Pandit Deendayal Energy University, Gandhinagar

School of Technology

Department of Computer Science & Engineering

Design Pattern Lab(20CP210P)

Name: **Padshala Smit Jagdishbhai**

Enrolment No: **21BCP187**

Semester: **IV Division**: **3 (G6)**

Branch: **Computer Science Engineering**

What is Design Pattern?

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

A Design pattern is a broad, repeatable solution to a problem that frequently arises in software design. Design patterns offer tried-and-true development paradigms, which can accelerate the development process. By making code more adaptable, readable, and maintainable, they can also raise the quality of software.

Design patterns are used in various areas of software engineering such as architecture, user interface design, and software development. They help developers solve recurring design problems by providing standard templates and approaches for creating software.

Design patterns can be used in a variety of ways, including:

1. Providing a common language for developers to communicate design ideas and solutions.
2. Improving the overall quality of software by promoting best practices and standardizing design decisions.
3. Making software development more efficient by reducing the time and effort required to solve common problems.

- **Creational Patterns: -**

1. **Factory Method:** Defines an interface for creating objects, but allows subclasses to decide which class to instantiate.
2. **Abstract Factory:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
3. **Builder:** Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.
4. **Prototype:** Allows copying existing objects without making code dependent on their classes.
5. **Singleton:** Ensures that a class has only one instance, and provides a global point of access to it.

1. Factory Pattern: -

The Factory Method Pattern is a creational design pattern that lets you create objects without specifying their exact class. It defines an interface for creating objects, but allows subclasses to decide which class to instantiate. This is useful when there are multiple related classes that need to be instantiated but the specific class to be used is not known at runtime.

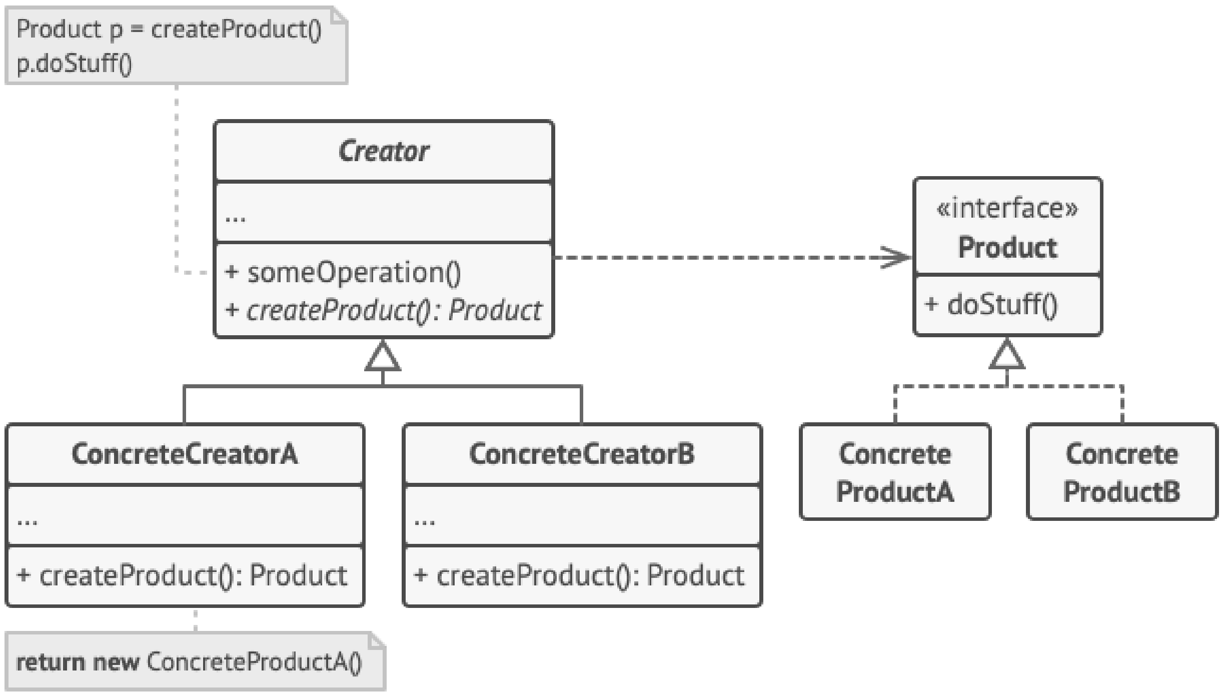
The pattern has four main components: Creator, Concrete Creator, Product, and Concrete Product. Creator is an abstract class or interface that defines the Factory Method. Concrete Creator is a subclass of Creator that implements the Factory Method and returns a Concrete Product. Product is an abstract class or interface that defines the object being created, and Concrete Product is a subclass of Product that provides the implementation of the object being created. Using this pattern can make code more modular and easier to maintain, since each class only needs to know about the interface and not the implementation details of other classes.

