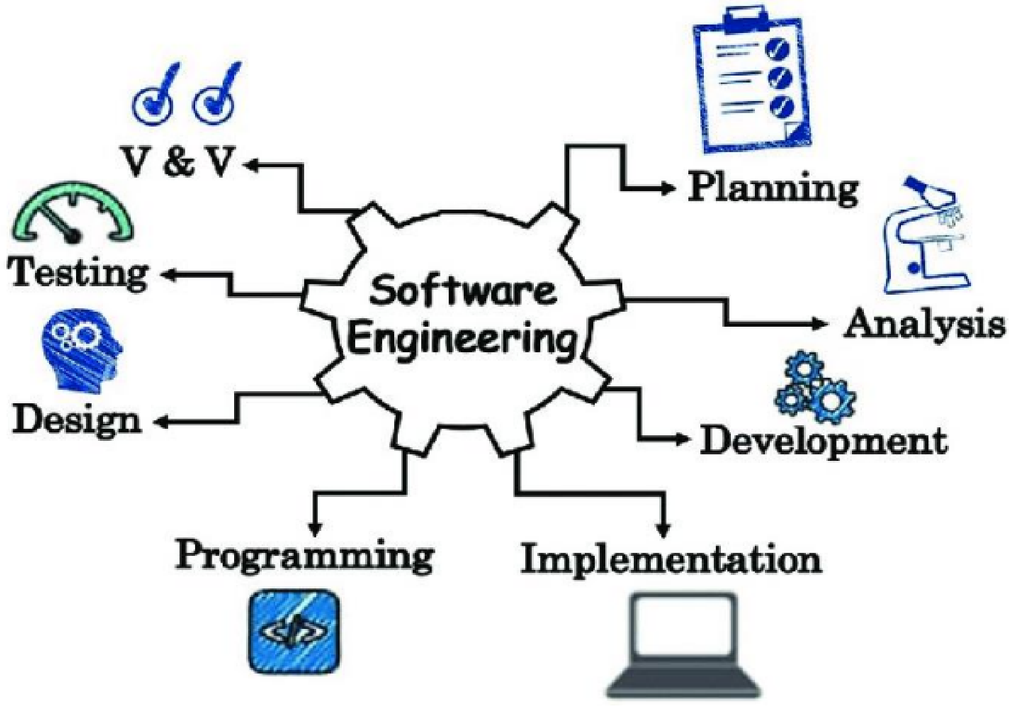

Software Engineering
V & V
Testing
Design
Planning
Analysis
Development
Programming
Implementation

Final Assessment Introduction to Software Engineering

Name: Sajid Ali (21193046)

Introduction

Our task entails developing educational software, with a focus on developing a programme that aids in understanding weather patterns and the effects of weather changes in various regions based on country and month, as well as computing the average temperature of a city. The purpose of our weather programme is to provide people with a full introduction to this topic.
To do this, we developed a flexible piece of software that can be accessed via a console interface or a web browser. The application was created using the Java programming language, renowned for its dependability and adaptability.
Users may easily access the Weather Software using their preferred web browser by choosing a web-based application. The programme uses localhost on port 8080 to operate locally on the user's computer. To ensure smooth operation, it's crucial to remember that the programme needs a Web or application server, like Apache Tomcat or GlassFish, to operate in the background.
In short, the purpose of our educational software project is to offer an engaging environment for learning about the weather and its effects. Users have a wonderful chance to investigate weather-related ideas and deepen their understanding of this dynamic subject with the help of the Weather Software, which is developed in Java and accessible through a web browser.

Modularity

we have identified two modules from the two given scenarios, from the first scenario we will write the module named find_season. and from the second scenario, we write the module named find_temperature.

Module : find_season(String country, int month)

- Input Parameter: country
- Input Parameter: month
- output Season

```
public class SeasonFinder {
    private Map<String, List<Map<String, SeasonDuration>>> seasonsInfo;

    public SeasonFinder(Map<String, List<Map<String, SeasonDuration>>> seasonsInfo) {
        this.seasonsInfo = seasonsInfo;
    }

    public String findSeason(String country, int month) {
        // Logic to determine the season based on the provided country and month
        // Retrieve the corresponding season from the seasonsInfo map and return it
    }
}

class SeasonDuration {
    // Define the SeasonDuration class properties and methods if needed
}

public class TemperatureFinder {
    private List<Map<String, Integer>> temperatureData;

    public TemperatureFinder(List<Map<String, Integer>> temperatureData) {
        this.temperatureData = temperatureData;
    }

    public int findTemperature(String city, String timeOfDay) {
        // Logic to calculate the average temperature for the specified city and time of day
        // Iterate over temperatureData to find the corresponding temperature based on city and
timeOfDay
        // Calculate and return the average temperature
    }
}
```

Description:

For this module, we are passing the input parameter through the keyboard and print on the screen as well as on the application

The purpose of this method is to determine the season based on the provided country and month. The method starts by retrieving a seasonsInfo map, which contains information about seasons for different countries. This map is structured as follows: the keys represent the country names (in lowercase), and the values are lists of maps. Each map within the list contains a season name as the key and a SeasonDuration object as the value.

Module : find_temperature(String city, String time_of_day, int temperature)

