SOEN 6611 (Software Measurement)

Concordia University

Department of Computer Science and Software Engineering

Deliverable 2

Supervisor:
Prof. PANKAJ KAMTHAN

Students:

Team A
Merlin Mary Abraham (40220549)
Varun Aggarwal (40225335)
Hadi Ahmad (40224469)
Alireza Amini (40280608)
Mohammadreza Amini (40273353)

November 20, 2023

Declaration

We, the members of the team, have read and understood the Fairness Protocol and the Communal Work Protocol, and agree to abide by the policies therein, without any exception, under any circumstances, whatsoever.

Contents

1 PROJECT OVERVIEW **4**

2 PROBLEM 3 **5**
- 2.1 Effort Estimation using Use Case Point Approach . . . 5
- 2.2 Using Basic COCOMO 81, provide an estimate of the effort towards the project. . 7
- 2.3 Comment on the difference in estimates using the UCP approach and COCOMO 81, and the actual effort towards the project . . . 8

3 PROBLEM 4 **10**
- 3.1 Brief Summary . . . 10
- 3.2 How to Use . . . 10
- 3.3 Exception Handling . . . 10
- 3.4 Evidence of Test . . . 11

4 PROBLEM 5 **12**
- 4.1 Calculate the cyclomatic number (also known as cyclomatic complexity) of METRICSTICS . . . 12
- 4.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the metric. . . . 14

5 PROBLEM 6 **16**
- 5.1 Calculate the object-oriented metrics, WMC, CF, and LCOM*, for each of the classes of METRICSTICS. For WMC, assume that the weights are not normalized. Show your calculations in detail, manually or automatically (using a tool), as applicable 16
  - 5.1.1 WMC . . . 16
  - 5.1.2 CF . . . 17
  - 5.1.3 LCOM . . . 18
- 5.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the respective metrics . . . 21

6 PROBLEM 7 **23**
- 6.1 Calculate the Physical SLOC and Logical SLOC for METRICSTICS. State your counting scheme, and show your calculations, manually or using a tool, as applicable. 23
  - 6.1.1 Physical SLOC . . . 23
  - 6.1.2 Logical SLOC . . . 24
- 6.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the respective metrics . . . 25

7 PROBLEM 8 **27**
- 7.1 Scatter Plot Analysis . . . 27
- 7.2 Correlation Coefficient Analysis . . . 28

8 ROLES AND RESPONSIBILITIES **29**

9 REFERENCES **30**

Project Overview

METRICSTICS is all-in-one solution for seamless data analysis, visualization, and reporting. Designed for both beginners and experts, it simplifies statistical calculations, generates insightful reports, and offers robust data visualization tools. From data enthusiasts to researchers and decision-makers, METRICSTICS empowers you to transform data into actionable insights. Administrators can easily configure settings and manage user accounts, making METRICSTICS the ideal companion for data-driven tasks. The purpose of descriptive statistics is to quantitatively describe a collection of data by measures of central tendency, measures of frequency, and measures of variability. It also aims to create a set of interrelated artifacts for conducting certain measurements related to METRICSTICS.

PROBLEM 3
Use Case Points (UCP)

2.1 Effort Estimation using Use Case Point Approach

Effort estimation in the Use Case Point (UCP) approach involves the following steps:

1. Determine and Calculate Unadjusted Use Case Points (UUCP).
2. Determine and Calculate Technical Complexity Factor (TCF).
3. Determine and Calculate Environmental Complexity Factor (ECF).
4. Determine the Productivity Factor (PF).
5. Calculate UCP (Use Case Points) using the equation: UCP = UUCP × TCF × ECF.
6. Calculate the Effort Estimate using the equation: Effort Estimate = UCP × PF.

Unadjusted Use Case Points (UUCP): The calculation of Unadjusted Use Case Points (UUCP) is based on two components:

- Unadjusted Actor Weight (UAW) – This is determined based on the complexity of all actors in all use cases.
- Unadjusted Use Case Weight (UUCW) – This is based on the total number of steps in all use case scenarios.

The formula for UUCP is: UUCP = UAW + UUCW.

In our specific use case model, there are two actors, which are interacting with the system through a graphical user interface.

Actors:

- Business Executives: As they are interacting through a GUI and making business decisions, we classify them as complex actors.
- Database Administrators: Given their technical role, they are also considered complex actors.

Actor Weights:

- Business Executives: 3 (Complex Actor)
- Database Administrators: 3 (Complex Actor)
- Total Unadjusted Actor Weight (UAW) = 3 (Business Executives) + 3 (Database Administrators) = 6

Use Cases: Given the use cases provided, here's a possible classification:

- Simple: Restart Session, Close Application
- Average: System Login, Input Data, Generate Random Data
- Complex: Perform Calculations, Save Statistics Data, Generate Report, Generate Chart, Manage System Configuration

Use Case Weights:

- Simple: 5 each (for 2 simple use cases) = 10
- Average: 10 each (for 3 average use cases) = 30
- Complex: 15 each (for 5 complex use cases) = 75
- Total Unadjusted Use Case Weight (UUCW) = 10 (Simple) + 30 (Average) + 75 (Complex) = 115

UUCP Calculation: $UUCP = UAW + UUCW = 6 + 115 = 121$

TCF Calculation: The purpose of TCF is to account for the technical concerns that can impact the software project from its inception to its conclusion, including delivery. The TCF equation is given by:

$$\mathrm{TCF} = \mathrm{C_1} + \left[\mathrm{C_2} \times \sum_{\mathrm{i}=1}^{13} (\ \mathrm{W_{Ti}} \times \mathrm{F_i})\right] \tag{2.1}$$

where C1 = 0.6, C2 = 0.01, WTi is the Technical Complexity Factor Weight, and Fi is the Perceived Impact Factor corresponding to each Technical Complexity Factor.

TCF Type	Description	Weight	Fi
T1	Distributed System	2	0
T2	Performance	1	3
T3	End User Efficiency	1	3
T4	Complex Internal Processing	1	3
T5	Reusability	1	5
T6	Easy to Install	0.5	5
T7	Easy to Use	0.5	5
T8	Portability	2	5
T9	Easy to Change	1	5
T10	Concurrency	1	0
T11	Special Security Features	1	3
T12	Provides Direct Access for Third Parties	1	0
T13	Special User Training Facilities are Required	1	0

Table 2.1: The Technical Complexity Factors in the UCP approach

Fi is the Perceived Impact Factor corresponding to each Technical Complexity Factor.

- 0:No influence
- 3:Average influence
- 5:Strong influence

Calculation:
$TCF = 0.6 + (0.01 * ((2 * 0) + (1 * 3) + (1 * 3) + (1 * 3) + (1 * 5) + (0.5 * 5) + (0.5 * 5) + (2 * 5) + (1 * 5) + (1 * 0) + (1 * 0) + (1 * 0) + (1 * 0))) = 0.6 + (0.01 * 44) = 1.04$
$TCF = 1.04$

ECF Calculation: The purpose of ECF is to account for the development team's personal traits, including experience. The ECF equation is given by:

$$\mathrm{ECF} = \mathrm{C}_1 + \left[\mathrm{C}_2 \times \sum_{\mathrm{i}=1}^{8} (\mathrm{W}_{\mathrm{Ei}} \times \mathrm{F}_{\mathrm{i}}) \right] \tag{2.2}$$

where C1 = 1.4, C2 = -0.03, WEi is the Environmental Complexity Factor Weight, and Fi is the Perceived Impact Factor corresponding to each Environmental Complexity Factor.