Advantages: -

1. Encapsulates object creation
2. Promotes code maintainability
3. Enhances code flexibility
4. Promotes code reuse
5. Reduces code complexity

Disadvantages: -

1. Can result in code bloat
2. Requires a good understanding of object-oriented programming
3. May be overkill for simple projects

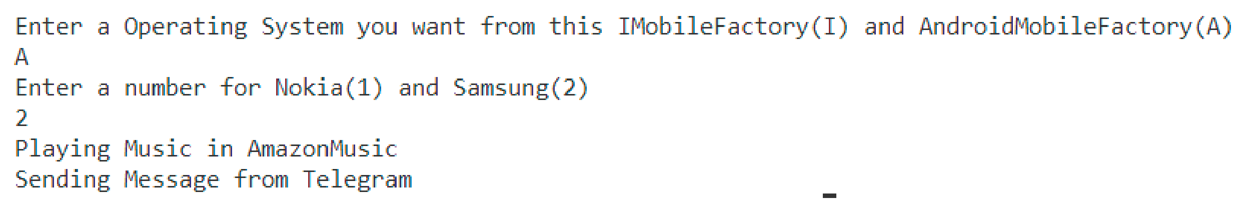


Code: -

```
import java.util.Scanner;

interface Mobile{
    String getMusic();
    String getSMS();

}

class Apple implements Mobile{

    @Override
    public String getMusic() {
        return "Playing Music in spotify";
    }

    @Override
    public String getSMS() {
        return "Sending Message from Instagram";
    }
```

```
}

class Nokia implements Mobile{

    @Override
    public String getMusic() {
        return "Playing Music in AmazonMusic";
    }

    @Override
    public String getSMS() {
        return "Sending Message from Telegram";
    }

}

class Samsung implements Mobile{

    @Override
    public String getMusic() {
        return "Playing Music in Youtube";
    }

    @Override
    public String getSMS() {
        return "Sending Message from Whatsapp";
    }

}

class MobileFactory{
    public  Mobile getMobile(String m){
        Scanner sc = new Scanner(System.in);
        switch(m){
            case "I":
            return new Apple();
            case "A":
                System.out.println("Enter a number for Nokia(1) and Samsung(2)");
                String st = sc.nextLine();
                switch(st){
                    case "1":
                    return new Samsung();
                    case "2":
                    return new Nokia();
                    default:
                    System.out.println("Enter valid mobile");
                    return null;
                }
            default:
                System.out.println("Enter valid MobileFactory ");
            return null;
        }
    }
}

public class Factory_Pattern_Mobile {
    public static void main(String[] args) {
        Scanner sa = new Scanner(System.in);
        System.out.println("Enter a Operating System you want from this 
IMobileFactory(I) and AndroidMobileFactory(A)");
        String str = sa.nextLine();
        MobileFactory MobileFactory = new MobileFactory();
        Mobile mo =  MobileFactory.getMobile(str);
        System.out.println(mo.getMusic());
        System.out.println(mo.getSMS());
    }
}
```

Output: -



```
Enter a Operating System you want from this IMobileFactory(I) and AndroidMobileFactory(A)
A
Enter a number for Nokia(1) and Samsung(2)
2
Playing Music in AmazonMusic
Sending Message from Telegram
```

2. Abstract Factory Pattern: -

The Abstract Factory Design Pattern is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It encapsulates a group of related factory methods in an interface and allows the creation of objects of these related classes without exposing the instantiation logic to the client.

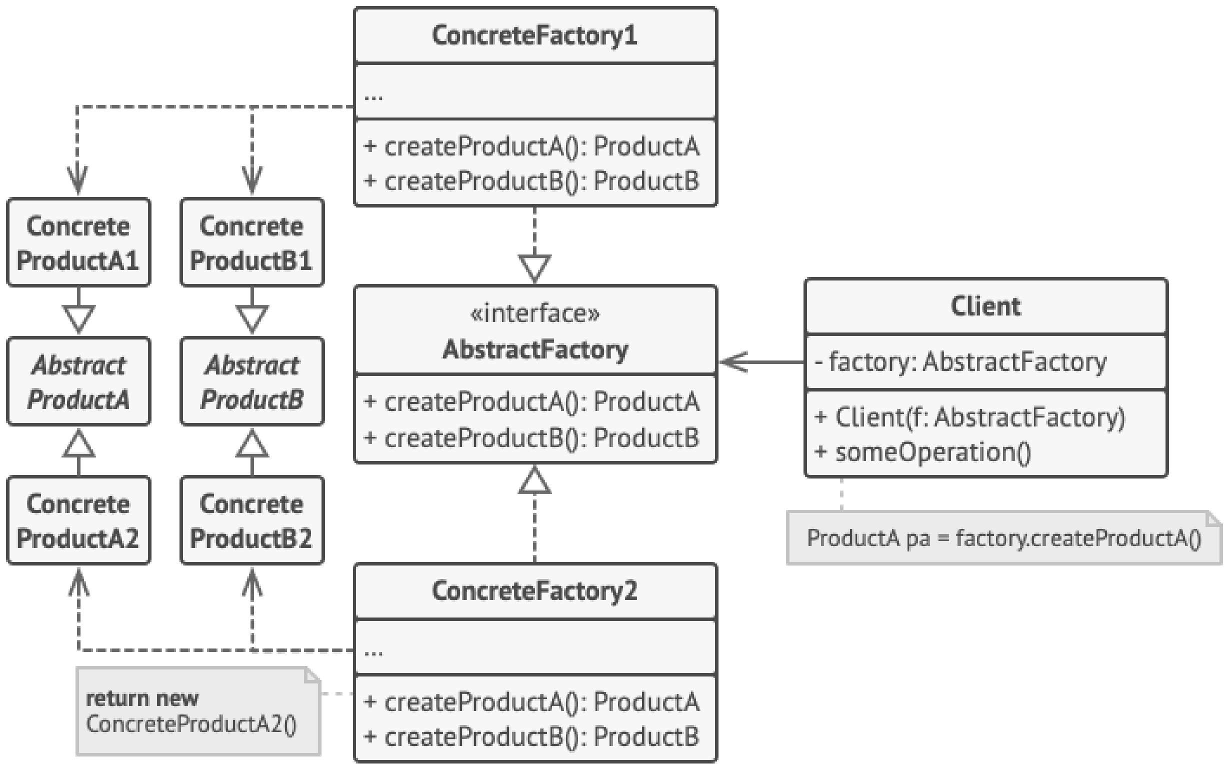
This pattern is useful for creating families those objects. It helps to reduce dependencies between objects and makes it easier to switch between of related objects that work together without the need to know the specific classes that implement families of related objects. The Abstract Factory Method is a powerful design pattern that simplifies object creation and makes an application more flexible and extensible.

