|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| Advanced Programming |
| |  |  |  | | --- | --- | --- | | S K D Tharindu Madusanka | 9/12/22 | Software Engineering | |

Contents

[1. Examine the key components related to the object-oriented programming paradigm, analyzing design pattern types. 3](#_Toc113900617)

[1.1. Programming Paradigm 3](#_Toc113900618)

[1.1.1. Characteristics of paradigms 3](#_Toc113900619)

[1.1.2. Object-Oriented Programming Paradigm 4](#_Toc113900620)

[1.1.3. Characteristics of OOP 5](#_Toc113900621)

[1.1.5. Relationships 30](#_Toc113900622)

[1.2. Design Patterns 37](#_Toc113900623)

[1.2.1. Types of design patterns 37](#_Toc113900624)

[1.3. Relationships between Design Patterns and OOP 52](#_Toc113900625)

[2. Part 02 54](#_Toc113900626)

[2.1. Documentation 54](#_Toc113900627)

[2.2. Use Case Diagram 54](#_Toc113900631)

[2.3. Activity Diagram 56](#_Toc113900632)

[2.4. Class Diagram 57](#_Toc113900633)

[3. Part 03 59](#_Toc113900634)

[3.1. Design Pattern 59](#_Toc113900635)

[3.2. Classes 61](#_Toc113900636)

[3.3. Interfaces 74](#_Toc113900640)

[3.3.1. Homepage 75](#_Toc113900641)

[3.3.2. Login Type 75](#_Toc113900649)

[3.3.3. Login Interface 76](#_Toc113900650)

[3.3.4. Dashboard 77](#_Toc113900651)

[3.3.5. Ticket Booking Interface 78](#_Toc113900652)

[3.3.6. Booking View Interface 79](#_Toc113900653)

[3.3.7. Booking Update Interface 79](#_Toc113900654)

[3.3.8. Movies Add Interface 80](#_Toc113900668)

[3.3.9. Movie Display Interface 80](#_Toc113900669)

[3.3.10. Movie Update Interface 81](#_Toc113900670)

[3.3.11. Register Interface 81](#_Toc113900671)

[3.3.12. Update Customer Interface 82](#_Toc113900672)

[3.3.13. Manager checking Interface 82](#_Toc113900673)

[4. Part 04 83](#_Toc113900674)

[4.1. Why I choose Factory method pattern? 83](#_Toc113900675)

[4.2. Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types. 83](#_Toc113900676)

[4.2.1. Creational Pattern 83](#_Toc113900677)

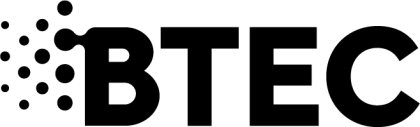
[4.2.2. Behavioral Pattern 85](#_Toc113900685)

[4.2.3. Structural Pattern 86](#_Toc113900686)

[4.2.4. Most appropriate Design Pattern for this scenario 88](#_Toc113900687)

[References 92](#_Toc113900688)





**Pearson**

**Higher Nationals in Computing**

**Unit 20: Advanced Programming**

**Assignment Brief Number: 1 - 4**

**Higher National Certificate/Diploma in**

**Computing**

|  |  |
| --- | --- |
| Student Name/ID Number | S K D Tharindu Madusanka |
| **Unit Number and Title** | 20: Advanced Programing |
| Academic Year | 2022 |
| Unit Tutor | Ms. Nethmi Hettiarachchi |
| Assignment Title | Advanced Programing |
| Issue Date | 24/02/2022 |
| Submission Date | 01/06/2022 |
| IV Name & Date | Mr. Aniz Saboordine / 14/05/2021 |

Assignment Brief

**Submission Format**

|  |
| --- |
| **Submission Format** |
| The submission is in the form of two documents:    Part 1: A report from 2000 to 3000 words in saved in either a Microsoft® Word® or PDF format    Part 2: A set of Star UML or Visio Diagrams, each diagram set must be clearly labelled with version numbers    Part 3: Code/Application with comments Softcopy, and evaluation report of the selected design pattern that you have applied    Part 4: Alternative Designs in UML that you have developed in Visio Diagrams    You are required to make use of headings, paragraphs, subsections and illustrations as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. |

# Examine the key components related to the object-oriented programming paradigm, analyzing design pattern types.

## Programming Paradigm

A paradigm is a way for solving a problem or completing a task. Programming paradigm is a means of solving a problem utilizing tools and techniques that are accessible to us while adhering to a certain approach. There are several programming languages that are well-known, but they all require some kind of methodology or strategy to be implemented, and this methodology/strategy is known as paradigms. Aside from different programming languages, there are a plethora of paradigms to meet any need.

### Characteristics of paradigms

* Procedural Paradigm
* Object-oriented Paradigm
* Functional Paradigm
* Logical Paradigm
* Database/Data driven programming approach

### Object-Oriented Programming Paradigm

The primary distinction between procedural and object-oriented programming is that object-oriented (OO) code will contain entities. A "message" in a system that may be "made hidden," "marked as read," "deleted," "sent," "drafted," and so on. Object-oriented programming is a programming paradigm. That uses objects (which include both data and methods) to achieve standard ,recyclable, and reusability. In order to develop applications and computer systems, objects, which are generally instances of classes, are utilized to communicate with one another.

In object-oriented analysis, object-oriented design, and object-oriented programming, the terms class and object are often used to express object-oriented ideas. I think class is a prototype, blueprint or plan that describes the common characteristics and operations that apply to a group of linked objects. A specific object is referred to as an instance of a class and is identifiable by a name or a number. The wide use of abstractions is a typical feature of object-oriented programming. A high-level description or model of a complicated or complex idea is called an abstraction. This “Abstractions” is not just found in OT. An object that is well-designed encompasses a single real-world entity with a set of properties and activities.

Object–oriented programming's key characteristics

* In software development, a bottom–up strategy is used.
* Object-oriented programs arranged into classes
* With ways to act on the data of objects, the focus is on data.
* Object-to-object interaction via functions
* Design reusability through the introduction of new classes and the addition of functionalities to existing ones

Examples of OOP languages – PHP, Java, C++, C#, Smalltalk, Delphi, Perl, Python, Ruby

### Characteristics of OOP

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

#### Encapsulation

The technique of encapsulating both properties and methods within a class is known as encapsulation. The internal information of a class may be hidden from the outer world through encapsulation. It allows just the interface supplied by the class to be used to access the parts of the class from the outside.

To get the combination of Java,

* Declare a class's variables to be private.
* To alter and inspect the variables' values, provide public setter and getter methods.

**This is illustrated by the following example to make it easier to understand.**

**Student Class**

package Encapsulation;

public class Student {

private String name;

private String id;

private int age;

public int getAge() {

return age;

}

public String getName() {

return name;

}

public String getId() {

return id;

}

public void setAge( int newAge) {

age = newAge;

}

public void setName(String newName) {

name = newName;

}

public void setIdNum( String newId) {

id = newId;

}

void run(){

System.out.println("Name: "+getName());

System.out.println("Age: "+getAge());

System.out.println("ID: "+getId());

}

}

**Run Class**

package Encapsulation;

public class Run {

public static void main(String args[]) {

Student card = new Student();

card.setName("James");

card.setAge(20);

card.setIdNum("12343ms");

card.run();

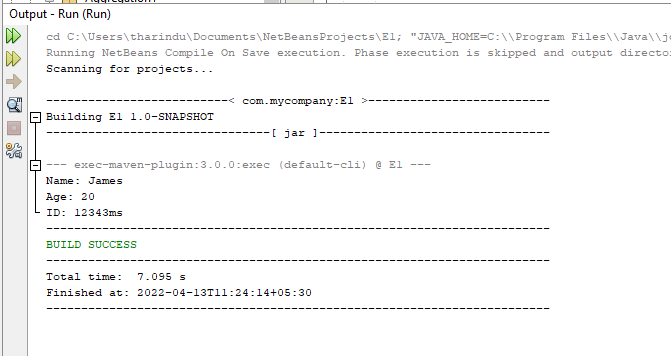
}

}

The example I am getting here is a display of name, identity card number and age for students.

The class Encapsulate is encapsulated in the given program since the variables are marked as private. The public get methods, such as getName(), getAge(), and getId(), are used to retrieve these variables. Setter methods are likewise specified as public and are used to set the values of variables. They include setName(), setAge(), and setId().

**Output:**



This is the output what I expected.

#### Abstraction

Simply said, abstraction "shows" just the important features of objects while "hiding" the irrelevant ones.

Abstraction is a key characteristic and building component of the Java language because it is an OOP language. An abstract class and interface are used to implement abstraction in Java.

So, with Java, how can we implement abstraction? For implementing abstraction, Java offers the non-access modifier "abstract." This abstract modification is only applicable to classes and methods, not variables.

The interface offers total abstraction, that is, it simply gives method prototypes rather than their implementation. An abstract class provides partial abstraction by obviating the need to implement at least one method.

**Abstract Method**

* An abstract class is one that has the abstract keyword in its name.
* The term "abstract method" refers to a method that is defined but not implemented.
* All abstract methods may or may not be present in an abstract class. Some of these might be concrete procedures.
* A method that is specified as abstract must always be redefined in the subclass, forcing overriding OR making the subclass abstract.
* Any class having one or more abstract methods must additionally include the abstract keyword in its declaration.
* An abstract class can't have any objects. An abstract class, on the other hand, cannot be directly created using the new operator.
* The default constructor is always present in an abstract class, and it can contain parameterized constructors.

**This is illustrated by the following example to make it easier to understand.**

**Abstract Class**

package Assignment;

abstract class Student {

abstract public void get();

}

We must be added “abstract” word. If you are not add it, it is not a abstract class.

A limited class that cannot be used to produce objects is an abstract class. It must be inherited from another class in order to be accessed.

**Student Details Class**

package Assignment;

public class StDetails extends Student{

@Override

public void get(){

System.out.println("Name: Tharindu");

System.out.println("Age: 21");

System.out.println("ID: 1020640");

System.out.println("Address: Mirigama");

System.out.println("Mobile: 0703585500");

System.out.println("Uni: BCAS");

}

}

**Exam Class**

package Assignment;

public class Exam extends Student{

@Override

public void get(){

System.out.println("Subject: Advance Programming");

System.out.println("Semester: 3");

System.out.println("Grade: Distinction");

}

}

**University Details** **Class**

package Assignment;

public class UniDetails extends Student{

@Override

public void get(){

System.out.println("Uni Name: British College of Applied Studies");

System.out.println("Department Computing");

System.out.println("Degree: Software Engineering");

}

}

**Run Class**

package Assignment;

import java.util.Scanner;

public class Run {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter number for get your details what you want \n Student Details =1 \n");

int numb = scan.nextInt();

System.out.println("Your number: "+numb);

if(numb==1){

Student S = new StDetails();

S.get();

}else if(numb==2){

Student E = new Exam();

E.get();

}else if(numb==3){

Student U = new UniDetails();

U.get();

}else{

Student S = new StDetails();

S.get();

Student E = new Exam();

E.get();

Student U = new UniDetails();

U.get();

}

}

}

#### Polymorphism

In object-oriented programming, polymorphism is a crucial notion. It merely indicates that there are several forms.

That is, in various contexts, the same item (method, operator, or object) might execute distinct activities.

Polymorphism have two types. First one is method “Runtime Polymorphism”.

* **Overriding Method**

Dynamic Method Dispatch is another name for it. It's the process of resolving a function call to an overridden method during runtime. Method Overriding is used to accomplish this form of polymorphism. The systematic rejection here occurs only when a derivative class has defined one member function of the main class. It's alleged that the basic function has been overridden.

