4. Behavioural Design Patterns II

Aim/Objective: To analyse the implementation of Observer Design Pattern & Template Design Pattern and Dependency Injection Design pattern for the real-time scenario.

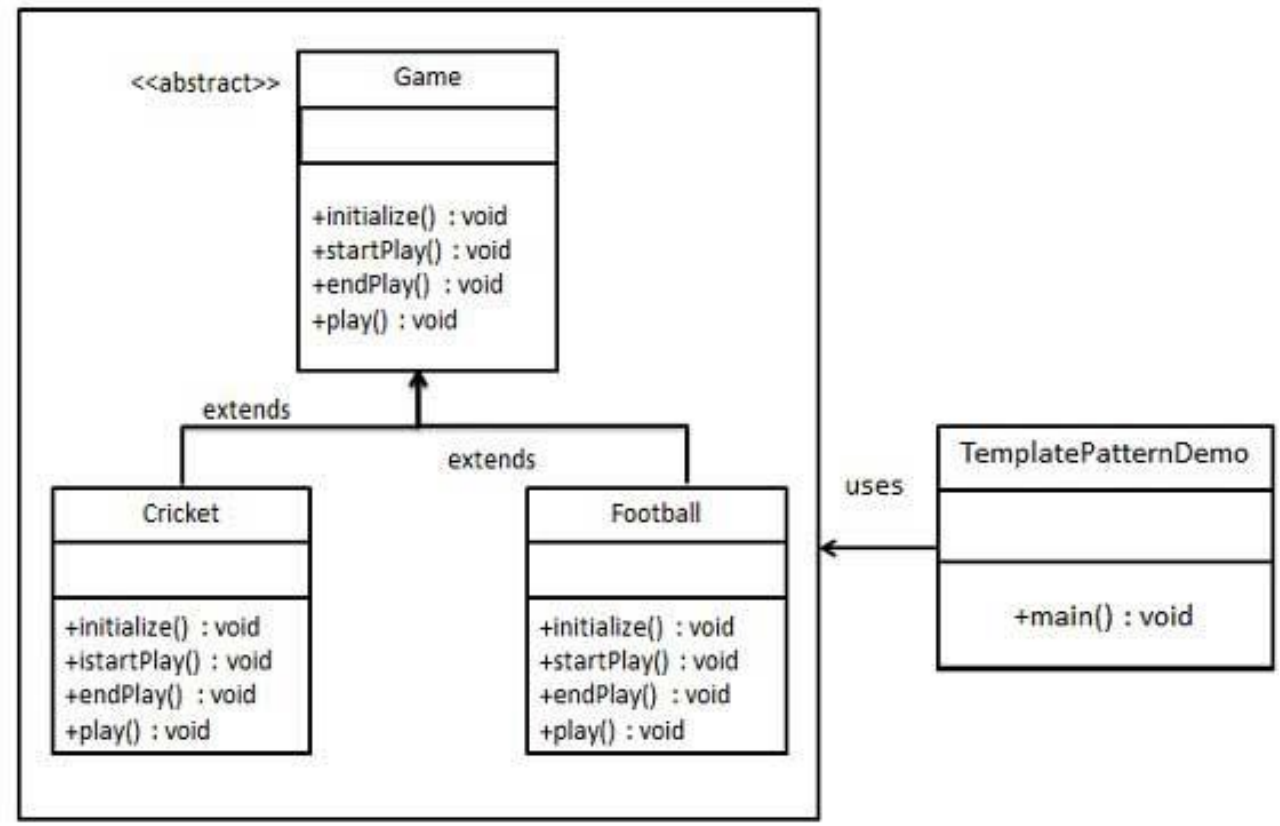
Description: To make student understand the application of behavioural design pattern in software applications.

Pre-Requisites: Classes and Objects in JAVA

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) Draw the UML Relationship Diagram for Template Design Pattern for customized Scenarios.



In-Lab:

1) Design an online auction system that utilizes the Observer and Template design patterns to manage auction events and bidding processes efficiently.

Requirements:

A. Observer Design Pattern:

- Use the Observer pattern to notify bidders about auction events such as item availability, bidding start, and bidding end.
- Bidders should be able to subscribe and unsubscribe to receive notifications.

