PROJECT REPORT

METRICSTICS

(DELIVERABLE - 2)

Prepared By

Sharanyu Pillai

Wei Qi

Anitha Ramakrishnan

Arshiya Sahni

Vinay Sahrawat

Under the Guidance of

Prof. Pankaj Kamthan

Submitted at

CONCORDIA UNIVERSITY

Github:

`https://github.com/23Fall-SOEN6611-TeamM/SOEN-6611`

By Team M

Contents

List of Symbols and Abbreviations

GQM	Goal Question Metric
UC	Use Case
SLOC	Source Line of Codes
SLOC(L)	Logical SLOC
UCP	Use Case Point
PF	Productivity Factor
UUCP	Unadjusted Use Case Points
TCF	Technical Complexity Factor
Ecf	Environmental Complexity Factor
UAW	Unadjusted Actor Weight
UUCW	Unadjusted Use Case Weight
WTi	Technical Complexity Factor Weight
Fi	Perceived Impact Weight
WEi	Environmental Complexity Factor Weight

List of Figures

List of Tables

Introduction

Understanding the intricacies of software project estimation and execution is pivotal in the field of software engineering. The delivery-2 offers a comprehensive analysis of such processes through the lens of the METRICSTICS project—a statistical analysis system. By juxtaposing the theoretical estimations of effort via the Use Case Points approach and Basic COCOMO 81 against practical implementation challenges in Python, the paper investigates the nuances of software metrics, design quality, and code complexity. The subsequent correlation analysis between various metrics further enriches the discourse on software project predictability and performance. This exploration aims to serve as a detailed narrative on the multifaceted nature of building a robust software system, from conception to completion.

1.1 System Information

The field of data analysis fundamentally relies on the ability to succinctly describe and interpret large datasets. Descriptive statistics are the tools through which this clarity is achieved, allowing complex data to be expressed through measures of central tendency, frequency, and variability. Central to this domain is the METRICSTICS system, a sophisticated framework engineered for calculating essential statistical metrics. The system is adept at processing a random variable, x, which encompasses a finite set of values $x_1, x_2, x_3, \ldots, x_n$, each with an equal likelihood of occurrence. It is equipped to ascertain the minimum m, maximum M, mode o, median d, arithmetic mean μ, mean absolute deviation (MAD), and standard deviation σ of a dataset. These statistical measures are critical in summarizing and understanding data, irrespective of whether the data is derived from empirical sources or generated via simulation.

1.2 Assumptions

The project has the following assumptions:

1. **Uniqueness of Minimum and Maximum:**

    - The probability distribution of the random variable is assumed to be uniform, as each value in the data set has the same probability.

2. **Uniqueness of Mode:**

    - The mode o may not be unique. There can be multiple modes if multiple values occur with the highest frequency.

3. **Calculation of Median for Even n:**

    - When the number of observations (n) is even, the median d is calculated as the arithmetic mean of the two middle values.

4. **Calculation of Standard Deviation:**

    - The standard deviation (σ) is calculated based on the squared differences between each observation and the arithmetic mean.

5. **Input Data Source:**

    - The system can accept data from different sources, including manual inputs, import from .csv files, and automatically generate data from system.

Problem 3: Use Case Points(UCP)

2.1 Project Estimation - UCP

Effort estimation for the METRICSTICS system development is methodically determined by the Use Case Points (UCP) and the Productivity Factor (PF).

$$Effort\ Estimate = UCP \times PF \quad \text{(Equation 1)}$$

UCP is a multi-faceted metric incorporating various dimensions of project complexity and it is computed as follows:

$$UCP = UUCP \times TCF \times ECF \quad \text{(Equation 2)}$$

where TCF is the Technical Complexity Factor adjusting for technical difficulty, and ECF is the Environmental Complexity Factor considering the development environment. The foundational element of this calculation is the Unadjusted Use Case Points (UUCP), which is derived from:

$$UUCP = UAW + UUCW \quad \text{(Equation 3)}$$

In this equation, UAW represents the Unadjusted Actor Weight, which assesses the complexity of the actors within the system, and $UUCW$ denotes the Unadjusted Use Case Weight, which measures the functional requirements through the total number of transactional steps in all use case scenarios. Together, UAW and $UUCW$ form the UUCP, providing a comprehensive measure of the project's functional size.

The detailed steps for calculating effort estimation are demonstrated below.

I. Calculate the Unadjusted Use Case Points (UUCP)

- The system has one actor, the user who interacts with the system using graphical user interface. So, the actor is a complex actor with 3 points. The UAW matrix shown in

Table 2.1. So,

$$UAW = 3 \times 1 = 3 \quad \text{(Equation 4)}$$

Actor Type	Description	Weight	Actor Count
A1	Simple Actor	1	0
A2	Average Actor	2	0
A3	Complex Actor	3	1

Table 2.1: Actor Weight and Acount Determination

- The application has 8 classes, which is average user case with weight 10 based on the standards outlined in Table 2.2. So,

$$UUCW = 10 \quad \text{(Equation 5)}$$

Use Case Type	Description	Weight
UC1	Simple Use Case	5
UC2	Average Use Case	10
UC3	Complex Use Case	15

Table 2.2: Use Case Type Description and Weight

- The value of UUCP is

$$UUCP = UAW + UUCW = 3 + 10 = 13 \quad \text{(Equation 6)}$$

II. The Technical Complexity Factor (TCF) is calculated using the formula:

$$TCF = C_1 + \left[C_2 \times \sum_{i=1}^{13} [W_{Ti} \times F_i] \right] \quad \text{(Equation 7)}$$

Where: C_1 is the constant 0.6, C_2 is the constant 0.01, W_{Ti} is the Technical Complexity Weight for factor i, and F_i is the Perceived Impact Factor corresponding to each Technical

Complexity Factor in the range of 0 to 5.

According to the assessment of the impact of each technical factor provided by team, table 2.3 presents the Technical Complexity Factor Weight for the system along with their corresponding Perceived Impact Factors(PIF). So, the value of TCF is

$$\begin{aligned} TCF &= 0.6 + (0.01 \times [(2 \times 0) + (1 \times 3) + (1 \times 3) + (1 \times 0) \\ &\quad + (1 \times 5) + (0.5 \times 5) + (0.5 \times 5) + (2 \times 5) + (1 \times 5) \\ &\quad + (1 \times 0) + (1 \times 0) + (1 \times 3) + (1 \times 0)]) \\ &= 0.6 + (0.01 \times 32.6) \\ &= 0.93 \end{aligned} \qquad \text{(Equation 8)}$$

TCF Type	Description	Weight	PIF
T1	Distributed System	2	0
T2	Performance	1	3
T3	End User Efficiency	1	3
T4	Complex Internal Processing	1	0
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	0
T12	Provides Direct Access for Third Parties	1	3
T13	Special User Training Facilities are Required	1	0

Table 2.3: TCF Type Description and Weight with Perceived Impact Factor

III. The Environmental Complexity Factors (ECF) is calculated using the formula:

$$ECF = C_1 + \left[C_2 \times \sum_{i=1}^{8} [W_{Ei} \times F_i] \right] \qquad \text{(Equation 9)}$$

Where: C_1 is the constant 1.4, C_2 is the constant -0.03, W_{Ei} is the the Environmental Complexity Weight for factor i, and F_i is the Perceived Impact Factor corresponding to each Environmental Complexity Factor in the range of 0 to 5.