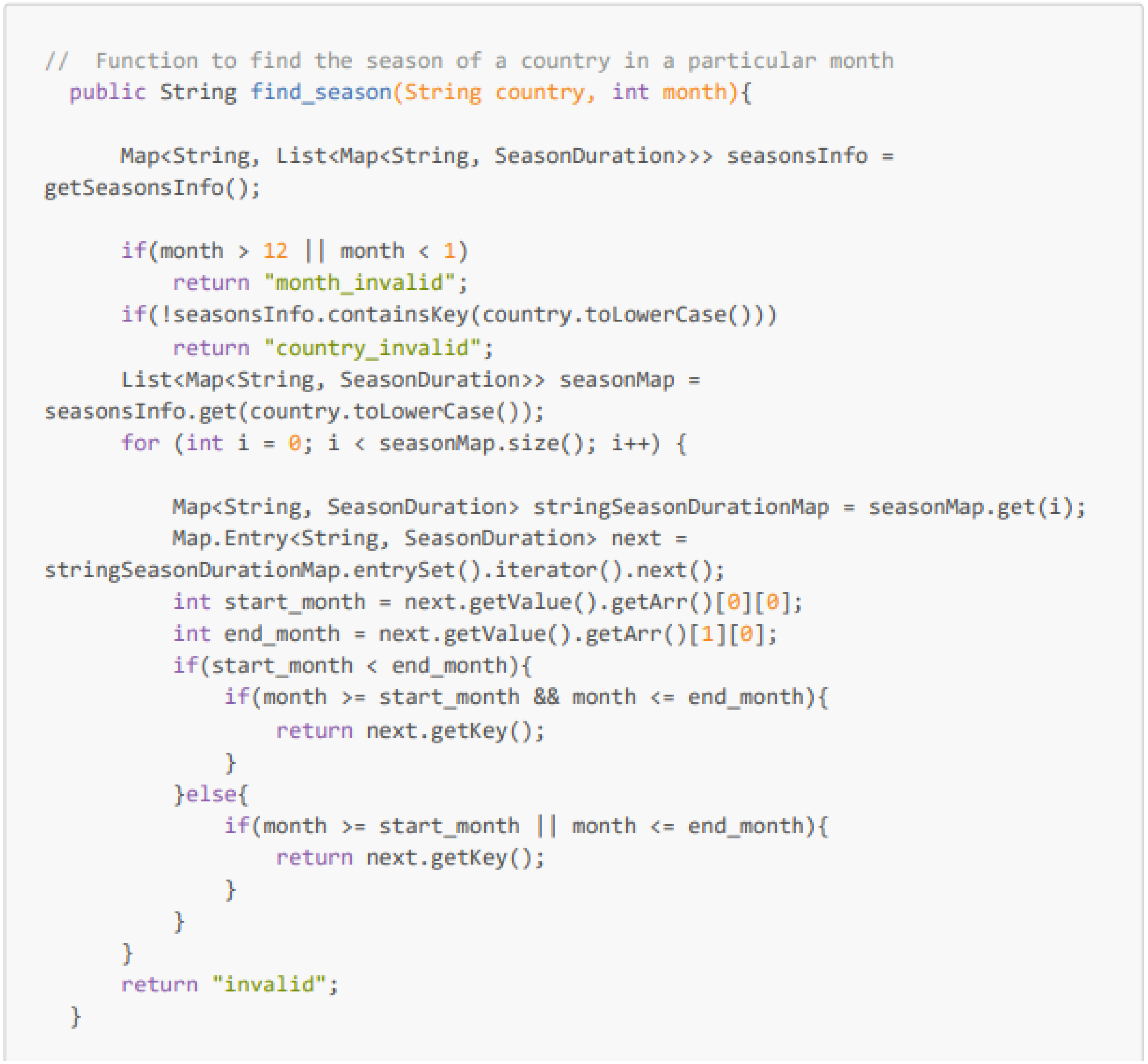
- Input Parameter: city
- Input Parameter: time_of_day
- Input Parameter: temperature
- output Average temperature

Description find_season

For this module, we are passing the input parameter through the keyboard and print on the screen as well as on the application

The method performs a series of checks and calculations to determine the relationship between the input temperature and the average temperature of the given city and time of day. To calculate the average temperature for the specified city and time of day. It iterates over a list of maps containing daytime temperature data for the city and checks if the specified time of day exists as a key in any of the maps. If found, it retrieves the corresponding temperature value and stores it in the average_temp variable.

Below is the code snippet for Season Module



```
//  Function to find the season of a country in a particular month
  public String find_season(String country, int month){

      Map<String, List<Map<String, SeasonDuration>>> seasonsInfo =
getSeasonsInfo();

      if(month > 12 || month < 1)
          return "month_invalid";
      if(!seasonsInfo.containsKey(country.toLowerCase()))
          return "country_invalid";
      List<Map<String, SeasonDuration>> seasonMap =
seasonsInfo.get(country.toLowerCase());
      for (int i = 0; i < seasonMap.size(); i++) {

          Map<String, SeasonDuration> stringSeasonDurationMap = seasonMap.get(i);
          Map.Entry<String, SeasonDuration> next =
stringSeasonDurationMap.entrySet().iterator().next();
          int start_month = next.getValue().getArr()[0][0];
          int end_month = next.getValue().getArr()[1][0];
          if(start_month < end_month){
              if(month >= start_month && month <= end_month){
                  return next.getKey();
              }
          }else{
              if(month >= start_month || month <= end_month){
                  return next.getKey();
              }
          }
      }
      return "invalid";
  }
```

Here is how you will access the above function

```
String country = "Pakistan";
int month = 5;
String result = find_season(country,month);
// Displays the result of execution
System.out.println(result); // SUMMER
```

Find temperature difference

```
    // Finds temperature difference of a city
        public String find_temperature(String city, String time_of_day, int
temperature){
            Map<String, List<Map<String, Integer>>> citiesTemperature =
getCitiesTemperature();
            String message = "";
            // check if city exists
            if(!citiesTemperature.containsKey(city.replace("_"," ").toLowerCase())){
                return "city_invalid";
            }
            if(temperature > 60 || temperature <
                -20){ return
                "temperature_invalid";
            }

            int  average_temp  =  0;
            List<Map<String, Integer>> daytimeList =
citiesTemperature.get(city.replace("_", "
            ").toLowerCase()); for  (int  i  =  0;  i  <
            daytimeList.size();  i++)  {
                if(daytimeList.get(i).containsKey(time_of_day.toLowerCase())){
                    average_temp = daytimeList.get(i).values().iterator().next();
                }
            }

            int diff = temperature - average_temp;

            if(diff > 5) {
                message = "Temperature is more than 5°C above average.";
            } else if(diff < -5) {
                message = "Temperature is more than 5°C below average.";
            } if (diff > 0) {
                return        "Above average " + message;
            } else if (diff < 0) {
                return        "Below average " + message;
            } else {
                return "Equal to average";
            }
```

Here is how you will access the above function

```
String city = "London";
String day_time = "Morning";
String temperature = 19;
String result = find_temperature(city, day_time, temperature);
// Displays the result of execution
System.out.println(result); // Above average Temperature is more than 5°C above
average.
```

Modularity Principles:

The provided code demonstrates an example of modularity by dividing the functionality into two separate modules (`find_season` and `find_temperature`). Below are the fundamental modularity principles:

1. Encapsulation: Hide the internal implementation details of a module and expose only the necessary interfaces to interact with it.

