Ramaiah Institute of Technology
Department of Information Science and Engg.

COURSE DESIGN, DELIVERY AND ASSESMENT

Semester: VI

Course Code: ISL66

Course Name: Object Oriented Analysis and Design Patterns Laboratory

Course Credits: 0:0:1

COURSE DESIGN, DELIVERY AND ASSESMENT

Course Title : OOADP Lab	Course Credits : 0:0:1
CIE : 50 Marks	SEE : 50Marks
Total No of Theory / Tutorial / Lab Hours : 0/0/28	

Prerequisites

Prerequisite Courses: Object Oriented Programming using Java/C++ Laboratory

Course Objectives

- Design a problem using UML
- Analyze the problem for possible restructuring
- Select candidate Design pattern for the problem
- Apply the most suitable design pattern for the problem
- Implement the problem using the selected design pattern

Syllabus

PART A

Case-study to understand the limitation of traditional Object Oriented Design and appreciate need for DesignPatterns. Use UML Notations to design.

You are a fresh analyst deputed to design the software for Decathlon Chain of Stores in Karnataka. You are informed about the Business Logic of Point of Sales criteria by Ms.Veronica Lodge, a dynamic business tycoon operating out of Decathlon Mumbai. She informs you that there are different types of Customers of Decathlon namely, Regular Customers, Senior Citizens and First Time Customers. Regular Customers are given a discount of 12%, Senior Citizens 10% and First

Time Customers 15%. Apart from this, based on the sales-index of previous day, a Store-level discount is determined every day. This is dynamic. **E.g**.Rs.100 off for every purchase above Rs.2000. Using the Object Oriented Principles of Encapsulation, Abstraction, Inheritance, Composition and Aggregation that you have studied until this semester, give at least two ways to design this system.

PART B

Common Case Study for Q#1 to Q#9
'Decathlon' is a Sports retail-store started in France. Today it spreads across 22 countries & has 900 outlets in these countries. It has a 'Point of Sale' software system called 'Decathlon POS', which uses various kinds of 3rd-party software sourced locally from the various countries they are established. You are a software consultant for Decathlon, in Bangalore, with a team of consultants reporting to you. When you analyze your answer for choosing a pattern, explain wherever applicable, keeping in mind the following four design principles:

- Separation of concerns
- Program to an interface, not a concrete implementation
- Prefer composition over inheritance
- Open-Close principle (Open for extension, Closed for modification)

1. **Adaptor (Structural):** To establish the 1st Decathlon store in Mauritius, you go along with Mr. Satya Nadella,an expert in finding 3rd-party partners. For e.g. a 3rd-party Tax-Calculator system to cater to the specifics of Salesand VAT (Value-added services Tax) tax calculations in different countries. He finds a 3rd-party Tax-Calculatorsystem called 'MauriTax' in Port Louis. The problem is, the APIs used by 'MauriTax' for tax-calculation is fixed& cannot be changed. *The 'MauriTax' APIs are incompatible with 'Decathlon POS'.*

   How will you use theAdaptor Pattern to design & implement?

2. **Strategy (Behavioural):** How will you use the Strategy Pattern to tackle the limitations of traditional ObjectOriented Design highlighted in PART A? *The design must handle varying price-schemes having different pricingalgorithms.* Design & implement.

3. **Factory Method (Creational):** The 'Decathlon POS' software system classifies its customers as senior-citizens(60 and above), First-Time customers, Regular Customers. There is a very high possibility that the CustomerType hierarchy will vary, depending upon the sales-pattern. **For e.g.** there could be the need to introduce newcategories based on the customer gender, different age groups for kids (0-5, 6-12), teenagers (13-19) and agegroups between 20 to 60(Twenties, 30s, 40s and 50s).

   You are advised by Mr.Sundar Pichai, the technical architect of your team, whom you trust, to use Factory MethodPattern in order to instantiate the above Customer Type hierarchy of concrete implementation of objects. Designand implement using this.

4. **Bridge (Structural):** You get a call from Ms.Masaba Gupta of Bangalore Decathlon office that there is a policy decision made globally to introduce discount slabs for a whole month twice in a year. The discount month will be in January and July after reviewing the sales made from Feb to June (first five months) and Aug to December (last five months) respectively. It is decided to provide four slabs of discounts in 2017, namely, 30%, 25%, 20% and 15%, based on the sports item purchased. **For e.g.** all tennis rackets could have a 20% discount while cricket bats could only have a 15% discount. All exercise tread-mills could be given a 30% discount while boxing-gloves could have a 25% discount. Point to be noted here is that, the slabs of discount may not remain the same in 2018. It is likely to vary year after year. The 'Decathlon POS' software system classifies its customers as Senior-Citizens (60 and above), First-Time Customers, Regular Customers as of now. There is a very high possibility that the Customer Type hierarchy will vary, depending upon the sales-pattern. **For e.g.** there could be the need to introduce new categories based on the customer gender.

   Use the Bridge Pattern to design & implement, *so that both the Customer Type hierarchy of classes as well as the Discount Percentage hierarchy of classes can both vary independently?* That is, they are not tied to each other.

5. **Observer (Behavioural):** There will be different discounts being offered for the sports items in Decathlon Stores across the globe for different festivals being celebrated in the various countries these stores are established. Assume that the Decathlon Chain of Stores fixes a particular discount slab for its items for a festival of a country.

   Use the Observer Pattern to design and implement a system to notify the customers of the Decathlon stores of that country about the various festival / seasonal discount rates as and when they are announced.

6. **Façade (Structural):** You get a call from Ms.Betty Cooper of Bangalore Decathlon office that there is a policy decision made globally to incorporate some new rules for 'Process Sale Use-Case'. **For e.g.** if payment is made via gift-certificate, the customer can buy only one item for the amount in the certificate. No other items can be bought with that gift-certificate. There must be no cash-back to the customer if the item costs less than amount specified in the gift-certificate. If the item costs more, the excess payment can be accepted via cash only & not credit / debit cards. When a new sale is created, these rules must become effective. You come to know from Mr. Satya Nadella, an expert in finding 3rd-party partners, that the Italian Competitor for Decathlon called 'Sport 2000' has a ready-made 'rule-engine' sub-system for this, whose specific implementation details is not known yet, as the business heads of Decathlon & Sport 2000 are chalking out the software purchase terms. This Sport 2000 rule-engine will be responsible for evaluating a set of rules against an operation & indicating if any of the rules invalidated the operation (e.g. 'makeNewSale' operation).

   How will you use the Façade pattern to provide a common unified interface to a dissimilar set of implementations, developed by a 3rd-party vendor, the implementation details are not known to you?