Based on the assessment of the impact of each environmental factor given by team, table 2.4 presents the Environmental Complexity Factor Weight for the system along with their corresponding Perceived Impact Factors(PIF). So, the value of ECF is

$$\begin{aligned} ECF &= 1.4 + (-0.03 \times [(1.5 \times 3) + (-1 \times 3) + (0.5 \times 5) \\ &\quad +(0.5 \times 3) + (1 \times 3) + (1 \times 0) \\ &\quad +(-1 \times 5) + (2 \times 3)]) \\ &= 1.4 + (-0.03 \times 6.27) \\ &= 1.4 - 0.1881 \\ &= 1.21 \end{aligned} \qquad \text{(Equation 10)}$$

ECF Type	Description	Weight	PIF
E1	Familiarity with Use Case Domain	1.5	3
E2	Part-Time Workers	-1	3
E3	Analyst Capability	0.5	5
E4	Application Experience	0.5	3
E5	Object-Oriented Experience	1	3
E6	Motivation	1	0
E7	Difficult Programming Language	-1	5
E8	Stable Requirements	2	3

Table 2.4: ECF Type Description and Weight with Perceived Impact Factor

IV. Calculate the Use Case Points (UCP) using the formula:

$$\begin{aligned} UCP &= UUCP \times TCF \times ECF \\ &= 13 \times 0.93 \times 1.21 \\ &= 14.63 \end{aligned} \qquad \text{(Equation 11)}$$

V. The initial estimate is 20 person-hours per use case point, so the final effort estimation is:

$$\begin{aligned} Effort\ Estimate &= UCP \times PF \\ &= 14.63 \times 20 \\ &= 292.60\ Hours \end{aligned} \qquad \text{(Equation 12)}$$

2.2 Project Estimation - COCOMO 81

In COCOMO 81, the effort estimate for a software project is calculated using the following formula:

$$\text{Effort} = a_1 \times (KLOC)^{b_1} \qquad \text{(Equation 13)}$$

Where:

- Effort represents the estimated number of person-months required to complete the project.
- $KLOC$ stands for Kilo Lines of Code.
- a_1 and b_1 are constants empirically determined based on project type and historical data.

The detailed steps for calculating effort estimation are demonstrated below.

I. The determination of the values of a_1 and b_1.

Considering the project is a small team-sized software, according to COCOMO 81 model, the values of a_1 and b_1 are:

$$a_1 = 2.4,\ b_1 = 1.05 \qquad \text{(Equation 14)}$$

II. Calculate KLOC. KLOC stands for "Kilo Lines of Code", the project's codebase has approximately 800 lines, so in KLOC, it would be:

$$\text{KLOC} = \frac{\text{lines of code}}{1000} = \frac{800}{1000} = 0.8 \qquad \text{(Equation 15)}$$

III. Finalize effort. According above data, the final effort is:

$$\text{Effort} = 2.4 \times (0.8)^{1.05} = 1.89\ Person - months = 288.64\ Hours \quad \text{(Equation 16)}$$

2.3 Estimation - UCP vs. COCOMO 81 vs. Actual Effort

I. The summary of two types of effort estimation and actual effort.

- **UCP Estimation:** The Use Case Points method estimated the project effort at 292.60 hours. This approach focuses on analyzing the project's use cases and adjusts for technical and environmental factors.
- **Basic COCOMO 81 Estimation:** The COCOMO 81 model estimated the effort to be 288.64 hours. This model is based on the size of the software measured in lines of code.
- **Actual Effort:** The actual effort expended was 260 hours. This is less than both the UCP and COCOMO estimates, which is due to team's higher productivity than expected and some underestimation of the team's capability.
- **Team Size and Work Hours:** With a five-person team working 15 hours per week, the total weekly capacity would be 75 hours. To complete 260 hours of work, the team spent slightly more than 3 weeks.

II. Commentary on the Differences.

- Both the UCP and COCOMO 81 estimates are conservative compared to the actual effort, suggesting that the team was more efficient than estimated.
- The UCP estimate is slightly higher than the COCOMO estimate, which might indicate that the technical and environmental factors included in the UCP calculation had a higher impact than the scale factors of COCOMO.
- The actual effort can be affected by various factors not accounted for in the estimates, such as team experience, the actual code complexity versus what was estimated, tool

usage, and project management efficiency.

- The productivity of the team and the actual hours worked can significantly influence the actual effort. For instance, during the development, some of our team members are highly experienced, the team completed the project with fewer hours than estimated.
- The COCOMO 81 model, being based on historical data, however, this is a new team and no previous data to reference, which could lead to a discrepancy in the estimate.

Problem 4: METRICSTICS System

3.1 System Implementation

I. Technology Stack and Programming Language

The implementation of the METRICSTICS system adopts the object-oriented programming paradigm and utilizes Python as the programming language. The implementation requirements stipulate that, apart from functions related to input and output, basic arithmetic, no native reuse mechanisms provided by the programming language or otherwise should be used. This necessitates the recursive implementation of certain primary functions, resulting in the creation of secondary functions. Furthermore, the implementation of a graphical user interface for METRICSTICS using Tkinter is a mandatory component as per the project's requirement.

II. Core Features

The METRICSTICS system offers the following core features, and the detailed system functions show in the section of 'System Testing'.

- **User Authentication and Login:** To ensure secure access to the system, users are required to authenticate themselves through a login process. Authentication helps protect sensitive data and ensures that only authorized individuals can utilize the system. (Figure3.1: System Login Window)
- **Data Input Method Selection:** Authenticated users have the option to import data using the following methods: manual input, automatic system generation, or importing from a CSV file.(Figure3.2: System Main Window 1)
- **Data Input and Processing:** Users can input data, and the system displays the entered data. The system is also capable of automatically computing and generating results, including the minimum value (m), maximum value (M), mode (o), median

(d), arithmetic mean (μ), mean absolute deviation (MAD), and standard deviation (σ).(Figure3.3: System Main Window 2)

- **Data Visualization:** The system can generate relevant data visualizations to provide a more intuitive representation of data characteristics and distributions.(Figure3.3: System Main Window 2)
- **Historical Data Records:** Users have the capability to review past data results, facilitating the tracking of historical data analysis records.
- **Export Historical Data:** Users can export historical data to their local storage for future reference or sharing.
- **Data Validation and Exception Handling:** The system incorporates robust data validation and exception handling mechanisms. These mechanisms serve to verify the integrity of user input and provide checks against illegal or erroneous operations.