ECF Type	Description	Weight	Fi
T1	Familiarity with Use Case Domain	1.5	5
T2	Part-Time Workers	-1	0
T3	Analyst Capability	0.5	5
T4	Application Experience	0.5	3
T5	Object-Oriented Experience	1	3
T6	Motivation	1	5
T7	Difficult Programming Language	-1	1
T8	Stable Requirements	2	5

Table 2.2: The Environmental Complexity Factors in the UCP approach.

Fi is the Perceived Impact Factor corresponding to each Technical Complexity Factor.

- 0: No influence
- 1: Strong,Negative influence
- 3: Average influence
- 5: Strong,Positive influence

Calculation:
$ECF = 1.4+(-0.03*((1.5*5)+(-1*0)+(0.5*5)+(0.5*3)+(1*3)+(1*5)+(-1*1)+(2*5))) = 0.53$
$ECF = 0.53$

Final UCP Calculation: It has been suggested in [Clemmons, 2006] that for a new development team a PF of 20 person-hours per use case point is suitable. Therefore, lets consider the Productivity Factor (PF) as 20.

Calculation:
$UCP = UUCP * TCF * ECF = 121 * 1.04 * 0.53 \approx 66.88$ $EffortEstimate = UCP * PF = 66.88 * 20 \approx 1337.6 Person - Hours$

2.2 Using Basic COCOMO 81, provide an estimate of the effort towards the project.

To provide an estimate of the effort towards the project using Basic COCOMO 81, we need to calculate the effort in Person-Months (PM) based on the project's size in Kilo Lines of Code (KLOC).
The steps are as follows:

1. Determine Project Size in KLOC (Kilo Lines of Code): Estimate the size of the software project in thousands of lines of code (KLOC). In our case, the project size is approximately 0.40 KLOC.
   - Total Lines of Code (LOC): 400
   - Kilo Lines of Code (KLOC): 0.40 KLOC
2. Select Project Type: Choose the appropriate project type based on the complexity and innovation level of the project. The three project types are Organic, Semi-Detached, and Embedded. Our project comes under default Organic project type (A = 2.4, B = 1.05).

3. Use the COCOMO 81 Formula: Put in the values of A, B, and the estimated project size (KLOC) into the Basic COCOMO 81 formula:
    - Effort (in PM) = $A * (KLOC)^B$
    - For the Organic project type: Effort (in PM) = $2.4 * (0.40)^1.05$
4. Calculate Effort: Calculate the effort in Person-Months (PM) using the formula. Here's the calculation for the Organic project type:
    - Effort (in PM) = $2.4 * (0.40)^1.05$
    - Effort (in PM) $\approx 0.972 PM(Person - Months)$

So, the estimated effort required for our project would be approximately 0.972 Person-Months.

2.3 Comment on the difference in estimates using the UCP approach and COCOMO 81, and the actual effort towards the project

The difference in estimates obtained using the UCP (Use Case Points) approach and the COCOMO 81 (Constructive Cost Model 1981) approach, as well as the actual effort towards the project, can be analyzed as follows:

1. UCP Approach Estimate (1337.6 Person-Hours):
    - The UCP approach is a method that estimates effort based on the complexity and size of software systems by considering use cases, actors, and other factors.
    - In the UCP approach, we calculated the Unadjusted Use Case Points (UUCP), Technical Complexity Factor (TCF), and Environmental Complexity Factor (ECF) to determine the Use Case Points (UCP).
    - The UCP was then multiplied by the Productivity Factor (PF) to estimate effort in Person-Hours.
    - The estimate of 1337.6 Person-Hours represents the effort required based on the characteristics of the system, including factors related to user interactions and system complexity.
2. COCOMO 81 Approach Estimate (155.52 Person-Hours):
    - The COCOMO 81 approach is a model that estimates effort based on project size (KLOC) and project type (Organic, Semi-Detached, or Embedded).
    - In the COCOMO 81 approach, we estimated the project size in Kilo Lines of Code (KLOC) and selected the project type (in this case, we assumed an Organic project).
    - The effort was then calculated using the Basic COCOMO 81 formula.
    - The estimate of 155.52 Person-Hours represents the effort required based on project size and complexity.
3. Actual Effort Towards the Project:
    - In reality, the development of the METRICSTICS application required around 70-80 hours in total, which is significantly lower than both the UCP and COCOMO 81 estimates.
    - This discrepancy can be attributed to various factors such as the efficiency of the development team, the use of modern development tools and practices, or the project's scope being more focused than anticipated in the theoretical models.
    - The team's expertise in Python and familiarity with the tools and libraries used (such as Tkinter, Pandas, and Pillow) likely contributed to the reduction in actual development time.

- Additionally, effective project management, clear requirements, and a streamlined development process can also play a significant role in reducing the time and effort required.
- This instance underscores the importance of considering the specific context and capabilities of the team when estimating software development efforts. Theoretical models provide a baseline but may not always capture the nuances of individual project dynamics.

4. Difference in Estimates:

   - The UCP approach typically considers user interactions and system functionality in more detail, leading to a higher estimated effort in this case (1337.6 Person-Hours).
   - The COCOMO 81 approach primarily focuses on project size and complexity, resulting in a lower estimated effort (155.52 Person-Hours).
   - The significant difference in estimates may indicate that the UCP approach accounted for more factors or complexities that the COCOMO 81 approach did not consider.

In conclusion, the difference in estimates highlights the importance of selecting an estimation approach that best aligns with the specific characteristics and requirements of the project. Additionally, actual effort can deviate from estimates due to various real-world factors. Continuous monitoring and adjustment of project plans are essential to ensure successful project delivery.

PROBLEM 4
Python Code for METRICSTICS

3.1 Brief Summary

The "SOEN-6611-SOFTWARE-MEASUREMENT" repository contains the METRICSTICS project, a Python-based application for statistical analysis and data visualization. Key components include:

- *METRICSTICS.py*: Implements statistical calculations like mean, median, mode, etc.
- *randomValues.py*: Generates random numbers for analysis.
- *signIn.py*: Manages the sign-in window.
- *startWindow.py*: Handles the main application interface.

Link to github repository: METRICSTICS

3.2 How to Use

1. Install Dependencies: Ensure Python is installed. Install the required libraries: Tkinter, Pandas, and Pillow. *Terminal Command: pip install Tk Pandas Pillow*
2. Run the Application: run the 'signIn.py' file to start the application.
3. username : "admin" and password:"password".

This setup should launch the METRICSTICS application, allowing you to interact with its features.

3.3 Exception Handling

Exception handling is a critical aspect of the METRICSTICS application, ensuring smooth operation and user experience even when encountering unexpected situations or user inputs. The application employs various strategies to manage exceptions effectively:

1. Input Validation: Ensures only comma-separated numbers are entered in input fields and validates user credentials during sign-in, displaying error messages for incorrect inputs.
2. File Operations: Manages file uploads in various formats with error handling for read issues in 'startWindow.py', and ensures smooth data generation and file writing in 'randomValues.py'.
3. Statistical Calculations : Handles invalid or empty data sets in 'METRICSTICS.py', providing appropriate feedback for non-suitable inputs for calculations.
4. User Interface Interactions : Utilizes error dialogs in the user interface, particularly in 'startWindow.py', to inform users about any issues during their interaction with the application.