Advantages: -

1. Provides a way to create families of related or dependent objects without specifying their concrete classes.
2. Allows a client to create objects that are part of a related family of objects without knowing the specific implementations of those objects.
3. Enhances code maintainability by reducing the coupling between the client code and the concrete classes of the objects.
4. Provides a way to enforce a consistent interface for a family of objects.
5. Enables the creation of new object variants without changing the client code.

Disadvantages: -

1. Can lead to code duplication if the families of objects have many common features.
2. Can increase the complexity of the code, especially if there are many families of objects with many variations.
3. Requires the creation of many new classes to implement the pattern, which can make the code more difficult to understand.


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
Abstract ProductA
Abstract ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

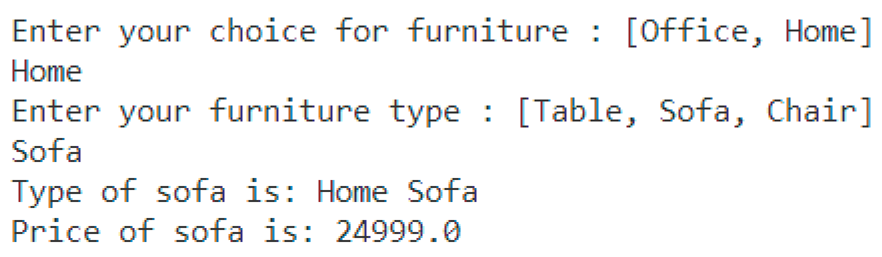
Code: -

```
import java.util.Scanner;

interface Chair{
    float Chairprice();
    String ChairType();
}

interface Table{
    float Tableprice();
    String TableType();
}

interface Sofa{
    float Sofaprice();
    String SofaType();
}

class OfficeChair implements Chair{
    @Override
    public float Chairprice() {
        return 899.99f;
    }

    @Override
    public String ChairType() {
        return "Office BMobileFactorys Chair\tOffice Employee Chair";
    }

}

class HomeChair implements Chair{

    @Override
    public float Chairprice() {
        return 999f;
    }

    @Override
    public String ChairType() {
        return "Guest Chair";
    }
}

class OfficeTable implements Table{

    @Override
    public float Tableprice() {
        return 1599f;
    }

    @Override
    public String TableType() {
        return "Plastic Table\tWood Table";
    }
}

class HomeTable implements Table{
    @Override
    public float Tableprice() {
        return 1999f;
    }

    @Override
    public String TableType() {
        return "Dining Table";
    }
}

class OfficeSofa implements Sofa{

    @Override
    public float Sofaprice() {
        return 19999f;
    }

    @Override
    public String SofaType() {
        return "Office Sofa";
    }
}

class HomeSofa implements Sofa{

    @Override
    public float Sofaprice() {
        return 24999f;
    }

    @Override
    public String SofaType() {
        return "Home Sofa";
    }
}

interface AbstractFactory{
    Chair createChair();
    Sofa createSofa();
    Table createTable();
}

class Office implements AbstractFactory{

    @Override
    public Chair createChair() {
        return new OfficeChair();
    }

    @Override
    public Sofa createSofa() {
        return new OfficeSofa();
    }

    @Override
    public Table createTable() {
        return new OfficeTable();
    }
}

class Home implements AbstractFactory{

    @Override
    public Chair createChair() {
        return new HomeChair();
    }

    @Override
    public Sofa createSofa() {
        return new HomeSofa();
    }

    @Override
    public Table createTable() {
        return new HomeTable();
    }
}

class FurnitureFactory{
```

```
public AbstractFactory getFurniture(String f){
    Scanner sf = new Scanner(System.in);
    switch(f.toLowerCase()){
        case "office" :
        System.out.println("Enter your furniture type : [Table, Sofa, Chair]");
        String of = sf.nextLine();
        switch(of.toLowerCase()){
            case "table" :
            OfficeTable ot = new OfficeTable();
            System.out.println("Type of table is: "+ot.TableType());
            System.out.println("Price of table is: "+ot.Tableprice());
            break;
            case "chair" :
            OfficeChair oc = new OfficeChair();
            System.out.println("Type of chair is: "+oc.ChairType());
            System.out.println("Price of chair is: "+oc.Chairprice());
            break;
            case "sofa" :
            OfficeSofa os = new OfficeSofa();
            System.out.println("Type of sofa is: "+os.SofaType());
            System.out.println("Price of sofa is: "+os.Sofaprice());
            break;
            default :
            System.out.println("Enter valid furniture type!!");
            break;
        }
        break;
        case "home" :
        System.out.println("Enter your furniture type : [Table, Sofa, Chair]");
        String hf = sf.nextLine();
        switch(hf.toLowerCase()){
            case "table" :
            HomeTable ht = new HomeTable();
            System.out.println("Type of table is: "+ht.TableType());
            System.out.println("Price of table is: "+ht.Tableprice());
            break;
            case "chair" :
            HomeChair hc = new HomeChair();
            System.out.println("Type of chair is: "+hc.ChairType());
            System.out.println("Price of chair is: "+hc.Chairprice());
```

```
                break;
                case "sofa" :
                HomeSofa hs = new HomeSofa();
                System.out.println("Type of sofa is: "+hs.SofaType());
                System.out.println("Price of sofa is: "+hs.Sofaprice());
                break;
                default :
                System.out.println("Enter valid furniture type!!");
                break;
            }
            break;
            default :
            System.out.println("Enter valid choice [Office, Home]");
            break;
        }
        return null;

    }
}

public class Abstarct_Factory_lab4 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your choice for furniture : [Office, Home]");
        String ch = sc.nextLine();
        FurnitureFactory ff = new FurnitureFactory();
        AbstractFactory af = ff.getFurniture(ch);
    }
}
```

