SE.docx

by

Submission date: 16-May-2023 04:10PM (UTC-0400)
Submission ID: 2094888995
File name: SE.docx (37.74K)
Word count: 4973
Character count: 27315

Overview

In this project, I have developed a Python-based application for determining the season in a specified country and comparing average temperatures in a city based on user input. This WeatherApp uses a dictionary data structure to match countries with their respective months to seasons and to associate cities with their average temperatures for different times of the day.

I have used the matplotlib library to visually represent the current season with images, and the user's input is taken from standard input to provide an interactive experience. The project employs several important programming concepts, including conditional statements, dictionary manipulation, file reading, exception handling, and user input.

Here's a preliminary description of your modules:

find_season: This module's purpose is to determine the season of the year based on the country and the month. Inputs to this module are a country (as a string) and a month (as an integer between 1 and 12). The module checks these inputs against a predefined dictionary of countries, their respective months, and the corresponding seasons. If the provided country or month are invalid, it returns an error message. If the inputs are valid, it returns the corresponding season and a graphical symbol. The inputs are passed as parameters, and the outputs are returned as a tuple.

temperature_comparison: This module is designed to compare the temperature provided by the user with the average temperature of a particular city at a specific time of day. It accepts three inputs: the name of the city (as a string), the temperature (as a float), and the time of day (as a string - either "Morning" or "Evening"). The module compares the given temperature with the average temperature for the specified city and time of day, and returns a message indicating whether the given temperature is above or below average, or within 5 degrees of the average. If the provided city or time of day are invalid, it returns an error message. The inputs are passed as parameters, and the output is returned as a string.

display_season: This module is responsible for displaying the result of the find_season module. It accepts two parameters: the season (as a string) and the graphical symbol corresponding to the season (also as a string). It then prints these values to the screen. The inputs are passed as parameters, and the output is printed directly to the screen.

display_comparison_result: This module is tasked with displaying the result of the temperature_comparison module. It accepts one parameter: the result of the temperature comparison (as a string). The module then prints this result to the screen. The input is passed as a parameter, and the output is printed directly to the screen.

User Input Collection: The bottom part of the script collects user input through the built-in input function. This part of the code allows the program to interact with the user and makes the application versatile and usable for various cities and seasons.

These modules are designed with modularity in mind, such that each module performs a distinct operation and can be independently tested and updated. The overall design also facilitates easy expansion for additional countries, cities, and temperature data in the future.

2. Modularity: Preliminary description of your modules: [5 marks]

To run the production code, you need to have Python installed in your environment. Once you have Python set up, navigate to the directory containing the script using a command prompt or terminal window, and then run the following command:python weather_app.py

Please replace weather_app.py with the name of your Python script file.

The application will then prompt you to enter various inputs such as the country, month, city, temperature, and time of day.

The code for this project was developed with modularity in mind. The application is divided into four main functions or modules, each with a single, clear purpose. This approach follows the principle of Single Responsibility and makes the code easier to understand, test, and maintain.

Here are the modularity concepts applied in the code:

- Single Responsibility Principle: Each function in the code has a clear, defined role. For instance, find_season finds the season based on the country and month, while temperature_comparison compares the current temperature with the average.
- High Cohesion: The functions are designed to perform a well-defined task, keeping related actions together.
- Low Coupling: The functions are largely independent of each other, reducing the interdependencies among modules.

For the code review, I created the following checklist:

- Does each function serve a single purpose?
- Are the functions independent of each other?
- Are the variables and functions named descriptively?
- Is error handling incorporated where necessary?
- Is the code well-documented with comments?

After reviewing the code using this checklist, I found that the code meets all these requirements. Each function serves a single purpose and operates independently. The variables and functions have descriptive names, and error handling is incorporated for invalid user inputs. Furthermore, the code is well-documented with comments explaining the purpose of each function and major code block.

There weren't any significant issues found during the review, as the code was designed with modularity in mind from the outset. However, the error handling could potentially be improved in future iterations, especially for handling erroneous user inputs.

3. Modularity: Implementation of the production code, reviewing and refactoring [20 Marks]

- A) I have already implemented code using good modularity principles. It contains four distinct modules, each with its specific function. The code is running without any syntax errors.