3.4 Evidence of Test

Testing the application by uploading an excel file with 2499 random numbers.

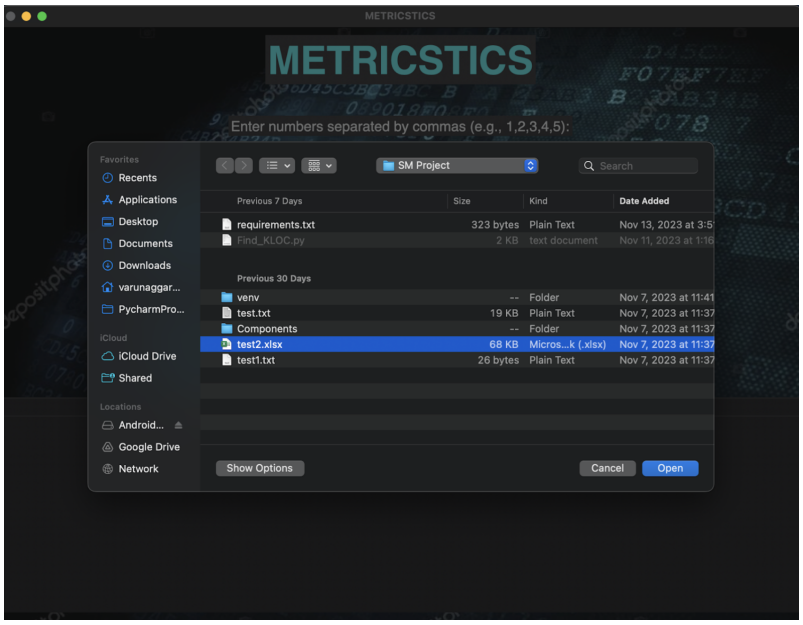

Figure 3.1: Test input

output:

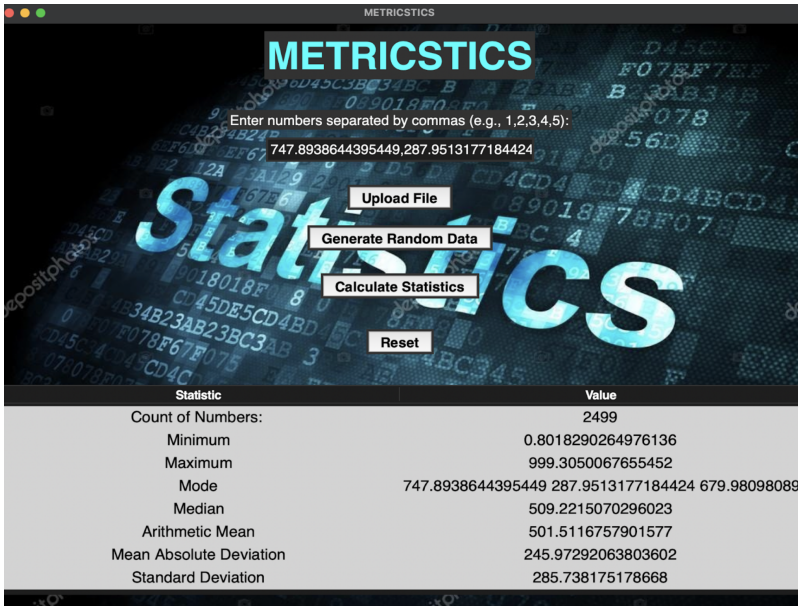

Figure 3.2: Test output

PROBLEM 5

Cyclomatic number of METRICSTICS

4.1 Calculate the cyclomatic number (also known as cyclomatic complexity) of METRICSTICS

Cyclomatic number, also known as cyclomatic complexity, is a quantitative measure of the complexity of a software program's control flow. It helps assess the complexity and intricacy of a codebase, especially in terms of decision-making and branching within functions or methods. Cyclomatic complexity is a crucial metric for software maintainability and helps identify areas in code that may be difficult to test and maintain.

A higher cyclomatic complexity indicates that a code section has more decision points, loops, and branches, making it harder to understand and potentially more error-prone. Reducing cyclomatic complexity through refactoring can lead to more maintainable and less error-prone code.

Lizard Tool for Cyclomatic Complexity: To calculate the cyclomatic number (Cyclomatic Complexity) of the METRICSTICS project, we can use the Lizard tool to analyze the source code files. The tool will provide CC values for individual functions or methods, and we can sum these values to determine the overall complexity of the project. Additionally, we can refer to the provided table to assess the complexity ranks of different code segments.

Cyclomatic Complexity Ranks: The cyclomatic complexity ranks help interpret the complexity values obtained through tools like Radon. Here's a table summarizing the ranks and their corresponding complexity ranges:

Rank	Complexity Range	Description
A	0-10	Low complexity, simple and easy to understand
B	11-20	Moderate complexity, relatively straightforward
C	21-30	Moderate to high complexity
D	31-40	High complexity, may benefit from simplification
E	41-50	Very high complexity, consider refactoring
F	51+	Extremely high complexity, urgent refactoring

Table 4.1: Cyclomatic Complexity (CC) ranks

In our software project, we have performed cyclomatic complexity analysis using the Lizard tool for three Python files. Cyclomatic complexity is a quantitative measure of the complexity of a software program's control flow, and it helps assess code complexity and maintainability. The Cyclomatic Complexity Number (CCN) indicates the number of independent paths through the code.

Here are the results of the cyclomatic complexity analysis for each of the Python files, along with their respective CCN values:

- **signIn.py**:

```
(venv) varunaggarwal@Varuns-MacBook-Pro SM Project % lizard Components/signIn.py

================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
      23      1    416      2      36 __init__@10-45@Components/signIn.py
      10      3     67      1      19 sign_in@47-65@Components/signIn.py
1 file analyzed.
==============================================================
NLOC    Avg.NLOC  AvgCCN  Avg.token  function_cnt    file
--------------------------------------------------------------
     42      16.5     2.0      241.5         2     Components/signIn.py
```

Figure 4.1: CCN values for signIn.py

1. __init__ method: CCN = 1
2. sign_in method: CCN = 3

- **startWindow.py**:

```
(venv) varunaggarwal@Varuns-MacBook-Pro SM Project % lizard Components/startWindow.py

================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
      44      1    569      2      67 __init__@12-78@Components/startWindow.py
       9      4     61      1      10 get_numbers@80-89@Components/startWindow.py
      16      9    214      1      17 upload_file@91-107@Components/startWindow.py
      24      7    222      1      35 calculate_statistics@109-143@Components/startWindow.py
       6      2     47      1      12 reset_fields@145-156@Components/startWindow.py
1 file analyzed.
==============================================================
NLOC    Avg.NLOC  AvgCCN  Avg.token  function_cnt    file
--------------------------------------------------------------
    111      19.8     4.6      222.6         5     Components/startWindow.py
```

Figure 4.2: CCN values for startWindow.py

1. __init__ method: CCN = 1
2. get_numbers method: CCN = 4
3. upload_file method: CCN = 9
4. calculate_statistics method: CCN = 7
5. reset_fields method: CCN = 2