Output: -



```
Enter your choice for furniture : [Office, Home]
Home
Enter your furniture type : [Table, Sofa, Chair]
Sofa
Type of sofa is: Home Sofa
Price of sofa is: 24999.0
```

3. Builder Design Pattern: -

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create multiple different representations.”

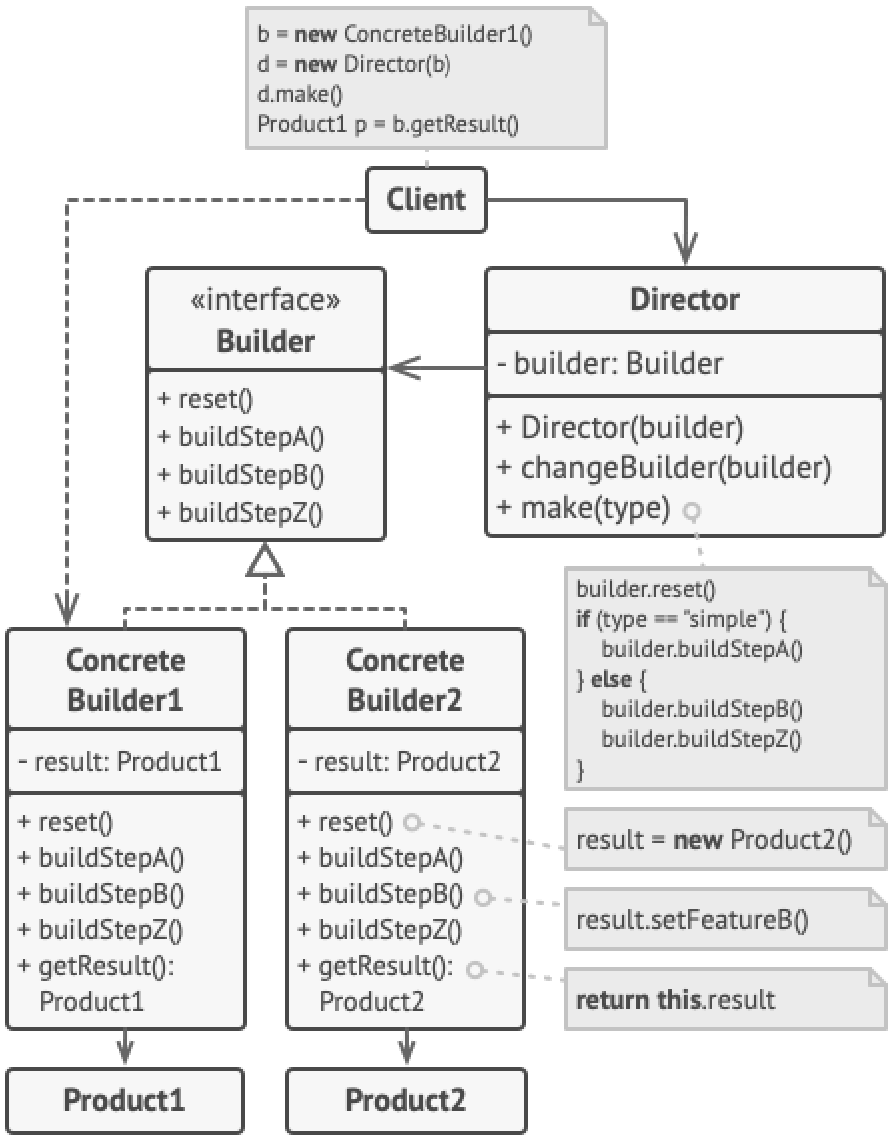
The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object it is to create.

Advantages: -

1. Provides a clear separation between the construction and representation of an object.
2. Allows the creation of complex objects through step-by-step construction.
3. Provides a flexible and scalable solution to object creation.
4. Allows the creation of different types and representations of an object using the same construction process.
5. Enhances code maintainability by reducing the complexity of the code.

Disadvantages: -

1. Can increase the complexity of the code, especially for simple objects.
2. Requires the creation of multiple classes to implement the pattern, which can make the code more difficult to understand.
3. Can lead to code duplication if not implemented correctly.