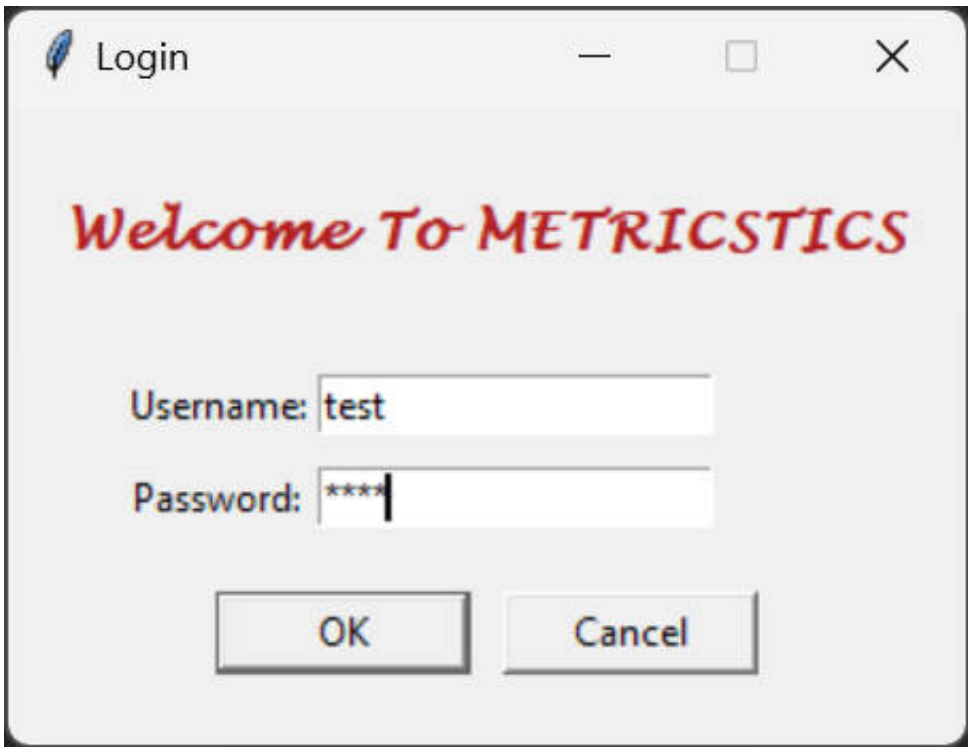


Figure 3.1: System Login Window

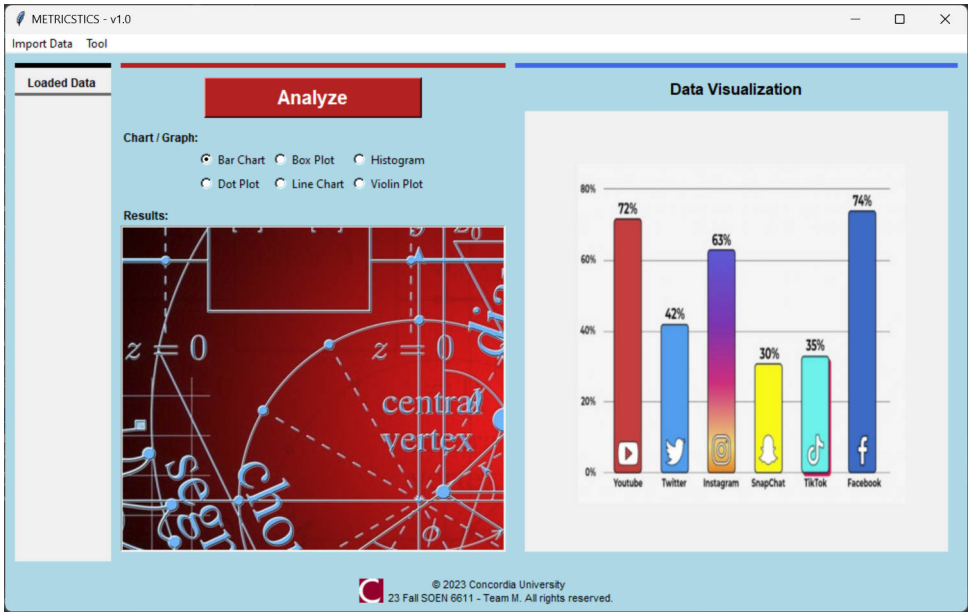


Figure 3.2: System Main Window 1

III. Implementation Process

The METRICSTICS system's implementation involves the creation of multiple classes, each serving a specific purpose in the software's functionality. In total, there are eight main classes (Figure 3.4: Code Structure) that collectively contribute to the system's capabilities:

- **Calculator Class:** The Calculator class is responsible for performing calculations based on user input. It encapsulates the logic for computing statistical metrics, including the minimum, maximum, mode, median, arithmetic mean (μ), mean absolute deviation (MAD), and standard deviation (σ) of the dataset. This class forms the core of the data analysis functionality.

- **EnterDataDialog Class:** The EnterDataDialog class is designed to handle the manual input of data by users. It provides a user-friendly interface for entering data into the system. Users can input data directly, and this class ensures that the entered data is appropriately processed and passed to the Calculator for analysis.

- **GenerateDataDialog Class:** The GenerateDataDialog class plays a crucial role in generating synthetic data within the system. It offers users the option to create datasets automatically. This feature is valuable for scenarios where users want to analyze data quickly without the need for manual input.

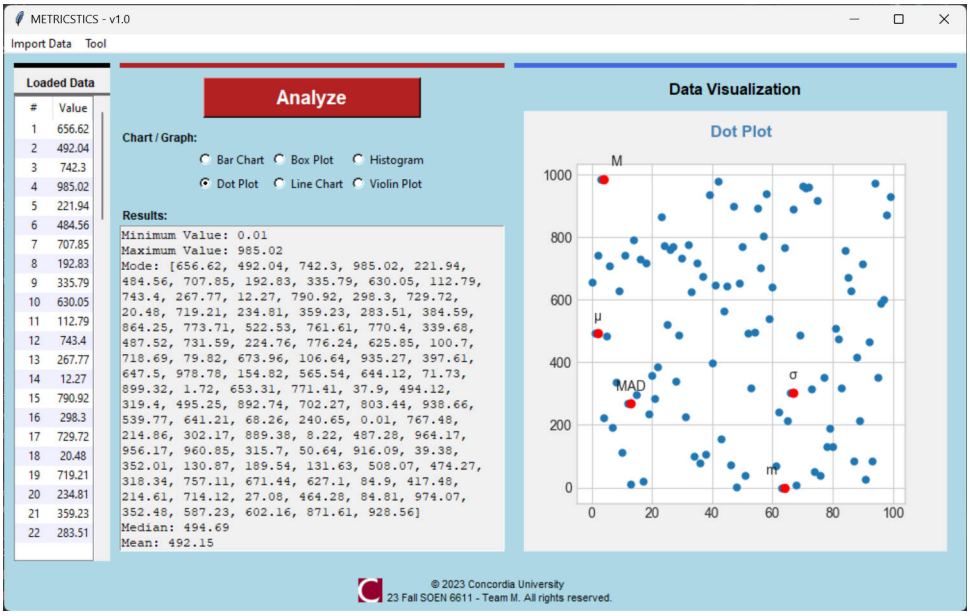


Figure 3.3: System Main Window 2

- **LeftFrame Class:** The LeftFrame class is responsible for displaying user-inputted data. It provides a visual representation of the data entered by users, allowing them to review and verify their input. This class ensures that the data is presented in a clear and organized manner within the user interface.

- **LoginDialog Class:** The LoginDialog class serves as the user login interface. It enables users to authenticate and access the METRICSTICS system securely. Users are required to log in before they can utilize the system's data analysis capabilities.

- **MetricsticsApp Class:** The MetricsticsApp class functions as the primary user interface, serving as the main window of the METRICSTICS application. It orchestrates the display of three essential components: LeftFrame, MiddleFrame, and RightFrame. Users interact with the system through this class, initiating data analysis and visualization.

- **MiddleFrame Class:** The MiddleFrame class is responsible for displaying the calculated results obtained by the Calculator class. It ensures that users can conveniently view essential statistical metrics, providing a clear summary of the data's characteristics.

- **RightFrame Class:** The RightFrame class is dedicated to displaying data visualizations. It generates graphical representations of the dataset, aiding users in understand-

ing the data's distribution and patterns visually.

These eight classes collectively form the foundation of the METRICSTICS system's implementation. The division of responsibilities among these classes allows for modularity, maintainability, and a seamless user experience. Users can interact with the system through the user interface provided by the MetricsticsApp class, input data via EnterDataDialog or GenerateDataDialog, and visualize results through RightFrame.

This structured approach to class design ensures that the METRICSTICS system is organized, user-friendly, and capable of delivering accurate and meaningful data analysis results.

3.2 Algorithm Implementation Of Calculator

The requirement asks an implementation of METRICSTICS must not make use of any reuse mechanism (such as built-in functions, libraries, or APIs) provided natively by the programming language or otherwise. This may lead to recursive implementation of certain primary functions, the result of which would be secondary functions. So, instead of using built-in functions, our system use Calculator class to implement related statistic calculation. The descriptions of algorithms used and the rationale for selecting those algorithms for Calculator class presented as below.