7. **Abstract Factory (Creational):** As an analyst in charge of designing the Decathlon POS Software, you realize the need to streamline the creation of objects belonging to different products in the Decathlon store. There are two major categories of products:
   a) For differently abled sports enthusiasts
   b) For able-bodied sports enthusiasts
   In each of the above categories there are products for outdoor adventure sports (e.g. trekking, para-gliding, bungee-jumping etc.), outdoor regular games (cricket, football, baseball etc.) indoor regular games (table tennis, squash etc.). There is a possibility of further class/object instantiation explosion with categories such as male & female sports enthusiasts and different equipment for them. *Objects need to be instantiated based on these categories*. Design & implement using Abstract Factory.
8. **Decorator(Behavioural – Structural according to GoF):** There is an existing interface method in the Decathlon POS software system called 'getCurrentStock' which is implemented by two concrete classes 'IndoorSports' and 'OutdoorSports', to get the number of stocks for the sports items belonging to these respective categories. On studying the Decathlon POS system, you as an analyst realize the need to get sports stock update of various items within:
   IndoorSports - 'GamesOnTable' (e.g. Table Tennis, Billiards, Snooker etc.)
   'BoardGames' (e.g. Carom, Chess etc.) 'CourtGames' (e.g. Basketball, Badminton, Kabaddi etc.)
   OutdoorSports – 'AdventureGames' (e.g. trekking, para-gliding, bungee-jumping etc.)
   'StadiumGames' (e.g. cricket, football, baseball etc.) 'Athletics' (e.g. different distances for running, high jump etc.)

   Use the Decorator pattern, decorating the 'getCurrentStock' method to design and implement this scenario.

9. **Template Method (Behavioural):** To keep up with the customer convenience of online ordering DecathlonChain of stores decides to have two modes of order-processing, namely 'online' and 'offline'. Both modes havethe same processing steps for order-processing, namely 'selectItem', 'doPayment' and 'doDelivery'. But, the waythese steps are done varies between the two modes.
   selectItem – online – gives tabular depiction of price comparison of the item chosen.Offline – allows trying out of the items in the store
   doPayment – online – net-banking payment; offline – pays through cash / swipe-carddoDelivery – online – needs to pay the charges for shipping & delivery address; offline – collect at the counter.

   Show how you as the analyst will use the Template Method pattern to design and implement this.

10. **Singleton (Creational):** A Browser's history has data of all the visited URLs across all tabs and windows of a browser. The history is saved such that the data persists even after closing the browser. How would you useSingleton Pattern to implement Browser History

such that on visiting a URL on any open tab of a browser theURL gets added to the existing history?

Course Outcomes

At the end of this course, the student will be able to:

CO#	Course Outcome Description
CO1	Design the given problem using standard UML (**PO 2, PO 5) (PSO 1, 2, 3)**
CO2	Identify maintenance issues in the existing solutions **(PO 2, PO 6) (PSO 1, 2, 3)**
CO3	Explain the use of design patterns **(PO 10) (PSO 1, 2, 3)**
CO4	Apply Design patterns to the problems**(PO 2, PO 4, PO 11) (PSO 1, 2, 3)**

PART-A

Case-study to understand the limitation of traditional Object Oriented Design and appreciate need for DesignPatterns. Use UML Notations to design.

You are a fresh analyst deputed to design the software for Decathlon Chain of Stores in Karnataka. You are informed about the Business Logic of Point of Sales criteria by Ms.Veronica Lodge, a dynamic business tycoon operating out of Decathlon Mumbai. She informs you that there are different types of Customers of Decathlon namely, Regular Customers, Senior Citizens and First Time Customers. Regular Customers are given a discount of 12%, Senior Citizens 10% and First Time Customers 15%. Apart from this, based on the sales-index of previous day, a Store-level discount is determined every day. This is dynamic. **E.g**.Rs.100 off for every purchase above Rs.2000. Using the Object Oriented Principles of Encapsulation, Abstraction, Inheritance, Composition and Aggregation that you have studied until this semester, give at least two ways to design this system.

PART-B

Common Case Study for Q#1 to Q#9
'Decathlon' is a Sports retail-store started in France. Today it spreads across 22 countries & has 900 outlets in these countries. It has a 'Point of Sale' software system called 'Decathlon POS', which uses various kinds of 3rd-party software sourced locally from the various countries they are established. You are a software consultant for Decathlon, in Bangalore, with a team of consultants reporting to you. When you analyze your answer for choosing a pattern, explain wherever applicable, keeping in mind the following four design principles:

- Separation of concerns
- Program to an interface, not a concrete implementation
- Prefer composition over inheritance
- Open-Close principle (Open for extension, Closed for modification)

1. **Adaptor (Structural):** To establish the 1st Decathlon store in Mauritius, you go along with Mr. Satya Nadella,an expert in finding 3rd-party partners. For e.g. a 3rd-party Tax-Calculator system to cater to the specifics of Salesand VAT (Value-added services Tax) tax calculations in different countries. He finds a 3rd-party Tax-Calculatorsystem called 'MauriTax' in Port Louis. The problem is, the APIs used by 'MauriTax' for tax-calculation is fixed& cannot be changed. *The 'MauriTax' APIs are incompatible with 'Decathlon POS'*.

   How will you use theAdaptor Pattern to design & implement?

   AdapterDemo.java

```
package tryAdapter;

public class AdapterDemo {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		CalcTax ct=new MauriTaxAdapter();
		//System.out.println(ct.taxAmount(1, 100));
		Item i1 = new Item("cycle",2,100,ct);
		i1.displayItem();

		i1.setTax(new GST());
		i1.displayItem();
	}

	}
```

CalcTax.java

```
	package tryAdapter;

public interface CalcTax {

	float taxAmount(int qty,float price);
}
```

GST.java

```
package tryAdapter;

public class GST implements CalcTax {

	@Override
	public float taxAmount(int qty, float price) {
		// TODO Auto-generated method stub
		return qty*price*0.18f;
	}

}
```

Item.java

```
package tryAdapter;

public class Item {
	String name;
	int qty;
	float price;
	CalcTax ct;
		public Item(String name,int qty,float price,CalcTax ct) {
		// TODO Auto-generated constructor stub
		this.name=name;
		this.price=price;
		this.qty=qty;
		this.ct=ct;
	}

	void setTax(CalcTax ct) {
		this.ct=ct;
	}

	void setQuantity(int qty) {
		this.qty=qty;
	}

	void displayItem() {
		System.out.println("\nName: "+name);
		System.out.println("Quantity: "+qty);
		System.out.println("Price: "+price);
		float tax=ct.taxAmount(qty, price);
		float billAmount=(qty*price)+tax;
		System.out.println("Tax Amount: "+tax);
		System.out.println("Bill Amount: "+billAmount);
	}
}
```

MauriTax.java

```
package tryAdapter;

public class MauriTax {

	float mauriTaxAmount(int qty,float price) {
		return qty*price*0.1f;
	}
}
```

MauriTaxAdapter.java

```
package tryAdapter;

public class MauriTaxAdapter implements CalcTax {

	MauriTax mt=new MauriTax();
```

```
    @Override
    public float taxAmount(int qty, float price) {
        // TODO Auto-generated method stub
        return mt.mauriTaxAmount(qty, price);
    }

}
```

2. **Strategy (Behavioural):** How will you use the Strategy Pattern to tackle the limitations of traditional ObjectOriented Design highlighted in PART A? *The design must handle varying price-schemes having different pricingalgorithms.* Design & implement.