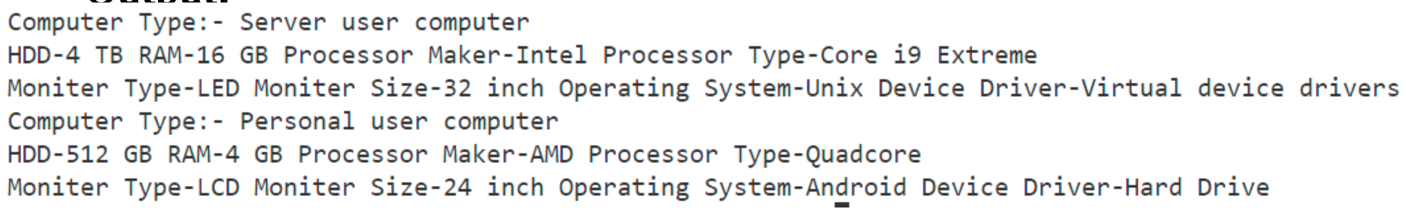


Code: -

```
class computer{
   private String HDD;
   private String RAM_size;
   private String Processormake;
   private String ProcessorType;
   private String MoniterSize;
   private String MoniterType;
   private String OSconfigure;
   private String DeviceDriver;
   private String type;
```

```
computer(String type) {
    this.type = type;
}
public computer setHDD(String HDD) {
    this.HDD = HDD;
    return this;
}
public computer setRAM_size(String RAM_size) {
    this.RAM_size = RAM_size;
    return this;
}
public computer setProcessormake(String Processormake) {
    this.Processormake = Processormake;
    return this;
}
public computer setProcessorType(String ProcessorType) {
    this.ProcessorType = ProcessorType;
    return this;
}
public computer setMoniterSize(String MoniterSize) {
    this.MoniterSize = MoniterSize;
    return this;
}
public computer setMoniterType(String MoniterType) {
    this.MoniterType = MoniterType;
    return this;
}
public computer setOSconfigure(String OSconfigure) {
    this.OSconfigure = OSconfigure;
    return this;
}
public computer setDeviceDriver(String DeviceDriver) {
    this.DeviceDriver = DeviceDriver;
    return this;
}

@Override
public String toString() {
        return "Computer Type:- "+type+"\nHDD-"+HDD+" RAM-"+RAM_size+"
Processor Maker-"+Processormake+" Processor Type-
"+ProcessorType+"\nMoniter Type-"+MoniterType+" Moniter Size-
"+MoniterSize+" Operating System-"+OSconfigure+" Device Driver-
"+DeviceDriver;
    }
    public static void describeComponent() {
    }

}

abstract class ComputerBuilder{
    protected computer comp;
    protected String type;

    public computer getcomp(){
        return comp = new computer(type);
    }

    public abstract void buildHDD();
    public abstract void buildRAM_size();
    public abstract void buildProcessor();
    public abstract void buildMoniter();
    public abstract void buildOS();
    public abstract void buildDeviceDriver();

}

class PersonalComputer extends ComputerBuilder{

    PersonalComputer(){
        super.type = "Personal user computer";
    }
    @Override
    public void buildHDD() {
        comp.setHDD("512 GB");
    }
    @Override
    public void buildRAM_size() {
        comp.setRAM_size("4 GB");
    }
    @Override
    public void buildProcessor() {
        comp.setProcessormake("AMD");
        comp.setProcessorType("Quadcore");
    }
    @Override
    public void buildMoniter() {
        comp.setMoniterSize("24 inch");
        comp.setMoniterType("LCD");
    }
    @Override
    public void buildOS() {
        comp.setOSconfigure("Android");
    }
    @Override
    public void buildDeviceDriver() {
        comp.setDeviceDriver("Hard Drive");
    }

}


class ServerComputer extends ComputerBuilder{
    ServerComputer(){
        super.type = "Server user computer";
    }
    @Override
    public void buildHDD() {
        comp.setHDD("4 TB");
    }
    @Override
    public void buildRAM_size() {
        comp.setRAM_size("16 GB");
    }
    @Override
    public void buildProcessor() {
        comp.setProcessormake("Intel");
        comp.setProcessorType("Core i9 Extreme");
    }
    @Override
    public void buildMoniter() {
        comp.setMoniterSize("32 inch");
        comp.setMoniterType("LED");
    }
    @Override
    public void buildOS() {
        comp.setOSconfigure("Unix");
    }
    @Override
    public void buildDeviceDriver() {
        comp.setDeviceDriver("Virtual device drivers");
    }
}

class ComputerEngineer{
    private ComputerBuilder computerBuilder;
    // public void setComputerBuilder(ComputerBuilder cb){
    //     computerBuilder = cb;
    // }
    // public computer getComputer(){
    //     return computerBuilder.getcomp();
    // }

    public ComputerEngineer(String a){
        if(a.equals("Server")){
            computerBuilder = new ServerComputer();
        }
        else if(a.equals("Personal")){
            computerBuilder = new PersonalComputer();

        }
    }
    public computer BuildComputer(){
        computer a = computerBuilder.getcomp();
        computerBuilder.buildHDD();
        computerBuilder.buildRAM_size();
        computerBuilder.buildProcessor();
        computerBuilder.buildMoniter();
        computerBuilder.buildOS();
        computerBuilder.buildDeviceDriver();
        System.out.println(a);
        return a;
    }
}

public class lab5_BuilderDesignPattern {
    public static void main(String[] args) {
        ComputerEngineer server = new ComputerEngineer("Server");
        server.BuildComputer();
        ComputerEngineer  personal = new ComputerEngineer("Personal");
        personal.BuildComputer();
    }
}
```

Output: -



```
Computer Type:- Server user computer
HDD-4 TB RAM-16 GB Processor Maker-Intel Processor Type-Core i9 Extreme
Moniter Type-LED Moniter Size-32 inch Operating System-Unix Device Driver-Virtual device drivers
Computer Type:- Personal user computer
HDD-512 GB RAM-4 GB Processor Maker-AMD Processor Type-Quadcore
Moniter Type-LCD Moniter Size-24 inch Operating System-Android Device Driver-Hard Drive
```