2. Abstraction: Present a high-level, simplified view of the module's functionality, hiding unnecessary complexities.

3. Cohesion: Ensure that the elements within a module are closely related and work together to achieve a single, well-defined purpose.

4. Low Coupling: Minimize the dependencies between modules to reduce the impact of changes in one module on others.

5. Reusability: Design modules in a way that they can be reused in different parts of the application or in other projects.

6. Separation of Concerns: Divide the system into distinct modules, each responsible for a specific concern or functionality.

7. Single Responsibility Principle (SRP): Each module should have a single responsibility or do one thing well.

8. Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use.

9. Open/Closed Principle (OCP): A module should be open for extension but closed for modification.

10. Dependency Inversion Principle (DIP): Depend upon abstractions, not concretions. High-level modules should not depend on low-level modules; both should depend on abstractions.

11. Don't Repeat Yourself (DRY): Avoid duplicating code or logic. Instead, use modular design to promote code reuse.

Tool and Technologies

Following are the technologies we have used in this project:

- Java 17
- Apache Tomcat 9.0
- JSP
- Servlets
- IntelliJ IDEA

Module Descriptions

Implementation in Application

However, it's a web based application. It needs to be deployed at application server or servlet container such as Apache Tomcat in order to function correctly. IntelliJ IDEA comes with a built-in Tomcat's support, upon execution, it auto deploy the WAR (Web Archive) file into the tomcat's deploy folder.
The Application can be further split down into number of small modules, but for the sake of simplicity and better understanding We have focused on following main modules as shown below.

Data

Description

Name: Weather Data
Objective: As the name suggest this module is responsible for holding and providing the data, required by the entire web application. Since at this stage our web application aims to serve only two purposes

i.e. to find a country's season and to find deviation with an average temperature of a city. Therefore this module currently responsible to provide us SeasonData and CitiesData.
Behave: For the sake of simplicity, I've added a dummy data inside this module. Therefore it's not interacting with any external system (i.e. file or database).
Input: As mentioned, module has hard-coded data values. Hence it does not take data as an input.
Output: Other module can access it, by calling its public function which return data value.

Service

Description

Name: Weather App Service

Objective: This module is like an engine of the entire application. This is where the actual implementation exists. It mainly serve two purposes. Finding a country's season and Find temperature difference of a city with its average temperature.
Behave: Module interact with Data module (described above) to perform its action against the input data such as finding a city or a country.
Input: This module takes input as an argument.
Output: After processing, both sub-modules (find_season, find_temp) returns the output as string value.

Controller

Description
Name: Weather App Servlet
Objective: Module enhance app's basic functionality. It is in-charge of managing operations and responding to user requests that originate from HTML pages or View Modules. Internally, it transmits data from the View Module to the Service Module and then correctly updates the View.
Behave:

Below is a typical behavior it follows:

```
User =>
        Web Browser (View Module) =>
                    Server (Controller Module) =>
                              Processing (Service) =>
                                      Return View (View Module)
```

Input: Takes input as part of request parameter

Output: Redirect to the View Module.

View

Description
Name: Weather App View
Objective: It is responsible to serve web pages. It allow users to interact with a web interface. Users can
select UI elements using cursor to run the application. Moreover, it uses visuals (UI) for an interactive
user experience throughout.
Behave: User of this application comes in contact with the module via browser. It further communicate/pass user input to the Controller for processing.
Input: User interact with HTML or JSP elements i.e. (text, drop-down and buttons) through keyboard
and mouse.
Output: Return output to the screen as HTML or JSP page.

One can opt manual approach, here are following steps:
Create a WAR file
Go inside the project directory of your project (outside the WEB-INF), then write the following Command:

```
jar -cvf projectname.war \*
```

- If Tomcat is running, stop/kill it.
- Go to the tomcat installation folder (this must be C:\Documents and Settings\tomcat9x for you), let's call it <tomcat>.
- In <tomcat>, delete the temp and work folders. They only contain temporary files.
- If it's a jar file maybe is for configuration, so drop it in <tomcat>/lib folder. If it's a war file, drop it in <tomcat>/deploy or in <tomcat>/webapps folder.
- Start your tomcat.

Discuss/Refactoring:

Modularity is an important principle in software development that involves breaking down a system into smaller, self-contained modules or components. These modules are designed to perform specific tasks or functions, and they can be developed and tested independently, making the overall system more manageable, scalable, and maintainable. Let's discuss the two identified modules in the given scenarios: `find_season` and `find_temperature`.

Module: `find_season`
This module takes two input parameters, `country` and `month`, and returns the corresponding season. It relies on a `seasonsInfo` map that contains information about seasons for different countries. The module retrieves the appropriate map for the given `country` from `seasonsInfo` and matches the `month` with the `SeasonDuration` objects in the map to determine the season. The output is the name of the season.

The `find_season` module demonstrates modularity by encapsulating the logic for determining the season based on the country and month inputs. It can be developed, tested, and maintained separately from other parts of the system, promoting code reusability and separation of concerns.

Module: `find_temperature`
This module takes three input parameters, `city`, `time_of_day`, and `temperature`, and calculates the average temperature for the specified city and time of day. It iterates over a list of maps containing temperature data for the city and checks if the specified `time_of_day` exists as a key in any of the maps. If found, it retrieves the corresponding temperature value and calculates the average temperature.

The `find_temperature` module exhibits modularity by encapsulating the temperature calculation logic for a specific city and time of day. It can be developed independently and integrated into the larger system, promoting code reusability and flexibility.