B. **Template Design Pattern**:

- Implement the Template pattern to define the structure and steps of the auction process.
- Customize specific steps for different types of auctions (e.g., standard auction, reserve auction).

Procedure/Program:

```
import java.util.*;

public class AuctionSystem {

  interface Observer {
    void update(String event);
  }

  static class Bidder implements Observer {
    private String name;

    public Bidder(String name) {
      this.name = name;
    }

    @Override
    public void update(String event) {
      System.out.println(name + " received: " + event);
    }
```

```
}

  static class Auction {
    private List<Observer> bidders = new ArrayList<>();
    private String item;

    public void subscribe(Observer bidder) {
      bidders.add(bidder);
    }

    public void unsubscribe(Observer bidder) {
      bidders.remove(bidder);
    }

    private void notifyBidders(String event) {
      bidders.forEach(b -> b.update(event));
    }

    public void startAuction(String item) {
      this.item = item;
      notifyBidders("Auction started for: " + item);
    }

    public void endAuction() {
      notifyBidders("Auction ended for: " + item);
    }
  }

  static abstract class AuctionTemplate {
    public final void conductAuction() {
      System.out.println("Starting " + getClass().getSimpleName());
      startBidding();
      endBidding();
    }

    protected abstract void startBidding();
    protected abstract void endBidding();
  }
```

```
static class StandardAuction extends AuctionTemplate {
    @Override
    protected void startBidding() {
      System.out.println("Standard auction bidding started.");
    }

    @Override
    protected void endBidding() {
      System.out.println("Standard auction bidding ended.");
    }
  }

  static class ReserveAuction extends AuctionTemplate {
    @Override
    protected void startBidding() {
      System.out.println("Reserve auction bidding started.");
    }

    @Override
    protected void endBidding() {
      System.out.println("Reserve auction bidding ended.");
    }
  }

  public static void main(String[] args) {
    Auction auction = new Auction();
    Bidder alice = new Bidder("Alice"), bob = new Bidder("Bob");

    auction.subscribe(alice);
    auction.subscribe(bob);
    auction.startAuction("Antique Vase");
    auction.endAuction();

    new StandardAuction().conductAuction();
    new ReserveAuction().conductAuction();
  }
}
```

OUTPUT

```
Alice received: Auction started for: Antique Vase
Bob received: Auction started for: Antique Vase
Alice received: Auction ended for: Antique Vase
Bob received: Auction ended for: Antique Vase
Starting StandardAuction
Standard auction bidding started.
Standard auction bidding ended.
Starting ReserveAuction
Reserve auction bidding started.
Reserve auction bidding ended.
```

2) In a software development project, you're tasked with implementing a data import module that processes different types of data files (CSV, XML, JSON) and performs specific operations based on the file type. The Template Design Pattern is suitable for this scenario to provide a structured way to define the steps of data processing while allowing customization for each file type.

Requirements

A. **Define an Abstract Data Importer (Template)**:
   - Create an abstract class DataImporter that defines the template method importData() and other common methods shared by all data importers.
   - The importData() method should outline the sequence of steps required for data import, such as reading data, parsing it, and saving it.

B. **Implement Concrete Importers for Each File Type**:
   - Implement concrete subclasses (CSVImporter, XMLImporter, JSONImporter) that extend DataImporter.
   - Each subclass should override specific methods as needed to handle file-specific operations like parsing and validation.

C. **Client Code to Use the Template**:
   - Develop client code (e.g., a main method or another service) that uses the template method pattern to invoke data import operations.
   - Demonstrate how different file types are imported using their respective concrete importers.

Procedure/Program:

```
abstract class DataImporter {
  public final void importData(String filePath) {
    System.out.println("Reading data from " + filePath);
    parseData();
    System.out.println("Saving data from " + filePath);
  }

  protected abstract void parseData();
}

class CSVImporter extends DataImporter {
  protected void parseData() {
    System.out.println("Starting to parse CSV data.");
    System.out.println("Parsing CSV data completed.");
  }
}
```

```
class XMLImporter extends DataImporter {
   protected void parseData() {
      System.out.println("Starting to parse XML data.");
      System.out.println("Parsing XML data completed.");
   }
}

class JSONImporter extends DataImporter {
   protected void parseData() {
      System.out.println("Starting to parse JSON data.");
      System.out.println("Parsing JSON data completed.");
   }
}

public class DataImportClient {
   public static void main(String[] args) {
      new CSVImporter().importData("data.csv");
      new XMLImporter().importData("data.xml");
      new JSONImporter().importData("data.json");
   }
}
```