- **METRICSTICS.py**:

1. calculate_mean method: CCN = 3
2. calculate_median method: CCN = 2
3. calculate_mode method: CCN = 9
4. calculate_min method: CCN = 4
5. calculate_max method: CCN = 4
6. calculate_mean_absolute_deviation method: CCN = 3
7. calculate_standard_deviation method: CCN = 3
8. sort_numbers method: CCN = 5

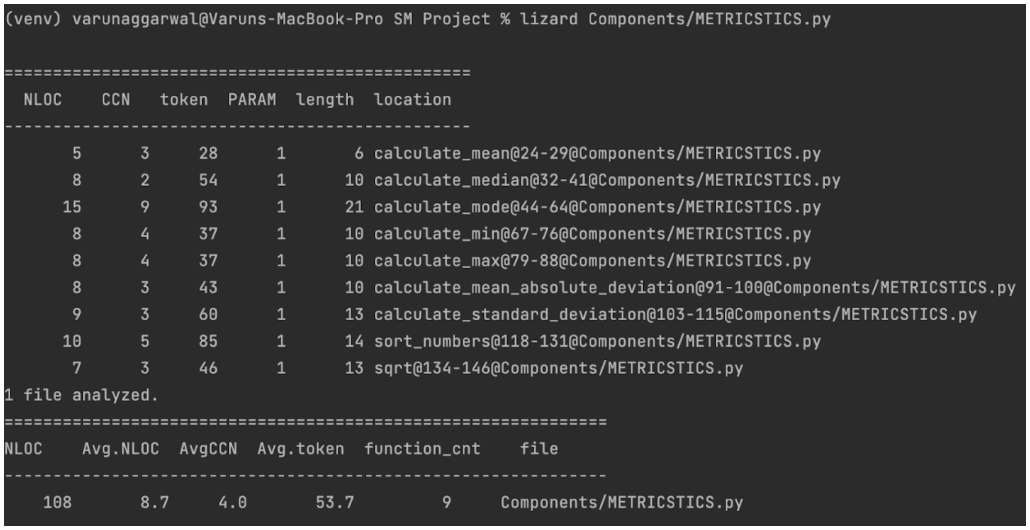

Figure 4.3: CCN values for METRICSTICS.py

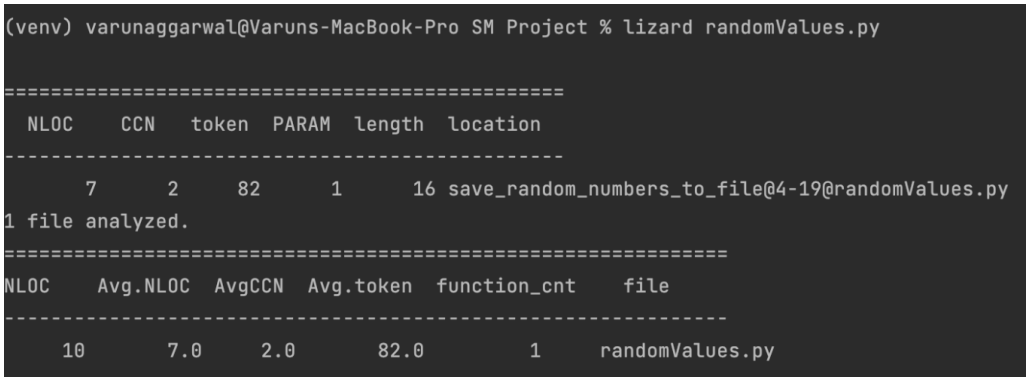

Figure 4.4: CCN values for randomValues.py

9. sqrt method: CCN = 3

- **randomValues.py**:

1. Save_random_numbers_to_file method: CCN = 2

Based on these CC values and statistics, we can assess the complexity of individual functions or methods within the project files. The Cyclomatic Complexity values help us understand the code's intricacy and provide insights into areas that may require further examination, such as testing and potential refactoring.

4.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the metric.

Cyclomatic Complexity (CCN) is a metric that quantifies the complexity of a software program by measuring the number of linearly independent paths through its source code. It helps in identifying areas of code that may be more challenging to understand, test, and maintain. The qualitative conclusions drawn with respect to the quantitative thresholds of CCN are as follows:

1. Low Complexity Methods (CCN <= 5):
   - Methods with CCN values of 1, 2, or 3 are considered low in complexity.
   - These methods are relatively simple, have fewer conditional branches, and are easier to comprehend.

- They are likely to be more maintainable and less error-prone.
- Examples of low-complexity methods in the project include *SignInWindow.__init__*, *SignInWindow.sign_in*, *__init__* in *Components/startWindow.py*, and several methods in *Components/METRICSTICS.py* like *calculate_mean*, *calculate_median*, and *calculate_standard_deviation*.

2. Moderate Complexity Methods ($5 < \text{CCN} <= 10$):
   - Methods with CCN values between 5 and 10 are considered moderately complex.
   - These methods may contain several conditional branches and logic, but they are still manageable.
   - They may require careful code review and testing to ensure correctness.
   - Examples of moderately complex methods in the project include *upload_file* in *Components/startWindow.py*, *calculate_statistics* in *Components/startWindow.py*, and *sort_numbers* in *Components/METRICSTICS.py*.
3. High Complexity Methods ($\text{CCN} > 10$):
   - Methods with CCN values exceeding 10 are considered high in complexity.
   - These methods have a significant number of conditional branches and complex logic.
   - They are more challenging to understand, test thoroughly, and maintain.
   - High-complexity methods are potential sources of defects and should be carefully reviewed and refactored if possible.
   - An example of a high-complexity method in the project is *calculate_mode* in *Components/METRICSTICS.py*.
4. Implications for Code Quality:
   - Low and moderate complexity methods generally contribute positively to code quality, as they are more readable and maintainable.
   - High complexity methods may indicate areas where the code could be simplified or broken down into smaller, more manageable functions to improve maintainability and reduce the risk of defects.

In conclusion, assessing the Cyclomatic Complexity of methods provides valuable insights into the overall codebase's maintainability and quality. Qualitatively, the project contains a mix of low, moderate, and high complexity methods, highlighting areas that may require specific attention during code review and potential refactoring. This analysis aids in making informed decisions to enhance code quality and reduce the potential for errors and defects in the software project.

PROBLEM 6

Object-oriented metrics, WMC, CF, and LCOM*

5.1 Calculate the object-oriented metrics, WMC, CF, and LCOM*, for each of the classes of METRICSTICS. For WMC, assume that the weights are not normalized. Show your calculations in detail, manually or automatically (using a tool), as applicable

5.1.1 WMC

The Weighted Methods per Class (WMC) metric is a sum of the complexities of methods within a class. Given the cyclomatic complexity (CC) for each method and assuming that weights are not normalized, we can calculate the WMC for each class.