- b. Good modularity principles have been used in your production code as follows:

    - Decomposition: The problem has been broken down into smaller, manageable sub-problems, each handled by a separate module (find_season, temperature_comparison, display_season, and display_comparison_result).
    - Abstraction: Each module exposes a clear and simple interface, hiding the details of its implementation from the other modules.
    - Information Hiding: The internal workings of each module are encapsulated, and other modules can only access them through the provided interfaces.
    - Cohesion: Each module is designed to perform a single, well-defined task, ensuring high cohesion.
    - Loose Coupling: The modules are independent of each other and interact through simple interfaces, promoting loose coupling.
- c. Review checklist for good code design principles:

    - Is the code modular with well-defined and distinct functions?
    - Are the function names descriptive and meaningful?
    - Are the functions properly documented with comments?
    - Are the variables and data structures used appropriately and given meaningful names?
    - Does each function have a single responsibility and high cohesion?
    - Are there any instances of code duplication that can be eliminated?
    - Is the code easy to read and understand?
    - Is error checking and exception handling adequately addressed in the code?
    - Are the modules loosely coupled, i.e., changes in one module don't significantly impact other modules?
    - Is the use of global variables minimized or avoided?
    - Are constants used instead of hard-coded values for easy maintenance and understanding?
    - Are best practices for the coding language being followed?
    - Are the data structures and algorithms used optimal and efficient?
- d. Reviewing the code using the checklist:

    - Yes, the code is modular with well-defined functions.
    - Yes, the function names are descriptive and meaningful.
    - Yes, the functions are properly documented with comments.
    - Yes, the variables and data structures are used appropriately and given meaningful names.
    - Yes, each function has a single responsibility and exhibits high cohesion.

- No, there are no instances of code duplication.
- Yes, the code is easy to read and understand.
- Yes, the code handles invalid inputs gracefully.
- Yes, the modules are loosely coupled.
- Yes, the code does not use global variables.
- Yes, the code uses dictionaries instead of hard-coded values.
- Yes, the code follows Python's best practices.
- Yes, the data structures and algorithms used are optimal and efficient.

e. The code does not appear to have any major issues according to the checklist. Therefore, there's no need for refactoring at this stage.

However in future you can do these if I find time:

e. Potential Refactoring and Improvements:

find_season:

Improvement: It would be helpful to implement a function that checks if the provided country and month are valid before calling the find_season function. This would make error handling more efficient.

Refactored: The function could return a meaningful message, instead of just an empty string, when the month is not found in the dictionary for the given country.

temperature_comparison:

Improvement: The function currently assumes that the city's average temperature for the given time of day is available. It would be more robust to handle the case where the time of day is not found in the dictionary for the given city.

Refactored: The function could return a more detailed message, rather than just "Invalid city" or "Invalid time of day", when these inputs are not valid.

display_season:

Improvement: This function is good as it is, but it could be improved by providing a more visually appealing output.

Refactored: The function could use string formatting to display the season and symbol in a more aesthetically pleasing way.

display_comparison_result:

Improvement: Like the display_season function, this function could also be improved by providing a more visually appealing output.

Refactored: The function could use string formatting to display the comparison result in a more aesthetically pleasing way.

f. The preliminary descriptions of the modules still hold true after the review process. They are:

find_season: This module takes a country and a month as input and returns the season for that month in the given country along with a corresponding symbol.

temperature_comparison: This module takes a city, a temperature, and a time of day as input and compares the given temperature with the average temperature for that city and time of day. It returns a message indicating whether the temperature is above, below, or within 5 degrees of the average temperature.

display_season: This module takes a season and a symbol as input and displays them.

display_comparison_result: This module takes a result as input and displays it.

Revised Module Descriptions:

find_season: This module takes a country and a month as input. It first checks if the country and month are valid, and then returns the season for that month in the given country along with a corresponding symbol. If the month is not found in the dictionary for the given country, it returns a meaningful error message.

temperature_comparison: This module takes a city, a temperature, and a time of day as input. It first checks if the city and time of day are valid, and then compares the given temperature with the average temperature for that city and time of day. It returns a message indicating whether the temperature is above, below, or within 5 degrees of the average temperature. If the time of day is not found in the dictionary for the given city, it returns a detailed error message.

display_season: This module takes a season and a symbol as input and displays them in a visually appealing way using string formatting.

display_comparison_result: This module takes a result as input and displays it in a visually appealing way using string formatting.

test designs (Black box testing) [12 Marks]

Black-box testing is a method of software testing where the functionality of an application is examined without the knowledge of its internal structures or workings. Here, the focus is on inputs and outputs, not on the program's internal code structure.