Customer.java

```
package tryStrategy;

public abstract class Customer {

    String id,name,typeOfCust;
    Discount d;

    public Customer(String id,String name) {
        // TODO Auto-generated constructor stub
        this.id=id;
        this.name=name;
    }

    void printBill(float amt) {
        System.out.println("\nID: "+id);
        System.out.println("Name: "+name);
        System.out.println("Type of Customer: "+typeOfCust);
        System.out.println("Gross Amount: "+amt);
        System.out.println("Discount: "+d.calcDiscount(amt));
        System.out.println("Amount Payable: "+(amt-d.calcDiscount(amt)));

    }

}
```

Discount.java

```
package tryStrategy;

public interface Discount {
    float calcDiscount(float amount);

}
```

FTCCustomer.java

```
package tryStrategy;

public class FTCCustomer extends Customer {

	public FTCCustomer(String id, String name) {
		super(id, name);
		this.d=new FTCDiscount();
		typeOfCust="First Time Customer";
		// TODO Auto-generated constructor stub
	}

}
```

FTCDiscount.java

```
package tryStrategy;

public class FTCDiscount implements Discount {

	@Override
	public float calcDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.15f*amount;
	}

}
```

RCCustomer.java

```
package tryStrategy;

public class RCCustomer extends Customer {

	public RCCustomer(String id, String name) {
		super(id, name);
		d=new RCDiscount();
		typeOfCust="Regular Customer";

		// TODO Auto-generated constructor stub
	}

}
```

RCDiscount.java

```
package tryStrategy;

import java.nio.file.DirectoryStream;

public class RCDiscount implements Discount{

	@Override
	public float calcDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.12f*amount;
	}

}
```

SCCustomer.java

```
package tryStrategy;

public class SCCustomer extends Customer {

	public SCCustomer(String id, String name) {
		super(id, name);
		this.d=new SCDiscount();
		this.typeOfCust="Senior Customer";
		// TODO Auto-generated constructor stub
	}

}
```

SCDiscount.java

```
package tryStrategy;

public class SCDiscount implements Discount {

	@Override
	public float calcDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.1f*amount;
	}

}
```

StrategyDemo.java

```
package tryStrategy;

public class StrategyDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Customer c1=new RCCustomer("rc1", "modi");
		c1.printBill(100);

		c1=new SCCustomer("sc1", "trump");
		c1.printBill(100);

		c1=new FTCCustomer("ftc1", "raga");
		c1.printBill(100);

	}

}
```

3. **Factory Method (Creational):** The 'Decathlon POS' software system classifies its customers as senior-citizens(60 and above), First-Time customers, Regular Customers. There is a very high possibility that the CustomerType hierarchy will vary, depending upon the sales-pattern. **For e.g.** there could be the need to introduce newcategories based on the customer gender, different age groups for kids (0-5, 6-12), teenagers (13-19) and agegroups between 20 to 60(Twenties, 30s, 40s and 50s).

   You are advised by Mr.Sundar Pichai, the technical architect of your team, whom you trust, to use Factory MethodPattern in order to instantiate the above Customer Type hierarchy of concrete implementation of objects. Designand implement using this.

Customer.java

```
package tryFactoryMethod;

public interface Customer {

	void getType();
}
```

CustomerFactory.java

```
package tryFactoryMethod;

public class CustomerFactory {
	Customer getCustomer(String type) {
		if(type.equalsIgnoreCase("first time"))
			return new FTCustomer();
		else if(type.equalsIgnoreCase("regular"))
			return new RegCustomer();
		else if(type.equalsIgnoreCase("senior citizen"))
			return new SCCustomer();
		else
			return null;
	}
}
```

<u>FTCustomer.java</u>

```
package tryFactoryMethod;

public class CustomerFactory {

	Customer getCustomer(String type) {
		if(type.equalsIgnoreCase("first time"))
			return new FTCustomer();
		else if(type.equalsIgnoreCase("regular"))
			return new RegCustomer();
		else if(type.equalsIgnoreCase("senior citizen"))
			return new SCCustomer();
		else
			return null;
	}
}
```

<u>MainClass.java</u>

```
package tryFactoryMethod;

public class MainClass {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		CustomerFactory cf=new CustomerFactory();
		Customer c1=cf.getCustomer("senior citizen");
		Customer c2=cf.getCustomer("first time");
		Customer c3=cf.getCustomer("regular");

		c1.getType();
		c2.getType();
		c3.getType();
	}

}
```

RegCustomer.java

```
package tryFactoryMethod;

public class RegCustomer implements Customer {

	@Override
	public void getType() {
		// TODO Auto-generated method stub
		System.out.println("Regular Customer");
	}

}
```

SCCustomer.java

```
package tryFactoryMethod;

public class SCCustomer implements Customer {
	@Override
	public void getType() {
		// TODO Auto-generated method stub
		System.out.println("Senior Citizen Customer");
	}
}
```

4. **Bridge (Structural):** You get a call from Ms.Masaba Gupta of Bangalore Decathlon office that there is a policy decision made globally to introduce discount slabs for a whole month twice in a year. The discount month will be in January and July after reviewing the sales made from Feb to June (first five months) and Aug to December (last five months) respectively. It is decided to provide four slabs of discounts in 2017, namely, 30%, 25%, 20% and 15%, based on the sports item purchased. **For e.g.** all tennis rackets could have a 20% discount while cricket bats could only have a 15% discount. All exercise tread-mills could be given a 30% discount while boxing-gloves could have a 25% discount. Point to be noted here is that, the slabs of discount may not remain the same in 2018. It is likely to vary year after year. The 'Decathlon POS' software system classifies its customers as Senior-Citizens (60 and above), First-Time Customers, Regular Customers as of now. There is a very high possibility that the Customer Type hierarchy will vary, depending upon the sales-pattern. **For e.g.** there could be the need to introduce new categories based on the customer gender.

Use the Bridge Pattern to design & implement, *so that both the Customer Type hierarchy of classes as well as the Discount Percentage hierarchy of classes can both vary independently?* That is, they are not tied to each other.