I. The function of minimum and maximum

- Algorithm: Iterates through the list once, keeping track of the smallest and largest values found.
- Rationale: This is the most straightforward approach and requires only one pass through the data, ensuring O(n) time complexity.

```
1     def minimum(self):
2     if not self.data:
3     print("Error: No data available. Please input data first.")
4     return None
5     min_value = self.data[0]
6     for value in self.data:
7     if value < min_value:
```

```
    min_value = value
    return self.round_number(min_value, 2)

```

Python Code 3.1: Minimum method

```
    def maximum(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    max_value = self.data[0]
    for value in self.data:
    if value > max_value:
    max_value = value
    return self.round_number(max_value, 2)

```

Python Code 3.2: Maximum method

II. The function of mode

- Algorithm: Uses a dictionary to count occurrences of each number. Then iterates through the dictionary to find the most frequent number(s).
- Rationale: Counting frequencies is a standard solution for finding the mode, which is efficient for data that fits in memory, typically O(n) time complexity.

```
    def mode(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    frequency = {}
    for value in self.data:
    if value in frequency:
    frequency[value] += 1
    else:
    frequency[value] = 1
    max_frequency = 0
    mode_values = []
    for key, value in frequency.items():
```

```
    if value > max_frequency:
    max_frequency = value
    mode_values = [key]
    elif value == max_frequency:
    mode_values.append(key)
    return mode_values

```

Python Code 3.3: Mode Method

III. The function of median

- Algorithm: Sorts the data and then chooses the middle element (or the average of two middle elements).
- Rationale: Sorting the data is a prerequisite to finding the median in an unordered list. The QuickSort algorithm is used for sorting because it's efficient on average, with time complexity O(n log n).

```
    def median(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    try:
    sorted_data = self.my_sorted(self.data)
    except Exception as e:
    print("Error: An error occurred while sorting the data.")
    print("Details:", e)
    return None
    n = len(sorted_data)
    if n % 2 == 1:
    return self.round_number(sorted_data[n // 2], 2)
    else:
    mid1 = sorted_data[(n - 1) // 2]
    mid2 = sorted_data[n // 2]
    return self.round_number((mid1 + mid2) / 2, 2)

```

Python Code 3.4: Median Method

IV. The function of mean

- Algorithm: Sums all the numbers and divides by the count of numbers.
- Rationale: This is the direct mathematical definition of the mean and is the most efficient way to calculate it, with O(n) time complexity.

```
    def mean(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    try:
    return self.round_number(sum(self.data) / len(self.data), 2)
    except ZeroDivisionError:
    print("Error: Division by zero occurred. This should not happen.")
    return None

```

Python Code 3.5: Mean Method

V. The function of mean_absolute_deviation

- Algorithm: First computes the mean, then calculates the absolute value of the deviation of each number from the mean, and finally averages those deviations.
- Rationale: This approach directly follows the mathematical definition of mean absolute deviation, with O(n) time complexity since it involves iterating through the data twice.

```
    def mean_absolute_deviation(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    mu = self.mean()
    if mu is None:
    return None
    return self.round_number(sum(self.absolute_value(x - mu) for x in
    self.data) / len(self.data), 2)

```

Python Code 3.6: Mean Absolute Deviation Method

VI. The function of standard_deviation

- Algorithm: Computes the mean, then the variance by averaging the squared deviations, and finally takes the square root of the variance to find the standard deviation.
- Rationale: This is the standard formula for standard deviation, which is computed efficiently in two passes through the data (O(n) complexity).

```
    def standard_deviation(self, decimal_places=2):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    mu = self.mean()
    if mu is None:
    return None
    variance = sum((x - mu) ** 2 for x in self.data) / len(self.data)
    std_dev = variance ** 0.5
    return self.round_number(std_dev, decimal_places)

```

Python Code 3.7: Standard Deviation Method

VII. The function of round_number

- Algorithm: Multiplies the number by a power of 10 to shift the decimal point, rounds the number, then shifts back.
- Rationale: This simple algorithm accurately rounds numbers to the desired number of decimal places. It's a basic and direct way to round numbers.

```
    def round_number(self, number, ndigits=None):
    if ndigits is None:
    ndigits = 0
    elif ndigits < 0:
    raise ValueError("ndigits must be non-negative")

    # Shift the decimal point to the right by ndigits
    # So we can work with the integer part for rounding
    shift = 10 ** ndigits
```

```
    temp = number * shift

    # Get the integer and the fractional part of the number
    integer_part = int(temp)
    fractional_part = temp - integer_part

    # Check the fractional part to decide how to round the integer part
    if fractional_part >= 0.5:
    # Round up
    integer_part += 1

    # Shift the decimal point back to the original position
    return integer_part / shift

```

Python Code 3.8: Round Number Method

VIII. The function of absolute_value

- Algorithm: Returns the number itself if it is positive or zero; otherwise, it returns the negation if the number is negative.
- Rationale: This is the most efficient way to determine the absolute value with O(1) time complexity.

```
    def absolute_value(self, number):
    """
    Return the absolute value of the number given.
    """
    # Check if the number is negative
    if number < 0:
    # If the number is negative, make it positive by multiplying by -1
    return -number
    else:
    # If the number is not negative, it is already positive, so return
    as is
    return number

```

Python Code 3.9: Absolute Value Method

IX. The function of quicksort

- Algorithm: Selects a pivot, partitions the list into elements less than and greater than the pivot, and then recursively sorts the partitions.
- Rationale: QuickSort is chosen for its average-case efficiency (O(n log n)). It's a common choice for in-memory sorting due to its good performance on average and its relatively low memory overhead compared to merge sort.

```
    def quicksort(self, data):
    if len(data) <= 1:
    return data
    else:
    pivot = data[0]
    less = [x for x in data[1:] if x < pivot]
    greater = [x for x in data[1:] if x >= pivot]
    return self.quicksort(less) + [pivot] + self.quicksort(greater)

```

Python Code 3.10: QuickSort Method

X. The function of my_sorted

- Algorithm: This method is a wrapper that calls the quicksort method on the class's data.
- Rationale: Provides a public interface to access the sorted data. It encapsulates the sorting functionality and allows the use of different sorting algorithms if needed.

```
    def my_sorted(self):
    if not self.data:
    print("Error: No data available. Please input data first.")
    return None
    return self.quicksort(self.data)

```

Python Code 3.11: My Sorted Method

3.3 System Testing

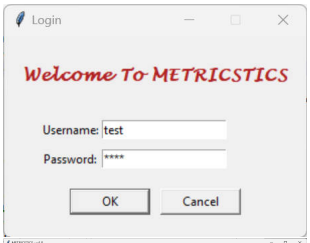
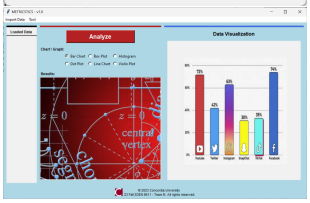
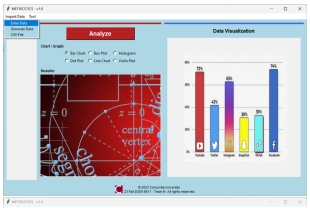
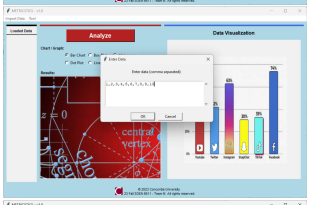
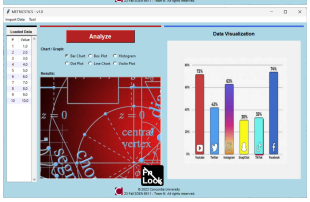
In the process of validating the METRICSTICS system, we employ a combination of automated and manual testing methods. Automated testing primarily focuses on the core "Calculator" class to ensure precise statistical calculations, while manual testing covers the entire user journey, including positive and negative scenarios.
Automated tests verify the accuracy of critical statistical metrics, such as minimum, maximum, mode, median, (σ). Tarithmetic mean (μ), mean absolute deviation (MAD), and standard deviation his ensures the foundational correctness of our data analysis functionality.
Manual testing evaluates the overall user experience, encompassing user interfaces, data input methods, data visualization, and interactions. It ensures the system's user-friendliness and error handling, allowing users to utilize METRICSTICS seamlessly. The following content provides detailed insights into our testing methodologies, test cases, and outcomes for a comprehensive view of METRICSTICS system validation.

