Date	

7. Behavioural Design Patterns II

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

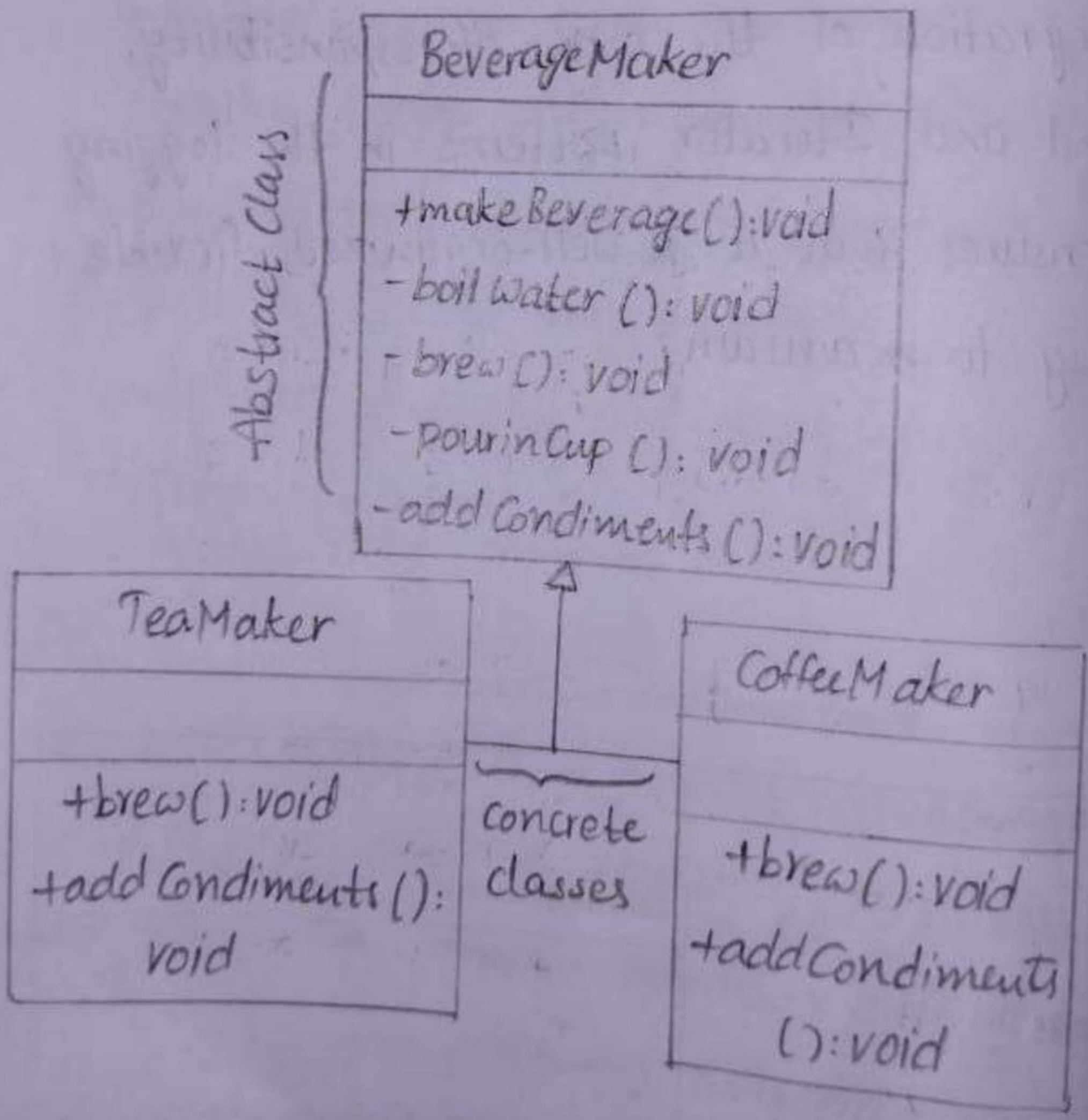
Description: To make student understand the application of behavioural desig software applications.

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 Scenarios.



In-Lab:

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

Requirements:

A. Observer Design Pattern:

- Use the Observer pattern to notify bidders about auction events, 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., Dutch auction, reserve auction).