**This is illustrated by the following example to make it easier to understand.**

* **Superclass**

package Polymorphism;

public class Student {

void Print()

{

// Print statement

System.out.println("Your Details \n");

}

}

* **Subclass1**

package Polymorphism;

public class StDetails extends Student{

@Override

public void Print(){

System.out.println("Name: Tharindu");

System.out.println("Age: 21");

System.out.println("ID: 1020640");

System.out.println("Address: Mirigama");

System.out.println("Mobile: 0703585500");

System.out.println("Uni: BCAS \n");

}

}

* **Subclass2**

package Polymorphism;

public class Exam extends Student{

@Override

public void Print(){

System.out.println("Subject: Advance Programming");

System.out.println("Semester: 3");

System.out.println("Grade: Distinction \n");

}

}

* **Subclass3**

package Polymorphism;

public class UniDetails extends Student{

@Override

public void Print(){

System.out.println("Uni Name: British College of Applied Studies");

System.out.println("Department Computing");

System.out.println("Degree: Software Engineering \n");

}

}

**Main class**

package Polymorphism;

public class Run {

public static void main(String[] args)

{

// Creating object of class 1

Student a = new Student();

a.Print();

// Now we will be calling print methods

// inside main() method

a = new StDetails();

a.Print();

a = new UniDetails();

a.Print();

a = new Exam();

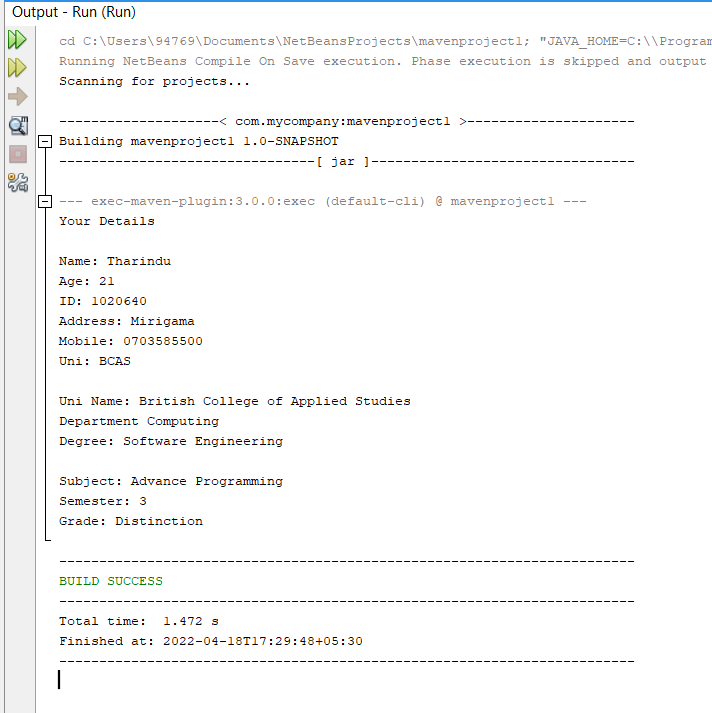
a.Print();

}

}

"Super Class" creates the main method. But running it there often doesn’t happen. But in this example I have used a few words to print it. The "Print ()" method here is implemented in "sub classes". The "@Override" method is used to implement it. If you do not use that method, you will not be able to run the "Print ()" method in sub classes under the same object name in the "Run Class". Also, here all sub classes must be made an extension of the super class. Or it may not be possible to run any of these methods and then it is not a "polymorphism" either.

* Output



**Why we use polymorphism?**

**Everyone living in this world has different commonalities. Those commonalities are with different differences from each other. It is used to present things or behaviors that exist in that or something under that common attribute. In the example above I am using a set of available information for "printing". Although there are similar features here, this method is important for presenting them separately.**

Second one is “**Compile - time Polymorphism**. Static polymorphism is another name for it. Function overloading or operator overloading is used to accomplish this form of polymorphism.

* **Overloading Method**

We can construct methods with the same name in a Java class if the parameters are different. Depending on the argument, the same method will do various tasks.

**This is illustrated by the following example to make it easier to understand.**

* Helper Class

//Helper class

//By using Different Types of Arguments

package Overloading;

public class Super {

//Method with 2 integer parameters

static int Addition (int part1, int part2){

//Returns product of integer numbers

return part1 + part2;

}

//Method with 2 double parameters. But use same name for both methods

static double Addition (double a, double b){

//Returns product of double numbers

return (a + b)/2;

}

}

* Main class

//Main class

package Overloading;

public class Run {

public static void main(String[] args) {

System.out.println(Super.Addition(78, 85));

System.out.println(Super.Addition(85.5,69.8));

}

}

Main driver method is running in this class. If it did not add, this class is not a main class and can’t run.

* Output



#### Inheritance

As we all know, a child inherits his parents' possessions. A similar technique is used in Java, where two classes are used.

1. Parent Class(Super class)
2. Child Class(Sub class)

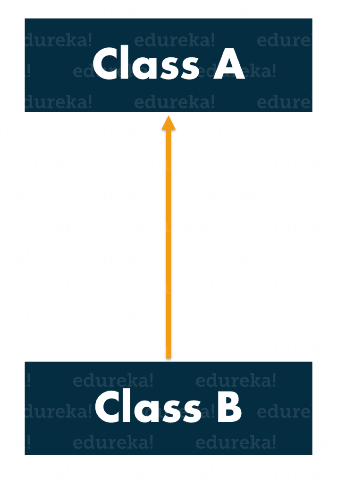
Here the class that inherits the property first is called the parent class. The property that exists in that parent class is then inherited class or several classes are referred to as children's classes. All property in the parent class belongs to these children's classes.

This is the basis of inheritance.

**Types of Inheritance**

Single Inheritance

Since there are only two classes here, the property of the parent class is inevitably inherited by the other class. Therefore, the code can be reused, and the necessary changes and special methods available only for the child class can be included.



* Parent Class

package Inherit1;

public class Animal {

public void eat(){

System.out.println("Eating foods");

}

}

* Child Class1

package Inherit1;

public class Dog extends Animal{

public void bark(){

System.out.println("Barking");

}

@Override

public void eat(){

System.out.println("Omnivorous");

}

}

* Main Class

package Inherit1;

public class Run {

public static void main(String[] args) {

Dog d = new Dog();

d.bark();

d.eat();

}

}

Multi-level Inheritance

* Parent Class

package Inherit1;

public class Animal {

public void eat(){

System.out.println("Eating foods");

}

}

* Child Class

package Inherit1;

public class Dog extends Animal{

public void bark(){

System.out.println("Barking");

}

@Override

public void eat(){

System.out.println("Omnivorous");

}

}

* Grandchild Class

package Inherit1;

public class Sheperds extends Dog{

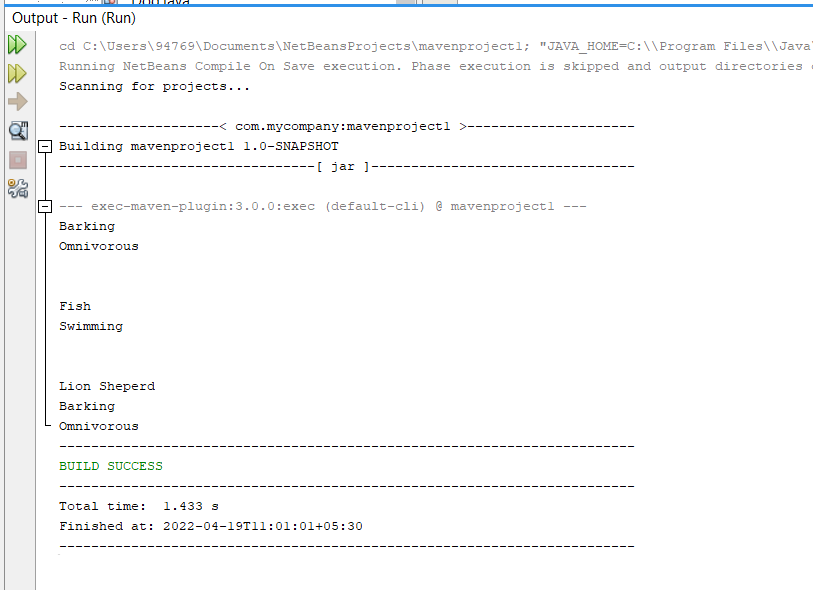
public void type(){

System.out.println("Lion Sheperd");

}

}

* Output



Hierarchical Inheritance

* Parent Class

package Inherit1;

public class Animal {

public void eat(){

System.out.println("Eating foods");

}

}

* Child Class1

package Inherit1;

public class Dog extends Animal{

public void bark(){

System.out.println("Barking");

}

@Override

public void eat(){

System.out.println("Omnivorous");

}

}

* Child class2

package Inherit1;

public class Swan extends Animal{

@Override

public void eat(){

System.out.println("Fish");

}

public void move(){

System.out.println("Swimming");

}

}

* Main class

package Inherit1;

public class Run {

public static void main(String[] args) {

Dog d = new Dog();

d.bark();

d.eat();

System.out.println("\n");

Swan s = new Swan();

s.eat();

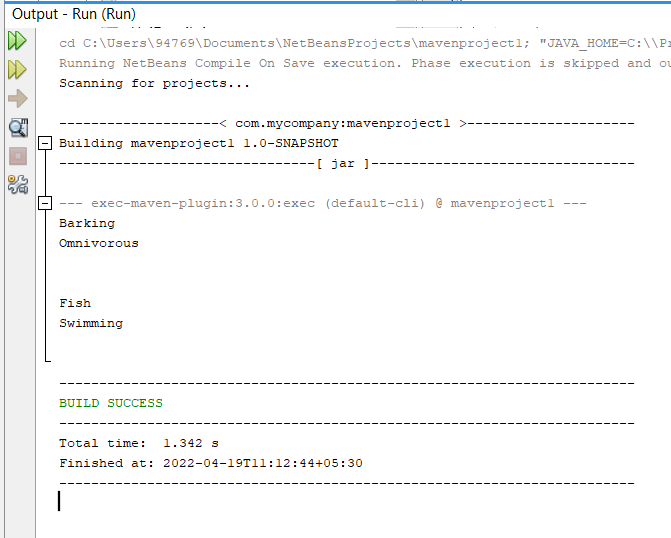
s.move();

System.out.println("\n");

}

}

* Output



#### Classes and Objects

##### Classes

A class is an object's blueprint. We must first declare the class before we can construct an object.

The class can be compared to a sketch (prototype) of a home. It covers all of the information regarding the floors, doors, and windows, among other things. We construct the home based on these descriptions. The item is a house.

We can make multiple objects from a class since many houses may be built from the same description.

Create class

package Assignment;

public class Class {

// fields

// methods

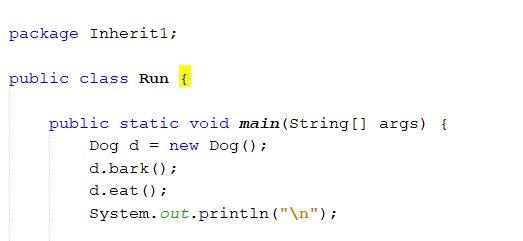
}

##### Objects

It represents real-life entities and is a fundamental unit of Object-Oriented Programming. A typical Java application produces a large number of objects, which interact via executing methods. An object is made up of the following elements:

* The state of an item is represented by its characteristics. It also reflects an object's attributes.
* The behavior of an object is represented by its methods. It also represents an object's interaction with other objects.
* Identity: It provides a thing a unique name and allows it to communicate with other objects.

Create object



### Relationships

#### Aggregation

Aggregation is what happens when a class has an entity reference. The HAS-A connection is represented via aggregation.

Consider the following scenario: the pet object has a variety of information, including an id, a name, age, and type. It also has another object called “details”, which has its own set of information, such as owner name, and address so on, as seen below.

Owner class

package Aggregation;

public class Owner {

String name, address;

public Owner(String name, String address){

this.name=name;

this.address=address;

}

}

Pet class

package Aggregation;

public class Pet {

String id, pname, type;

int age;

Owner owner;

public void card(String id, String pname, String type, int age, Owner owner){

this.id =id;

this.pname = pname;

this.type=type;

this.age=age;

this.owner=owner;

}

void display(){

System.out.println(id);

System.out.println(pname);

System.out.println(type);

System.out.println(age);

System.out.println("Owner details: "+owner.name+" - "+owner.address);

}

}

Main class

package Aggregation;

public class Run {

public static void main(String[] args) {

Owner details;

details = new Owner("Tharindu", "Mirigama");

Pet pet = new Pet();

pet.card("123wp", "Brown", "Sheperd", 6, details);

pet.display();

}

}

Here is a study on retrieving pet information. Two objects are created as "pet" and "details". The information under Details is brought to the "pet" object under "aggregation". The "Owner owner" variable in the pet class helps. If it had not been used, this would not have been possible under the "aggregation" method.

#### Composition

The "has-a" connection can be designed or implemented using the Composition. Both composition and inheritance are design methods. The "is-a" connection or relationship is implemented via the Inheritance. The "has-a" connection is employed in our software to ensure code reuse. We utilize an instance variable in Composition to refer to another object.

When one thing includes another object and that object is completely reliant on it, a composition connection between the two objects is conceivable. The enclosed object should not exist unless its parent object exists. To put it another way, it is a strategy for describing the relationship between two or more classes. We utilize the instance variable for this, which must be generated before it can be used.

#### Dependency

Dependency displays the interdependence of several elements within a system. A dependency relationship in UML is one in which a client (one element) is reliant on the provider (another element). It implies that a change to the provider needs a change to the customer in class diagrams, component diagrams, deployment diagrams, and use-case diagrams.

It has many types of relationships.

* <<derive>>
* <<friend>>
* <<instanceOf>>
* <<instantiate>>
* <<refine>>
* <<use>>
* <<substitute>>
* <<access>>
* <<import>>
* <<permit>>
* <<extend>>
* <<include>>

#### Generalization

It is the process by which two or more classes acquire common qualities and functions and act accordingly and maintain them as parent classes. There are many subclasses. Almost all of those subclasses are integrated with the parent class or super class. So we can say that subclass is "is-A" superclass.

Example:

* Parent class

package gene;

public class Animal {

String type;

String sound;

String legs;

}

* Sub class1

package gene;

public class Dog extends Animal{

String name;

String color;

int age;

public Dog(String name, String color, int age){

this.name=name;

this.age=age;

this.color=color;

}

public void get(){

System.out.println("Name: "+name);

System.out.println("Age: "+age);

System.out.println("Color: "+color);

}

}

* Main class

package gene;

public class Run {

public static void main(String[] args) {

Dog dog = new Dog("Brown", "White", 6);

dog.type = "Mammal";

System.out.println("Type: "+dog.type);

dog.get();

}

}

#### Association

## Design Patterns

A design pattern is a tried-and-true solution to a given issue or activity. This is the basic explanation of design patterns.