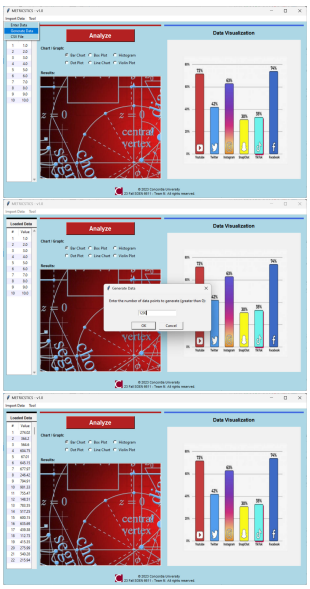
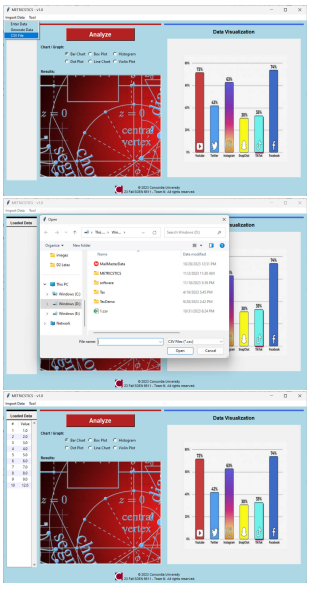
I. Automated Testing

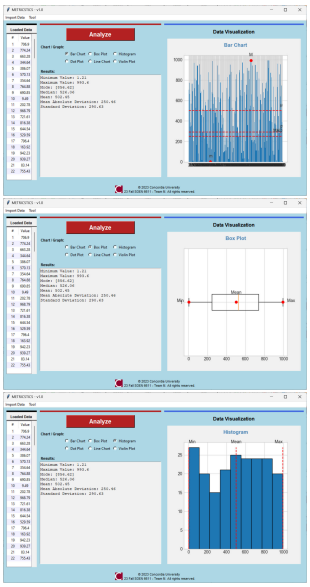
Automated testing was conducted using the pytest tool, and the test results is present in Figure 3.5, the Figure 3.6 is automated testing code segment, the full code is available in the GitHub.

II. Manual testing

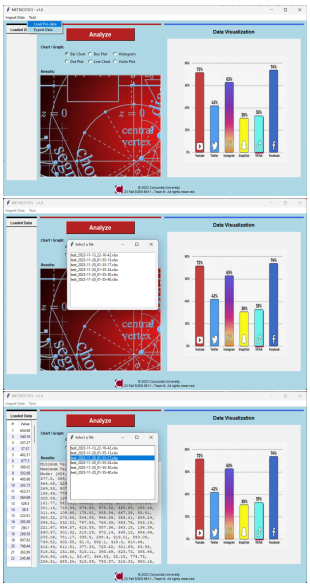
- Positive User Journey

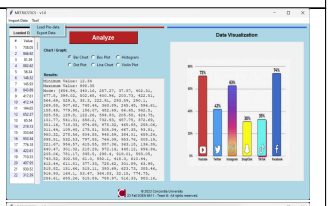
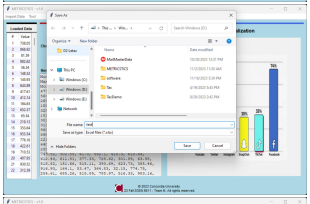
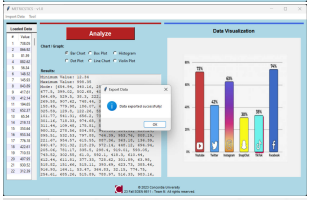
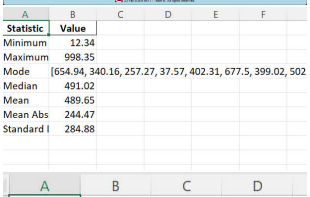
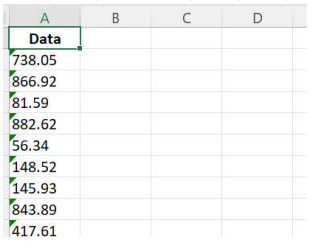
No.	Test Scenario	Test Steps and Expected Result	Result
1	Successful login	**Test Step:** 1. Open the login window. 2. Enter the username "test". 3. Enter the password "test". 4. Click the Ok button. **Expected Result:** The main window displayed	 
2	Enter data manually	**Test Step:** 1. Select the option of "Enter Data" from the menu of "Import Data". 2. Enter data. 3. Click Ok button. **Expected Result:** The data is loaded on the left frame	  

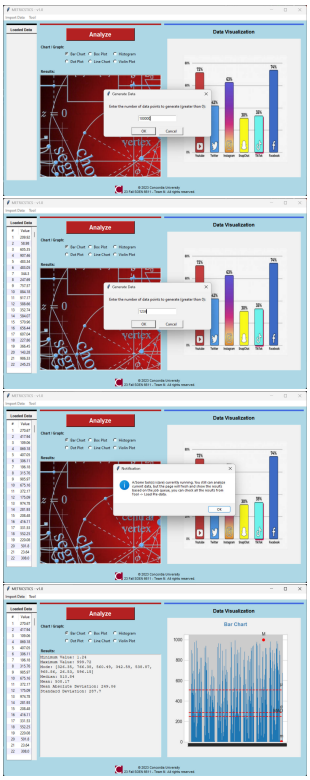
No.	Test Scenario	Test Steps and Expected Result	Result
3	Generate data	**Test Step:** 1. Select the option of "Generate Data" from the menu of "Import Data". 2. Enter the number of data. 3. Click Ok button. **Expected Result:** The data is loaded on the left frame	
4	Import data from cvs file	**Test Step:** 1. Select the option of "CVS File" from the menu of "Import Data". 2. Select cvs file from popup window. 3. Click Ok button. **Expected Result:** The data is loaded on the left frame	

No.	Test Scenario	Test Steps and Expected Result	Result
5	Analyze Data(show Bar Chart,Box Plot, and Histogram)	**Test Step:** 1. Import data. 2. Select Bar Chart. 3. Click Analyze button. 4. Select Box Plot. 5. Click Analyze button. 6. Select Histogram . 7. Click Analyze button. **Expected Result:** The analysis results and corresponding chart displayed	

No.	Test Scenario	Test Steps and Expected Result	Result
6	Analyze Data(Dot Plot,Line Chart, and Violin Plot)	**Test Step:** 1. Import data. 2. Select Dot Plot. 3. Click Analyze button. 4. Select Line Chart. 5. Click Analyze button. 6. Select Violin Plot. 7. Click Analyze button. **Expected Result:** The analysis results and corresponding chart displayed	

No.	Test Scenario	Test Steps and Expected Result	Result
7	Display historical data	**Test Step:** 1. Select "Load Prd-data" from Tool menu. 2. Select data from popup window. **Expected Result:** The historical data is displayed based on the selected record	

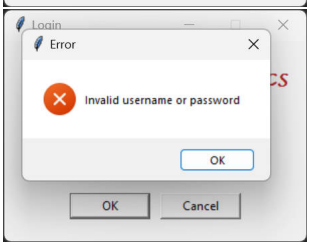
No.	Test Scenario	Test Steps and Expected Result	Result
8	Export Data	**Test Step:** 1. Select "Export Data" from Tool menu. 2. Save the data to the cvs file. 3. Open the cvs file. **Expected Result:** The data is exported to cvs file.	    