BridgeDemo.java

```
package tryBridge;

public class BridgeDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Customer c1=new RCCustomer("Modi", 61, new Discount1());
		c1.showBill(100);

		c1.setDiscount(new Discount2());
		c1.showBill(100);

		c1.setDiscount(new Discount3());
		c1.showBill(100);

		c1.setDiscount(new Discount4());
		c1.showBill(100);

		Customer c2=new FTCCustomer("Raga", 6, new Discount1());
		c2.showBill(100);

		c2.setDiscount(new Discount2());
		c2.showBill(100);

		c2.setDiscount(new Discount3());
		c2.showBill(100);

		c2.setDiscount(new Discount4());
		c2.showBill(100);
		}
}
```

Customer.java

```
package tryBridge;

public abstract class Customer {
	String name;
	int age;
	Discount d;
	String typeOfCust;

	public Customer(String name,int age,Discount d) {
		// TODO Auto-generated constructor stub
		this.name=name;
		this.age=age;
		this.d=d;
	}

	void setDiscount(Discount d) {
		this.d=d;
	}

	void showBill(float amt) {
		System.out.println("\nName: "+name);
		System.out.println("Age: "+age);
		System.out.println("Type of Customer: "+typeOfCust);
		System.out.println("Gross Cost: "+amt);
		System.out.println("Discount: "+d.getDiscount(amt));
		System.out.println("Payable Amount: "+(amt-d.getDiscount(amt)));
	}

}
```

Discount.java

```
package tryBridge;

public interface Discount {
	float getDiscount(float amount);
}
```

Discount1.java

```
package tryBridge;

public class Discount1 implements Discount {

	@Override
	public float getDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.3f*amount;
	}
}
```

Discount2.java

```
package tryBridge;

public class Discount2 implements Discount {

	@Override
	public float getDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.25f*amount;
	}

}
```

Discount3.java

```
package tryBridge;

public class Discount3 implements Discount{

	@Override
	public float getDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.2f*amount;
	}


}
```

Discount4.java

```
package tryBridge;

public class Discount4 implements Discount {

	@Override
	public float getDiscount(float amount) {
		// TODO Auto-generated method stub
		return 0.15f*amount;
	}
}
```

FTCCustomer.java

```
package tryBridge;
```

```
public class FTCCustomer extends Customer {

	public FTCCustomer(String name, int age, Discount d) {
		super(name, age, d);
		typeOfCust="First Time Customer";
		// TODO Auto-generated constructor stub
	}

}
```

RCCustomer.java

```
package tryBridge;

public class RCCustomer extends Customer {

	public RCCustomer(String name, int age, Discount d) {
		super(name, age, d);
		typeOfCust="Regular Customer";
		// TODO Auto-generated constructor stub
	}

}
```

SSCustomer.java

```
package tryBridge;

public class SSCustomer extends Customer {

	public SSCustomer(String name, int age, Discount d) {
		super(name, age, d);
		typeOfCust="Senior Customer";
		// TODO Auto-generated constructor stub
	}

}
```

5. **Observer (Behavioural):** There will be different discounts being offered for the sports items in Decathlon Stores across the globe for different festivals being celebrated in the various countries these stores are established. Assume that the Decathlon Chain of Stores fixes a particular discount slab for its items for a festival of a country.

   Use the Observer Pattern to design and implement a system to notify the customers of the Decathlon stores of that country about the various festival / seasonal discount rates as and when they are announced.

Customer.java

```
package tryObserver;

public class Customer extends Observer {

	Subject store;
	float discount;
	String name;
	public Customer(Subject subject,String name) {
		// TODO Auto-generated constructor stub
		this.name=name;
		store=subject;
		store.register(this);


	}
	@Override
	void update(float discount) {
		// TODO Auto-generated method stub
		this.discount=discount;
		System.out.println(name+ ",you get a discount of "+discount+"%");
	}

	public String toString() {
		return name;
	}

}
```

MainClass.java

```
package tryObserver;

public class MainClass {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Store s1=new Store("Store1", 10);
		Customer c1=new Customer(s1, "Modi");
		Customer c2=new Customer(s1,"Trump");

		s1.setDiscount("Holi", 5);

		s1.unregister(c2);
		s1.setDiscount("Diwali", 20);
		Customer c3=new Customer(s1, "Raga");
		s1.setDiscount("Ugadi", 15);  }}
```

Observer.java

```
package tryObserver;

abstract public class Observer {
	abstract void update(float discount);
}
```

Store.java

```
package tryObserver;

import java.util.ArrayList;

public class Store extends Subject {

	float discount;
	String name;
	ArrayList<Observer> ol;
	public Store(String name,float discount) {
		// TODO Auto-generated constructor stub
		this.name=name;
		this.discount=discount;
		ol=new ArrayList<Observer>();
	}
	@Override
	void register(Observer o) {
		// TODO Auto-generated method stub
		ol.add(o);
		System.out.println("Added Customer "+o+" to Store "+name);
	}

	@Override
	void unregister(Observer o) {
		// TODO Auto-generated method stub
		try {
			ol.remove(ol.indexOf(o));
			System.out.println("Removed Customer "+o+" from store "+name);
		}
		catch (NullPointerException e) {
			// TODO: handle exception
			System.out.println("No such Customer called "+o+" in store "+name);
		}
	}

	@Override
	void notifyObservers() {
		// TODO Auto-generated method stub
		for(Observer o:ol)
			o.update(discount);

	}

	void setDiscount(String festival,float d) {
		discount=d;
		System.out.println("New Discount Offer on Account of "+festival);
		notifyObservers();
	}

}
```

Subject.java

```
package tryObserver;

public abstract class Subject {

	abstract void register(Observer o);
	abstract void unregister(Observer o);
	abstract void notifyObservers();
}
```

6. **Façade (Structural):** You get a call from Ms.Betty Cooper of Bangalore Decathlon office that there is a policy decision made globally to incorporate some new rules for 'Process Sale Use-Case'. **For e.g.** if payment is made via gift-certificate, the customer can buy only one item for the amount in the certificate. No other items can be bought with that gift-certificate. There must be no cash-back to the customer if the item costs less than amount specified in the gift-certificate. If the item costs more, the excess payment can be accepted via cash only & not credit / debit cards. When a new sale is created, these rules must become effective. You come to know from Mr. Satya Nadella, an expert in finding 3rd-party partners, that the Italian Competitor for Decathlon called 'Sport 2000' has a ready-made 'rule-engine' sub-system for this, whose specific implementation details is not known yet, as the business heads of Decathlon & Sport 2000 are chalking out the software purchase terms. This Sport 2000 rule-engine will be responsible for evaluating a set of rules against an operation & indicating if any of the rules invalidated the operation (e.g. 'makeNewSale' operation).