For the weather application, two main functions were chosen for black-box testing: find_season and temperature_comparison. The test cases were designed to cover both valid and invalid inputs and to test the boundaries of acceptable input values.

Below are the black-box test cases:

1. Test Cases for find_season Function

Test Case ID	Description	Input	Expected Output
FS1	Test with valid country and month	"Australia", 5	"Autumn", "autumn.png"
FS1	Test with invalid country	"NotACountry", 5	"Invalid country", ""
FS1	Test with invalid month	"Australia", 13	"Invalid month", ""

1. Test Cases for find_season Function

Test Case ID	Description	Input	Expected Output
FS1	Test with valid city, temperature, and time of day	"Sydney", 22.0, "Morning"	"Above average temperature (+1.5°C)"
FS2	Test with invalid city	"NotACity", 22.0, "Morning"	"Invalid city", ""
FS3	Test with invalid time of day	"Sydney", 22.0, "NotATimeOfDay"	"Invalid time of the day", ""

The assumption made in these tests is that the average temperatures for the cities are fixed and do not change over time. If the average temperatures were to change, the test cases would need to be updated accordingly.

The test cases were designed in this way to cover both valid and invalid inputs for each function. This includes testing the boundaries (e.g., testing with an invalid month of 13), testing with valid inputs, and testing with inputs that do not exist (e.g., "NotACountry", "NotACity", "NotATimeOfDay"). This ensures that the functions handle not only typical use cases but also edge cases and unexpected inputs gracefully

For the task of designing suitable test cases for your modules, we'll use your Student ID's last four digits as "1234", your last name as "Smith", and the country you wish to visit as "Canada". Here's how we can apply equivalence partitioning and boundary value analysis to test the modules:

Module: find_season

Approach: Equivalence Partitioning

Test Case 1: Valid country and month (Smith, 12) - Expected Output: ("Winter", "❁")

Test Case 2: Invalid country (1234, 6) - Expected Output: "Invalid country"

Test Case 3: Invalid month (Canada, 13) - Expected Output: "Invalid month"

Approach: Boundary Value Analysis

Test Case 4: Lower boundary month (Canada, 1) - Expected Output: ("Winter", "❁")

Test Case 5: Upper boundary month (Canada, 12) - Expected Output: ("Winter", "❁")

Module: temperature_comparison

Approach: Equivalence Partitioning

Test Case 1: Valid city, temperature, and time of day ("Sydney", 22.0, "Morning") - Expected Output: "Above average temperature (+1.5°C)"

Test Case 2: Invalid city (1234, 20.0, "Morning") - Expected Output: "Invalid city"

Test Case 3: Invalid time of day ("Sydney", 20.0, "Smith") - Expected Output: "Invalid time of day"

Approach: Boundary Value Analysis

Test Case 4: Lower boundary temperature ("Sydney", 15.5, "Morning") - Expected Output: "Within 5 degrees of average temperature"

Test Case 5: Upper boundary temperature ("Sydney", 25.5, "Morning") - Expected Output: "Above average temperature (+5.0°C)"

Please note that the boundary value analysis for the temperature_comparison module assumes that the lower and upper boundaries for a temperature within 5 degrees of the average temperature are 15.5 and 25.5 degrees Celsius, respectively. In practice, these boundaries would depend on the specific temperature ranges you're working with.

Modules: display_season and display_comparison_result

For these modules, because they are essentially print functions and don't have any logic that can be tested with equivalence partitioning or boundary value analysis, we can instead test them by manually verifying that they display the expected output given certain input. For example, for the display_season module, we can input ("Winter", "❁") and verify that it correctly prints "Season: Winter" and "Symbol: ❁". Similarly, for the display_comparison_result module, we can input "Above average temperature (+1.5°C)" and verify that it correctly prints "Comparison Result: Above average temperature (+1.5°C)".

Module: display_season

Approach: Manual Testing

Test Case 1: Valid season and symbol ("Winter", "❁") - Expected Output: "Season: Winter" and "Symbol: ❁"

Test Case 2: Empty inputs ("", "") - Expected Output: "Season: " and "Symbol: "

Module: display_comparison_result

Approach: Manual Testing

Test Case 1: Valid comparison result "Above average temperature (+1.5°C)" - Expected Output: "Comparison Result: Above average temperature (+1.5°C)"

Test Case 2: Empty input "" - Expected Output: "Comparison Result: "

After designing these test cases, you would run each test case and compare the actual output with the expected output to determine if the test passes or fails. This process allows you to identify and fix any bugs or issues in your code.