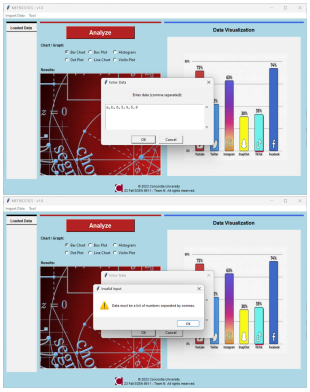
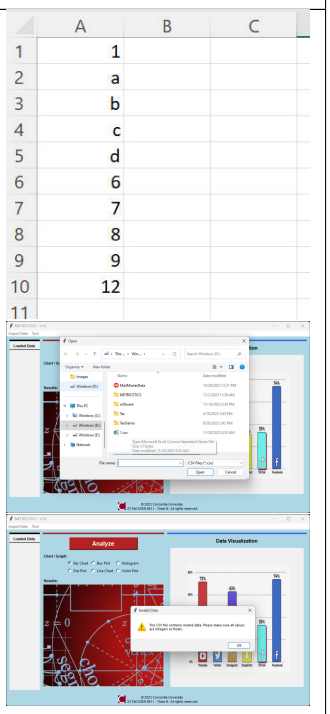
No.	Test Scenario	Test Steps and Expected Result	Result
9	Analyze high data load	**Test Step:** 1. Select "Generate Data" from top menu. 2. Enter 100000. 3. Click Ok. 4. Click Analyze Button 5. Select "Generate Data" from top menu again. 6. Enter 1234. 7. Click Ok. 8. Click Analyze Button. **Expected Result:** After clicking analyze button for the data load with the size of 100000, the system can proceed the data in the back-end and allow user generate another data load(1234) and analyze it. The system should popup window to remind users that some analysis is being proceed in the back-end.	

No.	Test Scenario	Test Steps and Expected Result	Result

Table 3.1: Positive Test Cases

- Negative User Journey

No.	Test Scenario	Test Steps and Expected Result	Result
1	Failed login	**Test Step:** 1. Open the login window. 2. Enter the username "tttt". 3. Enter the password "tttt". 4. Click the Ok button. **Expected Result:** Popup alter window	 

No.	Test Scenario	Test Steps and Expected Result	Result
2	Enter invalidate data	**Test Step:** 1. Select the option of "Enter Data" from the menu of "Import Data". 2. Enter "a,b,d,3,4,5,6" 3. Click Ok button. **Expected Result:** The alter window popup	
3	Select cvs file with invalidate data	**Test Step:** 1. Select the option of "CVS File" from the menu of "Import Data". 2. Choose a file with non-integer data". 3. Click Ok button. **Expected Result:** The alter window popup	

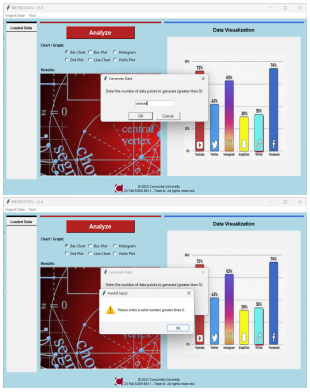
No.	Test Scenario	Test Steps and Expected Result	Result
4	Input invalidate data for the function of Generate data	**Test Step:** 1. Select the option of "Generate Data" from the menu of "Import Data". 2. Input "aaaaaa". 3. Click Ok button. **Expected Result:** The alter window popup	

Table 3.2: Negative Test Cases

3.4 Key Strengths of the System

1. **Concurrent Processing:** By implementing multithreading, the system can handle multiple operations simultaneously, which is particularly useful for performing complex calculations or managing large datasets without freezing the user interface.

2. **Enhanced Responsiveness:** The use of message queues allows the system to process user requests in an orderly fashion, preventing the system from becoming unresponsive during heavy loads. This means that while large data sets are being processed, users can still input new data, perform other analyses, or interact with the system in other ways without experiencing lag.

3. **Efficient Handling of Large Datasets:** The combination of message queues and multithreading prepares the system to scale up to handle larger datasets without a significant redesign. This makes the system future-proof and ready to grow with the user's needs.

4. **Multiple Data Input Options:** The system offers flexibility in data input methods, including manual entry, automatic generation, and importing from CSV files, catering to different user preferences and scenarios.

5. **Graphical Data Visualization:** The ability to generate various types of data visualizations such as bar charts, box plots, and histograms directly within the system allows for a more intuitive understanding of data distributions and patterns.

6. **Historical Data Visualization:** Users can review and export historical data records, which is invaluable for tracking changes over time and for record-keeping purposes.

7. **Data Validation and Error Handling:** Robust validation and exception handling mechanisms ensure data integrity and provide a safeguard against incorrect data inputs, enhancing the reliability of the system.