   How will you use the Façade pattern to provide a common unified interface to a dissimilar set of implementations, developed by a 3rd-party vendor, the implementation details are not known to you?

FacadeDemo.java

```
package tryFacade;

public class FacadeDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Sports2000Facade f=new Sports2000Facade(false, 100);
		f.displayItems();
		f.dispAmount();

	}

}
```

ItemPurchased.java

```
package tryFacade;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

public class ItemPurchased {
	HashMap<String, Integer> hm;
	int type;

	public ItemPurchased() {
		// TODO Auto-generated constructor stub
		hm=new HashMap<String, Integer>();
		storePurchase();
	}

	private void storePurchase() {
		// TODO Auto-generated method stub
		Scanner in=new Scanner(System.in);
		System.out.print("Enter number of types of Items: ");
		type=Integer.parseInt(in.nextLine());
		for(int i=1;i<=type;i++) {
			System.out.print("Enter name of Item "+i+": ");
			String name=in.nextLine();
			System.out.print("Enter Quantity of Item "+i+": ");
			int qty=Integer.parseInt(in.nextLine());
			hm.put(name, qty);
		}

	}

	int getTypeCount() {
		return type;
	}

	void showItems() {
		System.out.println("Items Purchased:-");
		Set<Map.Entry<String,Integer>> set=hm.entrySet();
		for(Map.Entry<String,Integer> i:set) {
			System.out.println(i.getKey()+":"+i.getValue());
		}
	}

}
```

Sports2000ProcessSales.java

```
package tryFacade;

import java.util.Scanner;

public class Sport2000ProcessSales {

	boolean giftCert;
	float amount;
	int qty;

	public Sport2000ProcessSales(boolean g,float a,int q) {
		// TODO Auto-generated constructor stub
		giftCert=g;
		amount=a;
		qty=q;
	}

	void processSales() {
		if(!giftCert) {
			System.out.println("You don't have a Gift Certificate!");
			System.out.println("Amount Payable: "+amount);
		}
		else {
			Scanner in=new Scanner(System.in);
			System.out.print("Enter Gift Certificate Value: ");
			float certValue=Float.parseFloat(in.nextLine());
```

```
				if(qty<=0) {
					System.out.println("No Items in Cart!");
				}
				else if(qty>1) {
					System.out.println("Only one Item can be purchased using Gift
Certificate");
				}
				else if(amount>certValue) {
					System.out.println("Please pay balance amount in Cash:
Rs."+(amount-certValue));
				}
				else if(amount<=certValue) {
					System.out.println("No cashback will be Refunded! Thankyou for
your Purchase!");

				}

			}

		}

}
```

Sports2000Facade.java

```
package tryFacade;

public class Sports2000Facade {
	ItemPurchased i;
	Sport2000ProcessSales s;

	public Sports2000Facade(boolean b,float a) {
		// TODO Auto-generated constructor stub
		i=new ItemPurchased();
		s=new Sport2000ProcessSales(b, a, i.getTypeCount());

	}

	void displayItems() {
		i.showItems();
	}

	void dispAmount() {
		s.processSales();
	}

}
```

7. **Abstract Factory (Creational):** As an analyst in charge of designing the Decathlon POS Software, you realize the need to streamline the creation of objects belonging to different products in the Decathlon store. There are two major categories of products:

a) For differently abled sports enthusiasts
b) For able-bodied sports enthusiasts

In each of the above categories there are products for outdoor adventure sports (e.g. trekking, para-gliding, bungee-jumping etc.), outdoor regular games (cricket, football, baseball etc.) indoor regular games (table tennis, squash etc.). There is a possibility of further class/object instantiation explosion with categories such as male & female sports enthusiasts and different equipment for them. *Objects need to be instantiated based on these categories*.

Design & implement using Abstract Factory.

BungeeJumpingDiffAbled.java

```
package tryAbstractFactory;

public class BungeeJumpingDiffAbled extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Differently Abled Bungee Jumping");
	}

}
```

BungeeJumpingRegular.java

```
package tryAbstractFactory;

public class BungeeJumpingRegular extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Regular Bungee-Jumping");
	}

}
```

CricketDiffAbled.java

```
package tryAbstractFactory;
```

```
public class CricketDiffAbled extends OutdoorRegularGames {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Differently Abled Cricket");
	}

}
```

CricketRegular.java

```
package tryAbstractFactory;

public class CricketRegular extends OutdoorRegularGames {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Regular Cricket");
	}

}
```

DiffAbledSportsFactory.java

```
package tryAbstractFactory;

public class DiffAbledSportsFactory implements SportsCategoryFactory {

	@Override
	public OutdoorAdventureSports getOutdoorAdventureSports(String name) {
		// TODO Auto-generated method stub
		if(name.equalsIgnoreCase("Bungee Jumping"))
			return new BungeeJumpingDiffAbled();
		else if(name.equalsIgnoreCase("Paragliding"))
			return new ParaglidingDiffAbled();
		else if(name.equalsIgnoreCase("Trekking"))
			return new TrekkingDiffAbled();
		else
			return null;
	}

	@Override
	public OutdoorRegularGames getOutdoorRegularGames() {
		// TODO Auto-generated method stub
		return new CricketDiffAbled();
	}

	@Override
	public IndoorRegularGames getIndoorRegularGames() {
		// TODO Auto-generated method stub
		return new TableTennisDiffAbled();
	}

}
```

IndoorRegularGames.java

```
package tryAbstractFactory;

public abstract class IndoorRegularGames {
	abstract void getSportName();

}
```

MainClass.java

```
package tryAbstractFactory;

public class MainClass {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		//Regular Sports
		SportsCategoryFactory reg=new RegularSportsFactory();
		OutdoorAdventureSports sp1=reg.getOutdoorAdventureSports("trekking");
		OutdoorRegularGames sp2=reg.getOutdoorRegularGames();
		IndoorRegularGames sp3=reg.getIndoorRegularGames();

		sp1.getSportName();
		sp2.getSportName();
		sp3.getSportName();

		//Differently Abled Sports
		SportsCategoryFactory diffabled=new DiffAbledSportsFactory();
		OutdoorAdventureSports
dsp1=diffabled.getOutdoorAdventureSports("Paragliding");
		OutdoorRegularGames dsp2=diffabled.getOutdoorRegularGames();
		IndoorRegularGames dsp3=diffabled.getIndoorRegularGames();

		dsp1.getSportName();
		dsp2.getSportName();
		dsp3.getSportName();


	}

}
```

<u>OutdoorAdventureSports.java</u>

```
package tryAbstractFactory;

public abstract class OutdoorAdventureSports {

	abstract void getSportName();
}
```