The finest practices utilized by skilled object-oriented software engineers or developers are represented by design patterns. This patterns are answers to common issues that software engineers confront while developing software. Several software engineers used trial and error to arrive at these solutions over a long period of time.

Really why we use these design patterns?

* They may be used in a variety of tasks.
* They offer solutions that aid in the definition of the system architecture.
* They document software engineering encounters.
* They make an application's design more transparent.
* Because they were created on the expertise and experience of skilled software engineers, they are well-proven and tested solutions.
* Design patterns do not ensure that an issue will be solved completely. They provide the system architecture clarity and the opportunity to design a better system.

### Types of design patterns

* Creational design patterns
* Structural design patterns
* Behavioral design patterns

#### Creational design patterns

The way objects are created is the subject of creational design patterns. When a choice must be taken during the instantiation of a class, these design patterns are employed.

As a example, creating a object for a class.



##### Factory Method Pattern

The factory pattern is one of Java's most popular design patterns. This design pattern is classified as a creational pattern since it gives one of the most effective ways to construct an item.

We generate objects without disclosing the creation mechanism to the client in the Factory design, and we refer to freshly formed objects using a common interface. Virtual Constructor is another name for the Factory Method Pattern.

Advantages of factory pattern

* Sub-classes can select the sort of object they want to produce using the Factory Method Pattern.
* It encourages loose coupling by removing the requirement for application-specific classes to be bound into the code. That is, the code only interacts with the resultant interface or abstract class, and it will function with any classes that implement or extend that interface or abstract class.

Usage opportunities

* When a class is unsure of which sub-classes it will need to generate
* When a class specifies the objects to be produced, its sub-classes must do so.
* When the parent classes decide whether or not to create objects for their sub-classes.

**Calculate Hotel Bill : A Real World Example of Factory Method**

* Abstract class

package com.company;  
  
abstract class Plan {  
  
 protected double rate;  
 abstract void getPrice();  
  
 public void calculate(int days){  
 System.*out*.println(days\*rate);  
 }  
}

* Sub class1

package com.company;  
  
public class Individual extends Plan{  
  
 public void getPrice(){  
 rate=5500;  
 }  
}

* sub class2

package com.company;  
  
public class Family extends Plan{  
 public void getPrice(){  
 rate=17000;  
 }  
}

* sub class3

package com.company;  
  
public class Customize extends Plan{  
 public void getPrice(){  
 rate=25000;  
 }  
}

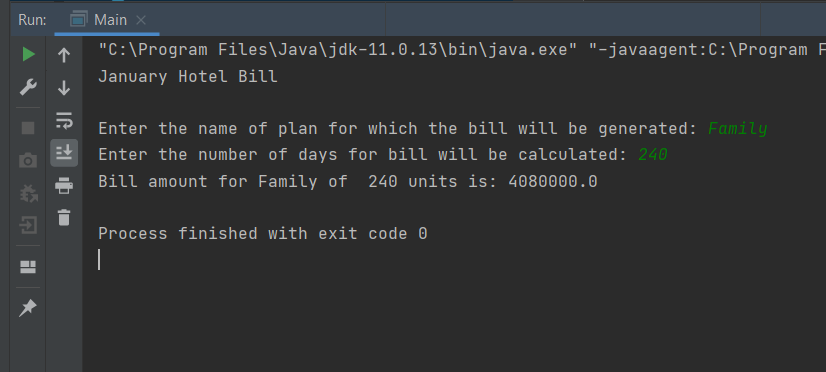
* pattern class

package com.company;  
  
public class GetPlan {  
 public Plan getPlan(String Type){  
 if(Type == null){  
 return null;  
 }  
 if(Type.equalsIgnoreCase("Family")) {  
 return new Family();  
 }  
 else if(Type.equalsIgnoreCase("Individual")){  
 return new Individual();  
 }  
 else if(Type.equalsIgnoreCase("Customize")) {  
 return new Customize();  
 }  
 return null;  
 }  
}

* Main class

public class Main {  
  
 public static void main(String[] args) throws IOException {  
 GetPlan planFactory = new GetPlan();  
  
 System.*out*.println("January Hotel Bill \n");  
  
  
 System.*out*.print("Enter the name of plan for which the bill will be generated: ");  
 BufferedReader br=new BufferedReader(new InputStreamReader(System.*in*));  
  
  
 String planName=br.readLine();  
 System.*out*.print("Enter the number of days for bill will be calculated: ");  
 int days=Integer.*parseInt*(br.readLine());  
  
 Plan p = planFactory.getPlan(planName);  
 //call getPrice() method and calculateBill()method of DomesticPlan.  
  
 System.*out*.print("Bill amount for "+planName+" of "+days+" units is: ");  
 p.getPrice();  
 p.calculate(days);  
  
 }  
}

* Output

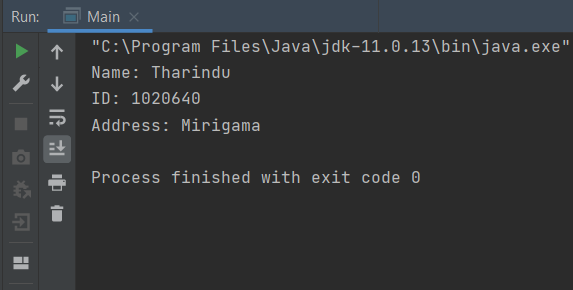


##### Singleton Pattern

The singleton pattern is one of Java's most basic design patterns. This design pattern is classified as a creational pattern since it gives one of the most effective ways to construct an item.

This pattern uses a single class that is in charge of creating an object while ensuring that only one object is generated. This class provides a method to access its single object, which may be accessible without having to initialize the class's object.

* Singleton Class
* package com.company;  
    
  public class SingleObj {  
   int id;  
   String name;  
   String address;  
    
   private static final SingleObj *instance* = new SingleObj();  
    
   private SingleObj(){  
   }  
   public static SingleObj getInstance(){  
   return *instance*;  
   }  
    
   public void display(){  
   System.*out*.println("Name: "+name);  
   System.*out*.println("ID: "+id);  
   System.*out*.println("Address: "+address);  
   }  
  }
* Main class
* package com.company;  
    
  public class Main {  
    
   public static void main(String[] args) {  
    
   SingleObj obj = SingleObj.*getInstance*();  
   obj.id=1020640;  
   obj.name="Tharindu";  
   obj.address="Mirigama";  
   obj.display();  
   }  
  }
* output



#### Structural Patterns

The composition of classes and objects to construct bigger structures is the subject of structural design patterns. By recognizing the linkages, structural design patterns simplify the structure.

These patterns are concerned with how classes inherit from one another and how they are made up of other classes.

It have 7 types of patterns.

* Adapter Pattern
* Bridge pattern
* Composite pattern
* Decorator pattern
* Façade pattern
* Flyweight pattern
* Proxy pattern

##### Adapter pattern

This an Adapter Pattern simply "helps to convert the interface of a class into another interface that a customer desires," according to the definition.

To put it another way, to offer an interface that meets the needs of the client while utilizing the functions of a class with a different interface. Wrapper is another name for the Adapter Pattern.

**Advantages**

* It enables the interaction of two or more previously incompatible things.
* It allows existing functionality to be reused.

Usage opportunities

* When an object has to use a class that has an incompatible interface.
* When you need to make a reusable class that works with other classes that don't have the same interface.
* When you need to make a reusable class that works with other classes that don't have the same interface.

**Example**

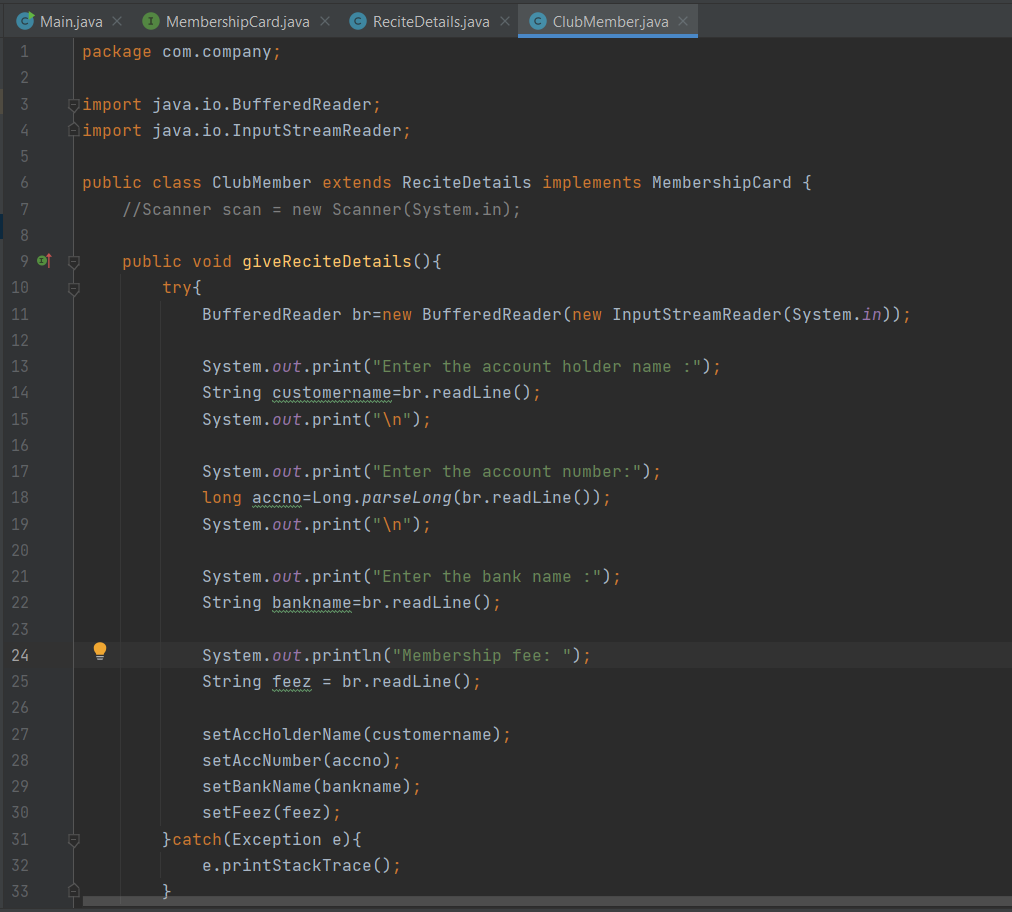
Interface Class

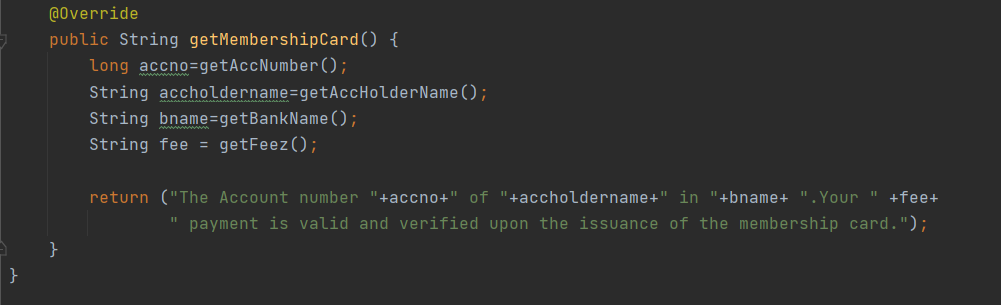


Adaptee Class

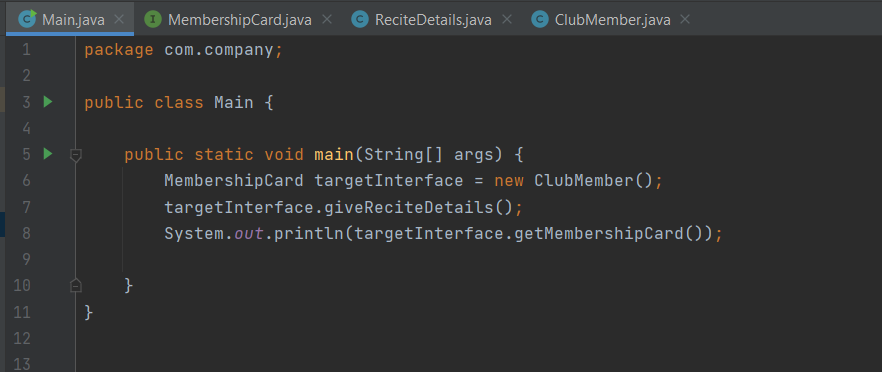


Adapter Class

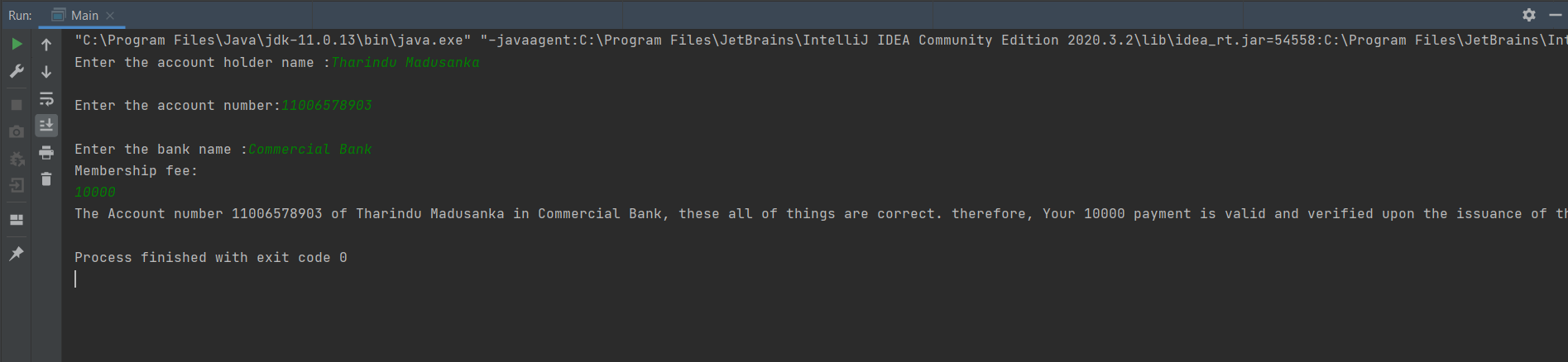




Main Class



Output



##### Bridge Pattern

When we need to separate an abstraction from its implementation so that both may change independently, we utilize Bridge. This design pattern is classified as a structural pattern because it provides a bridge structure that connects implementation and abstract classes.