4. Prototype Design Pattern: -

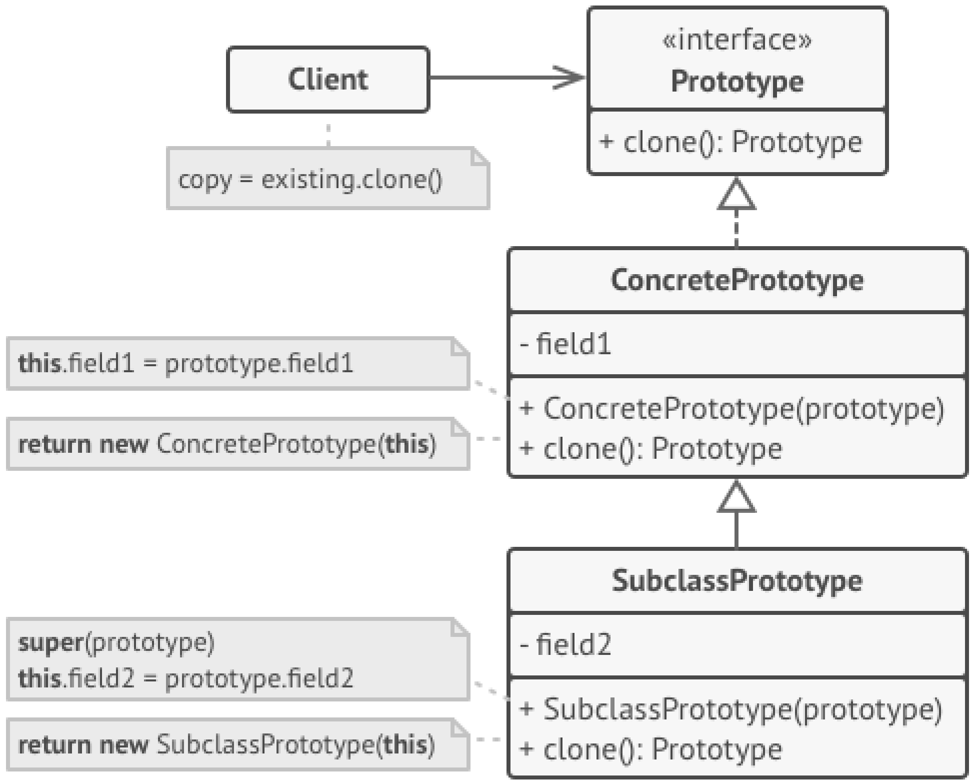
The Prototype Design Pattern is a creational design pattern that allows you to create new objects based on existing ones. It involves creating a clone of an existing object and then modifying it as needed, rather than creating a new object from scratch. This pattern can be useful in situations where creating a new object from scratch is expensive or time-consuming.

Advantage: -

1. Simplifies object creation: You can create new objects by cloning existing ones, rather than creating them from scratch.
2. Reduces code duplication: By using prototypes, you can avoid duplicating code and make your code more efficient.
3. Provides flexibility: You can easily modify existing objects or create new ones based on different prototypes.
4. Encourages loose coupling: The Prototype pattern promotes loose coupling between objects, which can make your code more modular and easier to maintain.
5. Improves performance: Creating objects by cloning existing ones is generally faster than creating them from scratch, especially when creating complex objects.

Disadvantages: -

1. Increases memory usage: Using the Prototype pattern can increase memory usage, especially if you're creating many copies of an object.
2. Can be complex: The Prototype pattern can be more complex than other creational patterns, such as the Factory Method or Singleton patterns.
3. Requires careful design: To use the Prototype pattern effectively, you need to carefully design your objects and their clones, which can require more upfront work than other approaches to object creation.

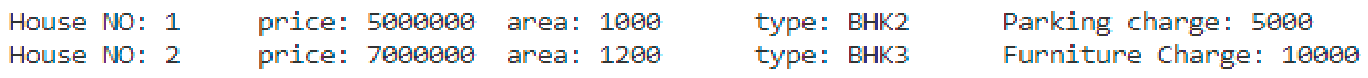

Client
copy = existing.clone()
«interface»
Prototype
+ clone(): Prototype
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)