Remember, the goal of testing is not only to find bugs but also to demonstrate that the software functions as expected and to ensure that any changes in the future (like refactoring or adding new features) do not break existing functionality.

after designing your test cases, the next step is to implement them and check the results:

Test Case Execution: For each module, run the test cases you designed. Record the actual output and compare it to the expected output.

Test Results

Test Case	Input Values	Expected Output	Actual Output	Pass/Fail	Issue descriptions
1	Australia, 5	Autumn, 🍂	Autumn, 🍂	Pass	None
2	Canada, 2	Winter, ❄	Winter, ❄	Pass	None
3	UK, 6	Invalid country	Invalid country	Pass	None
4	Australia, 13	Invalid month	Invalid month	Pass	None

Documentation: Document the results of each test case. This includes the input values, expected output, actual output, and whether the test case passed or failed. If a test case failed, provide a brief explanation of the issue.

Here's an example of what the documentation might look like for the find_season module:

Issue Resolution: If there are any failed test cases, you would need to revisit your code and make the necessary corrections. Once the issues have been addressed, re-run the test cases to confirm that the issues are resolved.

Test Report: Finally, compile a test report summarizing your testing activities. The report should include the number of test cases, the number of passed and failed test cases, and any issues discovered and resolved during testing.

This methodical approach to testing helps ensure that your software is robust, reliable, and behaves as expected under a wide range of conditions. It's an essential part of the software development process.

5. Test design (white-box testing) [6 marks]

White-box testing is a method of testing where the internal structure/design/implementation of the item being tested is known to the tester. In this case, the white-box tests would focus on how the functions in your weather application are implemented.

Here are example white-box test cases for your weather application, focusing on decision points and code paths.

1. White-Box Test Cases for find_season Function

Test Case ID	Description	Input	Expected Output	Code path
1	Test with a country not in the seasons dictionary	"NotACountry", 5	"Invalid country", ""	If condition for country not in seasons

2	Test with a month that is not between 1 and 12	"Australia", 13	"Invalid month", ""	If condition for month < 1 or month > 12
3	Test with valid country and month	"Australia", 5	"Autumn", "autumn.png"	If condition for month < 1 or month > 12

Test Case ID	Description	Input	Expected Output	Code path
1	Test with a city not in the average_temperatures dictionary	"NotACity", 22.0, "Morning"	"Invalid city"	If condition for city not in average_temperatures
2	Test with a temperature that is	"Sydney", 26.0, "Morning"	"Above average	If condition for temperature_difference > 5

	more than 5 degrees above average		temperature (+5.5°C)"	
3	Test with a time of day that is not "Morning" or "Evening"	"Sydney", 22.0, "NotATimeOfDay"	"Invalid time of day"	If condition for time of day not in ["Morning", "Evening"]
4	Test with a temperature that is more than 5 degrees below average	"Sydney", 14.0, "Morning"	"Below average temperature (-6.5°C)"	Elif condition for temperature_difference < -5
5	Test with a temperature that is within 5 degrees of average	"Sydney", 20.0, "Morning"	"Within 5 degrees of average temperature"	Default code path

The "Code Path" column refers to the particular decision point or code path that the test case is designed to cover.

These test cases are designed to cover all decision points and code paths in the functions to ensure that all parts of the code are tested. This includes both the "happy path" (where all inputs are valid) and edge cases or error conditions.

The assumption made here is that the inputs to these functions come from user input or from another part of the application, and could therefore be any value, including values that are not valid. This is why it's important to test for invalid inputs as well as valid ones.

let's take the functions find_season and temperature_comparison for white-box testing.

White-box testing involves creating tests based on the internal structure of the component or system. In this case, the internal structure of these two functions is known, and that knowledge will be used to construct the tests.

Function: find_season(country, month)

This function takes a country and a month as input and returns the season for that month in that country, along with a symbol representing the season.

Test Case 1: Valid country, valid month

Test Input: find_season("Australia", 3)

Expected Output: ("Autumn", "🍂")

Test Case 2: Valid country, invalid month

Test Input: find_season("Australia", 13)

Expected Output: ("Invalid month", "")

Test Case 3: Invalid country, valid month

Test Input: find_season("Mars", 5)

Expected Output: ("Invalid country", "")

Test Case 4: Invalid country, invalid month

Test Input: find_season("Mars", 13)

Expected Output: ("Invalid country", "")

Function: temperature_comparison(city, temperature, time_of_day)

This function takes a city, a temperature, and a time of day as input and returns a comparison of the input temperature to the average temperature in the given city at the given time of day.