Procedure/Program:

(A.)

```
public interface Observer {
    void update(String message);
} public class Bidder implements Observer{
    private String name;
    public Bidder(String name) {
        this.name = name;
    } public void update(String message) {
        System.out.println(name + "received notification: "
                + message);
    }
}
```

```
public interface Subject {
 void registerObserver (observer o);
 void removeObserver (observer o);
 void notifyObservers ();
} import java.util.ArrayList;(
   import java.util.List;
public class Auction implements Subject {
 private List<Observer> observers = new
                                ArrayList<> ();
 private String eventMessage;
 public void registerObserver (observer o) {
    observers.add (o);
 } public void removeObserver (observer o) {
    observers.remove (o);
 } public void notifyObservers () {
   for (Observer observer: observers) {
   observer.update (eventMessage);
    }
 } Public void setEvent Message (String message) {
 this.eventMessage = message;
    } }
```

B. public abstract class AuctionTemplate {

public final void conductAuction() {

setupAuction();

startBidding();

endBidding();

concludeAuction();

} protected abstract void setupAuction();

protected abstract void startBidding();

protected abstract void endBidding();

private void concludeAuction() {

System.out.println("Auction Concluded");

}

}

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:

```
public abstract class DataImporter {
public final void importData() {
    readData();
    parseData();
    validateData();
    saveData();
} protected abstract void readData();
  protected abstract void parseData();
  protected abstract void validateData();
```

```
protected abstract void saveData();

}
```

CSV Importer:

```
Public class CSVImporter extends DataImporter{
protected void readData() {
System.out.println ("Reading data from CSV file.");
} protected void parseData (){
System.out.println ("Parsing CSV data");
} protected void validateData (){
System.out.println ("Validating CSV data");
} protected void saveData (){
System.out.println ("Saving CSV data to database.");
}}
```

XML Importer:

```
public class XMLImporter extends DataImporter{
protected void readData (){
System.out.println ("Reading data from XML file");
} protected void parseData(){
```

```
System.out.println("Parsing XML data");
} protected void validateData() {
System.out.println("Validating XML data");
} protected void saveData() {
System.out.println("Saving XML data to database.");
}}
```

JSONImporter:

```
public class JSONImporter extends DataImporter {
  protected void readData() {
  System.out.println("Reading data from JSON file");
} protected void parseData() {
System.out.println("Parsing JSON data");
} protected void validateData(){
System.out.println("Validating JSON data");
} protected void saveData() {
 System.out.println("saving JSON data to database");
}}
```

Experiment#		Student ID	
Date		Student Name	

```
public class DataImportClient {
public static void main (String[] args) {
DataImporter csvImporter=new CSVImporter();
DataImporter xmlImporter= new XMLImporter();
DataImporter jsonImporter= new
                              JSONImporter();
System.out.println ("Importing CSV Data:");
csvImporter.importData();
System.out.println ("Importing XML Data:");
xmlImporter.importData();
System.out.println ("Importing JSON Data:");
jsonImporter.importData();
  }
}
```

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

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

Observer Design pattern is ideal for scenarios where objects need to remain synchronized with each other in a loosely coupled manner, especially when one object's state changes need to be communicated to multiple other objects.

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

Pros : Code Reusability, consistent structure, encapsulation, maintainability, extension

Cons : Class Proliferation, limited flexibility, complex inheritance, subclass overhead, Tight coupling

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

Pros : Loose coupling, dynamic subscriptions, automatic updates, scalability, consistency

Cons : Potential performance issues, Complexity in debugging, unpredictable order of updates

```
void update (float temperature, float humidity,
             float pressure); }

import java.util.ArrayList;
import java.util.List;
public class WeatherStation implements
                    WeatherStationSubject {
private List<WeatherObserver> observers;
private float temperature;
private float humidity;
 private float pressure;
 Public WeatherStation (){
observers=new ArrayList<>();
} public void registerObserver (WeatherObserver
                                observer) {
 observers.add (observer);
} public void removeObserver (WeatherObserver
                              observer) {
 observers.remove (observers);
} public void notifyObservers (WeatherObserver observer){
 observer.update (temperature, humidity, pressure);
```

Experiment#		Student ID	
Date		Student Name	

```
}}
public void setMeasurements(float temperature,
                            float humidity, float pressure){
    this.temperature = temperature;
    this.humidity = humidity;
    this.pressure = pressure;
     notifyObservers();
    }}
Public class CurrentConditions implements
                                WeatherObserver{
    privati float temperature;
    privati float humidity;
  Public void updati(float temperature, float
                     humidity, float pressure) {
  this.temperature = temperature;
  this.humidity = humidity;
     display();
  } public void display(){
  System.out.println("Current Conditions:" +
  temperature + "F degrees and" + humidity + "%.
```

Post-Lab:

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

Requirements:

A. Subject Interface: WeatherStationSubject
   a. Define an interface WeatherStationSubject that methods to register, remove, and notify observers.
B. Observer Interface: WeatherObserver
   a. Define an interface WeatherObserver that declares for updating when notified by the subject.
C. Concrete Subject: WeatherStation
   a. Implement the WeatherStationSubject interface called WeatherStation. This class will maintain observers and notify them when weather data cha
D. Concrete Observers: Display Devices
   a. Implement WeatherObserver interface in variou devices such as CurrentConditionsDisplay, Statisti and ForecastDisplay. These displays will upd information whenever the weather data changes.