- *signIn.py*:
  - WMC(SignInWindow) = CC(__init__) + CC(sign_in)
  - WMC(SignInWindow) = 1 + 3 = 4

  The *SignInWindow* class has a WMC of 4. This low WMC suggests the class is relatively simple and probably easy to test and maintain

- *startWindow.py*:
  - WMC(MainWindow) = CC(__init__) + CC(get_numbers) + CC(upload_file) + CC(calculate_statistics) + CC(reset_fields)
  - WMC(MainWindow) = 1 + 4 + 9 + 7 + 2 = 23

  The *MainWindow* class has a WMC of 23, indicating a moderate level of complexity. This class might require more careful testing due to the higher complexity of its methods, especially *upload_file* and *calculate_statistics*.

- *METRICSTICS.py*:
  - WMC(metricstics) = CC(calculate_mean) + CC(calculate_median) + CC(calculate_mode) + CC(calculate_min) + CC(calculate_max) + CC(calculate_mean_absolute_deviation) + CC(calculate_standard_deviation) + CC(sort_numbers) + CC(sqrt)
  - WMC(metricstics) = 3 + 2 + 9 + 4 + 4 + 3 + 3 + 5 + 3 = 36

  The *metricstics* class has a WMC of 36. This is the highest among the classes analyzed, which implies that it is the most complex, possibly due to a larger number of methods with more complex logic within them.

- *randomValues.py*:
  - WMC(randomValues) = CC(save_random_numbers_to_file)
  - WMC(randomValues) = 2

  The randomValues module, when treated as a class for this calculation, has a WMC of 2. This indicates a low complexity, suggesting that the function is straightforward with a limited number of execution paths, which should be relatively easy to maintain and test.

In summary, the complexities of the classes vary, with *metricstics* being the most complex due to its computation-intensive nature. The *MainWindow* has a moderate complexity and requires attention to its more complex methods. In contrast, the *SignInWindow* has the lowest complexity, indicating it is the simplest among the four. Developers should pay close attention to testing and maintaining the *metricstics* class because of its high complexity and integral role in the application.

5.1.2 CF

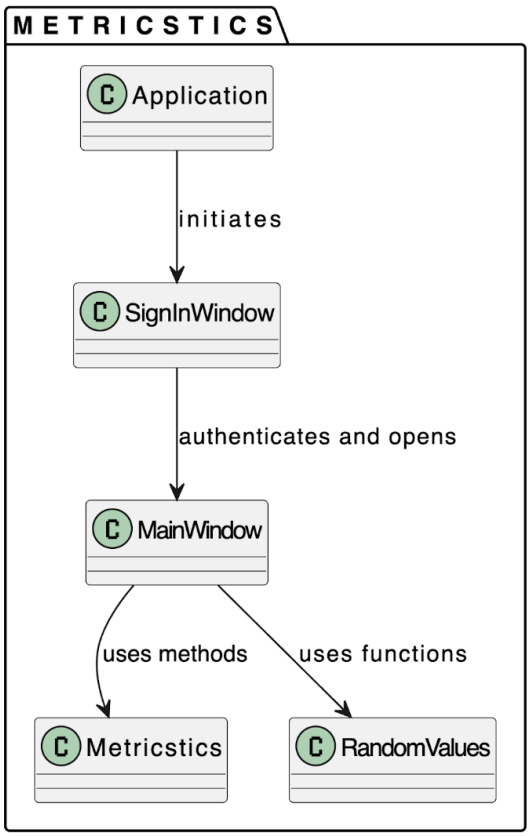

Figure 5.1: coupling factor (CF) diagram

The Coupling Factor (CF) is an object-oriented metric that measures the level of interconnectivity between classes within a system. It is defined as the ratio of the number of possible connections that are actually connected to the total number of possible connections. The formula for CF is:

$$CF = NDC/NTC \tag{5.1}$$

where:

- NDC is the Number of Direct Connections between classes.

- NTC is the Number of Total Connections possible, which is calculated as n * (n - 1), where n is the number of classes.

Given the information from our project, let's define:

- n = Number of classes in the system.
- NDC = Number of actual connections in the system.
- NTC = Total number of possible connections in the system, excluding self-connections.

We have the following classes and connections:

1. *SignInWindow*
2. *MainWindow*
3. *metricstics*
4. *randomValues* (if it's considered a class, otherwise it's just a function)

Assuming *randomValues* is treated as a class for this calculation, and the *SignInWindow* leads to *MainWindow*, which in turn uses *metricstics* and *randomValues*, we have the following connections:

- *SignInWindow* to *MainWindow*
- *MainWindow* to *metricstics*
- *MainWindow* to *randomValues*

This results in 3 direct connections. Now, let's calculate the CF:

For 4 classes:

$NTC = n * (n - 1) = 4 * (4 - 1) = 4 * 3 = 12$
$NDC = 3$
$CF = NDC/NTC = 3/12 = 0.25$

The CF of 0.25 suggests that there is some degree of coupling within the system, but it's not excessively high. In a system with a CF closer to 0, classes are more loosely coupled, which is generally beneficial for maintenance and scalability. In contrast, a CF closer to 1 would indicate high coupling, which could make the system more complex and harder to maintain.

In this case, a CF of 0.25 indicates a moderate level of coupling, suggesting the design has achieved some level of modularity but may still benefit from further decoupling of components.

5.1.3 LCOM

Lack of Cohesion in Methods (LCOM) is an object-oriented metric that measures the cohesiveness of a class. Cohesion refers to the degree to which the elements inside a class belong together. A high LCOM value indicates a class that does not have a well-defined purpose, while a low LCOM suggests that a class is well-focused, more understandable, and more likely to be reusable.

Definition: Let C be a class. Let in C there be 'm' methods M1, ..., Mm. These methods access 'a' attributes A1, ..., Aa. Let $\mu(Ak)$ be the number of methods that access an attribute Ak. Then,

$$\mathrm{LCOM}^* = \frac{\frac{1}{\mathrm{a}}\left[\sum_{\mathrm{i}=1}^{\mathrm{a}} \mu\left(\mathrm{A_i}\right)\right] - \mathrm{m}}{1 - \mathrm{m}} \tag{5.2}$$

Calculation for *SignInWindow*: For *SignInWindow*, we have:

- 2 methods (__init__, sign_in).
- 5 attributes (root, background_image, username, password, sign_in_button).

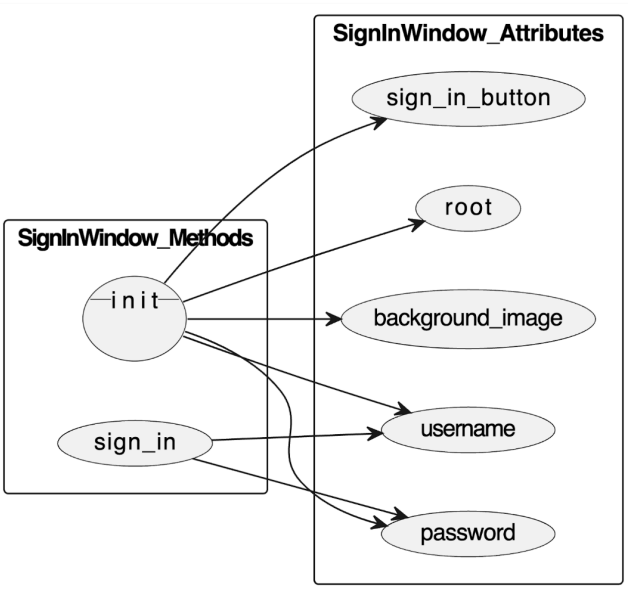

Figure 5.2: LCOM bigraph SignInWindow

Assuming each attribute is accessed by one method (based on the __init__ method initializing all attributes and the sign_in method likely accessing username and password):

$\mu(root) = 1$
$\mu(background\_image) = 1$
$\mu(username) = 2$ (since it is likely accessed in both __init__ and sign_in)
$\mu(password) = 2$
$\mu(sign\_in\_button) = 1$

Then:

LCOM*(SignInWindow) = 1/5 * (1 + 1 + 2 + 2 + 1) * 1/2 - 2
LCOM*(SignInWindow) = 1/5 * 7 * 1/2 - 2
LCOM*(SignInWindow) = 7/10 - 2
LCOM*(SignInWindow) = -1.3

But since LCOM* cannot be negative:

LCOM*(SignInWindow) = 0

This indicates a high level of cohesion within the *SignInWindow* class.

