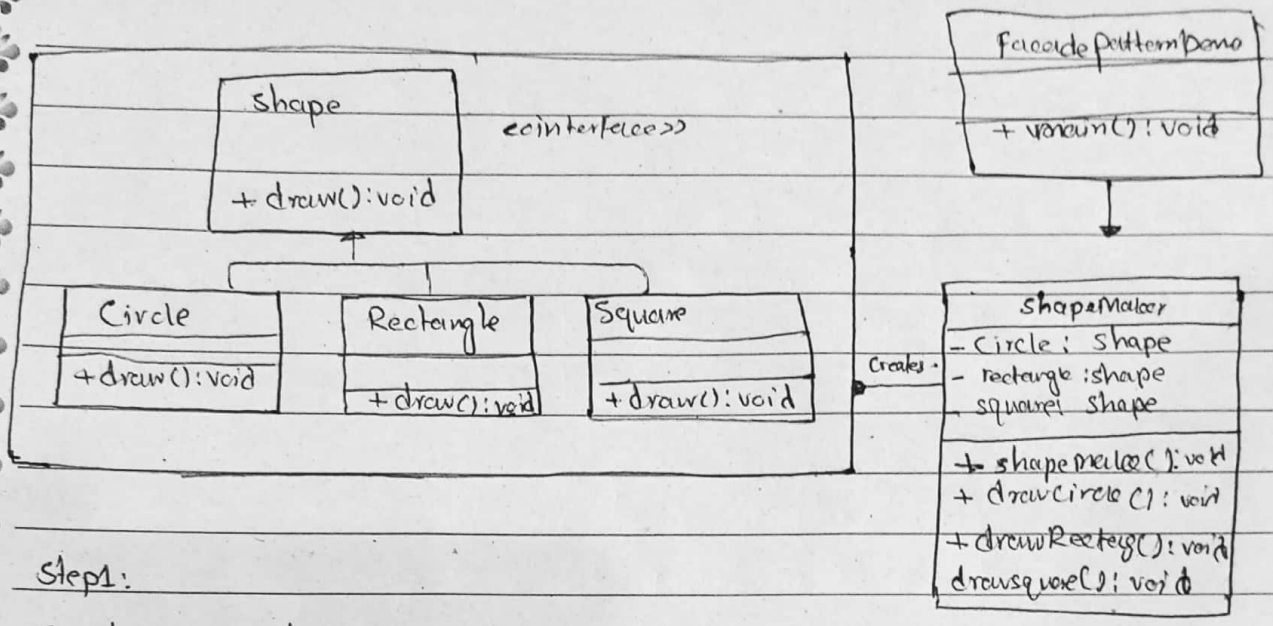
facade design pattern

The Facade design pattern is a structural design pattern that provides a simplified interface to a complex subsystem.
it is often use to hide the internal workings of a system and to make it easier for clients to use.
A Facade is a single class that provide a simplified interface to a set of classes or objects

usages

- provide a simplified interface to a complex library or framework
  Eg: The Java swing framework provide a facade that makes it easier to use the undelying graphic classes.
- Exposing a limit set of functionality to a client. Eg:
  A facade can be used to expose a limited set of functionalty to a client, while hiding the rest of the system's functionality.
- Creating a layer of abstraction between a client and a subsystem
  Eg:
  A Facade can be use to create a layer of abstraction between a client and a subsystem.
  making it easier to change the subsystem without affecting the client.

The Facade design pattern is a powerful tool that can be used to simplify complex systems and to make them easier to use and maintain.



Step1:

Create an interface.