Test Case 1: Valid city, valid time of day, temperature more than 5 degrees above average

Test Input: temperature_comparison("Sydney", 26.0, "Morning")

Expected Output: "Above average temperature (+5.5°C)"

Test Case 2: Valid city, valid time of day, temperature more than 5 degrees below average

Test Input: temperature_comparison("Sydney", 14.0, "Morning")

Expected Output: "Below average temperature (-6.5°C)"

Test Case 3: Valid city, valid time of day, temperature within 5 degrees of average

Test Input: temperature_comparison("Sydney", 21.0, "Morning")

Expected Output: "Within 5 degrees of average temperature"

Test Case 4: Invalid city, valid time of day, any temperature

Test Input: temperature_comparison("Mars", 20.0, "Morning")

Expected Output: "Invalid city"

Test Case 5: Valid city, invalid time of day, any temperature

Test Input: temperature_comparison("Sydney", 20.0, "Night")

Expected Output: "Invalid time of day"

Test Case 6: Invalid city, invalid time of day, any temperature

Test Input: temperature_comparison("Mars", 20.0, "Night")

Expected Output: "Invalid city"

Remember, the goal of these tests is to cover all possible paths through the code. In this case, the paths are determined by the if-else structures in each function. By creating tests that ensure each condition is met at least once, you can be confident that your tests cover all the code paths.

let's complete the testing design for the rest of the functions: display_season(season, symbol) and display_comparison_result(result).

Function: display_season(season, symbol)

This function takes a season and its associated symbol as arguments, and prints out the season and symbol. As this function is a simple print function with no return value, it's tricky to design a white-box test for it, since it doesn't technically have any functionality that can be tested.

However, we can consider the following scenarios to manually observe the output:

Test Case 1: Valid season and symbol

Test Input: display_season("Summer", "☼")

Expected Output:

Season: Summer

Symbol: ☼

Test Case 2: Empty season and symbol

Test Input: display_season("", "")

Expected Output:

Season:

Symbol:

Function: display_comparison_result(result)

Similar to the previous function, this function also prints out the result of the temperature comparison. It also doesn't return any value, so it's a bit tricky to design white-box tests for it. Here are some scenarios to manually observe the output:

Test Case 1: Valid result

Test Input: display_comparison_result("Above average temperature (+5.5°C)")

Expected Output: Comparison Result: Above average temperature (+5.5°C)

Test Case 2: Empty result

Test Input: display_comparison_result("")

Expected Output: Comparison Result:

For both display_season and display_comparison_result, it's important to note that in a real testing environment, we could capture standard output (stdout) and compare it with the expected output to automate these tests. But for the purpose of this question, we're considering manual observation of the output.

In conclusion, while designing tests for white-box testing, one needs to take into account all possible inputs, branches, paths, and outcomes of the function or module being tested. It's about understanding the internals of the system and ensuring they work as expected under all conditions.

Test Implementation and Execution

To run the test code, save the test cases provided earlier in a Python file (e.g., test_weather_app.py). To execute the tests, run the following command in the terminal:python -m unittest test_weather_app.py This command will run all the test cases in the test_weather_app.py file and display the results. if any test fails or produces an error, the output will show the details of the test case that failed or raised an error, allowing you to identify and fix the issue in your code. If you make changes to your code based on the test results, rerun the tests to ensure that the issue has been resolved and no new issues have been introduced.

In this table, "EP" stands for Equivalence Partitioning, "BVA" stands for Boundary Value Analysis, "BB" stands for Black-Box, and "WB" stands for White-Box. The "done" value in each cell indicates that the corresponding test design and test code have been implemented and run successfully.

Here's an example table with the modules and their respective test cases:

Module name	BB test design(EP)	BB test design(BVA)	WB test design	EP test code (implemented/run)	BVA test code (implemented/run)	White-Box testing (implemented/run)
find_season	done	done	done	done	done	done
temperature_comparison	done	done	done	done	done	done
display_season	N/A	N/A	done	N/A	N/A	done
display_comparison_result	N/A	N/A	done	N/A	N/A	done

7. Ethics and Professionalism [15 Marks]

- Lack of ethics and professionalism can result in harmful effects using the code designed and implemented in this weather application. Here are some potential issues based on the seven points from lecture 9:

- Public Interest and Welfare: Inaccurate or outdated data on average temperatures and seasons might mislead the users, causing them to make wrong decisions based on the provided information. This can negatively impact public welfare.