<u>OutdoorRegularGames.java</u>

```
package tryAbstractFactory;

public abstract class OutdoorRegularGames {

	abstract void getSportName();}
```

<u>ParaGlidingDiffAbled.java</u>

```
package tryAbstractFactory;

public class ParaglidingDiffAbled extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Differently Abled Paragliding");

	}

}
```

<u>ParaglidingRegular.java</u>

```
package tryAbstractFactory;

public class ParaglidingRegular extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
```

```
			System.out.println("Regular Paragliding");
		}

}
```

RegularSportsFactory.java

```
package tryAbstractFactory;

public class RegularSportsFactory implements SportsCategoryFactory {

		@Override
		public OutdoorAdventureSports getOutdoorAdventureSports(String name) {
			// TODO Auto-generated method stub
			if(name.equalsIgnoreCase("Bungee Jumping"))
				return new BungeeJumpingRegular();
			else if(name.equalsIgnoreCase("Paragliding"))
				return new ParaglidingRegular();
			else if(name.equalsIgnoreCase("Trekking"))
				return new TrekkingRegular();
			else
				return null;
		}

		@Override
		public OutdoorRegularGames getOutdoorRegularGames() {
			// TODO Auto-generated method stub
			return new CricketRegular();
		}

		@Override
		public IndoorRegularGames getIndoorRegularGames() {
			// TODO Auto-generated method stub
			return new TableTennisRegular();
		}

}
```

SportsCategoryFactory.java

```
package tryAbstractFactory;

public interface SportsCategoryFactory {
		OutdoorAdventureSports getOutdoorAdventureSports(String name);
		OutdoorRegularGames getOutdoorRegularGames();
		IndoorRegularGames getIndoorRegularGames();
}
```

<u>TableTennisDiffAbled.java</u>

```
package tryAbstractFactory;

public class TableTennisDiffAbled extends IndoorRegularGames {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Differently Abled Table Tennis");
	}

}
```

<u>TableTennisRegular.java</u>

```
package tryAbstractFactory;

public class TableTennisRegular extends IndoorRegularGames {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Regular Table Tennis");
	}

}
```

<u>TrekkingDiffAbled.java</u>

```
package tryAbstractFactory;

public class TrekkingDiffAbled extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Differently Abled Trekking");
	}

}
```

TrekkingRegular.java

```
package tryAbstractFactory;

public class TrekkingRegular extends OutdoorAdventureSports {

	@Override
	void getSportName() {
		// TODO Auto-generated method stub
		System.out.println("Regular Trekking");
	}

}
```

8. **Decorator(Behavioural – Structural according to GoF):** There is an existing interface method in the Decathlon POS software system called 'getCurrentStock' which is implemented by two concrete classes 'IndoorSports' and 'OutdoorSports', to get the number of stocks for the sports items belonging to these respective categories. On studying the Decathlon POS system, you as an analyst realize the need to get sports stock update of various items within:
   IndoorSports - 'GamesOnTable' (e.g. Table Tennis, Billiards, Snooker etc.)
   'BoardGames' (e.g. Carom, Chess etc.) 'CourtGames' (e.g. Basketball, Badminton, Kabaddi etc.)
   OutdoorSports – 'AdventureGames' (e.g. trekking, para-gliding, bungee-jumping etc.)
   'StadiumGames' (e.g. cricket, football, baseball etc.) 'Athletics' (e.g. different distances for running, high jump etc.)

   Use the Decorator pattern, decorating the 'getCurrentStock' method to design and implement this scenario.

AdventureGames.java

```
import java.util.*;

/**
 *
 */
public abstract class AdventureGames extends OutdoorSportsDecorator {

   /**
    * Default constructor
    */
   public AdventureGames() {
   }
}
```

Athletics.java

```
import java.util.*;

/**
 *
 */
public abstract class Athletics extends OutdoorSportsDecorator {

    /**
     * Default constructor
     */
    public Athletics() {
    }

}
```

Badminton.java

```
import java.util.*;

/**
 *
 */
public class Badminton extends CourtGames {

    /**
     * Default constructor
     */
            Sports sports;
    public Badminton(Sports sports) {
            this.sports=sports;
    }

            @Override
            public int getCurrentStock() {
                        // TODO Auto-generated method stub
                        return 2+sports.getCurrentStock();
            }
}
```

Baseball.java

```
import java.util.*;

/**
 *
 */
public class Baseball extends StadiumGames {

    /**
     * Default constructor
     */
        Sports sports;
    public Baseball(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

Basketball.java

```
import java.util.*;

/**
 *
 */
public class Basketball extends CourtGames {

    /**
     * Default constructor
     */
        Sports sports;
    public Basketball(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

Billiards.java

```
import java.util.*;

/**
 *
 */
public class Billiards extends GamesOnTable {

    /**
     * Default constructor
     */
        Sports sports;
    public Billiards(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

BoardGames.java

```
import java.util.*;

/**
 *
 */
public abstract class BoardGames extends IndoorSportsDecorator {

    /**
     * Default constructor
     */
    public BoardGames() {
    }

}
```

BungeeJumping.java

```
import java.util.*;

/**
 * 
 */
public class BungeeJumping extends AdventureGames {

    /**
     * Default constructor
     */
        Sports sports;
    public BungeeJumping(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
            return 2+sports.getCurrentStock();
            // TODO Auto-generated method stub

        }

}
```

Carrom.java

```
import java.util.*;

/**
 * 
 */
public class Carrom extends BoardGames {

    /**
     * Default constructor
     */
        Sports sports;
    public Carrom(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

Chess.java

```
import java.util.*;

/**
 *
 */
public class Chess extends BoardGames {

    /**
     * Default constructor
     */
        Sports sports;
    public Chess(Sports sports) {
        this.sports=sports;
    }

        @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

<u>CourtGames.java</u>

```
import java.util.*;

/**
 *
 */
public abstract class CourtGames extends IndoorSportsDecorator {

    /**
     * Default constructor
     */
    public CourtGames() {
    }

}
```

<u>Cricket.java</u>

```
import java.util.*;

/**
 *
 */
public class Cricket extends StadiumGames {

    /**
     * Default constructor
     */
            Sports sports;
    public Cricket(Sports sports) {
            this.sports=sports;
    }

    @Override
            public int getCurrentStock() {
                        return 2+sports.getCurrentStock();
                        // TODO Auto-generated method stub

            }
}
```

Football.java

```
import java.util.*;

/**
 *
 */
public class Football extends StadiumGames {

    /**
     * Default constructor
     */
            Sports sports;
    public Football(Sports sports) {
            this.sports=sports;
    }

    @Override
            public int getCurrentStock() {
                        return 2+sports.getCurrentStock();
                        // TODO Auto-generated method stub

            }


}
```

GamesOnTable.java

```
import java.util.*;