This design uses an interface as a bridge, separating the functionality of concrete classes from the functionality of interface implementer classes. Structures of both sorts of classes can be changed without impacting the other.

The following example illustrates how to use the bridge pattern.

**Advantages**

* It allows the implementation and the user interface to be separated.
* It increases the extensibility of the system.
* It allows the client to be unaware of implementation specifics.

##### Decorator Pattern

A user can use the Decorator pattern to add additional functionality to an existing object without changing its structure. This design pattern is classified as a structural pattern since it functions as a wrapper for an existing class.

This approach produces a decorator class that encapsulates the original class and adds functionality while leaving the signature of the class methods intact.

We'll show how to utilize the decorator pattern in the following example, in which we'll colorize a shape without changing its class.

**Example**

Interface

package com.company;  
  
public interface Interface {  
 void print();  
}

Class A

package com.company;  
  
public class A implements Interface{  
 public void print(){  
 System.*out*.println("++++");  
  
 }  
}

Class B

package com.company;  
  
abstract class B implements Interface{  
 private Interface newInter;  
  
 public B(Interface newInter){  
 this.newInter=newInter;  
 }  
  
 @Override  
 public void print() {  
 newInter.print();  
 }  
}

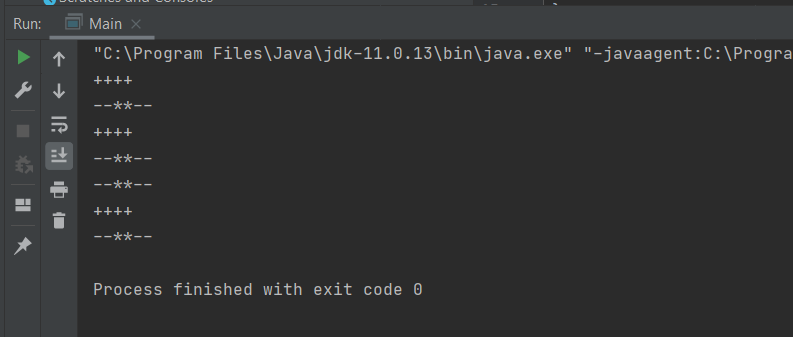
Class C

package com.company;  
  
public class C extends B{  
 public C(Interface newInter){  
 super(newInter);  
 }  
  
 @Override  
 public void print() {  
 super.print();  
 doPrint();  
 }  
 private void doPrint(){  
 System.*out*.println("--\*\*--");  
 }  
}

Main class

package com.company;  
  
public class Main {  
  
 public static void main(String[] args) {  
  
 Interface[] interfaces = {new C(new A()), new C(new C(new A())), new C(new A())};  
 for (Interface anInterface : interfaces) {  
 anInterface.print();  
 //System.out.print(" ");  
 }  
 }  
}

Output



#### Behavioral Patterns

A type of design that takes into account the existing interactions and responsibilities of objects. The interaction between the items in these design patterns should be such that they may readily communicate with one another while being loosely connected.

There are a number of different activities that can be done to avoid hard coding and dependencies. There the activation and the client loosely connect.

It have 12 types of patterns.

* Chain of Responsibility Pattern
* Command Pattern
* Interpreter Pattern
* Iterator Pattern
* Mediator Pattern
* Memento Pattern
* Observer Pattern
* State Pattern
* Strategy Pattern
* Template Pattern
* Visitor Pattern
* Null Object

##### Iterator Pattern

It provides a sequential approach to the raw material of an object as a whole without revealing its functionality.

This is also known as the cursor. We are now utilizing Iterator instead of Enumeration in the collection structure. It also has the following advantages:

* It allows you to traverse a collection in different ways.
* It streamlines the collection's user interface.

These are used when the object needs to support the collection when it has multiple movements, and when it is necessary to access an object collection without exposing the internal representation.

## Relationships between Design Patterns and OOP

An approach to programming or a notion that organizes codes, objects, and relationships is called object-oriented programming. Proposals tried-and-true patterns that produce squares or other objects to address a programming event.

In programming, design patterns are developed to address issues of the same nature. These design patterns all rely on classes and objects in this situation. The three primary classes and object types are as follows. On OOP alone, several patterns were developed. Experienced object-oriented software engineers adhere to design patterns as their best practices.

Instead of employing objects that are rapidly triggered by the new operator, these design patterns offer a means to build things while concealing the design logic. This allows the software more freedom to select the appropriate objects for a specific usage scenario.

Additionally, we may state that relationships and things have coordinates. proposes tried-and-true design methods for making objects to address a program event. On the OOP program, design patterns are typical methods of issue solution. Utilizing design patterns will increase reuse and decrease connection. The search for a universal pattern language may be traced back to the early days of active programs, bridges, or architecture. A certain conceptual structure that matches particular programming patterns is called object-oriented.

Classes and objects are the foundation of the OOP concept. To facilitate and automate operations that are now being performed or are anticipated to be performed, an individual or organization requires software. The problem is divided into portions before being given to the software engineers who will build the application. They exist as classes and objects. all things connected to the issue as objects. The items are then divided into molds that may be made. There are one or more connections between these classes when they are divided.

For instance, lessons in a program for supermarket management cover both products and patrons. When the clients receive the items, a new class labeled "Receipts or acquisitions" should be added so that they may be recorded and added to the database using the same method. The three groups of commodities, clients, and receivables or acquisitions are then related to one another.

A person who understands design patterns is not always a special OP. Programmers frequently employ.net and Java patterns without having studied them. The majority of Java Io namespaces contain classes that utilize the decorating pattern. Knowing that not only aids in comprehending things but also helps one become more or less of an expert in OOP.

These classes and objects are frequently utilized by programmers in their creation of programs, where they employ various techniques. But in this case, the software developers discovered that there was a pattern or repetition in these links. As software design patterns, the associated patterns were displayed. This OOP notion is what produces these patterns.

Pattern class and object composition are important considerations in these designs. The idea of legacy is used to create interfaces and to specify how to put pieces of an item together to create new functionality. A design pattern is, as the name suggests, a method for organizing how to overcome particular problems and challenges. There is a serious misunderstanding of design patterns. They are neither designs nor solutions. They serve as recommendations (patterns) for creating a real solution. A design pattern has restrictions that apply to a particular situation. Beginners frequently make the error of believing that design patterns are a model for programming; rather, programs should be seen as a collection of interactive design patterns. This is untrue. In light of this, they frequently attempt to program a solution by determining what pertains to a design pattern.

Solutions to problems in software engineering or programming technology should be reusable. The best programming techniques for solving typical issues while creating an application or system are design patterns.

A programming format is only a programming method or style. One illustration is object-oriented programming. You can utilize objects to access the entire program if you stick to this approach. You can effectively complete a job that you can easily connect to because objects include data and activity. Through a "supermarket management program," I already described this.

A design pattern now has the potential to be a tried-and-true response to a typical programming pattern. It is frequently regarded as the ideal procedure. There are a variety of design patterns available when creating a software using OOP that you may use to address certain issues.

We strive to apply design patterns to execute codes in a decent and effective way since effective and accurate code is crucial in software development. Patterns offer quick fixes for issues Design patterns are nothing more than a robust, scalable, readable, and adaptable solution to a recurrent issue.

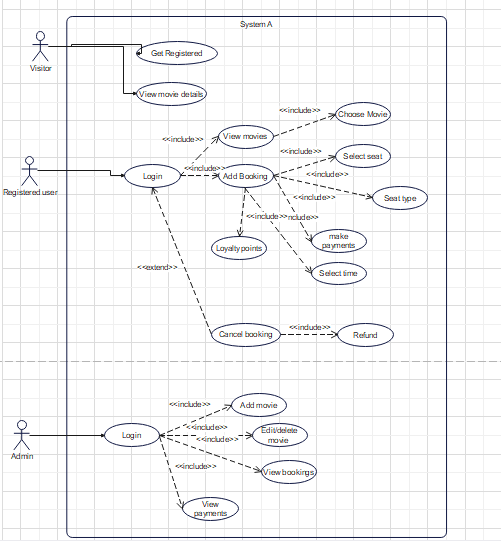
# Part 02

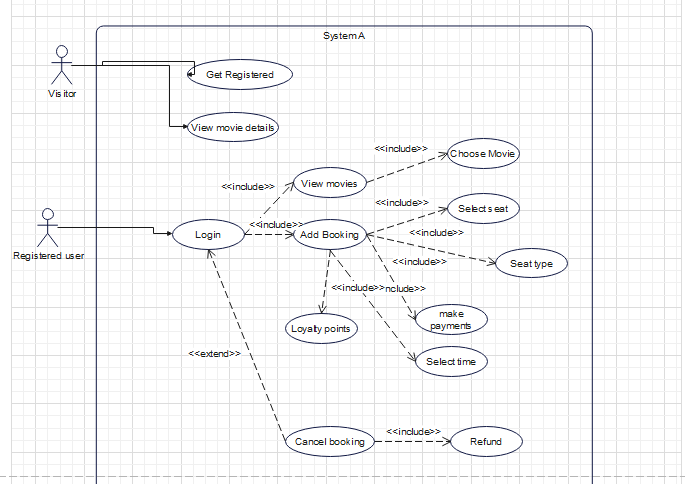
### Documentation

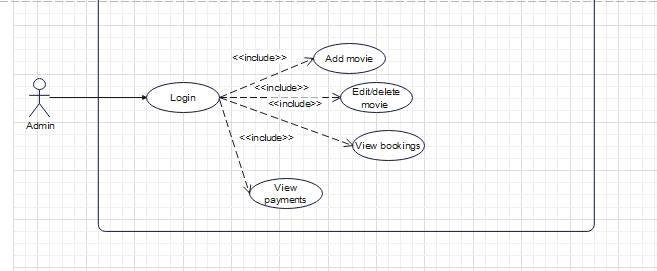
Here we study how to create software before creating it. After that study, several diagrams are created using the available facilities and possible tasks of the software. Before starting the implementation of this in any software, class diagram, UML diagram (use case diagram) and other diagrams should be created for the project. Because all those who contribute to this project should be given a good understanding about it. And this gives ease to find solutions by solving the problems that arise. Therefore, time can be saved. This will also contribute to the control of the budget in some way. You can study the diagrams below.