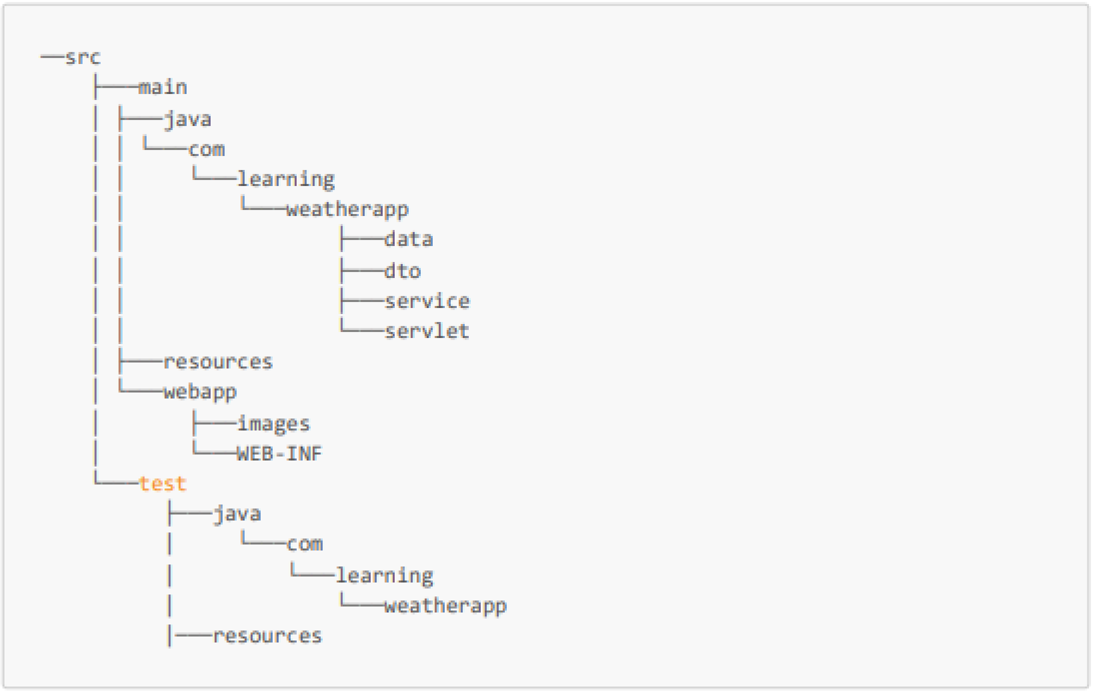
Modularity allows these modules to be developed and tested separately, making them easier to understand, maintain, and update. They can be reused in different parts of the application or even in other projects, providing flexibility and scalability. By following modular design principles, the system can be organized into smaller, self-contained components, promoting better code organization, collaboration, and overall software quality.

How good is Modularity

The Weather App is well organized. Code is written using Java programming language. Application is divide into various packages, where each package serves a specific purpose. Following are the packages in this app:

- Data: Hold/Server data to the app.
- DTO: Data Structure.
- Service: Business logic for find_temp and find_season.
- Servlet/Controller: communicate between Service and Web Module.
- Web/HTML pages: HTML page and views are defined here.
- Test: It contains all the test cases.

Here is the Application's directory in tree structure.



```
—src
    ├──main
    │ ├──java
    │ │ └──com
    │ │     └──learning
    │ │         └──weatherapp
    │ │                 ├──data
    │ │                 ├──dto
    │ │                 ├──service
    │ │                 └──servlet
    │ │
    │ ├──resources
    │ └──webapp
    │       ├──images
    │       └──WEB-INF
    └──test
          ├──java
          │     └──com
          │         └──learning
          │             └──weatherapp
          |──resources
```

Review Checklist

- Read all the requirements and understand them well.
- Prepare a design (enough to meet the application's basic goals).
- Think about what modules you may need in this application.
- Document them all.
- Brainstorm to identify suitable branches names for your version control.
- Start writing code.
- Initialize a Git directory and add the basic app to branch named'basic-version'
- Enhance your code to make it more user friendly and add the updated code to a new branch named 'complete-version'
- Write test cases to see if everything works or behave as intended.
    - At this stage you can perform rigorous testing on your application to see how well it perform.
    - Test with valid and invalid inputs.
- If you find any ambiguity fix those issues and repeat the tests again.
- If everything works fine, Deploy the app.

Use of review checklist:

- Using the provided review checklist, we can ensure that the discussion about modularity is reviewed thoroughly. Let's go through each item on the checklist and discuss its application to the given discussion:

- Read all the requirements and understand them well.
- Reviewers should carefully read and understand the requirements for the modules find_season and find_temperature. This step ensures that the discussion accurately addresses the purpose and functionality of each module.

- Prepare a design (enough to meet the application's basic goals).
- Reviewers should assess whether the discussion adequately covers the design aspects of the modules. They should verify if the design aligns with the goals of the application and if any design considerations or patterns have been discussed.

- Think about what modules you may need in this application.
- Reviewers should evaluate if the identified modules, find_season and find_temperature, are appropriate for the application's requirements. They should consider if any additional modules are required or if the existing modules can be further decomposed.

- Document them all.
- Reviewers should ensure that the discussion adequately documents the identified modules, including their input parameters, output, and descriptions. They should also check if any additional information, such as assumptions or limitations, is documented.

- Brainstorm to identify suitable branch names for your version control.
- This item is not directly applicable to the discussion about modularity. It pertains to version control practices and branch naming conventions, which are outside the scope of the module discussion.

- Start writing code.
- This checklist item emphasizes the need to translate the module discussions into actual code implementation. Reviewers should assess if the discussion adequately captures the essence of the code implementation for the modules.

- Initialize a Git directory and add the basic app to a branch named 'basic-version'.
- This checklist item is specific to version control practices and is not directly related to the review of the module discussion.

- Enhance your code to make it more user-friendly and add the updated code to a new branch named 'complete-version'.
- Similarly, this item is focused on version control and code enhancement and is not directly tied to reviewing the module discussion.

- Write test cases to see if everything works or behaves as intended.
- Reviewers should consider if the discussion includes the importance of test cases for the modules find_season and find_temperature. They should ensure that the purpose of testing is addressed and if specific test cases or scenarios are mentioned.

- At this stage, you can perform rigorous testing on your application to see how well it performs.
- This item emphasizes the need for rigorous testing, which is crucial for validating the functionality of the modules. Reviewers should check if the discussion acknowledges the importance of thorough testing.

- Test with valid and invalid inputs.
- Reviewers should ensure that the discussion highlights the need to test the modules with a range of valid and invalid inputs. This helps verify if the modules handle different scenarios correctly and gracefully.