Procedure/Program:

```
public interface WeatherStationSubject {
void registerObserver (WeatherObserver observer);
void removeObserver (WeatherObserver observer);
void notifyObservers();
} public interface WeatherObserver {
```

```
humidity");
}}
public class StatisticsDisplay implements WeatherObserver
{ private float temperatureSum = 0.0f;
  private float humiditySum = 0.0f;
  private int numReadings = 0;
public void update (float temperature, float
                    humidity, float pressure) {
temperatureSum += temperature;
 humiditySum += humidity;
 numReadings++;
  display();
} public void display (){
 System.out.println ("Average temperature: " +
(temperatureSum / numReadings) + "F");
 System.out.println ("Average humidity: " +
(humiditySum / numReadings) + "%");
  }
}
```

Experiment#		Student ID	
Date		Student Name	

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

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

Procedure/Program:

```
public abstract class NotificationTemplate {
public final void sendNotification(String
                    recipient, String message){
    prepareMessage(message);
    send(recipient);
    logDelivery();
}protected abstract void prepareMessage
                    (String message);
protected abstract void send(String recipient);
protected void logDelivery() {
System.out.println("Notification sent successfully.");
  }
}
```

```
public class EmailNotification extends NotificationTemplate
{
protected void prepareMessage (String message) {
System.out.println ("Preparing email message:" + message);
} protected void send (String recipient) {
System.out.println ("sending email to:" + recipient);
} }
public class SMSNotification extends Notification
                                        Template {
protected void prepareMessage (String message) {
System.out.println ("Preparing SMS message:" +
                                        message);
} protected void send (String recipient) {
System.out.println ("sending SMS to:" + recipient);
} }
public class PushNotification extends Notification
                                        Template {
protected void prepareMessage (String message) {
System.out.println ("Preparing push notification:"
+message);
```

```
} protected void send (string recipient) {
System.out.println ("Sending push notification to:
" + recipient);
} }
Public class Notifier {
private NotificationTemplate notificationService;
public Notifier (NotificationTemplate notification
                                    Service) {
this.notificationService = notificationservice;
} public void notifyUser (String recipient, String
                                    message) {
notificationService.sendNotification (recipient,
                                    message);
}
}
public interface UserEventObserver {
void update (String erentType, String reciptent,
                                    String message);
}
public class UserRegistrationObserver implements
                                    UserEventObserver {
```

```
private Notifier notifier;
public UserRegistrationObserver (Notifier notifier) {
    this.notifier = notifier;
} public void update (String eventType, String
                      recipient, String message) {
if ("UserRegistration".equals (eventType)) {
notifier.notifyUser (recipient, "welcome!" + message);
    }
  }
}
public class PasswordResetObserver implements
                                UserEventObserver {
  private Notifier notifier;
  public PasswordResetObserver (Notifier notifier) {
      this.notifier = notifier;
  } public void update (String eventType, String recipient,
                                  String massage) {
  if ("PasswordReset".equals (eventType)) {
  notifier.notifyUser (recipient, "Password reset
  request: " + message);
  }
```

```
    }
}
import java.util.ArrayList;
 import java.util.List;
Public class UserEventPublisher {
private List<UserEventObserver> observers=new
                                ArrayList<>();
Public void registerObserver(UserEventObserver
                                observer){
    observers.add(observer);
} public void removeObserver(UserEventObserver
                                observer) {
    observers.add remove(observer);
} public void notifyObservers(String eventType,
                    String recipient, String message){
    for (UserEventObserver observer : observers) {
observer.update(eventType, recipient, message);
    }
}
}
public class NotificationClient {
```

```
public static void main (String[] args) [
    NotificationTemplate emailService = new EmailNotification();
    NotificationTemplate smsService=new SmsNotification();
    NotificationTemplate pushService = new PushNotification();
    Notifier emailNotifier = new Notifier (emailService);
    Notifier smsNotifier = new Notifier (smsService);
    Notifier pushNotifier = new Notifier (pushService);
    UserRegistration Observer registrationObserver=new
                                UserRegistrationObserver (emailNotifier);
    ~~UserRegistration~~ PasswordResetObserver resetObserver=new
                                PasswordResetObserver (smsNotifier);
    UserEventPublisher eventPublisher=new UserEventPublisher();
    eventPublisher. registerObserver (registrationObserver);
    eventPublisher. registerObserver (resetObserver);
    eventPublisher. notifyObservers ("UserRegistration",
    "user@example.com", "Thankyou for registering!");
    eventPublisher. notifyObservers ("PasswordReset",
    "user@example.com", "Click here to reset your password.");
        }
    }
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Preparing email message: Welcome! Thank you for registering!
Sending email to: user@example.com
Notification sent successfully.

✓ **Analysis and Inferences:**

Each notification service ('EmailNotification', 'SMSNotification', 'PushNotification') prepares the message and sends it and then logs the successful delivery.

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

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