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

step2:

Create concret classes implementing the same interface.

```
public class Rectangle implements shape {
    @Override
    public void draw() {
        System.out.print("Rectangle: Draw()");
    }
}
```

Square.Java

```
public class Square implements shape {
    @Override
    public void draw() {
        syster.out.print("square draw()");
    }
}
```

Circle.java

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

~~Create~~ step3 create a facade class.

```
public class ShapeMaker {
    private Shape circle;
    private Shape rectangle;
    private Shape square;

    public ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
    }

    public void drawCircle() {
        circle.draw();
    }

    public void drawRectangle() {
        rectangle.draw();
    }

    public void drawSquare() {
        square.draw();
    }
}
```

Step 4:
use the Facade to draw various types of shapes.

```
public class FacadePatternDemo{
    public static void main(String[] args){
        ShapeMaker shapeMaker = new ShapeMaker();
        shapeMaker.drawCircle();
        shapeMaker.drawRectangle();
        shapeMaker.drawSquare();
    }
}
```

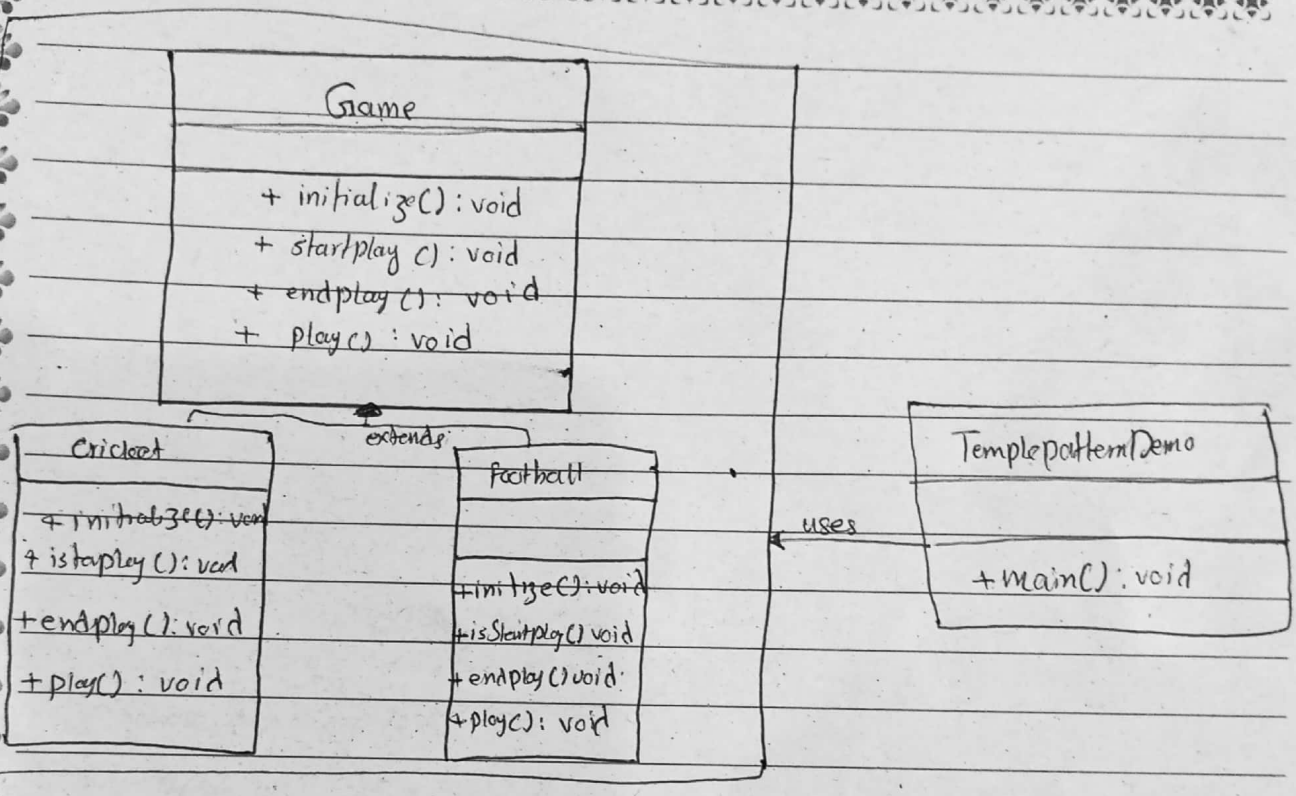
Template design pattern

The template design pattern is a behavioral design pattern that defines a skeleton of an algorithm in an operation, deffering some steps to subsclass.
It allows subclasses to redefine certain steps of the algorithm without changing the algorithm structure.