Code: -

```
import java.util.HashMap;
import java.util.Map;

abstract class House implements Cloneable{

    protected int House_no;
    protected long House_price;
    protected int House_area;
    protected String HouseType;
    protected Owner owner;
    // abstract void addHouse();
    public void setHouse_no(int house_no) {
        House_no = house_no;
    }
    public void getHouse_area(int House_area) {
        this.House_area = House_area;
    }
    public void setHouse_price(long house_price) {
        House_price = house_price;
    }
    public int getHouse_no() {
        return House_no;
    }
    public int getHouse_area() {
        return House_area;
    }
    public long getHouse_price() {
        return House_price;
    }
    public String getHouseType() {
        return HouseType;
    }
    public void setHouseType(String HouseType) {
        this.HouseType = HouseType;
    }
    public Object clone(){
        Object clone = null;
        try{
            clone = super.clone();
        } catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return clone;
    }
    @Override
    public String toString() {
        return "House NO: "+House_no+"\tprice: "+House_price+"\tarea: 
"+House_area+"\ttype: "+HouseType;
    }
}


class BHK2 extends House{
    protected int ParkingCharge;
    public int getParkingCharge() {
        return ParkingCharge;
    }
    public void setParkingCharge(int parkingCharge) {
        ParkingCharge = parkingCharge;
    }
    @Override
    public String toString() {
        return super.toString()+"\tParking charge: "+ParkingCharge;
    }
}

class BHK3 extends House{
    protected int furnitureCharge;
    public int getFurnitureCharge() {
        return furnitureCharge;
    }
    public void setFurnitureCharge(int furnitureCharge) {
        this.furnitureCharge = furnitureCharge;
    }
    @Override
    public String toString() {
        return super.toString()+"\tFurniture Charge: "+furnitureCharge;
    }
}

class Owner{

}
class SampleHouse {
        private static Map<String, House> houseMap = new HashMap<String,
House>();

        static {
            houseMap.put("BHK2", new BHK2());
            houseMap.put("BHK3", new BHK3());
        }

        public static House getHouse(String type) {
            return (House) houseMap.get(type).clone();
        }
}
```

```
public class lab6_prototype {
    public static void main(String[] args) throws CloneNotSupportedException{
        SampleHouse sampleHouse = new SampleHouse();

        House bhk2 = sampleHouse.getHouse("BHK2");
        bhk2.setHouse_no(1);
        bhk2.setHouse_price(5000000);
        bhk2.getHouse_area(1000);
        bhk2.setHouseType("BHK2");
        ((BHK2) bhk2).setParkingCharge(5000);

        House bhk3 = sampleHouse.getHouse("BHK3");
        bhk3.setHouse_no(2);
        bhk3.setHouse_price(7000000);
        bhk3.getHouse_area(1200);
        bhk3.setHouseType("BHK3");
        ((BHK3) bhk3).setFurnitureCharge(10000);

        System.out.println(bhk2);
        System.out.println(bhk3);
    }
}
```

Output: -



```
House NO: 1     price: 5000000  area: 1000      type: BHK2      Parking charge: 5000
House NO: 2     price: 7000000  area: 1200      type: BHK3      Furniture Charge: 10000
```

5. Singleton Design Pattern: -

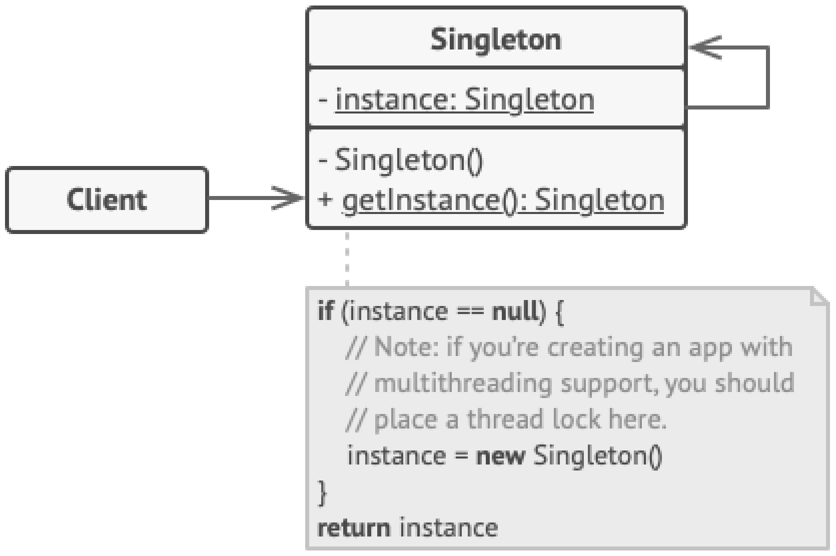
The Singleton design pattern is a creational design pattern that ensures a class has only one instance, and provides a global point of access to that instance. In other words, it allows you to control the number of instances of a class that can be created, and ensures that there is always only one instance available for use.

Advantages: -

1. Guarantees a single instance: Singleton pattern ensures that only one instance of the class exists in the entire application, which makes it useful for situations where you want to limit the number of objects of a particular class.
2. Global access: The Singleton pattern provides a global point of access to the instance, so you don't need to pass around references to the object all over your codebase.
3. Lazy initialization: The Singleton pattern allows you to defer the creation of the object until it is actually needed, which can help with performance and memory usage in some cases.

Disadvantages: -

1. Difficult to test: Singleton objects are globally accessible, which makes them difficult to test in isolation. Since they can be accessed from anywhere in your code, it can be hard to isolate them and test them in a controlled environment.
2. Can be overused: The Singleton pattern can be overused, leading to unnecessary constraints on your code. In some cases, it might be more appropriate to allow multiple instances of a class.
3. Can create dependency issues: Since Singleton objects are globally accessible, they can create dependency issues in your code. If one part of your code depends on a Singleton object, changes to that object can have unintended consequences throughout your codebase.