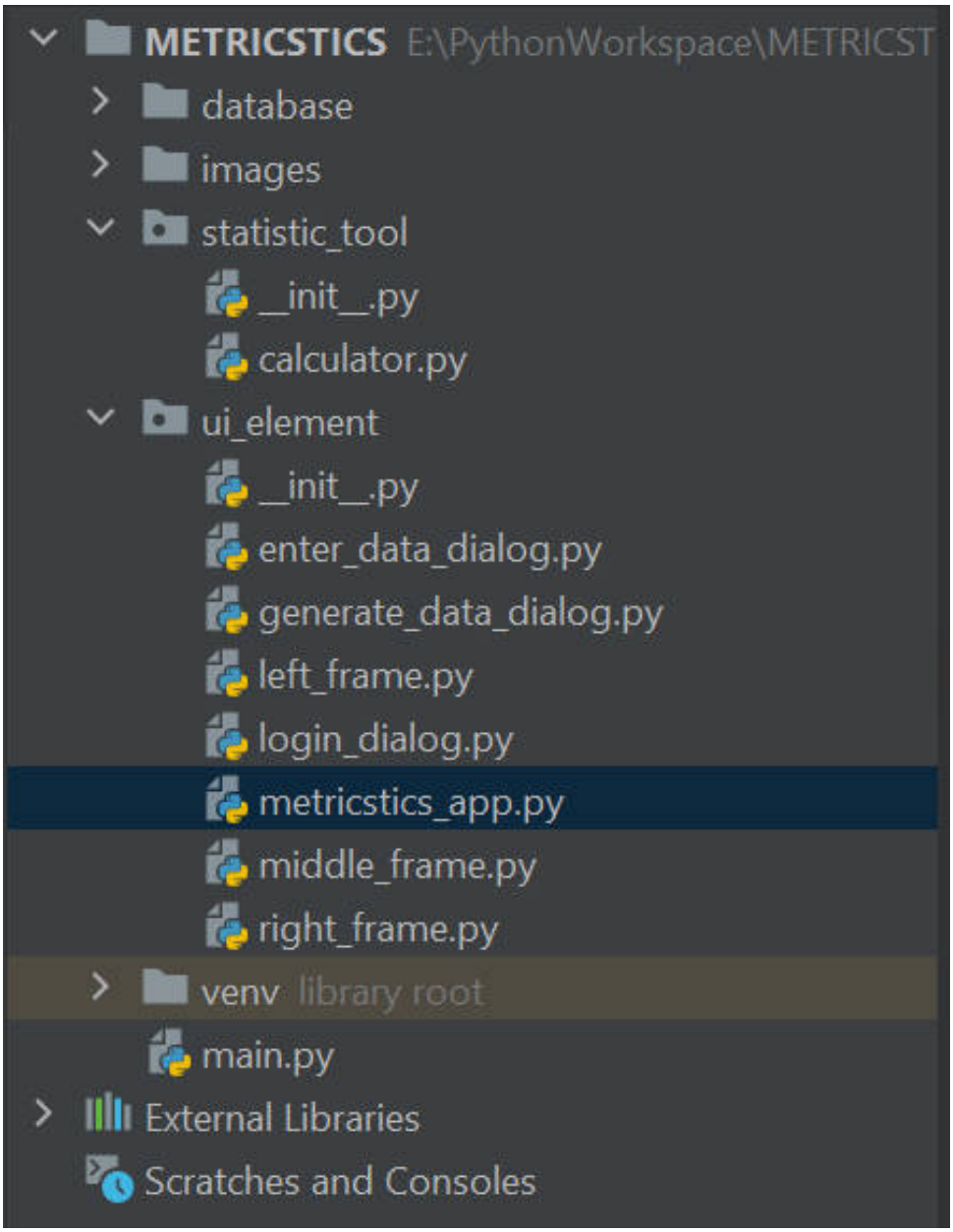
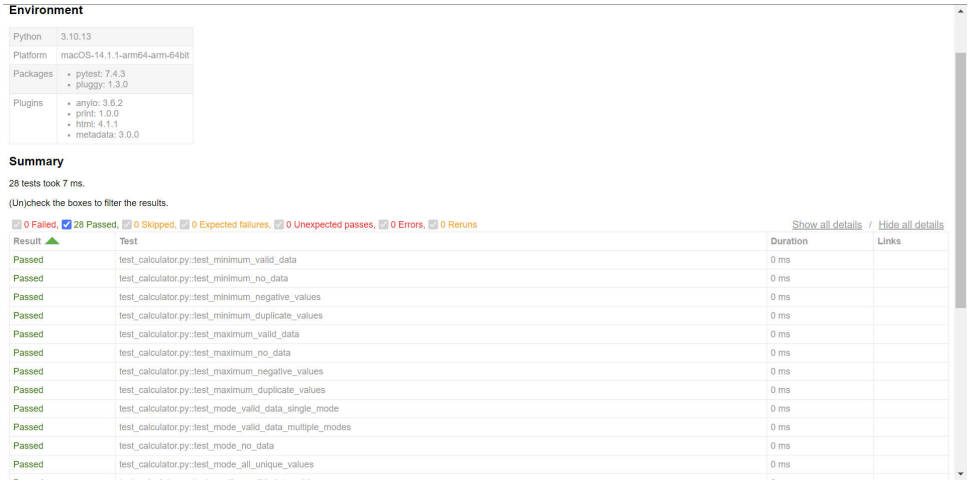


Figure 3.4: Code Structure



Environment

Python	3.10.13
Platform	macOS-14.1.1-arm64-arm-64bit
Packages	• pytest: 7.4.3 • pluggy: 1.3.0
Plugins	• anyio: 3.6.2 • print: 1.0.0 • html: 4.1.1 • metadata: 3.0.0

Summary

28 tests took 7 ms.

(Un)check the boxes to filter the results.

0 Failed, 28 Passed, 0 Skipped, 0 Expected failures, 0 Unexpected passes, 0 Errors, 0 Reruns — Show all details / Hide all details

Result	Test	Duration	Links
Passed	test_calculator.py::test_minimum_valid_data	0 ms	
Passed	test_calculator.py::test_minimum_no_data	0 ms	
Passed	test_calculator.py::test_minimum_negative_values	0 ms	
Passed	test_calculator.py::test_minimum_duplicate_values	0 ms	
Passed	test_calculator.py::test_maximum_valid_data	0 ms	
Passed	test_calculator.py::test_maximum_no_data	0 ms	
Passed	test_calculator.py::test_maximum_negative_values	0 ms	
Passed	test_calculator.py::test_maximum_duplicate_values	0 ms	
Passed	test_calculator.py::test_mode_valid_data_single_mode	0 ms	
Passed	test_calculator.py::test_mode_valid_data_multiple_modes	0 ms	
Passed	test_calculator.py::test_mode_no_data	0 ms	
Passed	test_calculator.py::test_mode_all_unique_values	0 ms	

Figure 3.5: Auto Test Result

test_calculator.py

```
5   # Test cases for minimum function
6   def test_minimum_valid_data():
7       calculator = Calculator()
8       calculator.input_data("3, 1, 4, 1, 5, 9, 2")
9       assert calculator.minimum() == 1.0
10
11
12  def test_minimum_no_data():
13      calculator = Calculator()
14      assert calculator.minimum() is None
15
16
17  def test_minimum_negative_values():
18      calculator = Calculator()
19      calculator.input_data("-3, -1, -4, -1, -5, -9, -2")
20      assert calculator.minimum() == -9.0
21
22
23  def test_minimum_duplicate_values():
24      calculator = Calculator()
25      calculator.input_data("3, 1, 4, 1, 5, 9, 2, 1")
```

Figure 3.6: Automated Testing Code Segment

Problem 5: Cyclomatic Number

Cyclomatic complexity quantifies the complexity of a program's control flow structure. It is based on the number of linearly independent paths within the code, reflecting the decision points and the intricacy of the program's logic. In the METRICSTICS system, comprising eight core classes—Calculator, EnterDataDialog, GenerateDataDialog, LeftFrame, LoginDialog, MetricsticsApp, MiddleFrame, and RightFrame. The calculation of the cyclomatic number is crucial for assessing code maintainability and for identifying areas that may pose risks due to high complexity.
To ensure the accuracy of the cyclomatic complexity evaluation, this analysis utilizes two reputable tools: Lizard and Radon. These tools have been selected for their proven track record in static code analysis. Lizard offers a multi-faceted assessment of code metrics, including cyclomatic complexity, and Radon provides a Python-specific analysis, which is particularly pertinent given that METRICSTICS is developed in Python. By applying both tools in tandem, the aim is to cross-validate the results, thereby enhancing the reliability of the findings.

The section 4.1 presents the cyclomatic numbers ascertained by each tool for the individual classes.

4.1 Cyclomatic Complexity Matrix

The tables from 4.1 to 4.8 provide a comprehensive overview of the Cyclomatic Numbers associated with individual methods within each class. These Cyclomatic Numbers Matrix serve as a measure of the control flow complexity for each method, helping to assess the code's structural intricacies and potential areas for optimization or refactoring.

Class	Method	Cyclomatic number
Calculator	__init__	1
	absolute_value	2
	descriptive_statistics	2
	input_data	5
	maximum	4
	mean	3
	mean_absolute_deviation	4
	median	4
	minimum	4
	mode	7
	round_number	4
	standard_deviation	4

Table 4.1: Cyclomatic Number Matrix - Calculator

Class	Method	Cyclomatic number
EnterDataDialog	__init__	1
	body	1
	validate	8

Table 4.2: Cyclomatic Number Matrix - EnterDataDialog

4.2 Qualitative Analysis of Cyclomatic Number Thresholds

The cyclomatic complexity metric offers insight into the control flow of a program's source code, influencing both its maintainability and testability. Applying the defined ranges to the METRICSTICS system's classes, one can draw several qualitative conclusions regarding the system's architecture and potential areas for improvement.