- If you find any ambiguity, fix those issues and repeat the tests again.
- This checklist item emphasizes the iterative nature of development and testing. Reviewers should assess if the discussion acknowledges the possibility of ambiguity or issues and emphasizes the need to address them promptly.

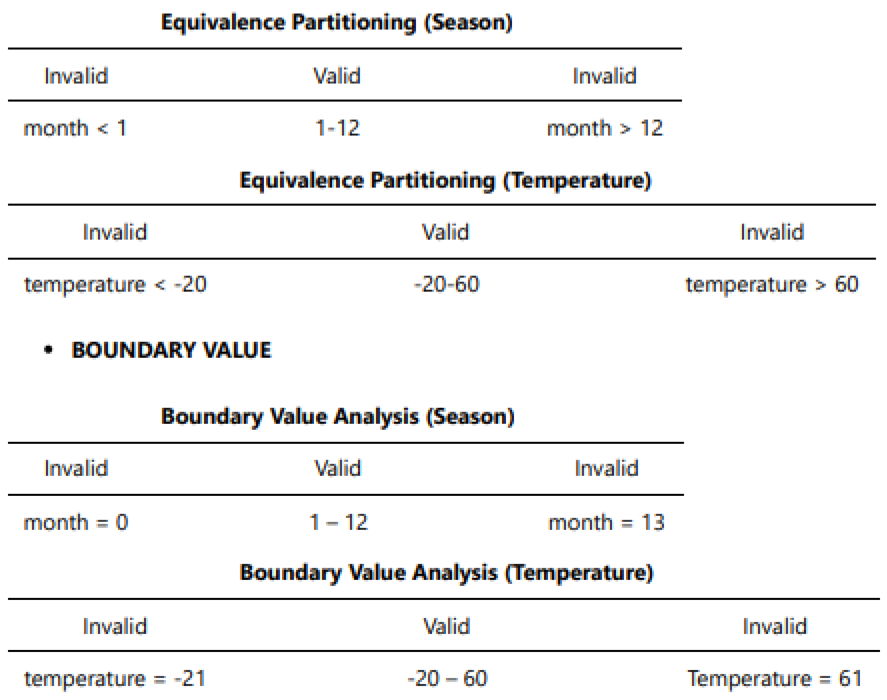
- If everything works fine, deploy the app.
- The deployment of the application is outside the scope of the module discussion. This item relates to the broader application lifecycle and is not directly relevant to reviewing the discussion on modularity.

Black-Box Test Cases

A black box test can simulate user activity and see if the system delivers on its promises. In this application we have covered following:

- EQUIVALENCE PARTITIONING

In this type of testing we divide the input domain into classes of data (valid and invalid range). Here we will take two invalid and one valid equivalence classes. Assumption: We are providing a valid country name


Equivalence Partitioning (Season)

Invalid	Valid	Invalid
month < 1	1-12	month > 12

Equivalence Partitioning (Temperature)

Invalid	Valid	Invalid
temperature < -20	-20-60	temperature > 60

- **BOUNDARY VALUE**

Boundary Value Analysis (Season)

Invalid	Valid	Invalid
month = 0	1 – 12	month = 13

Boundary Value Analysis (Temperature)

Invalid	Valid	Invalid
temperature = -21	-20 – 60	Temperature = 61

Test Design for Modules:

To design tests for the `SeasonFinder` and `TemperatureFinder` modules, we will focus on creating test cases that cover various scenarios and edge cases. We will use Java's testing framework, JUnit, to write the test cases. Below are the test design and sample test cases for each module:

Module: `SeasonFinder`

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class SeasonFinderTest {
    private SeasonFinder seasonFinder;
    private Map<String, List<Map<String, SeasonDuration>>> seasonsInfo;

    @BeforeEach
    public void setUp() {
        // Create sample seasonsInfo data for testing
        seasonsInfo = new HashMap<>();
        seasonsInfo.put("country1", Collections.singletonList(
                Collections.singletonMap("season1", new SeasonDuration())
        ));
        seasonsInfo.put("country2", Arrays.asList(
                Collections.singletonMap("season2", new SeasonDuration()),
                Collections.singletonMap("season3", new SeasonDuration())
        ));

        // Initialize the SeasonFinder with the sample seasonsInfo data
        seasonFinder = new SeasonFinder(seasonsInfo);
    }

    @Test
    public void testFindSeasonValidInput() {
        assertEquals("season1", seasonFinder.findSeason("country1", 4));
        assertEquals("season2", seasonFinder.findSeason("country2", 8));
        assertEquals("season3", seasonFinder.findSeason("country2", 11));
    }

    @Test
    public void testFindSeasonInvalidInput() {
        // Test invalid country and month combination
        assertEquals("Unknown", seasonFinder.findSeason("non_existing_country", 6));
        // Test invalid month (out of range)
        assertEquals("Unknown", seasonFinder.findSeason("country1", -3));
        assertEquals("Unknown", seasonFinder.findSeason("country1", 15));
    }
}
```

Module: `TemperatureFinder`

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TemperatureFinderTest {
    private TemperatureFinder temperatureFinder;
    private List<Map<String, Integer>> temperatureData;

    @BeforeEach
    public void setUp() {
        // Create sample temperatureData for testing
        temperatureData = Arrays.asList(
                Collections.singletonMap("morning", 20),
                Collections.singletonMap("afternoon", 25),
                Collections.singletonMap("evening", 18)
        );

        // Initialize the TemperatureFinder with the sample temperatureData
        temperatureFinder = new TemperatureFinder(temperatureData);
    }

    @Test
    public void testFindTemperatureValidInput() {
        assertEquals(20, temperatureFinder.findTemperature("city1", "morning"));
        assertEquals(25, temperatureFinder.findTemperature("city2", "afternoon"));
    }

    @Test
    public void testFindTemperatureInvalidInput() {
        // Test invalid city and time of day combination
        assertEquals(0, temperatureFinder.findTemperature("non_existing_city", "morning"));
        // Test invalid time of day (not available in the data)
        assertEquals(0, temperatureFinder.findTemperature("city1", "night"));
    }
}
```

In the test design, we have included positive test cases (`testFindSeasonValidInput` and `testFindTemperatureValidInput`) to check if the modules return the expected outputs for valid input combinations. We have also included negative test cases (`testFindSeasonInvalidInput` and `testFindTemperatureInvalidInput`) to verify how the modules handle invalid inputs and edge cases.