### Use Case Diagram



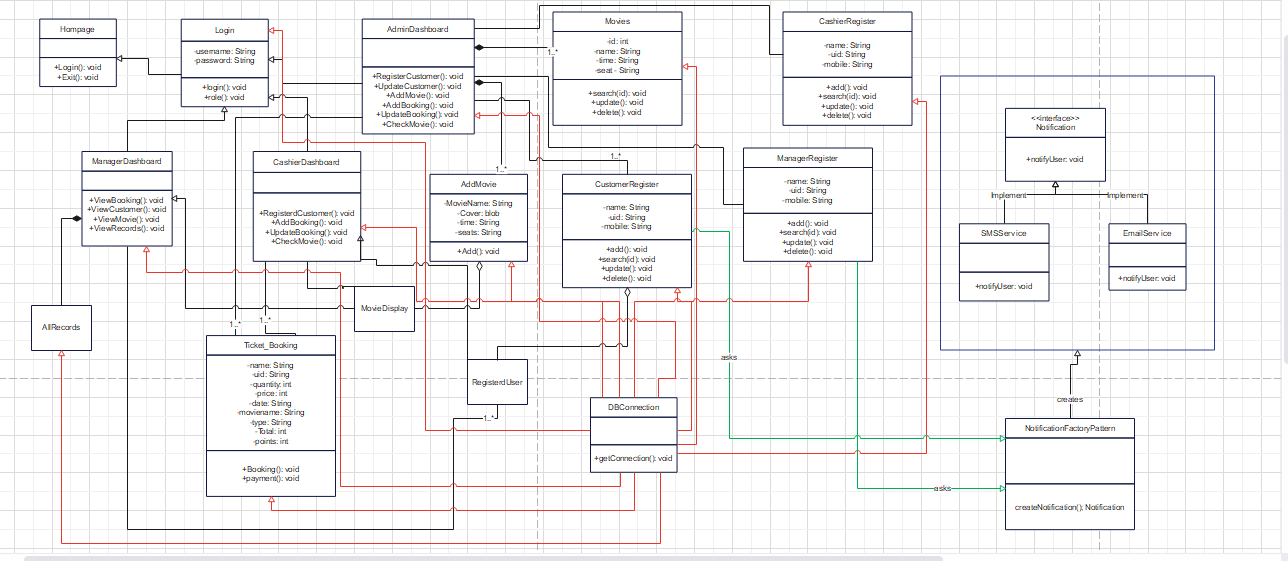


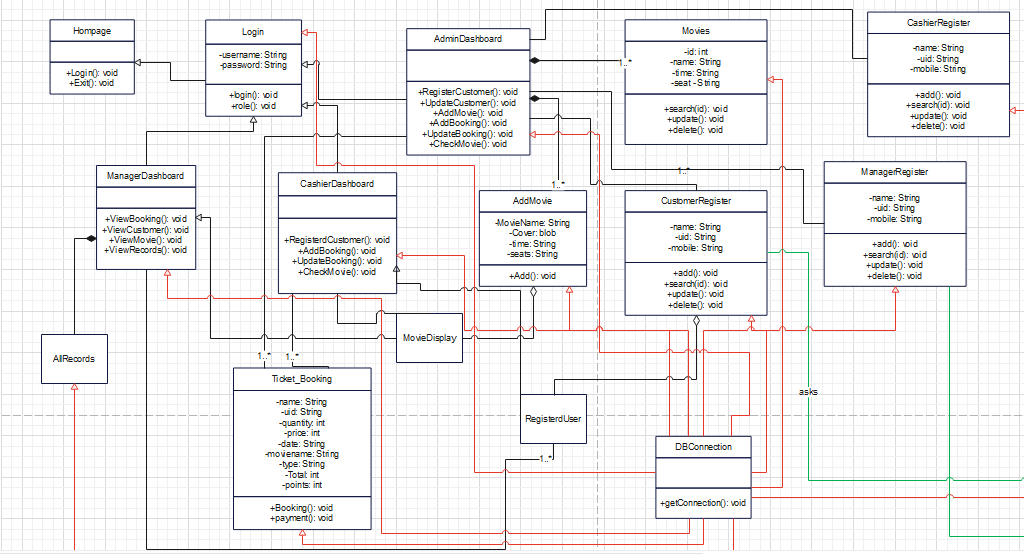


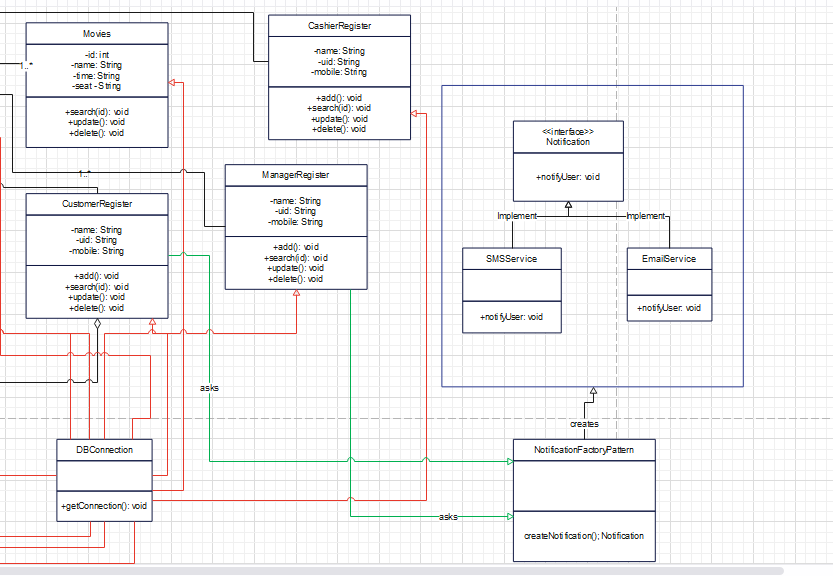
### Activity Diagram



### Class Diagram







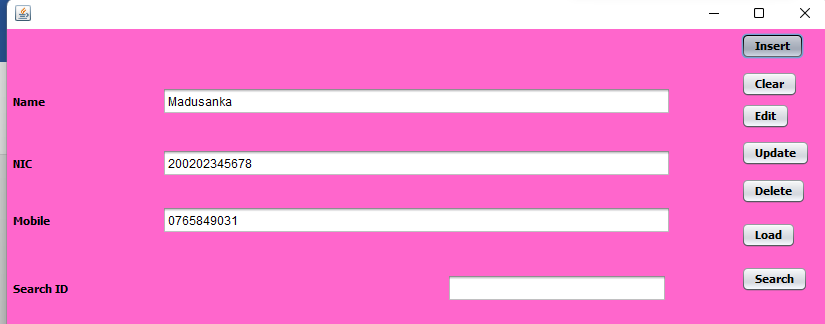
# Part 03

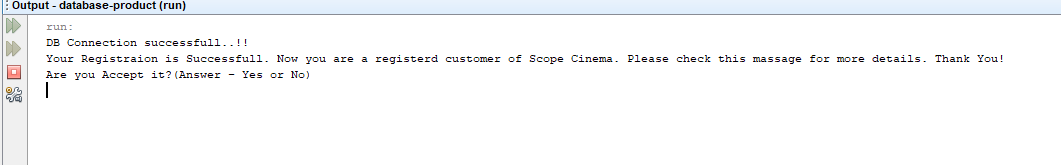
## Design Pattern

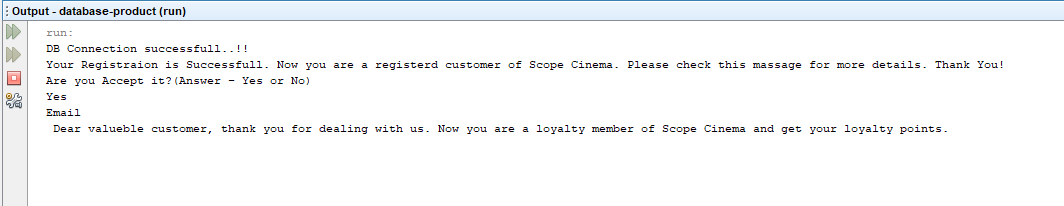
In this case, we used Factory Pattern. The factory design pattern is used when a superclass has several subclasses and we need to return one of the subclasses based on input. In this method, the factory class takes over the client program's responsibility for class instantiation.

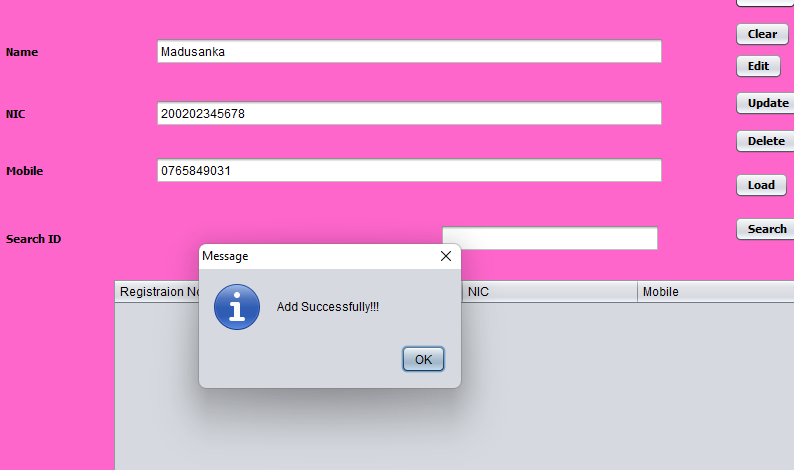
I am using the design pattern here to send a message to the people who are newly registering in the system. Here the next process is based on giving "Yes" or "No" to the message sent through an SMS. A message is actually sent to the employee's phone number of the registered customer. I have only created a sample method. After entering the data and clicking the insert button, any message will be displayed in the output of netbeans. When the question is answered as "Yes", the data entered will be stored in the database and if "No" is given, no data will be entered in the database.

If the answer is “Yes”, a message will be displayed in the interface with the following information entered.

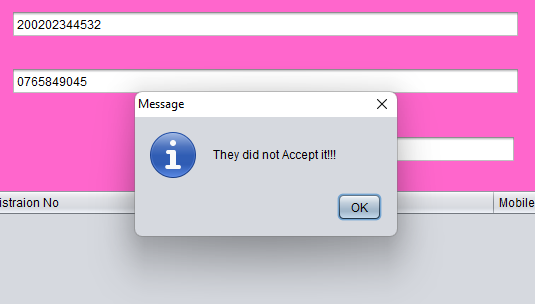








If it is checked as No, it will be displayed as below.



## Classes

* Notification Interface

public interface Notification {

void notifyUser();

}

* SMSService Class

package ScopeCinema;

public class SMSService implements Notification{

@Override

public void notifyUser()

{

// TODO Auto-generated method stub

System.out.println("Your Registraion is Successfull. Now you are a registerd customer of Scope Cinema. Please check your email for more details. Thank You!");

}

}

* EmailService

package ScopeCinema;

public class EmailService implements Notification{

@Override

public void notifyUser()

{

// TODO Auto-generated method stub

System.out.println("Email" +'\n'+" Dear valueble customer, thank you for dealing with us. please check below link for get your details and how many loyalty points your earned.");

}

}

* NotificationFactoryPattern

package ScopeCinema;

public class NotificationFactoryPattern {

public Notification createNotification(String media)

{

if (media == null || media.isEmpty())

return null;

switch (media) {

case "SMS":

return new SMSService();

case "EMAIL":

return new EmailService();

default:

throw new IllegalArgumentException("Unknown media "+media);

}

}

}

* CustomerRegister

package ScopeCinema;

//import library\_app.\*;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Scanner;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.swing.JOptionPane;

import javax.swing.table.DefaultTableModel;

public class CustomerRegister extends javax.swing.JFrame {

Connection con;

public CustomerRegister() {

initComponents();

createConnection();

}

void createConnection(){

try {

Class.forName("com.mysql.cj.jdbc.Driver");

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/database1", "root", "1234567890tm");

System.out.println("DB Connection successfull..!!");

} catch (ClassNotFoundException | SQLException ex) {

Logger.getLogger(CustomerRegister.class.getName()).log(Level.SEVERE, null, ex);

}

}

@SuppressWarnings("unchecked")

// <editor-fold defaultstate="collapsed" desc="Generated Code">

private void initComponents() {

jButton1 = new javax.swing.JButton();

jPanel1 = new javax.swing.JPanel();

jLabel2 = new javax.swing.JLabel();

jLabel3 = new javax.swing.JLabel();

jLabel4 = new javax.swing.JLabel();

nameTxt = new javax.swing.JTextField();

genderTxt = new javax.swing.JTextField();

schoolTxt = new javax.swing.JTextField();

clearBtn = new javax.swing.JButton();

insertBtn = new javax.swing.JButton();

editBtn = new javax.swing.JButton();

deleteBtn = new javax.swing.JButton();

loadBtn = new javax.swing.JButton();

jScrollPane1 = new javax.swing.JScrollPane();

jTable1 = new javax.swing.JTable();

id = new javax.swing.JLabel();

ID = new javax.swing.JTextField();

updateBtn = new javax.swing.JButton();

searchbtn = new javax.swing.JButton();

jButton1.setText("jButton1");

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT\_ON\_CLOSE);

jLabel2.setText("Name");

jLabel3.setText("NIC");

jLabel4.setText("Mobile");

genderTxt.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

genderTxtActionPerformed(evt);

}

});

clearBtn.setText("Clear");

clearBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

clearBtnActionPerformed(evt);

}

});

insertBtn.setText("Insert");

insertBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

insertBtnActionPerformed(evt);

}

});

editBtn.setText("Edit");

editBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

editBtnActionPerformed(evt);

}

});

deleteBtn.setText("Delete");

deleteBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

deleteBtnActionPerformed(evt);

}

});

loadBtn.setText("Load");

loadBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

loadBtnActionPerformed(evt);

}

});

jTable1.setModel(new javax.swing.table.DefaultTableModel(

new Object [][] {

},

new String [] {

"Title 1", "Title 2", "Title 3", "Title 4"

}

));

jScrollPane1.setViewportView(jTable1);

id.setText("Search ID");

updateBtn.setText("Update");

updateBtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

updateBtnActionPerformed(evt);