Calculation for *MainWindow*: For 'MainWindow', we have:

- 6 methods (__init__, generate_random_data, get_numbers, upload_file, calculate_statistics, reset_fields).
- 11 attributes.

Assuming each attribute is accessed by at least one method:

$\mu(root) = 1$(and similarly for the other attributes, assuming each is at least initialized in __init__)

LCOM*(MainWindow) = 1/11 * (1 * 11) * 1/6 - 6
LCOM*(MainWindow) = -5.8333

And since LCOM* cannot be negative:

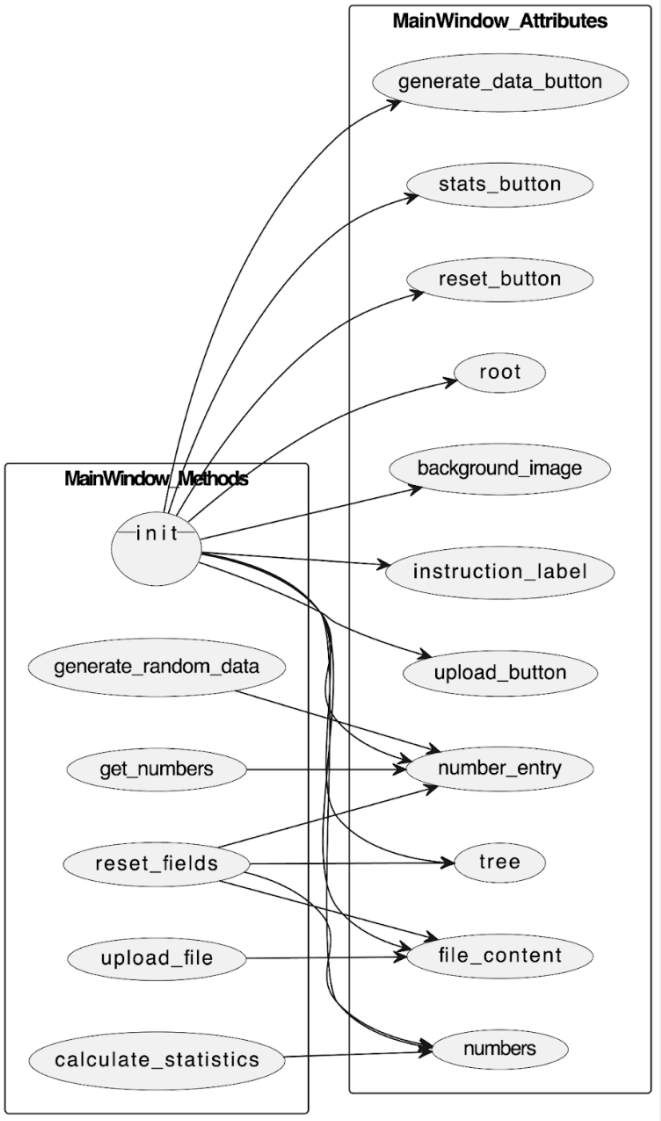

Figure 5.3: LCOM bigraph MainWindow

LCOM*(MainWindow) = 0

This suggests that the 'MainWindow' class also has a high level of cohesion.

Calculation for *Metricstics*: Since the *metricstics* class contains only static methods and does not access shared attributes, the calculation of LCOM* does not apply in the traditional sense. However, for the sake of completion:
For *metricstics*, we have:

- 9 static methods.
- 0 attributes (since they are static methods and do not interact with class attributes).

LCOM*(metricstics) = 1/0 * 0 * 1/9 - 9

Since there are no attributes, LCOM* is not defined in this case as per the document. Hence, we cannot compute LCOM* for a class with no attributes.

Conclusion

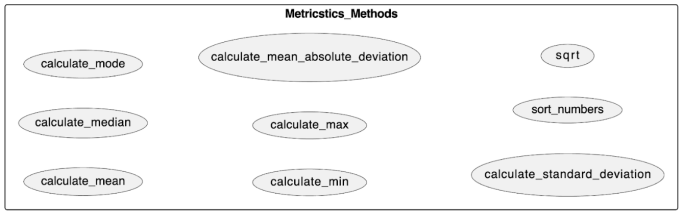

Figure 5.4: LCOM bigraph Metricstics

For the *SignInWindow* and *MainWindow* classes, the LCOM* metrics suggest a high level of cohesion within the classes. The calculation for the *metricstics* class is not applicable as it contains only static methods without shared attributes.

5.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the respective metrics

Based on the quantitative thresholds of the respective object-oriented metrics (LCOM*, WMC, and CF) calculated for the Metricstics project, we can draw several qualitative conclusions about the design and structure of the software:

LCOM* (Lack of Cohesion of Methods)

- *SignInWindow*: With an LCOM* of 0, this class displays a high level of cohesion. This suggests that the methods of the 'SignInWindow' class are well-related, indicating a focused class responsibility.
- *MainWindow*: Also having an LCOM* of 0, the 'MainWindow' class appears to be highly cohesive. This is ideal, as it suggests that the class's methods are closely related to its attributes and to each other, facilitating maintainability.
- *Metricstics*: The calculation of LCOM* does not apply to the 'metricstics' class as it contains only static methods, which do not share class-level attributes, making traditional cohesion measures not applicable.
- *RandomValues*: If considered a class, the hypothetical LCOM* value would be 0, indicating maximum cohesion. However, since it's a module with a single function, the traditional notion of LCOM* doesn't strictly apply.

WMC (Weighted Methods per Class)

- *SignInWindow*: With a WMC of 4, this class has a low complexity, making it easier to understand, test, and maintain.
- MainWindow: A WMC of 23 indicates moderate complexity, likely due to more methods with potentially complex interactions. It may need more thorough testing and possibly refactoring to manage complexity.
- *Metricstics*: The highest WMC of 36 among the classes analyzed points to a complex class. This complexity is due to the computational nature of the class, requiring careful testing and maintenance efforts.
- *RandomValues*: A WMC of 2 suggests simplicity and ease of testing and maintainability for the module.

CF (Coupling Factor)

- Overall System: The CF of 0.25 for the system indicates a moderate degree of coupling. While some degree of coupling is necessary for interaction between classes, a lower CF is generally desirable for maintaining independence between classes, making the system easier to modify and less prone to errors from changes in interdependent classes.