⇒ usages
- in sorting algorithm, the template design pattern can be use to define the common steps of sorting algorithm such as swapping elements and comparing elements.
- In document creation process, the template design pattern can be use to define the common steps of the process.

→ In web application framework:

The template design pattern can be use to define the common steps of processg a web request. such as parsing the request.



Step 1:

Create an abstract class with a template method being final.

```
public abstract class Game {
    abstract void initialize();
    abstract void startplay();
    abstract void endplay();

    public final void play() {
        initialize();
        startplay();
        endplay();
    }
}
```

Step 2:

Create a concret classes extending the above class.

```
public class Cricket extends Game {
    @Override
    void endplay() {
        System.out.print("Cricket game finished");
    }
    @Override
    void initialize() {
        system.out("cricket game initialize, start playing.)
    }
    @Override
    void startplay() {
        system.out.prit("Cricket Game started");
    }
}
```

Football.java

```
public class Football extends Game {
    @Override
    void endplay() {
        system.out.print("Football game finished");
    }
    @Override
    void initialize() {
        system.out.println("Footbal game initialized");
    }
    @Override
    void startplay() {
        System.out.print("football game started")
    }
}
```

Step3:

Use the Game's template method play() to demonstrate a defined way of playing Game.

```
public class TemplatePatternDemo {
    public static void main(String[] args) {
        Game game = new Cricket();
        game.play();
        game = new Football();
        game.play();
    }
}
```

Date Month Year

Builder Design pattern

The builder design pattern is a creational design pattern that separates the construction of a complex object from its representation.
It is one of the Gang of four design patterns.
The builder pattern provides a flexible solution to various object creation problems in object oriented-programing.

usages

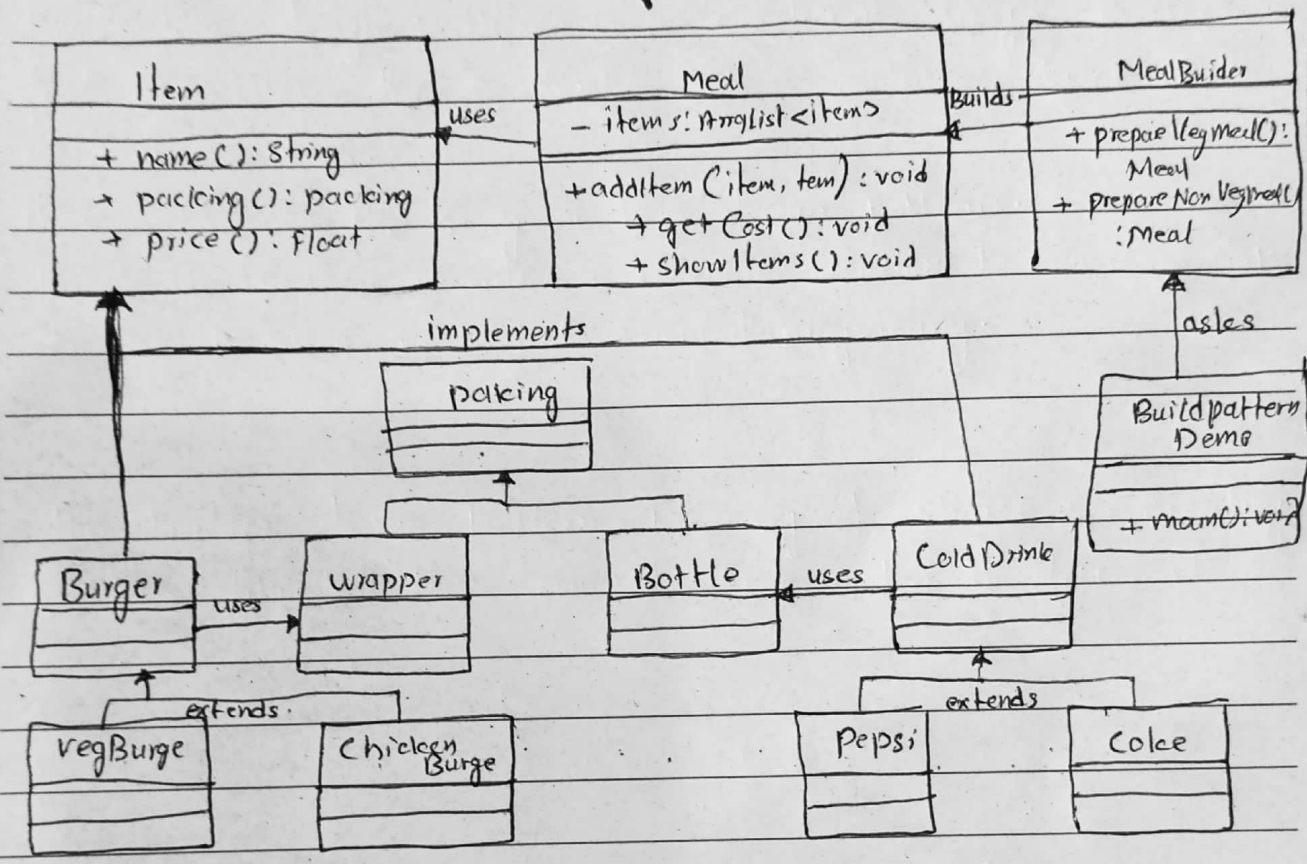
The builder design pattern is a good choice for creating complex objects that can be represented in different ways.