OUTPUT

```
Reading data from data.csv
Starting to parse CSV data.
Parsing CSV data completed.
Saving data from data.csv
Reading data from data.xml
Starting to parse XML data.
Parsing XML data completed.
Saving data from data.xml
Reading data from data.json
Starting to parse JSON data.
Parsing JSON data completed.
Saving data from data.json
```

VIVA-VOCE Questions (In-Lab):

1) State at which situation that we need Observer Design Pattern.

- A **one-to-many** dependency exists between objects.
- Changes in one object should notify multiple dependents.
- Implementing **event-driven systems** (GUIs, notifications, stock updates).

2) Discuss the Pros and Cons of Template Design Pattern.

✅ **Pros:**

- Code reuse (common logic in a base class).
- Enforces a standard structure.
- Reduces redundancy.

❌ **Cons:**

- Less flexibility for subclasses.
- Changes in base class affect all subclasses.
- Can lead to complex hierarchies.

3) Discuss the Pros and Cons of Observer Design Pattern.

✅ **Pros:**

- **Loose coupling** (independent components).
- Supports **real-time updates**.
- **Scalable** (multiple observers).

❌ **Cons:**

- **Performance overhead** with too many observers.
- Hard to debug.
- Risk of **memory leaks** if not managed properly.

Post-Lab:

1) Imagine you are developing a weather monitoring system that notifies various displays when the weather conditions change. The system involves multiple displays that need to stay updated with the latest weather data in real-time. Implement the Observer design pattern to achieve this functionality.

Requirements:

A. Subject Interface: WeatherStationSubject
   - **a.** Define an interface WeatherStationSubject that declares methods to register, remove, and notify observers.

B. Observer Interface: WeatherObserver
   - **a.** Define an interface WeatherObserver that declares a method for updating when notified by the subject.

C. Concrete Subject: WeatherStation
   - **a.** Implement the WeatherStationSubject interface in a class called WeatherStation. This class will maintain a list of observers and notify them when weather data changes.

D. Concrete Observers: Display Devices
   - **a.** Implement WeatherObserver interface in various display devices such as CurrentConditionsDisplay, StatisticsDisplay, and ForecastDisplay. These displays will update their information whenever the weather data changes.

Procedure/Program:

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

public class WeatherApp {

  public interface Subject {
    void addObserver(Observer obs);
    void removeObserver(Observer obs);
    void notifyObservers();
  }
```

```
public interface Observer {
    void update(float temp, float hum, float press);
  }

  public static class Station implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private float temp, hum, press;

    @Override
    public void addObserver(Observer obs) {
      observers.add(obs);
    }

    @Override
    public void removeObserver(Observer obs) {
      observers.remove(obs);
    }

    @Override
    public void notifyObservers() {
      for (Observer obs : observers) {
        obs.update(temp, hum, press);
      }
    }

    public void setData(float temp, float hum, float press) {
      this.temp = temp;
      this.hum = hum;
      this.press = press;
      notifyObservers();
    }
  }

  public static class CurrentDisplay implements Observer {
    @Override
    public void update(float temp, float hum, float press) {
      System.out.println("Current: " + temp + "°C, " + hum + "% humidity");
    }
  }
```

```
public static class StatsDisplay implements Observer {
    private float maxTemp = Float.MIN_VALUE, minTemp = Float.MAX_VALUE,
sumTemp = 0;
    private int count = 0;

    @Override
    public void update(float temp, float hum, float press) {
      sumTemp += temp;
      count++;
      if (temp > maxTemp) maxTemp = temp;
      if (temp < minTemp) minTemp = temp;
      System.out.println("Avg/Max/Min temp = " + (sumTemp / count) + "/" +
maxTemp + "/" + minTemp);
    }
  }

  public static class ForecastDisplay implements Observer {
    private float curPress = 29.92f, lastPress;