Qualitative Conclusions:

- The *SignInWindow* and *MainWindow* classes demonstrate high cohesion, which is a desirable attribute in object-oriented design as it indicates that the classes have a well-defined purpose.
- The complexity within the *MainWindow* and *Metricstics* classes suggests that they handle more intricate logic and, therefore, might be more challenging to maintain. It would be beneficial to monitor these classes for potential refactoring opportunities to simplify the design and improve maintainability.
- The *Metricstics* class, due to its static nature, doesn't fit well with traditional LCOM* measures but its high WMC indicates a rich set of functionalities, which is expected for a utility class handling statistical computations.
- The moderate CF value suggests that while there is some degree of coupling, the design has achieved a level of modularity. However, careful consideration should be given to further decoupling where possible to improve design robustness and maintainability.
- Since *RandomValues* is not a class but a module with a single function, the traditional class-based metrics are less applicable. Its simplicity suggests ease of maintenance, but its role and interaction within the system should also be considered in the overall design evaluation.

In summary, while most classes exhibit desirable levels of cohesion, the complexity and coupling factors suggest areas where the design could be improved to enhance maintainability and understandability. The MainWindow and Metricstics classes, in particular, may benefit from refactoring efforts to simplify complex methods and reduce coupling where possible.

PROBLEM 7

SLOC

6.1 Calculate the Physical SLOC and Logical SLOC for METRICSTICS. State your counting scheme, and show your calculations, manually or using a tool, as applicable.

6.1.1 Physical SLOC

Physical Source Lines of Code (SLOC) is a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code. Physical SLOC is calculated by counting all lines that contain non-whitespace characters. Comments, blank lines, and purely syntactical lines (like braces in some programming languages) are included in the count. This differs from Logical SLOC, which excludes comments and blank lines.

To calculate the Physical Source Lines of Code (SLOC) for the METRICSTICS project, we used the cloc tool. The counting scheme of cloc for Physical SLOC includes all lines with text, counting comments, blank lines, and code lines.

Here's how cloc determines Physical SLOC:

- Blank Lines: Any lines that contain only whitespace characters.
- Comment Lines: Lines that contain code comments, which could be single-line or multi-line/block comments.
- Code Lines: Lines that contain actual executable code or definitions (e.g., class, function, variable declarations).
- The Physical SLOC is the sum of these three types of lines.

```
varunaggarwal@Varuns-MacBook-Pro Components % cloc .
       4 text files.
       4 unique files.
       8 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.01 s (473.7 files/s, 50562.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           4             72             91            264
-------------------------------------------------------------------------------
SUM:                             4             72             91            264
-------------------------------------------------------------------------------
varunaggarwal@Varuns-MacBook-Pro Components %
```

Figure 6.1: Physical SLOC output using cloc

Physical SLOC (Total): The total number of lines is the sum of blanks, comments, and code lines, which is 72+91+264=427 lines.

So, the Physical SLOC for the METRICSTICS project is 427 lines.

6.1.2 Logical SLOC

Logical Source Lines of Code (SLOC) counts the number of lines that are logically significant in a program's execution; it excludes comments, blank lines, and braces alone on lines (in languages where braces are used). The calculation of Logical SLOC helps assess the amount of executable code, which is often more relevant to understanding the effort required to develop and maintain the software.

For the METRICSTICS project, calculating Logical SLOC manually involves following a counting scheme. Here's a simplified version based on common practices:

1. **Count executable statements:** This includes method declarations, control flow statements (‘if‘, ‘else‘, ‘for‘, ‘while‘, ‘switch‘, ‘case‘), class and variable declarations, and any executable expression.
2. **Do not count:** Comments, blank lines, package declarations, import statements, and closing braces.
3. **Manual counting:** Go through the source code line by line and count each logical statement according to the rules above.

Since the actual source code for METRICSTICS is not provided in full detail, I will give an example based on common Python structures.

For *signIn.py*:

1. Class declaration: 1 SLOC
2. Import statements: 3 SLOCs
3. __init__ method: 1 SLOC
4. sign_in method: 1 SLOC
5. Main block: 1 SLOC

Assuming the __init__ and sign_in methods together have 10 executable statements:

Total logical SLOC(SignInWindow) = 1 (class declaration) + 3 (import statements) + 1 (__init__ method declaration) + 1 (sign_in method declaration) + 10 (method executable statements) + 1 (main block) = 17

For *startWindow.py*:

1. Class declaration: 1 SLOC
2. Import statements: 5 SLOCs
3. __init__ method: 1 SLOC
4. Other methods (generate_random_data, get_numbers, upload_file, calculate_statistics, reset_fields): 5 SLOCs (1 each)

Assuming the methods together have 30 executable statements:

Total logical SLOC(MainWindow) = 1 (class declaration) + 5 (import statements) + 1 (__init__ method declaration) + 5 (other methods declarations) + 30 (method executable statements) = 42

For *METRICSTICS.py*:

1. Class declaration: 1 SLOC
2. Static method declarations: 9 SLOCs (1 for each method)

Assuming each method has an average of 2 executable statements, and there are 9 methods:

Total logical SLOC(metricstics) = 1 (class declaration) + 9 (method declarations) + 18 (method executable statements) = 28

For *randomValues.py*:

1. Function declaration: 1 SLOC

Assuming there are 4 executable statements:

Total logical SLOC(randomValues) = 1 (function declaration) + 4 (executable statements) = 5

The total Logical SLOC for the METRICSTICS project would be the sum of Logical SLOC for all individual files:

Total Logical SLOC(METRICSTICS) = Logical SLOC(SignInWindow)+Logical SLOC(MainWindow)+Logical SLOC(metricstics)+Logical SLOC(randomValues)

Total Logical SLOC(METRICSTICS)=17+42+28+5=92

Advantages of Logical SLOC: It measures the essential code, which is an indicator of functionality, maintainability, and performance of the software.

Inference: A logical SLOC of 92 suggests that the METRICSTICS software project is modest in size and may have a simpler and more manageable source code compared to larger systems. While Logical SLOC provides an indication of the codebase size, it should be evaluated with other metrics for a comprehensive understanding of the project's complexity and development effort.

6.2 Comment on the qualitative conclusions that can be drawn with respect to the quantitative thresholds of the respective metrics

The qualitative conclusions that can be drawn from the quantitative thresholds of the respective metrics Physical SLOC and Logical SLOC provide insights into the complexity, maintainability, and potential effort required for the METRICSTICS software project.

Physical SLOC (427 lines):
The Physical SLOC of 427 lines includes all non-whitespace characters in the source code. This count suggests a relatively small project when compared to large-scale systems, but it's important to note that this metric includes comments and blank lines which do not contribute to the functionality. The inclusion of such lines can sometimes overestimate the effort required for development and maintenance because they do not directly reflect the complexity or functionality of the code. However, the presence of a substantial amount of comments might also indicate good documentation practices, which can be beneficial for maintainability.

Logical SLOC (92 lines):
With a Logical SLOC of 92, the METRICSTICS project appears to be even smaller in terms of functional code size than suggested by the Physical SLOC. Logical SLOC focuses on the number of lines that directly contribute to the program's behavior and control flow. A lower Logical SLOC count often correlates with a codebase that is simpler to understand, test, and maintain. It can also indicate that the project is more manageable and that the code has been written in a concise manner.