Benefits of Builder Design pattern

- improved readability and maintainability:
The builder pattern makes the construction of complex objects more readable and maitainable by breaking it down into series of steps.

- Increased Flexibility:
The builder pattern makes it easier to create different representation of the same object by providing a way to set the attribute of the objects step by step.

UML Diagram



Step 1:

Create an interface item representing food item and packing.

```
public interface Item {
    public string name();
    public paking paking();
    public float price();
}
public interface paking {
    public string pack();
}
```

step 2:

Create concret classes implementing the packet interface.

```
public class Wrapper implements packing {
    @Override
    public String pack() {
        return "Wrapper";
    }
}

public class Bottle implement packing {
    @Override
    public string pack() {
        return "Bottle";
    }
}
```

Step 3:

Create abstract classes implementing the item interface providing default functionalities.

```
public abstract class Burger implements Item {
    @Override
    public Packing packing() {
        return new Wrapper();
    }
    @Override
    public abstract float price();
}

public ~~class~~ abstract class ColdDrink implements Item {
    @Override
    public packing packing() {
        return new Bottle();
    }
    @Override
    public abstract float price();
}
```

Step 4:

Create Concrete classes extending Burger and ColdDrink classes

```
public class VegBurger extends Burger {
    @Override
    public Float price(){
        return 25.0f;
    }
    @Override
    public String name(){
        return "veg Burger";
    }
}

public class ChickenBurger extends Burger {
    @Override
    public Float price()
        return 50.5f;
    }
    @Override
    public String name(){
        return "Chienken Burger";
    }
}

public Class Coke extends ColdDrink{
    @Override
    public Float price(){
        return 3.0f;
    }
    @Override
    public String name(){
        return "Coke";
    }
}
```