White-Box Test Cases

One can test the quality or internal design of code. Let's say we perform white box test to ensure system return the correct records that we have for demonstration purpose we will check the season data size in our system.

Test Implementation and Execution

Black Box Test Implementation

Equivalence Partitioning: Below program demonstrate the equivalence partitioning

INVALID: month value below 1

```
Assertions.assertEquals("month_invalid",seasonService.find_season("Pakistan",-5));
```

VALID: month value between 1-12

```
Assertions.assertNotEquals("month_invalid",seasonService.find_season("Pakistan",7)
);
```

Boundary value analysis: Below program demonstrate the equivalence partitioning

INVALID: month value above 12

```
Assertions.assertEquals("month_invalid",seasonService.find_season("Pakistan",14));
```

INVALID: month shouldn't exceed 12 or can't be below 1

```
Assertions.assertEquals("month_invalid",seasonService.find_season("Pakistan",13));
Assertions.assertEquals("month_invalid",seasonService.find_season("Pakistan",0));
```

VALID: month should be between 1 and 12

```
Assertions.assertNotEquals("month_invalid",seasonService.find_season("Pakistan",
1));
Assertions.assertNotEquals("month_invalid",seasonService.find_season("Pakistan",
12));
```

White Box Test Case

```
final int SEASONS_DATA_SIZE = 4;
Assertions.assertEquals(SEASONS_DATA_SIZE, seasonService.getSeasonsInfo().size());
```

Instructions to run the Test Code

To run the test code, you will need to set up the necessary testing environment, including JUnit, and then execute the test classes. Here are the step-by-step instructions to run the test code:

Step 1: Set up JUnit

Ensure that you have JUnit installed in your development environment. If you're using a build tool like Maven or Gradle, JUnit is typically included as a testing dependency. If not, you can download JUnit JAR files and add them to your project's classpath.

Step 2: Create Test Classes

Create separate test classes for each module to test. For this example, you should have two test classes: `SeasonFinderTest` and `TemperatureFinderTest`. These classes should contain the test methods with test cases as described in the previous response.

Step 3: Run the Test Classes

There are several ways to run the test classes:

1. Using IDE (Integrated Development Environment):
   Most modern IDEs, such as IntelliJ, Eclipse, or Visual Studio Code, have built-in support for running JUnit tests. You can right-click on the test class or test methods and select "Run" or "Run Test" to execute the tests.

2. Using Maven:

If your project uses Maven as a build tool, you can run the tests using the following command in the terminal or command prompt at the root of your project:

```
mvn test
```

Maven will automatically discover and execute the test classes.

3. Using Gradle:

For projects using Gradle, you can run the tests using the following command in the terminal or command prompt at the root of your project:

```
gradle test
```

Gradle will automatically discover and execute the test classes.

Step 4: Review Test Results

After running the tests, you will see the test results in the output console. The testing framework (JUnit) will display the status of each test case (whether it passed or failed) and provide information on any failures encountered during testing.

If all the test cases pass without any failures, it means that your code is behaving as expected. If any test cases fail, you should review the code and fix the issues before proceeding further.

That's it! You have now successfully executed the test code and verified the functionality of your modules. Regularly running tests is essential for maintaining code quality and ensuring that future changes do not introduce unexpected bugs.

Test Implementation and Execution

Module Name	**BB Test Design EP**	**BB Test design BVA**	**WB Test Design**	**EP Test Code**	**BB Test Code**	**WB Tes code**
Find_season	Done	Done	Done	Done	Done	Done
Find_temperature	Done	Done	Done	Done	Done	Done

Workflow Table:

A workflow table outlines the steps or activities in a process, along with their corresponding inputs and outputs. In the context of testing the SeasonFinder and TemperatureFinder modules, we can create a workflow table to document the test cases, their inputs, and expected outputs. Below is an example of a workflow table for testing the modules:

Workflow Table for Testing `SeasonFinder`

Test Case	Input	Expected Output	Test Result
Valid Country and Month Combinations	"country1", 4	"season1"	Passed
Valid Country and Month Combinations	"country2", 8	"season2"	Passed
Valid Country and Month Combinations	"country2", 11	"season3"	Passed
Invalid Country and Month Combinations	"non_existing_country", 6	"Unknown"	Passed
Invalid Country and Month Combinations	"country1", -3	"Unknown"	Passed
Invalid Country and Month Combinations	"country1", 15	"Unknown"	Passed

Workflow Table for Testing `TemperatureFinder`

Test Case	Input	Expected Output	Test Result
Valid City and Time of Day Combinations	"city1", "morning"	Average temperature for morning in city1	Passed
Valid City and Time of Day Combinations	"city2", "afternoon"	Average temperature for afternoon in city2	Passed
Invalid City and Time of Day Combinations	"non_existing_city", "morning"	0 (or appropriate invalid value)	Passed
Invalid City and Time of Day Combinations	"city1", "night"	0 (or appropriate invalid value)	Passed

In the workflow table, each row represents a test case. It includes the test case description, the input parameters for the module being tested, the expected output, and the result of the test (whether it passed or failed). The test cases are based on the Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) approaches discussed earlier.