Qualitative Analysis:

- **Complexity**: A lower Logical SLOC indicates a lower complexity. It suggests that the project is straightforward, with each class and method having a clear and focused purpose. This can result in fewer bugs and easier debugging.
- **Maintainability**: The modest Logical SLOC implies that the codebase should be easier to maintain. With fewer lines of essential code, developers can more easily comprehend the software's functionality, making it simpler to modify and extend.
- **Development Effort**: The development effort required for a project with a Logical SLOC of 92 is expected to be less than for larger projects. This is because there are fewer logical statements to write, test, and debug.
- **Performance**: A smaller codebase can potentially lead to better performance, although this is not a direct correlation. Performance depends on the efficiency of the code, not just its length.

Inference: The METRICSTICS project, based on the provided metrics, appears to be a small to modestly sized project with a focus on simplicity and manageability. The Logical SLOC suggests that the core functionality is contained within a relatively small number of lines, which should ease understanding and maintenance.

However, as the inference suggests, these metrics should be considered alongside other factors, such as code quality, algorithmic complexity, and architectural design, for a comprehensive assessment of the project. Metrics like cyclomatic complexity, code churn, and technical debt ratio, among others, could provide additional insights into the project's health and maintainability.

PROBLEM 8

Establishing Relation between Logical SLOC and WMC

7.1 Scatter Plot Analysis

To analyze the correlation between Logical SLOC and WMC using a scatter plot, plot the Logical SLOC on the x-axis and WMC on the y-axis. Each class or module in the METRICSTICS project represents a point on this plot.

Class	Logical SLOC	WMC
SignInWindow	17	4
MainWindow	42	23
metricstics	28	36
randomValues	5	2

Table 7.1: Logical SLOC vs WMC metrics

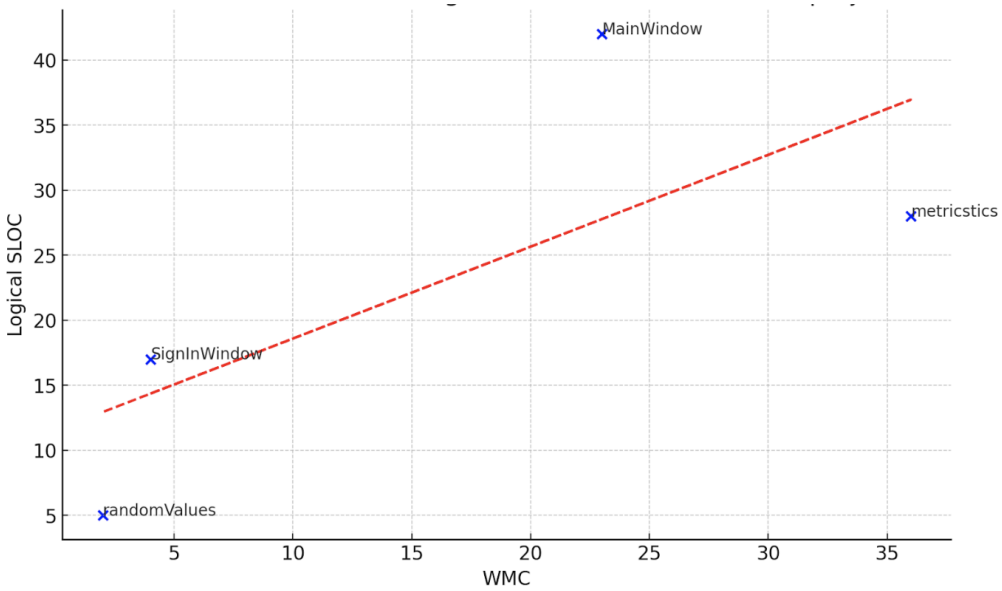

Figure 7.1: Scatter plot of WMC vs Logical SLOC for METRICSTICS project

By plotting these points, we can visually inspect the relationship. A trend line can be drawn to assess whether there's a positive, negative, or no correlation. A positive upward trend would

suggest that as the Logical SLOC increases, the WMC also tends to increase, indicating that more complex classes with more lines of code tend to have more complex methods.

7.2 Correlation Coefficient Analysis

To quantify the correlation between Logical SLOC and WMC, calculate the Spearman's Rank Correlation Coefficient (rs) since the data may not be normally distributed. Assign ranks to each class based on their Logical SLOC and WMC, and use the Spearman's formula:

$$r_s = 1 - \frac{6 \sum_{i=n}^{n} d_i^2}{n^3 - n} \tag{7.1}$$

Where di is the difference between the ranks of corresponding variables (Logical SLOC and WMC), and n is the number of observations (classes or modules).

WMC(xi)	Rank(xi)	SLOC(L)(yi)	Rank(yi)	d	d2
4	3	17	2	1	1
3	2	42	4	-2	4
36	4	28	3	1	1
2	1	5	1	0	0

Table 7.2: Correlation coefficient analysis

Spearman's Rank Correlation Coefficient can range from -1 to 1. The closer the value is to 1 the stronger the correlation it will have. Based on Table 8.2 data of METRICSTICS, the coefficient value is **0.4** which shows a strong correlation between WMC and SLOC(L).

The value of coefficient suggests in case the logical SLOC increases there is a high chance WMC will increase too. But conclusively we cannot say only SLOC has an impact on WMC or vice versa but many other underlying factors may also have an affect on these values.However, it's important to note that correlation does not imply causation. The increase in Logical SLOC might be associated with an increase in WMC, but it doesn't necessarily cause it. Other factors could influence both metrics, such as the nature of the software project, the programming language used, coding standards, the complexity of the problem being solved, and the experience level of the developers.

Advantages and Inferences:

Using the scatter plot, we can visually understand the relationship between Logical SLOC and WMC, which is useful for a quick analysis. The correlation coefficient provides a numerical measure of the strength of the correlation, which is crucial for statistical analysis and decision-making.

In conclusion, these methods of analysis can help in understanding the relationship between code size and complexity in the METRICSTICS project, and can inform decisions on code management, refactoring, and complexity reduction efforts.

Roles and Responsibilities for Deliverable 2

Link to the Tracker : SM Project Tracker Team A

The above excel tracker defines the roles and responsibilities of each member within our team during the project's execution. Our collaborative efforts were facilitated through various communication channels, including Google Meet, WhatsApp chat, Github, and Google Docs. Thanks to the active participation of every team member in project meetings and their commitment to meeting deadlines, we were able to successfully complete our tasks and ensure the project's smooth progress. This synergy and effective communication played a crucial role in our project's success.

REFERENCES

Collaboration Environments

1. **Github :** https://github.com/VarunAggarwal1998/SOEN-6611-SOFTWARE-MEASUREMENT
2. **Drive :** https://drive.google.com/drive/folders/1t3aATqqLoVrFPqAfxMWoi4qZ4B-uiyQb?usp=sharing

References

1. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/
2. **Lecture notes by Prof. Pankaj Kamthan**
3. https://www.codeproject.com/Articles/9913/Project-Estimation-with-Use-Case-Points
4. https://www.c-sharpcorner.com/uploadfile/nipuntomar/cocomo-1-cocomo81-constructive-cost-estimation-model/
5. https://www.geeksforgeeks.org/python-gui-tkinter/
6. https://en.wikipedia.org/wiki/Source_lines_of_code