/**
 *
 */
public abstract class GamesOnTable extends IndoorSportsDecorator {

    /**
     * Default constructor
     */
    public GamesOnTable() {
    }
}
```

HighJump.java

```
import java.util.*;

/**
 *
 */
public class HighJump extends Athletics {

    /**
     * Default constructor
     */
            Sports sports;
    public HighJump(Sports sports) {
            this.sports=sports;
    }

    @Override
            public int getCurrentStock() {
                        return 2+sports.getCurrentStock();
                        // TODO Auto-generated method stub

            }

}
```

IndoorSports.java

```
import java.util.*;

/**
 *
 */
public class IndoorSports extends Sports {

    /**
     * Default constructor
     */
    public IndoorSports() {
    }
    /**
     * @return
     *
     */
    public int getCurrentStock() {
                    return 0;
            }

}
```

IndoorSportsDecorator.java

```
import java.util.*;

/**
 *
 */
public abstract class IndoorSportsDecorator extends Sports {

    /**
     * Default constructor
     */

    public IndoorSportsDecorator() {
    }


    /**
     * @return
     *
     */
    public abstract int getCurrentStock();

}
```

Kabbadi.java

```
import java.util.*;

/**
 *
 */
```

```
public class Kabaddi extends CourtGames {

    /**
     * Default constructor
     */
            Sports sports;
    public Kabaddi(Sports sports) {
            this.sports=sports;
    }

    @Override
            public int getCurrentStock() {
                        return 2+sports.getCurrentStock();
                        // TODO Auto-generated method stub

            }


}
```

LongJump.java

```
import java.util.*;

/**
 *
 */
public class LongJump extends Athletics {

    /**
     * Default constructor
     */
            Sports sports;
    public LongJump(Sports sports) {
            this.sports=sports;
    }

    @Override
            public int getCurrentStock() {
                        return 2+sports.getCurrentStock();
                        // TODO Auto-generated method stub

            }
}
```

MainClass.java

```
public class MainClass {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                //Assuming stock of each sport is 2

                Sports sp1=new IndoorSports();
                System.out.println("Total Indoor Sports Stock:"+sp1.getCurrentStock());
                sp1=new Billiards(sp1);
                System.out.println("Total Indoor Sports Stock:"+sp1.getCurrentStock());
                sp1=new Carrom(sp1);
                System.out.println("Total Indoor Sports Stock:"+sp1.getCurrentStock());
                sp1=new Badminton(sp1);
                System.out.println("Total Indoor Sports Stock:"+sp1.getCurrentStock());

                Sports sp2=new OutdoorSports();
                System.out.println("\nTotal Outdoor Sports Stock:"+sp2.getCurrentStock());
                sp2=new Trekking(sp2);
                System.out.println("Total Outdoor Sports Stock:"+sp2.getCurrentStock());
                sp2=new Cricket(sp2);
                System.out.println("Total Outdoor Sports Stock:"+sp2.getCurrentStock());
                sp2=new LongJump(sp2);
                System.out.println("Total Outdoor Sports Stock:"+sp2.getCurrentStock());


        }

}
```

OutdoorSports.java

```
import java.util.*;

/**
 *
 */
public class OutdoorSports extends Sports {

   /**
    * Default constructor
```

```
     */
    public OutdoorSports() {
    }

    /**
     * @return
     *
     */
    public int getCurrentStock() {
                    return 0;
            }

}
```

OutdoorSportsDecorator.java

```
import java.util.*;

/**
 *
 */
public abstract class OutdoorSportsDecorator extends Sports {

    /**
     * Default constructor
     */
    public OutdoorSportsDecorator() {
    }


    /**
     * @return
     *
     */
    public abstract int getCurrentStock();

}
```

Paragliding.java

```
import java.util.*;

/**
```

```
 *
 */
public class Paragliding extends AdventureGames {

   /**
    * Default constructor
    */
         Sports sports;
   public Paragliding(Sports sports) {
         this.sports=sports;
   }

   @Override
         public int getCurrentStock() {
                  return 2+sports.getCurrentStock();
                  // TODO Auto-generated method stub

         }


}
```

Snooker.java

```
import java.util.*;

/**
 *
 */
public class Snooker extends GamesOnTable {

   /**
    * Default constructor
    */
         Sports sports;
   public Snooker(Sports sports) {
         this.sports=sports;
   }

   @Override
         public int getCurrentStock() {
                  return 2+sports.getCurrentStock();
                  // TODO Auto-generated method stub

         }
}
```

Sports.java

```
import java.util.*;

/**
 *
 */
public abstract class Sports {

    /**
     * Default constructor
     */
    public Sports() {
    }

    /**
     *
     */
    public abstract int getCurrentStock();

}
```

StadiumGames.java

```
import java.util.*;

/**
 *
 */
public abstract class StadiumGames extends OutdoorSportsDecorator {

    /**
     * Default constructor
     */
    public StadiumGames() {
    }

}
```

TableTennis.java

```
import java.util.*;

/**
 *
 */
public class TableTennis extends GamesOnTable {

    /**
     * Default constructor
     */
        Sports sports;
    public TableTennis(Sports sports) {
        this.sports=sports;
    }

    @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }


}
```

Trekking.java

```
import java.util.*;

/**
 *
 */
public class Trekking extends AdventureGames {

    /**
     * Default constructor
     */
        Sports sports;
    public Trekking(Sports sports) {
        this.sports=sports;
    }

    @Override
        public int getCurrentStock() {
                return 2+sports.getCurrentStock();
                // TODO Auto-generated method stub

        }

}
```

9. **Template Method (Behavioural):** To keep up with the customer convenience of online ordering DecathlonChain of stores decides to have two modes of order-processing, namely ‘online’ and ‘offline’. Both modes havethe same processing steps for order-processing, namely ‘selectItem’, ‘doPayment’ and ‘doDelivery’. But, the waythese steps are done varies between the two modes.
   selectItem – online – gives tabular depiction of price comparison of the item chosen.Offline – allows trying out of the items in the store
   doPayment – online – net-banking payment; offline – pays through cash / swipe-carddoDelivery – online – needs to pay the charges for shipping & delivery address; offline – collect at the counter.

   Show how you as the analyst will use the Template Method pattern to design and implement this.

Cycle.java

```
package tryTemplate;

import java.util.ArrayList;

public class Cycle implements Menu {
        ArrayList<Item> al=new ArrayList<Item>();

        public Cycle() {
                // TODO Auto-generated constructor stub
                al.add(new Item(1, "cycle1", 1000));
                al.add(new Item(2, "cycle2", 2000));
                al.add(new Item(3, "cycle3", 3000));
                al.add(new Item(4, "cycle4", 4000));
                al.add(new Item(5, "cycle5", 5000));