Date Month Year

steps

```
public class Meal {
    private List<Item> items = new ArrayList<Item>();
    public void addItem(Item item){
        items.add(item);
    }
    public Float getCost(){
        Float cost = 0.0f;
        for(Item item : items){
            cost += item.price();
        }
        return cost;
    }
    Public void showItem(){
        for(Item item : items){
            System.out.print("Item" + item.name());
            System.out.print(", packing" + item.packing().pack());
        }
    }
}
```

step6: Create a MealBuilder class, the actual builder class responsible to create Meal object.

```
public class MealBuilder {
    public Meal prepareVegMeal(){
        Meal meal = new Meal();
        meal.addItem(new VegBurger());
        meal.addItem(new Coke());
        return meal;
    }
    public Meal prepareNonVegMeal(){
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;
    }
}

public class BuilderPatternDemo {
    public static void main(String[] args){
        MealBuilder mealBuilder = new MealBuilder();
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("non-veg meal");
        nonVegMeal.showItem();
```

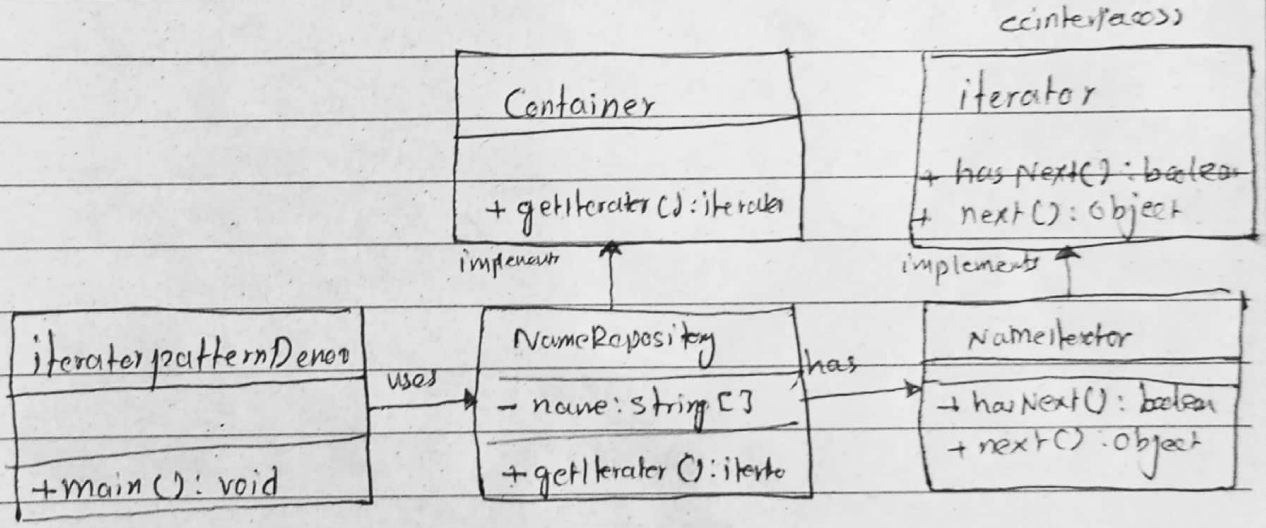
Date Month Year

Iterator Design pattern

The iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

usages of iterator Design pattern

The iterator design pattern is a good choice for traversing collections of objects that are implemented in different ways.



Step 1:

Create interfaces.

```
public interface Iterator {
    public boolean hasNext();
    public object next();
}
```

Date Month Year

```
public interface Container {
    public Iterater getIterater();
}
```

step 2:

Create Concret class implementing the Container interface: This class has inner class NameIterator implementing the iterator interface

```
public class NameRepository implements Container{
public string names[] = {"Robert", "John"};
.
@Override
public Iterater getIterater(){
    return new NameIterater();
}
private class NameIterater implements Iterater{
    int index;
    @Override
    public boolean hasNext(){
    if(index < namo.length){
        return true;
        return false;
    }
    @Override
    public object next(){
        if(this.hasNext()){
            return name[index++];
        }
        return null
    }
}

public class IteraterpatterDemo {
    public static void main(string[] args){
        NameRepository namesRepository.getIterater(); iter.hasNext(){
            string nam = (string) iter.next();
            System.out.println("Name:"+name);
        }
    }
}
```