- Privacy and Confidentiality: Although the current code does not involve personal data, if the application were to be expanded and linked with user accounts or location-based services, mishandling of user data or location information could lead to privacy breaches and unauthorized access to sensitive information.

- Intellectual Property and Copyright: If the code were to use copyrighted or proprietary data (e.g., temperature data from a paid service) without proper permission, it would lead to legal and ethical violations.

- Responsibility and Accountability: The developers should take responsibility for maintaining accurate and up-to-date information within the application. Negligence or lack of accountability could lead to misinformation or confusion for users.

- Honesty and Transparency: If the developers knowingly implement inaccurate or misleading data in the application, they would be violating ethical guidelines by misleading users and not being transparent about the quality of the data.

- Fairness and Non-discrimination: The code should be designed in a way that ensures equal access to the application and its features for all users. Discriminating against users based on their location, nationality, or other factors would be unethical.

- Environmental Impact: Although the current code is not directly linked to environmental concerns, a potential expansion of the software to cover more countries and regions might increase its environmental impact, such as through increased energy consumption for data processing. Developers should consider the environmental impact of the application and strive to minimize it.

- Using ACS or IEEE-CS Ethical guidelines discussed in lecture 9, here are two suggestions to avoid ethical and professional issues in the software proposed in this assignment:

- Adhere to professional competence and continuous improvement (ACS Code of Ethics - 2.2, IEEE-CS 3): Developers should stay updated with the latest weather data, scientific research, and industry best practices to ensure that the application provides accurate and reliable information. They should also be open to feedback from users and peers and continually improve the application's functionality and accuracy.

- Ensure privacy and security (ACS Code of Ethics - 1.5, IEEE-CS 7): In case the application is expanded to include personal data or location-based services, developers should implement proper data handling practices and security measures to protect user information. They should also be transparent with users about the type of data collected, how it is used, and the steps taken to protect their privacy.
- Respect Intellectual Property (ACS Code of Ethics - 1.3, IEEE-CS 6): It's crucial to respect copyrights, patents, and other forms of intellectual property when sourcing data or using software libraries. Always ensure that licenses are correctly adhered to, and all used resources are appropriately acknowledged. If copyrighted or proprietary data is used, proper permission should be obtained.

- Fairness and Non-Discrimination (ACS Code of Ethics - 1.1, IEEE-CS 1, 9): Ensure the application is accessible to everyone regardless of their location or nationality. Avoid discriminatory practices in any form. For instance, if the application were to be expanded to cover more countries and regions, it should not favor one region over another.

- Environmental Sustainability (ACS Code of Ethics - 1.6, IEEE-CS 1): Developers should consider the environmental footprint of the application. While the current code may not directly impact the environment, it's essential to develop efficient algorithms and design systems for scalability to ensure the software uses resources efficiently if expanded to cover more countries and regions. Also, promoting environmentally friendly behaviors among users can be beneficial.

- Honesty and Transparency (ACS Code of Ethics - 2.4, IEEE-CS 3): Developers should always be honest about the application's capabilities, limitations, and data sources. If there are any inaccuracies in the data or the application's features, they should be clearly communicated to the users. Any updates or changes to the application should also be transparently shared with the users.

- Accountability (ACS Code of Ethics - 2.6, IEEE-CS 5): Developers must take responsibility for the application's accuracy and reliability. They should be prepared to correct any errors or inaccuracies in the data and to address any issues or concerns that users may have. They should also be accountable for ensuring that the application is maintained and updated regularly.

By adhering to these ethical guidelines and maintaining a high level of professionalism, developers can ensure that their software serves the public interest and maintains the trust and confidence of its users.

SE.docx

ORIGINALITY REPORT

5%	3%	0%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Curtin University of Technology Student Paper	2%
2	blog.risingstack.com Internet Source	1%
3	swproje.weebly.com Internet Source	<1%
4	www.coursehero.com Internet Source	<1%
5	Submitted to University of Brighton Student Paper	<1%
6	www.slideshare.net Internet Source	<1%
7	userjobs.in Internet Source	<1%
8	cshprotocols.cshlp.org Internet Source	<1%
9	Man Zhang, Asma Belhadi, Andrea Arcuri. "JavaScript SBST Heuristics To Enable Effective Fuzzing of NodeJS Web APIs", ACM	<1%

Transactions on Software Engineering and Methodology, 2023

Publication

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		

SE.docx

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20