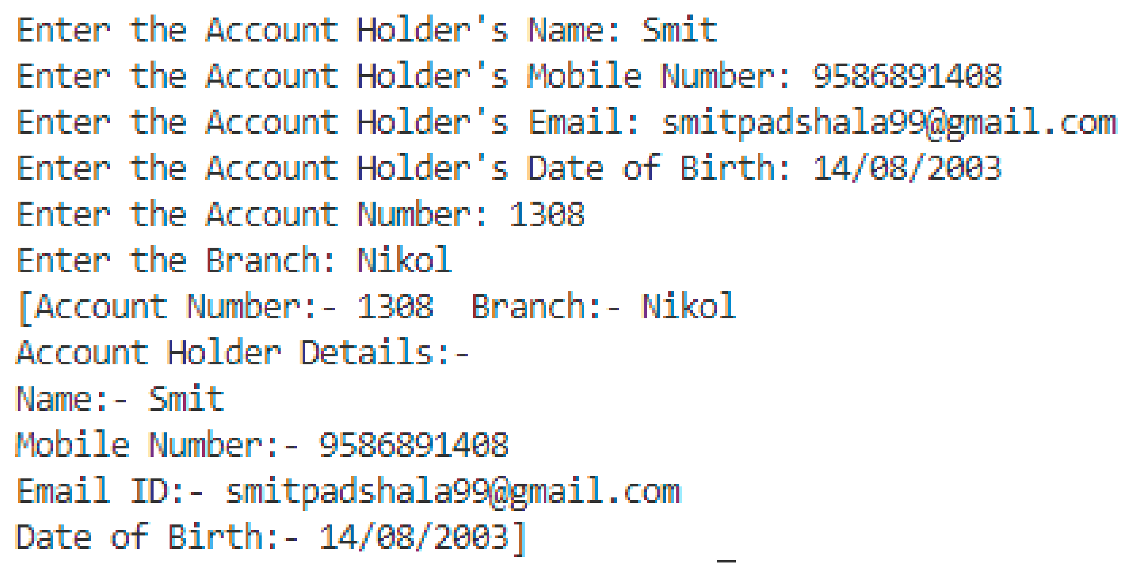

Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
if (instance == null) {
// Note: if you're creating an app with
// multithreading support, you should
// place a thread lock here.
instance = new Singleton()
}
return instance

Code: -

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

class Account_Holder{
   String Name;
   String Mobile_no;
   String Email;
   String DOB;

   public void setName(String name) {
      Name = name;
   }
   public String getName() {
      return Name;
   }
   public void setMobile_no(String mobile_no) {
      Mobile_no = mobile_no;
   }
   public String getMobile_no() {
      return Mobile_no;
   }
   public void setEmail(String email) {
      Email = email;
```

```
    }
    public String getEmail() {
        return Email;
    }
    public void setDOB(String dOB) {
        DOB = dOB;
    }
    public String getDOB() {
        return DOB;
    }
    @Override
    public String toString() {
        return "\nAccount Holder Details:- \nName:- "+Name+"\nMobile Number:-
"+Mobile_no+"\nEmail ID:- "+Email+"\nDate of Birth:- "+DOB;
    }
}
class Account{
    private long Account_no;
    private String Branch;
    private Account_Holder Account_Holder;
    Account(long Account_no, String Branch, Account_Holder Account_Holder ){
        this.Account_no=Account_no;
        this.Branch=Branch;
        this.Account_Holder=Account_Holder;
    }

    public void setAccount_no(long account_no) {
        Account_no = account_no;
    }
    public long getAccount_no() {
        return Account_no;
    }
    public String getBranch() {
        return Branch;
    }
    public void setBranch(String branch) {
        Branch = branch;
    }
    public void setAccount_Holder(Account_Holder account_Holder) {
        Account_Holder = account_Holder;
    }
    public Account_Holder getAccount_Holder() {
        return Account_Holder;
    }
    @Override
    public String toString() {
        return "Account Number:- "+Account_no+"\tBranch:- "+Branch+Account_Holder;
    }
}

class Banker{
    private static Banker banker=null;
    private static List<Account> account = new ArrayList<Account>();

    static{
        banker = new Banker();
    }
    private Banker(){
    }
    public static synchronized Banker getInstance(){
        if(banker == null){
            banker = new Banker();
        }
        return banker;
    }
    @Override
    public String toString() {
        return account.toString();
    }
    public void addAccount(Account account2) {
        Banker.account.add(account2);
    }
}



public class Lab7_SingletonPattern {

    public static void main(String[] args) {
        Scanner sc =new Scanner(System.in);

        Account_Holder accountHolder = new Account_Holder();

        // Ask the user to input the Account Holder's details
        System.out.print("Enter the Account Holder's Name: ");
        String name = sc.nextLine();
        accountHolder.setName(name);

        System.out.print("Enter the Account Holder's Mobile Number: ");
        String mobile = sc.nextLine();
        accountHolder.setMobile_no(mobile);

        System.out.print("Enter the Account Holder's Email: ");
        String email = sc.nextLine();
        accountHolder.setEmail(email);

        System.out.print("Enter the Account Holder's Date of Birth: ");
        String dob = sc.nextLine();
        accountHolder.setDOB(dob);

        // Create an Account object using the Account_Holder object
        System.out.print("Enter the Account Number: ");
        long accountNumber = sc.nextLong();
        System.out.print("Enter the Branch: ");
        sc.nextLine();
        String branch = sc.nextLine();
        Account account = new Account(accountNumber, branch, accountHolder);
        Banker b1 = Banker.getInstance();
        b1.addAccount(account);
        System.out.println(b1.toString());
    }
}
```

Output: -



```
Enter the Account Holder's Name: Smit
Enter the Account Holder's Mobile Number: 9586891408
Enter the Account Holder's Email: smitpadshala99@gmail.com
Enter the Account Holder's Date of Birth: 14/08/2003
Enter the Account Number: 1308
Enter the Branch: Nikol
[Account Number:- 1308  Branch:- Nikol
Account Holder Details:-
Name:- Smit
Mobile Number:- 9586891408
Email ID:- smitpadshala99@gmail.com
Date of Birth:- 14/08/2003]
```