    @Override
    public void update(float temp, float hum, float press) {
      lastPress = curPress;
      curPress = press;
      if (curPress > lastPress) {
        System.out.println("Forecast: Improving weather!");
      } else if (curPress == lastPress) {
        System.out.println("Forecast: Stable weather");
      } else {
        System.out.println("Forecast: Cooler, rainy weather");
      }
    }
  }
```

```
public static void main(String[] args) {
        Station station = new Station();
        station.addObserver(new CurrentDisplay());
        station.addObserver(new StatsDisplay());
        station.addObserver(new ForecastDisplay());

        station.setData(25.0f, 65.0f, 1013.0f);
        station.setData(27.0f, 70.0f, 1011.0f);
        station.setData(22.0f, 60.0f, 1010.0f);
    }
}
```

OUTPUT

```
Current: 25.0°C, 65.0% humidity
Avg/Max/Min temp = 25.0/25.0/25.0
Forecast: Improving weather!
Current: 27.0°C, 70.0% humidity
Avg/Max/Min temp = 26.0/27.0/25.0
Forecast: Cooler, rainy weather
Current: 22.0°C, 60.0% humidity
Avg/Max/Min temp = 24.666666/27.0/22.0
Forecast: Cooler, rainy weather
```

2) Design a scenario that incorporates the Template Method, Dependency Injection, and Observer patterns in Java. The scenario will be a notification system where different types of notifications (Email, SMS, and Push) are sent based on user events (e.g., user registration, password reset).

A. **Template Method Pattern**: Use this pattern to define the steps of sending a notification, allowing subclasses to implement specific steps for different notification types.
B. **Dependency Injection**: Use this pattern to inject the specific notification service into a notifier class.
C. **Observer Pattern**: Use this pattern to observe user events and trigger the appropriate notifications.

Procedure/Program:

```
import java.util.*;

class User {
   String name, email;
   public User(String name, String email) { this.name = name; this.email = email;
}
}

abstract class Notification {
   public final void send(User user, String event) {
      System.out.println("Choosing recipient: " + user.email);
      System.out.println(getType() + " Message: " + event + " for " + user.name);
      System.out.println("Notification logged successfully.");
   }
   protected abstract String getType();
}

class EmailNotification extends Notification { protected String getType() {
return "Email"; } }
class SMSNotification extends Notification { protected String getType() { return
"SMS"; } }
class PushNotification extends Notification { protected String getType() { return
"Push"; } }

interface Observer { void update(User user, String event); }
```

```
class EventPublisher {
   private List<Observer> observers = new ArrayList<>();
   public void addObserver(Observer observer) { observers.add(observer); }
   public void notifyObservers(User user, String event) { observers.forEach(o ->
o.update(user, event)); }
}

public class NotificationSystem {
   public static void main(String[] args) {
      User user1 = new User("John Doe", "john@example.com");
      User user2 = new User("Jane Doe", "jane@example.com");

      EventPublisher publisher = new EventPublisher();
      publisher.addObserver((user, event) -> new EmailNotification().send(user,
event));
      publisher.addObserver((user, event) -> new SMSNotification().send(user,
event));
      publisher.addObserver((user, event) -> new PushNotification().send(user,
event));

      publisher.notifyObservers(user1, "User Registered");
      publisher.notifyObservers(user2, "Password Reset");
   }
}
```

OUTPUT

```
Choosing recipient: john@example.com
Email Message: User Registered for John Doe
Notification logged successfully.
Choosing recipient: john@example.com
SMS Message: User Registered for John Doe
Notification logged successfully.
Choosing recipient: john@example.com
Push Message: User Registered for John Doe
Notification logged successfully.
Choosing recipient: jane@example.com
Email Message: Password Reset for Jane Doe
Notification logged successfully.
Choosing recipient: jane@example.com
SMS Message: Password Reset for Jane Doe
Notification logged successfully.
Choosing recipient: jane@example.com
Push Message: Password Reset for Jane Doe
Notification logged successfully.
```

Experiment#		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

- ✓ **Data and Results:**

DATA:

User notifications are sent using email, SMS, and push notifications.

RESULT:

Notifications are successfully delivered based on user-triggered events dynamically.

- ✓ **Analysis and Inferences:**

ANALYSIS:

Observer pattern ensures event-driven updates, while DI improves flexibility.

INFERENCES:

This approach enhances modularity, maintainability, and scalability of notifications.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.