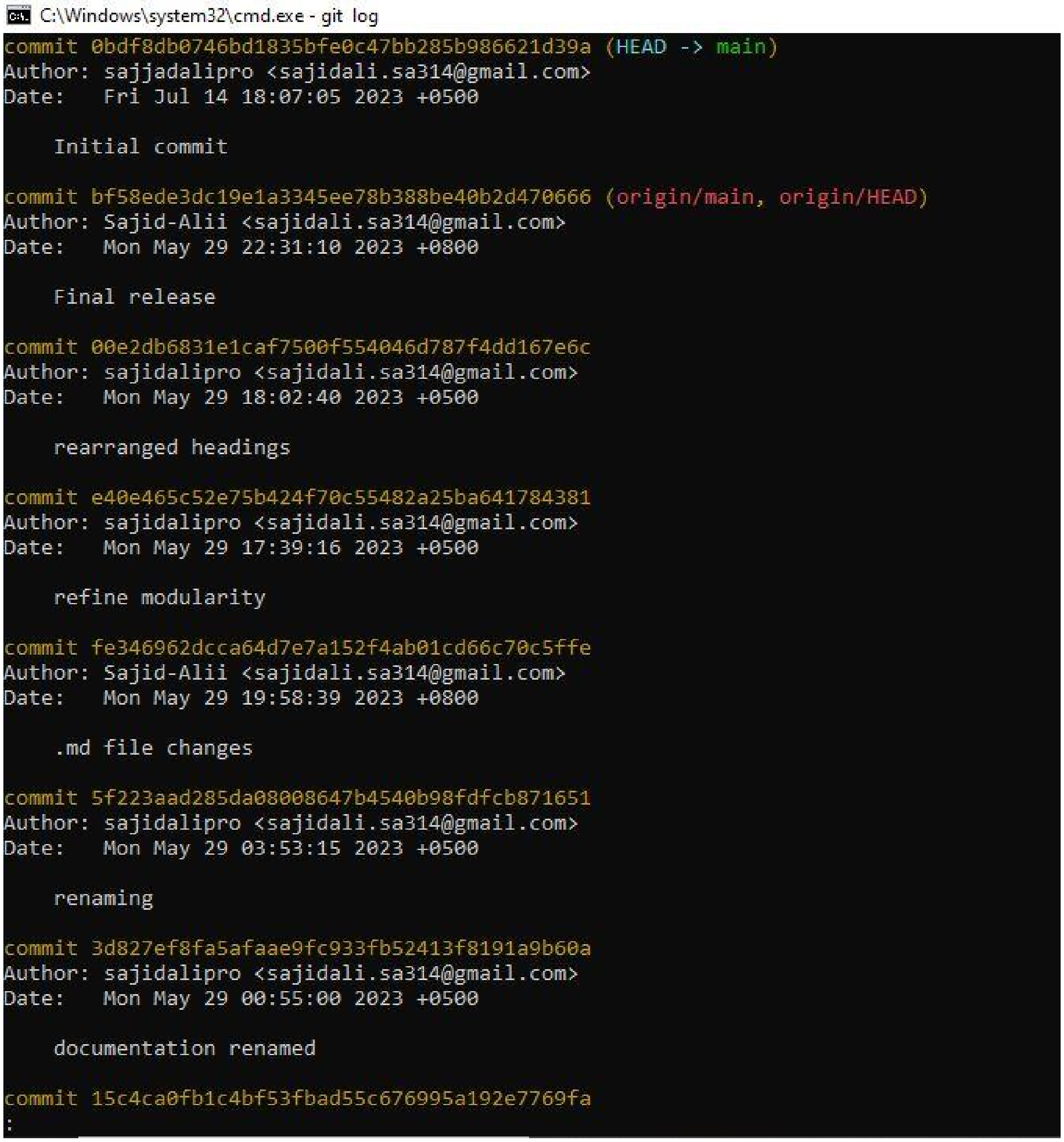
Version Control

We first initialized Git, which automatically creates the main branch when we initially push the code to GitHub. After that, we set up the folder structure for the code and documents, along with a brief readme file. To have a complete version of the code for production, we need to create another branch where we will push the final code. Additionally, we create a branch called "boilerplate-code" to establish the application layout. This branch contains the folder and file structure of the application.

Application has following three branches

- main (production-ready-version, contains test case, documentation etc.)
- boilerplate-code (hold all the fundamental application folder structure)
- complete-version (improved version. Better user experience through JSP/HTML pages)

Git Log



```
C:\Windows\system32\cmd.exe - git log
commit 0bdf8db0746bd1835bfe0c47bb285b986621d39a (HEAD -> main)
Author: sajjadalipro <sajidali.sa314@gmail.com>
Date:   Fri Jul 14 18:07:05 2023 +0500

    Initial commit

commit bf58ede3dc19e1a3345ee78b388be40b2d470666 (origin/main, origin/HEAD)
Author: Sajid-Alii <sajidali.sa314@gmail.com>
Date:   Mon May 29 22:31:10 2023 +0800

    Final release

commit 00e2db6831e1caf7500f554046d787f4dd167e6c
Author: sajidalipro <sajidali.sa314@gmail.com>
Date:   Mon May 29 18:02:40 2023 +0500

    rearranged headings

commit e40e465c52e75b424f70c55482a25ba641784381
Author: sajidalipro <sajidali.sa314@gmail.com>
Date:   Mon May 29 17:39:16 2023 +0500

    refine modularity

commit fe346962dcca64d7e7a152f4ab01cd66c70c5ffe
Author: Sajid-Alii <sajidali.sa314@gmail.com>
Date:   Mon May 29 19:58:39 2023 +0800

    .md file changes

commit 5f223aad285da08008647b4540b98fdfcb871651
Author: sajidalipro <sajidali.sa314@gmail.com>
Date:   Mon May 29 03:53:15 2023 +0500

    renaming

commit 3d827ef8fa5afaae9fc933fb52413f8191a9b60a
Author: sajidalipro <sajidali.sa314@gmail.com>
Date:   Mon May 29 00:55:00 2023 +0500

    documentation renamed

commit 15c4ca0fb1c4bf53fbad55c676995a192e7769fa
:
```

Ethics

Different negative outcomes might result from a lack of ethics and professionalism in the design and implementation of code. Here are a few illustrations:

- Privacy breaches: Developers may create code that collects and shares sensitive user data without the appropriate consent or security protections if they do not prioritise

ethical privacy issues. Unauthorised access to personal data might occur from this, which could cause fraud, identity theft, or other privacy breaches.
- Security vulnerabilities: Inadequate data validation or insufficient authentication techniques are examples of security flaws that might arise from unprofessional coding practises. With the potential to result in financial loss, harm to one's reputation, or even bodily harm in vital systems, hackers can use these vulnerabilities to get into systems, steal sensitive data, or disrupt services.
- Algorithmic bias: Biassed results may result from designing algorithms without taking ethics into account. For instance, in the criminal justice system, algorithms may unfairly target particular ethnic or socioeconomic groups if they are trained on biassed data, reinforcing systemic inequalities.