        }

        @Override
        public void displayMenu() {
                // TODO Auto-generated method stub
                System.out.println("List of Items:-");
                for(Item i:al) {
                        System.out.println("\nID: "+i.id+"\nName: "+i.name+"\nPrice: "+i.price);
```

```
		}

	}

}
```

Item.java

```
package tryTemplate;

public class Item {
	String name;
	float price;
	int id;

	public Item(int i,String n,float p) {
		// TODO Auto-generated constructor stub
		name=n;
		price=p;
		id=i;
	}

	int getID() {
		return id;
	}


}
```

Menu.java

```
package tryTemplate;

public interface Menu {

	void displayMenu();
}
```

OfflineOrder.java

```
package tryTemplate;

import java.util.Scanner;
```

```
public class OfflineOrder extends OrderProcessing {

        Cycle menu;
        public OfflineOrder() {
                // TODO Auto-generated constructor stub
                menu=new Cycle();
        }

        @Override
        Item selectItem() {
                // TODO Auto-generated method stub
                //menu.displayMenu();
                for(Item i:menu.al) {
                        System.out.println("\nID: "+i.id+"\nName: "+i.name+"\nPrice: "+i.price);
                        System.out.println("Do you wish to select this product?(y/n): ");
                        Scanner in=new Scanner(System.in);
                        String c=in.nextLine();
                        if(c.equals("y")) {
                                return i;
                        }

                }

                                //Item i=null;
/*              for(Item l:menu.al) {
                        if(l.getID()==c) {
                                return l;
                        }
                        }
*/
                System.out.println("No More Items to show!");
                return null;
        }

        @Override
        void doPayment(Item i) {
                // TODO Auto-generated method stub
                System.out.println("\nSelected Item:-");
                System.out.println("ID: "+i.id);
                System.out.println("Name: "+i.name);
                System.out.println("Price: "+i.price);
                System.out.println("\nPayment Modes:-\n1.Cash\n2.Card");
                //System.out.print("Enter Your Choice: ");
                //Scanner in=new Scanner(System.in);
                //int c=Integer.parseInt(in.nextLine());
```

```
		int c;
		do {
			System.out.print("Enter Your Choice: ");
			Scanner in=new Scanner(System.in);
			c=Integer.parseInt(in.nextLine());

		switch(c) {
		case 1:
			cash();
			break;
		case 2:
			card();
			break;
		default:
			System.out.println("Invalid Payment Option!Try Again!");
		}
	}while(c!=1 && c!=2);



	}

	private void card() {
		// TODO Auto-generated method stub
		System.out.println("Thanks for the Card Payment!");

	}

	private void cash() {
		// TODO Auto-generated method stub
		System.out.println("Thanks for the Cash Payment!");
	}

	@Override
	void doDelivery() {
		// TODO Auto-generated method stub
		System.out.println("Your product will be delivered at your Address!");

	}

}
```

OnlineOrder.java

```
package tryTemplate;

import java.util.Scanner;
```

```
public class OnlineOrder extends OrderProcessing {

	Cycle menu;
	public OnlineOrder() {
		// TODO Auto-generated constructor stub
		menu=new Cycle();
	}

	@Override
	Item selectItem() {
		// TODO Auto-generated method stub
		menu.displayMenu();
		System.out.println("Enter ID of Product: ");
		Scanner in=new Scanner(System.in);
		int c=Integer.parseInt(in.nextLine());
		//Item i=null;
		for(Item l:menu.al) {
			if(l.getID()==c) {
				return l;
			}

		}
		System.out.println("Item not Found!");
		return null;
	}

	@Override
	void doPayment(Item i) {
		// TODO Auto-generated method stub
		System.out.println("\nSelected Item:-");
		System.out.println("ID: "+i.id);
		System.out.println("Name: "+i.name);
		System.out.println("Price: "+i.price);
		System.out.println("\nPayment Modes:-\n1.Paytm\n2.Card");
		//System.out.print("Enter Your Choice: ");
		//Scanner in=new Scanner(System.in);
		//int c=Integer.parseInt(in.nextLine());
		int c;
		do {
			System.out.print("Enter Your Choice: ");
			Scanner in=new Scanner(System.in);
			c=Integer.parseInt(in.nextLine());

		switch(c) {
		case 1:
				paytm();
				break;
			case 2:
				card();
				break;
			default:
				System.out.println("Invalid Payment Option!Try Again!");
			}
		}while(c!=1 && c!=2);



		}

		private void card() {
			// TODO Auto-generated method stub
			System.out.println("Thanks for the Card Payment!");

		}

		private void paytm() {
			// TODO Auto-generated method stub
			System.out.println("Thanks for the Paytm Payment!");
		}

		@Override
		void doDelivery() {
			// TODO Auto-generated method stub
			System.out.println("Your product will be delivered at your Address!");

		}

	}
```

__OrderProcessing.java__

```
package tryTemplate;

public abstract class OrderProcessing {

	abstract Item selectItem();
	abstract void doPayment(Item i);
	abstract void doDelivery();

	void purchaseItem() {
		Item i=selectItem();
		if(i!=null) {
```

```
                doPayment(i);
                doDelivery();
            }
        }


}
```

<u>TemplateDemo.java</u>

```
package tryTemplate;

public class TemplateDemo {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                OnlineOrder ol=new OnlineOrder();
                //ol.purchaseItem();

                OfflineOrder off=new OfflineOrder();
                off.purchaseItem();

        }

}
```

10. **Singleton (Creational):** A Browser's history has data of all the visited URLs across all tabs and windows of a browser. The history is saved such that the data persists even after closing the browser. How would you useSingleton Pattern to implement Browser History such that on visiting a URL on any open tab of a browser theURL gets added to the existing history?

<u>SingletonDemo.java</u>

```
package singleton;

public class SingletonDemo {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

                TabWindow s1=TabWindow.getInstance();
                //System.out.println(s1);
                TabWindow s2=TabWindow.getInstance();
                //System.out.println(s2);
                s1.addUrl("www.google.com");
```

```
		s2.addUrl("www.facebook.com");
		TabWindow s3=TabWindow.getInstance();
		s3.showUrls();

	}
}
```

<u>TabWindow.java</u>

```
package singleton;

import java.util.ArrayList;

public class TabWindow {

	public static TabWindow sc;
	ArrayList<String> urls;
	private TabWindow() {
		// TODO Auto-generated constructor stub
		urls=new ArrayList<String>();

	}

	public static TabWindow getInstance() {
		if(sc==null)
			sc=new TabWindow();
		return sc;
	}

	public void addUrl(String url) {
		urls.add(url);
	}

	public void showUrls() {
		for(String u:urls) {
			System.out.println(u);
		}
	}



}
```