}

});

searchbtn.setText("Search");

searchbtn.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

searchbtnActionPerformed(evt);

}

});

javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);

jPanel1.setLayout(jPanel1Layout);

jPanel1Layout.setHorizontalGroup(

jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, jPanel1Layout.createSequentialGroup()

.addContainerGap(157, Short.MAX\_VALUE)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED\_SIZE, 375, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addGroup(jPanel1Layout.createSequentialGroup()

.addComponent(ID, javax.swing.GroupLayout.PREFERRED\_SIZE, 220, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addGap(80, 80, 80)

.addComponent(searchbtn)))

.addContainerGap())

.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, jPanel1Layout.createSequentialGroup()

.addContainerGap()

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(jPanel1Layout.createSequentialGroup()

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(jLabel2)

.addComponent(jLabel3)

.addComponent(jLabel4))

.addGap(112, 112, 112)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(nameTxt)

.addComponent(genderTxt)

.addComponent(schoolTxt))

.addGap(70, 70, 70))

.addGroup(jPanel1Layout.createSequentialGroup()

.addComponent(id)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, javax.swing.GroupLayout.DEFAULT\_SIZE, Short.MAX\_VALUE)))

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(loadBtn)

.addComponent(clearBtn)

.addComponent(insertBtn)

.addComponent(editBtn)

.addComponent(deleteBtn)

.addComponent(updateBtn))

.addGap(18, 18, 18))

);

jPanel1Layout.setVerticalGroup(

jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(jPanel1Layout.createSequentialGroup()

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(jPanel1Layout.createSequentialGroup()

.addGap(58, 58, 58)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

.addComponent(jLabel2)

.addComponent(nameTxt, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE)))

.addGroup(jPanel1Layout.createSequentialGroup()

.addGap(4, 4, 4)

.addComponent(insertBtn)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)

.addComponent(clearBtn)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addComponent(editBtn)))

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(jPanel1Layout.createSequentialGroup()

.addGap(32, 32, 32)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

.addComponent(jLabel3)

.addComponent(genderTxt, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE))

.addGap(17, 17, 17)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

.addComponent(jLabel4)

.addComponent(schoolTxt, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE))

.addGap(40, 40, 40)

.addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

.addComponent(id)

.addComponent(ID, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE)))

.addGroup(jPanel1Layout.createSequentialGroup()

.addGap(11, 11, 11)

.addComponent(updateBtn)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)

.addComponent(deleteBtn)

.addGap(18, 18, 18)

.addComponent(loadBtn)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)

.addComponent(searchbtn)))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, 27, Short.MAX\_VALUE)

.addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED\_SIZE, 275, javax.swing.GroupLayout.PREFERRED\_SIZE))

);

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());

getContentPane().setLayout(layout);

layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(jPanel1, javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, Short.MAX\_VALUE)

);

layout.setVerticalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addComponent(jPanel1, javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, Short.MAX\_VALUE)

);

pack();

}// </editor-fold>

private void clearBtnActionPerformed(java.awt.event.ActionEvent evt) {

nameTxt.setText(" ");

genderTxt.setText(" ");

schoolTxt.setText(" ");

}

private void genderTxtActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:

}

private void insertBtnActionPerformed(java.awt.event.ActionEvent evt) {

String name = nameTxt.getText();

String uid = genderTxt.getText();

String mobile = schoolTxt.getText();

try {

Statement stmt = con.createStatement();

String inserQuery = ("INSERT INTO user (UserId,NIC,Name,Mobile) VALUES ('"+uid+"','"+uid+"','"+name+"','"+mobile+"')");

stmt.execute(inserQuery);

stmt.close();

NotificationFactoryPattern notificationFactory = new NotificationFactoryPattern();

Notification notification = notificationFactory.createNotification("SMS");

notification.notifyUser();

Scanner myObj = new Scanner(System.in);

String name1 = myObj.nextLine();

System.out.println("Check email?"+ name1);

if("yes".equals(name1)){

Notification email = notificationFactory.createNotification("EMAIL");

email.notifyUser();

}else{

System.out.println("Thank You");

}

} catch (SQLException ex) {

Logger.getLogger(CustomerRegister.class.getName()).log(Level.SEVERE, null, ex);

}

JOptionPane.showMessageDialog(this, "Add Successfully!!!");

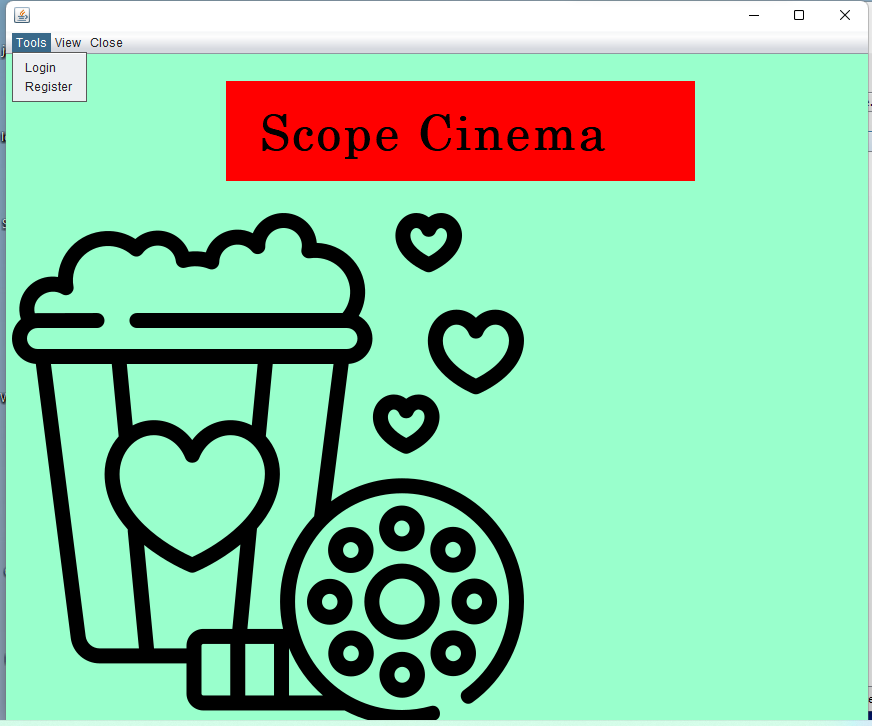
}

What I am using here is the "factory pattern". By using that pattern, what is done here is to convey a message to customers who are looking to register. A factory method is defined in the "Notification" interface. It is called "notifyUser()". The above factory method is implemented for the two subclasses "EmailService, SMSService" and the necessary objects are created. After that a concrete class is created for this. It is called "NotificationFactoryPattern" and contains the logic that this pattern needs to execute. The existing logic finally executes this pattern in the "CustomerRegister" class. You can observe the output below after running like this. This pattern handles the messaging required by a number of other main classes.