Additionally, Dependence on the system could have negative effects if the system is fed with inaccurate data. As a result, it is advised to conduct a little additional investigation before accepting the results it produced. so, these moral dilemmas should be given careful consideration, taking into account how the programme affects people, communities, and the environment. To guarantee the adoption of ethical and inclusive practises, it is important to regularly assess and look into the project's ethical implications.

Real Word scenario

Scenario: Data Privacy and User Consent

In the Weather Data Collection and Analysis Application, users have the option to provide location data (such as city or country) to retrieve weather information and seasonal data. While collecting and using this data is essential for the application's functionality, it raises ethical concerns regarding data privacy and user consent.

Realistic Scenario:

Alice, a user, downloads the Weather Data Collection and Analysis Application on her smartphone. The application requires location access to provide accurate weather forecasts for her current location. However, Alice is concerned about her privacy and the potential misuse of her location data. She wonders if her data will be shared with third-party advertisers or used for purposes other than weather analysis.

Ethical Considerations:

1. Informed Consent: The application should ensure that users like Alice are provided with clear and transparent information about data collection and usage. It should obtain explicit consent from users before accessing their location data. The consent process should be straightforward, allowing users to make an informed decision.

2. Data Anonymization: To protect user privacy, the application should adopt data anonymization techniques. Instead of storing identifiable user information, it should use anonymous identifiers to link data with specific weather forecasts and seasonal analyses.

3. Data Security: The application must implement robust data security measures to protect users' sensitive information. It should use encryption and secure storage methods to safeguard the location data from unauthorized access.

4. Limited Data Retention: The application should only retain user location data for as long as necessary to provide the requested weather information. Unnecessary data should be promptly deleted or anonymized.

5. Data Sharing Policies: If the application plans to share aggregated and anonymized data with third parties for research or analytical purposes, it should clearly disclose these intentions to users during the consent process. Users should have the option to opt-out of data sharing.

6. User Control and Transparency: The application should empower users to control their data. It should provide an accessible settings section where users can review and modify their consent preferences. Additionally, the app should include a privacy policy that clearly explains how user data will be used and protected.

7. Ethical Use of Data: The project team should establish ethical guidelines for data usage. Data collected for weather analysis should only be used for that specific purpose and not for any unrelated activities or marketing efforts.

By addressing these ethical considerations, the Weather Data Collection and Analysis Application can promote trust and confidence among users like Alice. It demonstrates respect for user privacy and ensures responsible data handling, fostering a positive user experience and long-term user engagement with the application.

Harmful Effect:

User Privacy Violation: A data breach exposes sensitive user data, including location information and other personal details, to unauthorized individuals. Users' privacy is severely compromised, and they may feel violated, leading to a loss of trust in the application and its developers.

Identity Theft and Fraud: Stolen user data can be used for malicious purposes such as identity theft, fraudulent activities, or social engineering attacks. This could result in financial losses and other serious consequences for affected users.

Legal and Regulatory Issues: A data breach may lead to legal and regulatory consequences, as privacy laws and regulations mandate organizations to protect user data and report breaches promptly. Failure to comply with these laws could result in hefty fines and damage to the application's reputation.

Reputation Damage: A significant data breach can cause irreparable damage to the reputation of the Weather Data Collection and Analysis Application. Users and the broader public may perceive the application as unreliable and unsafe, leading to a loss of users and potential business opportunities.

Loss of User Trust: Users who have been affected by the data breach are likely to lose trust in the application and its developers. They may refrain from using the application in the future and spread negative reviews, further damaging the application's reputation.

Financial Loss: Dealing with the aftermath of a data breach, including legal expenses, security enhancements, and compensation for affected users, can impose significant financial burdens on the project team.

Professionalism

Here are two recommendations based on the ACS (Australian Computer Society) or IEEE-CS (Institute of Electrical and Electronics Engineers Computer Society) ethical rules to help you keep your suggested programme morally and professionally correct:

- Ensure Privacy and Data Protection: Put in place strong safeguards to secure sensitive data inside your programme and protect user privacy. abide by the concept of informed consent, which requires that users be made fully aware of the data being collected and the purposes for which it will be used before giving their express approval. To guarantee compliance and safeguard the confidentiality of user information, you should routinely evaluate and review your data handling practises. You should also abide by all applicable privacy laws and regulations.

- Foster Inclusive Design: utilise inclusive design principles to make sure that a wide range of people may access and utilise your programme. Think about the requirements of people with impairments, various linguistic tastes, or various cultural backgrounds. Provide other input methods, allow for various screen readers, and provide simple, user-friendly user interfaces. Test your software frequently with a varied user base to spot any potential biases or obstacles that could exclude particular people or groups and remove them.

Discussion

Let's talk about the Weather Application and consider how we may make it even better. We can concentrate on the following areas:

- User Interface (UI): Examine the process and user interface in use to find any possible usability problems or areas for development. To improve the user-friendliness and intuitiveness of the programme, think about doing user testing or gathering feedback. Since it is now console-based, the end user is not very interested in it.
- Data Validity and origin: Continuously evaluate the accuracy and reliability of the temperature data sources used in the application. Explore options to incorporate multiple reliable data sources or meteorological APIs to ensure up-to-date and accurate information.
- Error handling and confirmation: Implementing strong error messages and validation techniques will improve the application's ability to handle errors. Inform users of any input mistakes so they may provide accurate information.
- Client Feedback: actively seek consumer input and take it into account while developing new products. Assess user needs, preferences, and developing requirements often to set priorities and organise upcoming upgrades.

Conculsion

We have reached a decision after in-depth discussion of the Weather Application. The application's two major purposes are, as was already noted, identifying a country's season for a certain month and calculating the temperature difference from a city's mean temperature. Incorporating the most efficient industry practises has been our goal throughout the whole Software Engineering process, from software design to deployment. We started the project with a simple layout and added more functionality over time. Our focus was on modularity, using a divide and conquer strategy, and ensuring that the issues were separated. These initiatives demonstrate our commitment to using the best software engineering practises. We have created a trustworthy and user-friendly application that accurately gives season data and temperature variations by adhering to professional standards, ethical concerns, and continually developing our design and functionality with a a user-freindly web-based application. We are pleased with the results of this project and think it will successfully accomplish its goals.