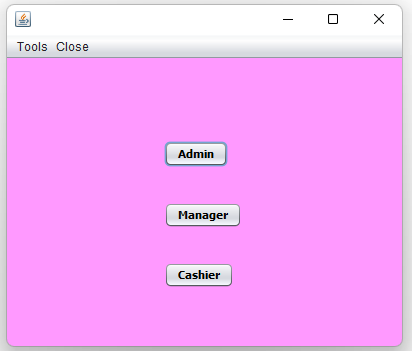
## Interfaces

### Homepage



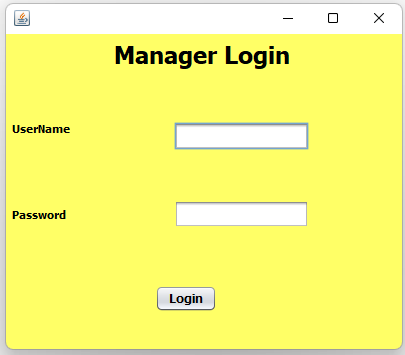


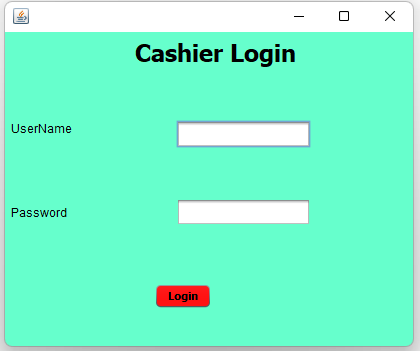
### Login Type



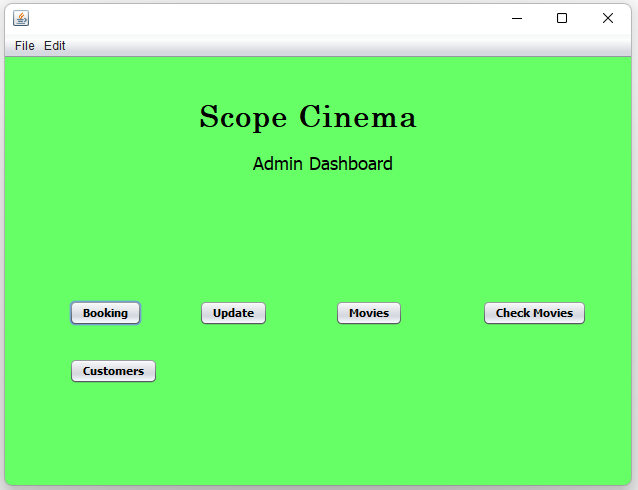
### Login Interface

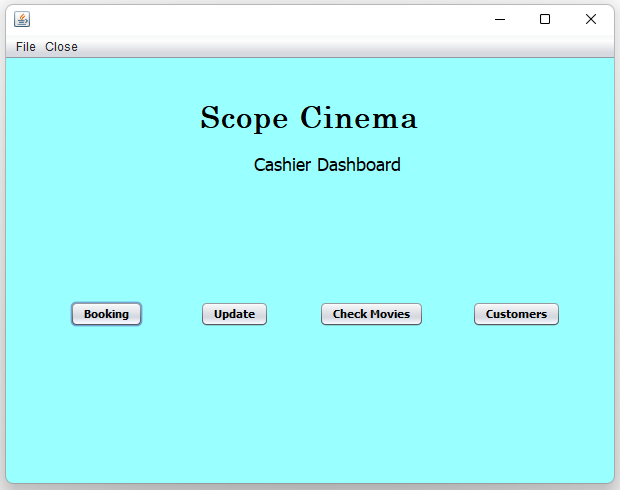


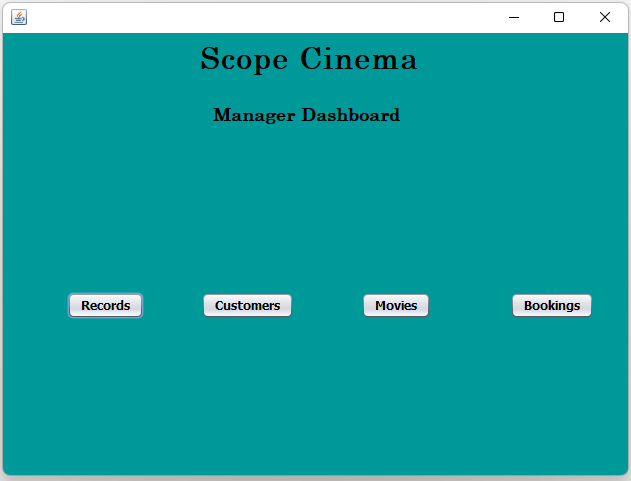




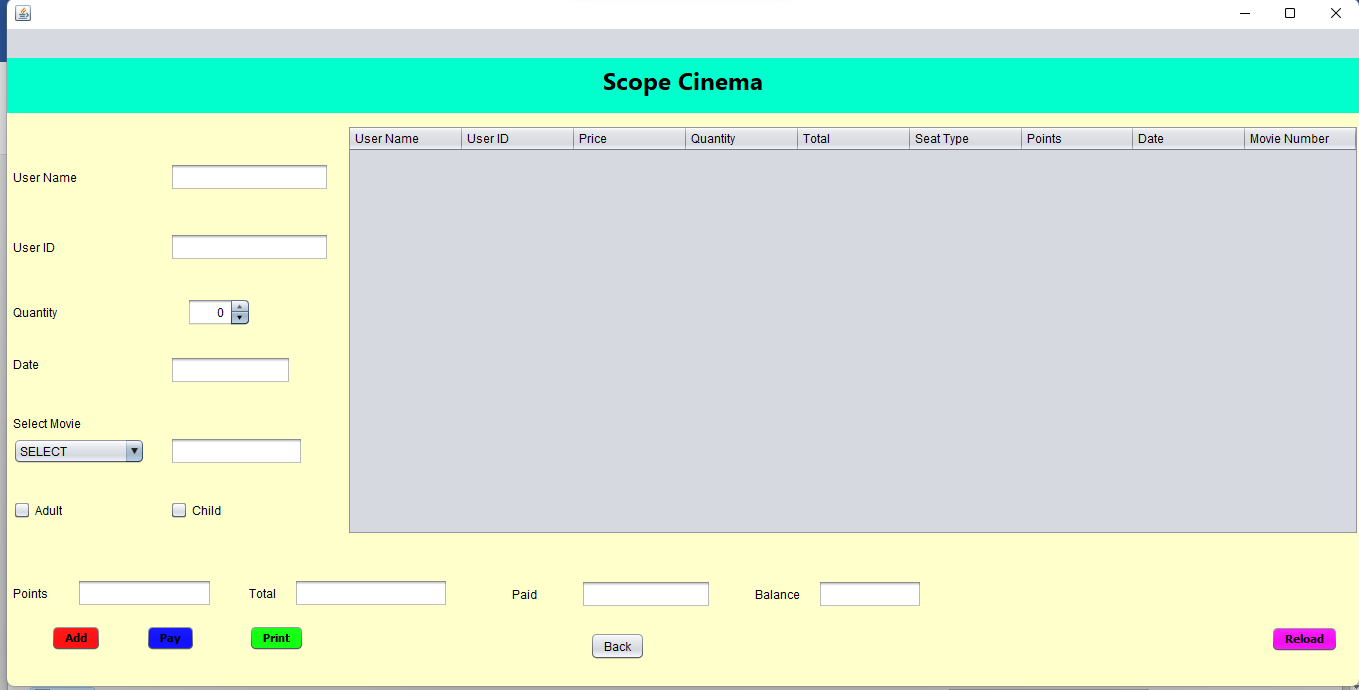
### Dashboard







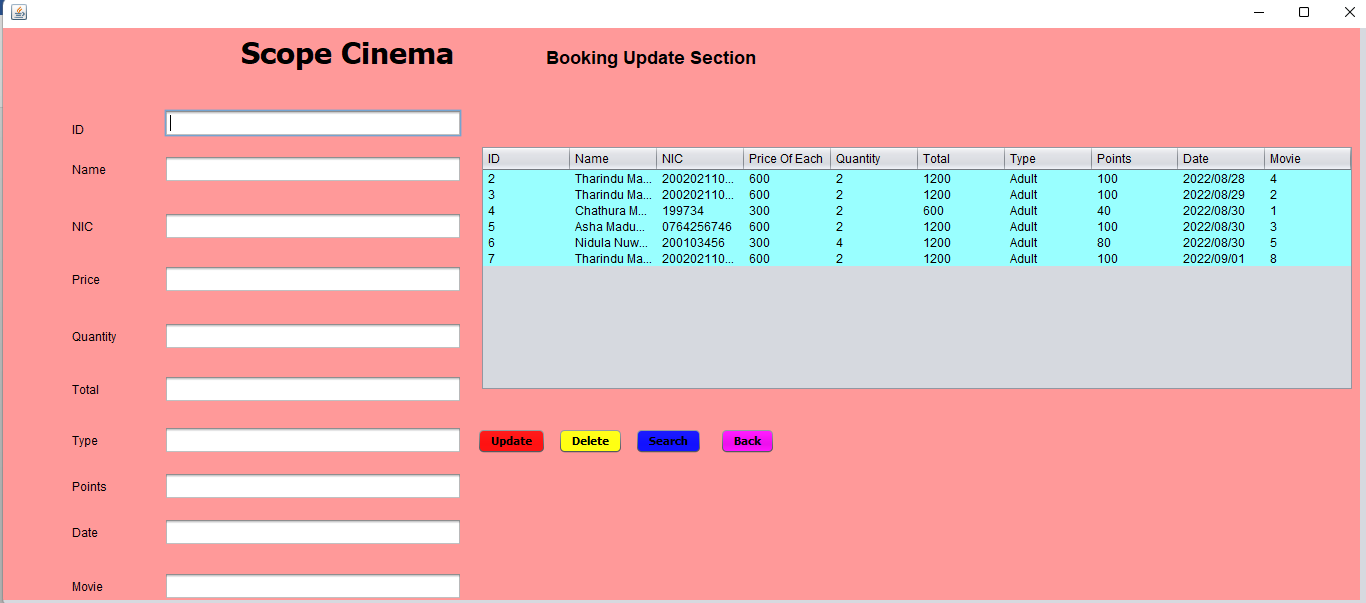
### Ticket Booking Interface



### Booking View Interface

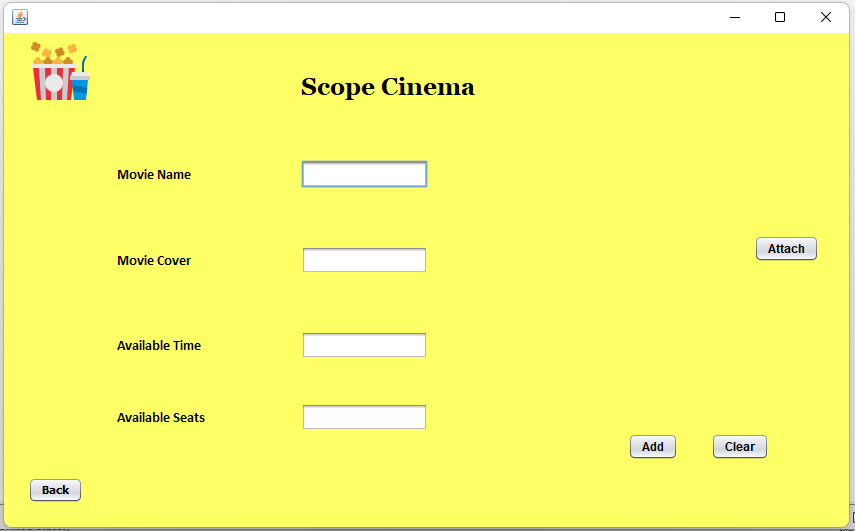


### Booking Update Interface

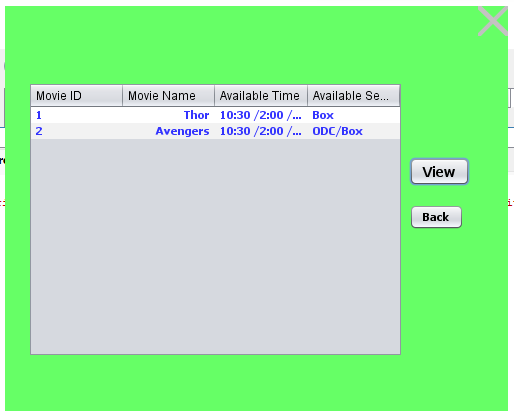




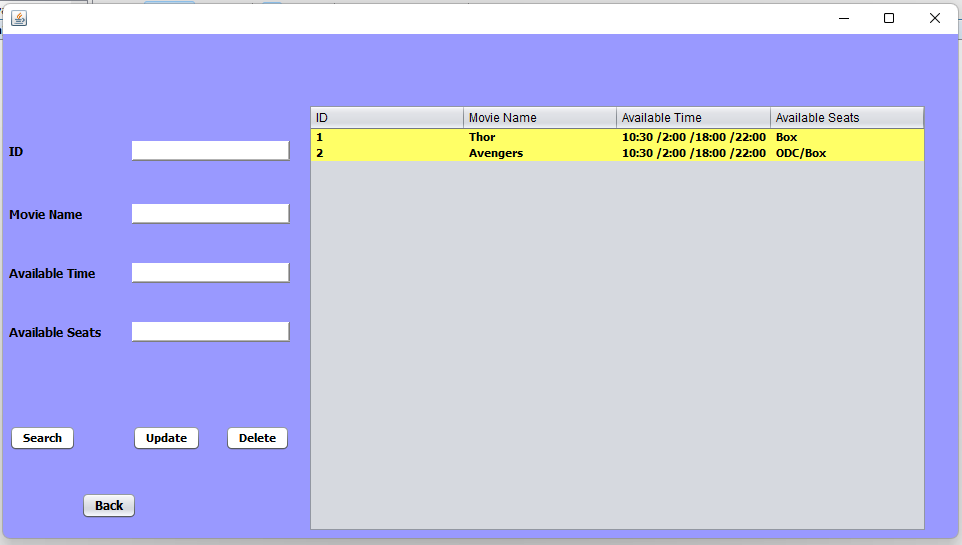
### Movies Add Interface



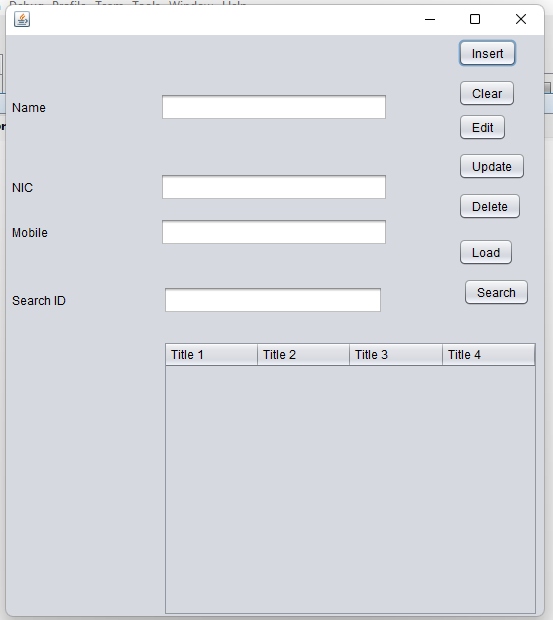
### Movie Display Interface



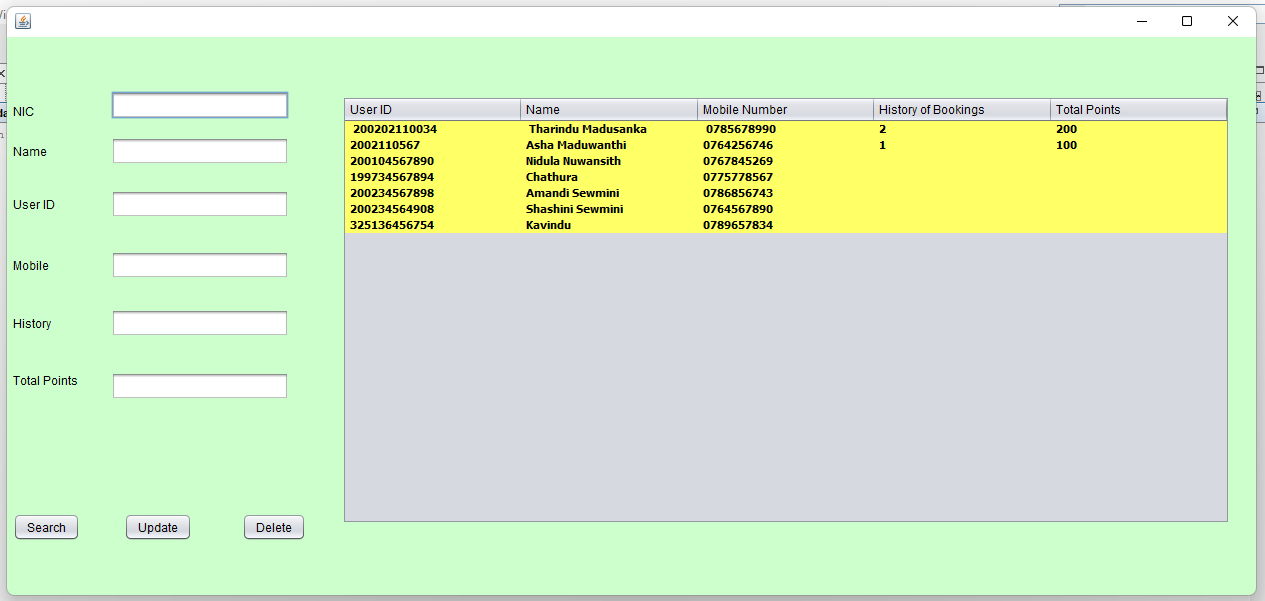
### Movie Update Interface



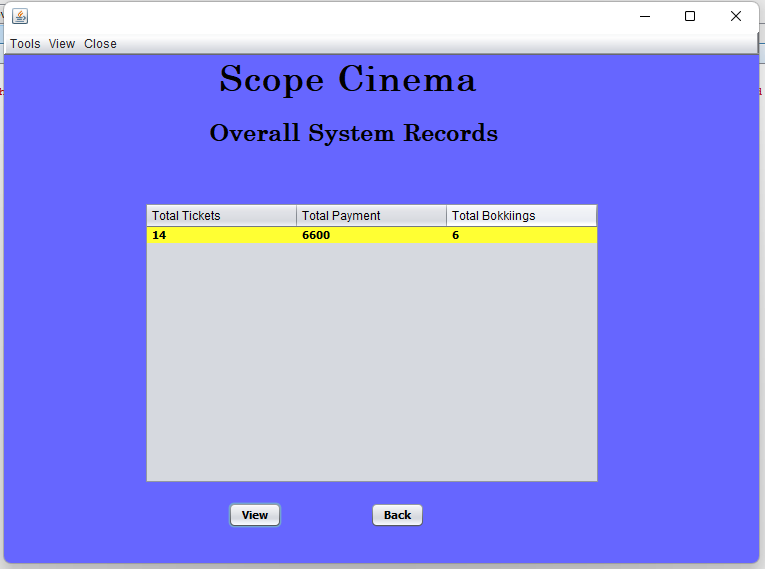
### Register Interface



### Update Customer Interface



### Manager checking Interface



# Part 04

## Why I choose Factory method pattern?

* As a software engineer, I picked factory design patterns because they allow us to reuse previously produced goods, conserving system resources.
* The factory method pattern gives subclasses the option of selecting the kind of objects to produce.
* By removing the requirement to tie application-specific classes into the code, it encourages loose coupling. This implies that the code only interacts with the resulting interface or abstract class, allowing it to function with any classes that extend or implement that interface.
* This technique might potentially be applied to let users of a library or framework enhance its elements.

## Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types.

Here I have used the Factory Method pattern as a design pattern. But from my point of view I need to confirm how the other existing patterns fit or not. Before that, one sub-pattern of each of the three main patterns, namely Creational, Behavioral and Structural, will be described with examples.

### Creational Pattern

#### Singleton Pattern

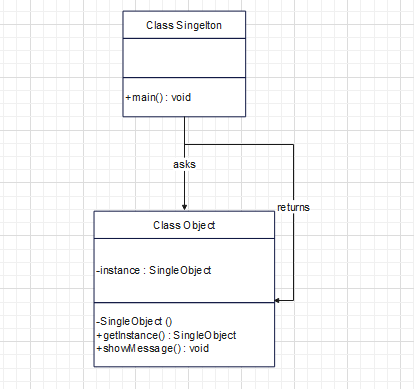
This is the simplest design pattern. This pattern defines a class that has only one instance. It then provides a global point of access. Runs the existing class through that global point.

In this pattern, a single class is in charge of creating an object while ensuring that only one object is produced. This class offers a method of immediately accessing the class's only object, eliminating the need to initialize it first.

To put it another way, a class must make sure that only one instance is produced and that only one object may be utilized by all other classes.

Implementation,

A class called "Object" will be formed. A static instance of an object class's constructor is also present. A static method is provided by the "Object" class to return its static instance to the outside world. Our demonstration class, Singleton, will use the Object class to obtain a SingleObject object.



This pattern has many advantages. because an object is not generated with every request, memory is saved. Only one instance is continuously used.

#### Factory Method Pattern

One of the most popular design patterns in Java is the factory pattern. This kind of design pattern falls under the category of a creational pattern since it offers one of the finest ways to make an object.

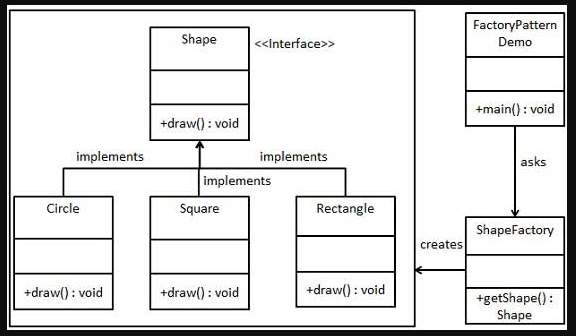
In the factory design, we build objects without disclosing the creation logic to the client and make use of a standard interface to refer to recently created objects.

The factory method pattern gives subclasses the option of selecting the kind of objects to produce. By removing the requirement to tie application-specific classes into the code, it encourages loose coupling. This implies that the code only interacts with the resulting interface or abstract class, allowing it to function with any classes that extend or implement that interface.

* When a class is unsure of the subclasses it will need to develop
* When a class requests that the objects to be produced be specified by its subclasses.
* When the parent classes decide to create objects for their child classes.

This is used in such situations.

Implementation,





### Behavioral Pattern

These patterns are used to recognize typical communication designing patterns between objects in software engineering. By doing this, these patterns boost the software development's communication's flexibility. They are interested in the relationship and accountability of items. where objects should be able to effortlessly communicate with one another during interactions. Therefore, the client implementation must be loosely coupled in order to prevent hard coding and dependencies.

#### Iterator Pattern

It provides a sequential approach to the raw material of an object as a whole without revealing its functionality.

This is also known as the cursor. We are now utilizing Iterator instead of Enumeration in the collection structure. It also has the following advantages:

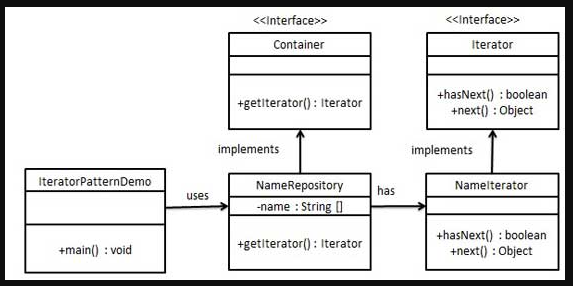
* It allows you to traverse a collection in different ways.
* It streamlines the collection's user interface.

These are used when the object needs to support the collection when it has multiple movements, and when it is necessary to access an object collection without exposing the internal representation.

Implementation,

A Container interface that repeats the iterator and an Iterator interface that describes the navigation function will be created. Concrete classes that implement the Container interface are in charge of implementing and utilizing the Iterator interface.

To print a Names saved as a collection in NamesRepository, IteratorPatternDemo, our sample class, will use NamesRepository, a concrete class implementation.



### Structural Pattern

#### Adapter Pattern

This an Adapter Pattern simply "helps to convert the interface of a class into another interface that a customer desires," according to the definition.

To put it another way, to offer an interface that meets the needs of the client while utilizing the functions of a class with a different interface. Wrapper is another name for the Adapter Pattern.

**Advantages**

* It enables the interaction of two or more previously incompatible things.
* It allows existing functionality to be reused.

Usage opportunities

* When an object has to use a class that has an incompatible interface.
* When you need to make a reusable class that works with other classes that don't have the same interface.
* When you need to make a reusable class that works with other classes that don't have the same interface.

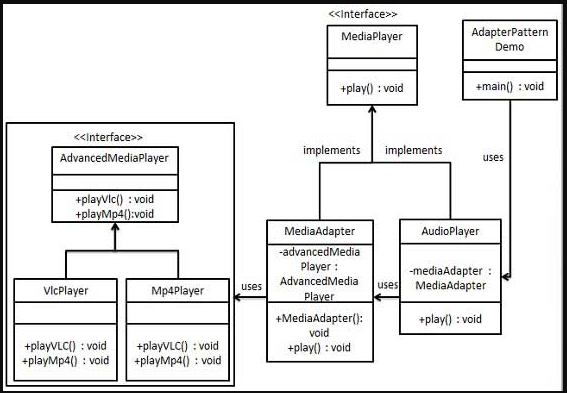
Implementation,

We have an AudioPlayer concrete class that implements the MediaPlayer interface, as well as a MediaPlayer interface. By default, AudioPlayer can play audio files in the mp3 format.

We also have concrete classes that implement the AdvancedMediaPlayer interface and another interface called AdvancedMediaPlayer. VLC and MP4 files can be played by these classes.

We want to provide support for more file types in AudioPlayer. To achieve this, we developed a media adapter class called MediaAdapter that implements the MediaPlayer interface and makes use of AdvancedMediaPlayer objects to play the necessary formats.

Without knowing the actual class that can play the desired format, AudioPlayer uses the adapter class MediaAdapter, to which the desired audio type is passed. Our demonstration class, AdapterPatternDemo, will play various formats using the AudioPlayer class.



### Most appropriate Design Pattern for this scenario

Under the above topic, I have discussed 4 sub-design patterns under 3 main designs. Under that discussion, I intend to certify that the most suitable design pattern is the most appropriate from my point of view.

* **Singleton pattern – Factory method pattern**

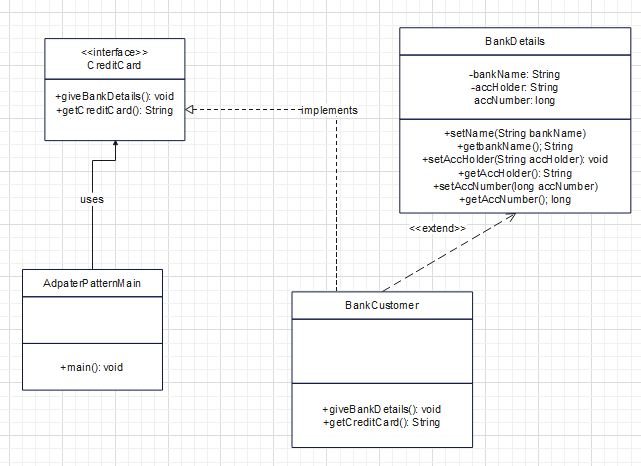
One of the simplest design patterns in Java is the singleton pattern. This kind of design pattern falls under the category of a creational pattern since it offers one of the finest ways to make an object. In singleton pattern only one object is created. I do all the necessary codes and message encoding. But in my program, I realized that doing it in one class is not the most appropriate. Because what works here is to send a message to a customer when he registers with our system. After confirming the message, the customer is included in the system. Therefore, it requires the contribution of several classes.

The factory pattern allows you to create different types of code and programs using separate classes without any confusion, and it is also easy to run them in my program. A "method" created using an interface is implemented in other concrete classes. That method is common to all classes and the necessary programs can be created in it. Thus, I have created several programs in several concrete classes and have included objects and methods in other classes as needed.

because an object is not generated with every request, memory is saved. Only one instance is continuously used. Although I can get these advantages using the singleton pattern, I can get more advantages using the factory pattern. Out of these 2 patterns I think the factory pattern is more appropriate. It was chosen due to the fact that many objects can be created, many different code programs in many classes can be created under one method and they can be run separately where needed.

**Adapter pattern – Factory method pattern**

The adapter design pattern allows two or more previously incompatible objects to interact. That is, it does the necessary work to connect several objects that cannot be connected together. It also provides the opportunity to reuse existing functionality. This converts an interface of a class to another interface.



You can prove it by this example. Another program has an extension attached to it while another method is running. The whole program is done in two parts and after joining a complex program is created.

But the factory pattern allows subclasses of the program to decide which class to instantiate using an interface. In the above adapter pattern, although methods are created through interfaces, in this pattern, unlike that pattern, other unrelated programs are not connected. Codes are created in required classes and an object is created in another class to run them. By that class, the examples or programs in the concrete classes run and execute the programs in the main class.

I was able to get a better advantage using this than the adapter method. The factory method pattern allows subclasses to choose the object type to create. Therefore, any necessary changes can be made to concrete classes. But in the adapter pattern, since the main class is connected to the interface, the adapter class and Adaptee classes are also affected.

It can also eliminate the need to bind to application-specific class code. Therefore, it can be easily integrated with other programs. In this project I have used the design pattern to register customers. After entering the customer's data, a message will be sent to their email account and after giving "Yes" to it, the information will be entered into the database. Linking the two programs was made easier by this factory pattern. Furthermore, another class can be added to this at any time and the necessary programs can be coded and connected to the main project. All that is needed is an extension with the interface.

But there is a complicated process for that in the adapter pattern. Therefore, I think that the factory pattern is more suitable for this project than the adapter pattern.

Adapter patterns are used to create programs by adapting incompatible interfaces or classes. It cannot be said that it is not one hundred percent suitable for this project. But I feel it is best to use the factory pattern in a program where all the subclasses to be created are not properly known. I see the Factory Method pattern as the design pattern that is more flexible, can solve problems more easily, can add many new classes, is less complicated, and can be used for many projects. I think it is the best for this project.

# References

Anon., n.d. *Design Patterns.* [Online]   
Available at: https://www.bing.com/ck/a?!&&p=850b4726b9e403b1JmltdHM9MTY2MjY4MTYwMCZpZ3VpZD0wMzEzZGY2Zi1hNDJmLTY2OTktMjlmYS1jZWE0YTU4MzY3NGEmaW5zaWQ9NTE4MA&ptn=3&hsh=3&fclid=0313df6f-a42f-6699-29fa-cea4a583674a&u=a1aHR0cHM6Ly9yZWZhY3RvcmluZy5ndXJ1L2Rlc2lnbi1wYXR0ZXJucw

freeCodeCamp, n.d. *What exactly is a programming paradigm?.* [Online]   
Available at: https://www.bing.com/search?q=programming+paradigm&cvid=b4fc020fddc7420da01ad021221662f3&aqs=edge.0.0l9.7966j0j1&pglt=171&FORM=ANNTA1&PC=U531

GeeksForGeeks, n.d. *Design Patterns.* [Online]   
Available at: https://www.bing.com/ck/a?!&&p=cb65a39a5a35096aJmltdHM9MTY2MjY4MTYwMCZpZ3VpZD0wMzEzZGY2Zi1hNDJmLTY2OTktMjlmYS1jZWE0YTU4MzY3NGEmaW5zaWQ9NTIyNQ&ptn=3&hsh=3&fclid=0313df6f-a42f-6699-29fa-cea4a583674a&u=a1aHR0cHM6Ly93d3cuZ2Vla3Nmb3JnZWVrcy5vcmcvZGVzaWduLXBhdH

GeeksForGeeks, n.d. *Introduction of Programming Paradigms.* [Online]   
Available at: https://www.bing.com/search?q=programming+paradigm&cvid=b4fc020fddc7420da01ad021221662f3&aqs=edge.0.0l9.7966j0j1&pglt=171&FORM=ANNTA1&PC=U531