Class	Method	Cyclomatic number
GenerateDataDialog	__init__	1
	apply	4
	body	1
	validate	3

Table 4.3: Cyclomatic Number Matrix - GenerateDataDialog

Class	Method	Cyclomatic number
LeftFrame	__init__	1
	get_data	4
	set_data	6
	update_width	3

Table 4.4: Cyclomatic Number Matrix - LeftFrame

I. Calculator Class

With a complexity of 44, the Calculator class is categorized as 'very complex', though still manageable. While the class remains testable, the high value suggests that the logic could be intricate, which may complicate the testing process and increase the risk of defects. Refactoring could be beneficial to streamline the control flow and improve the class's maintainability.

II. EnterDataDialog Class

The complexity level of 10 places this class at the threshold between 'simple' and 'complex is well-structured, but any additional complexity could push it into a higher category, potentially complicating future modifications.

III. GenerateDataDialog Class

With a score of 9, this class is deemed to have a 'simple' control flow. Its straightforward structure likely results in ease of testing and a lower likelihood of bugs, making it an exemplar of clean design within the system.

Class	Method	Cyclomatic number
LoginDialog	__init__	1
	apply	2
	body	1
	get_username	2
	ok	2

Table 4.5: Cyclomatic Number Matrix - LoginDialog

Class	Method	Cyclomatic number
MetricsticsApp	__init__	2
	enter_data	3
	export_data	6
	generate_data	3
	load_csv	7
	load_pre_data	6
	login	2
	update_frames	1

Table 4.6: Cyclomatic Number Matrix - MetricsticsApp

IV. LeftFrame Class

A complexity score of 14 categorizes the LeftFrame class as 'complex'. This suggests that while the class is manageable and testable, careful consideration should be given to any additional complexity that could hinder its understandability.

V. LoginDialog Class

The LoginDialog class, with a score of 8, is considered 'simple'. This indicates a clean and clear control flow, which is advantageous for both testing and maintenance purposes.

Class	Method	Cyclomatic number
MiddleFrame	__init__	2
	calculate	2
	calculate_in_thread	8
	generate_graphs	7
	get_results_data	6
	process_queue	3
	save_results_to_excel	6
	uset_results	2
	show_error_msg	1
	show_notification	1

Table 4.7: Cyclomatic Number Matrix - MiddleFrame

VI. MetricsticsApp Class

With a cyclomatic complexity of 30, the MetricsticsApp class is 'very complex', but within the realm of manageability. Careful management and potential simplification are recommended to prevent it from becoming unwieldy.

VII. MiddleFrame Class

The complexity score of 38 also places the MiddleFrame class in the 'very complex' category. This complexity can be managed with diligent oversight, but simplification could greatly benefit future maintenance and testing efforts.

VIII. RightFrame Class

The RightFrame class has an exceptionally low complexity with consistent scores of 1, falling into the 'simple' category. This suggests that the class is highly maintainable and poses minimal risk for errors, exemplifying a streamlined control flow.

In summation, the METRICSTICS system displays a varied range of complexities across its classes. While several classes maintain a 'simple' control flow, which is ideal, there are notable instances of 'complex' and 'very complex' categorizations. These

Class	Method	Cyclomatic number
RightFrame	__init__	1
	clear	1
	draw_bar_chart	1
	draw_bar_plot	1
	draw_dot_plot	1
	draw_histogram	1
	draw_line_chart	1
	draw_violin_plot	1
	set_background	1
	handle_exception	3

Table 4.8: Cyclomatic Number Matrix - RightFrame

findings highlight areas where the system could be optimized to reduce complexity, thereby enhancing maintainability and reducing the cognitive load on developers and testers. It is imperative that the 'very complex' classes undergo careful scrutiny to ensure they do not evolve into unmanageable or untestable segments of the system.

Problem 6: WMC, CF, and LCOM*

5.1 WMC Calculations

Let C be a class with methods $M_1, \ldots, M_n$. Let $c_1(M_1), \ldots, c_n(M_n)$ be the complexity (weights) of the methods. Then, Weighted Method Per Class (WMC)is given by:

$$WMC = \sum_{i=1}^{n} c_i(M_i) \qquad \text{(Equation 17)}$$

Assume that the weights are not normalized, and due to cyclomatic numbers have been previously calculated for each method (Table 4.1 to 4.8), so to compute the WMC for a class, we sum the weights assigned to the cyclomatic numbers of all methods within that class.

I. Calculator Class

From Table 4.1, the Calculator class has a cyclomatic number of:

$$\begin{aligned} \text{Calculator - cyclomatic number} &= 1+2+2+5+4+3+4+4 \\ &\quad +4+7+4+4 \\ &= 44 \end{aligned}$$

II. EnterDataDialog Class

From Table 4.2, the EnterDataDialog class has a cyclomatic number of:

$$\text{EnterDataDialog - cyclomatic number} = 1+1+8 = 10$$

III. GenerateDataDialog Class

From Table 4.3, the GenerateDataDialog class has a cyclomatic number of:

$$\text{GenerateDataDialog - cyclomatic number} = 1 + 4 + 1 + 3 = 9$$

IV. LeftFrame Class

From Table 4.4, the LeftFrame class has a cyclomatic number of:

$$\text{LeftFrame - cyclomatic number} = 1 + 4 + 6 + 3 = 14$$

V. LoginDialog Class

From Table 4.5, the LoginDialog class has a cyclomatic number of:

$$\text{LoginDialog - cyclomatic number} = 1 + 2 + 1 + 2 + 2 = 8$$

VI. MetricsticsApp Class

From Table 4.6, the MetricsticsApp class has a cyclomatic number of:

$$\begin{aligned}\text{MetricsticsApp - cyclomatic number} &= 2 + 3 + 6 + 3 + 7 + 6 + 2 + 1 \\ &= 30\end{aligned}$$

VII. MiddleFrame Class

From Table 4.7, the MiddleFrame class has a cyclomatic number of:

$$\begin{aligned}\text{MiddleFrame - cyclomatic number} &= 2 + 2 + 8 + 7 + 6 + 3 \\ &\quad + 6 + 2 + 1 + 1 \\ &= 38\end{aligned}$$

VIII. RightFrame Class

From Table 4.8, the RightFrame class has a cyclomatic number of:

$$\begin{aligned}\text{RightFrame - cyclomatic number} &= 1+1+1+1+1+1 \\ &\quad +1+1+1+3 \\ &= 12\end{aligned}$$

5.2 CF Calculations

The Coupling Factor (CF) is a metric used to evaluate the average coupling between classes in an Object-Oriented Design (OOD) while excluding inheritance relationships. In the context of OOD, let $C_1, C_2, \ldots, C_n$ represent the classes, where n is greater than 1.

The CF formula is defined as follows:

$$CF = \frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{n} \text{IsClient}(C_i, C_j) \right)}{n^2 - n} \qquad \text{(Equation 18)}$$

Where:

IsClient(C_i, C_j) is a characteristic function that equals 1 if class $C_i$has a relationship with class C_j, and 0 otherwise. This relationship typically signifies interactions like method calls, references, or attribute access.

$\sum_{i=1}^{n}$ computes the sum of relationships for each class C_i.

$\sum_{j=1}^{n}$ sums the relationships of class C_i with all other classes C_j (excluding itself).

The denominator $n^2 - n$ represents the maximum number of possible relationships between classes, excluding self-relationships.

Building upon the provided equation, the following content provides a comprehensive explanation and calculation of CF tailored to our project.

In the class diagram shown in Figure 5.1, n = 8. The IsClient for the MetricsticsApp

class is 6, for the MiddleFrame class is 1, for the Calculator class is 0, for the EnterDataDialog class is 0, for the GenerateDataDialog class is 0, for the LeftFrame class is 0, for the RightFrame class is 0, for the LoginDialog class is 0.

Therefore,

$$CF = \frac{6+1+0+0+0+0+0+0}{64-8} = \frac{7}{56} = 0.125 \qquad \text{(Equation 19)}$$

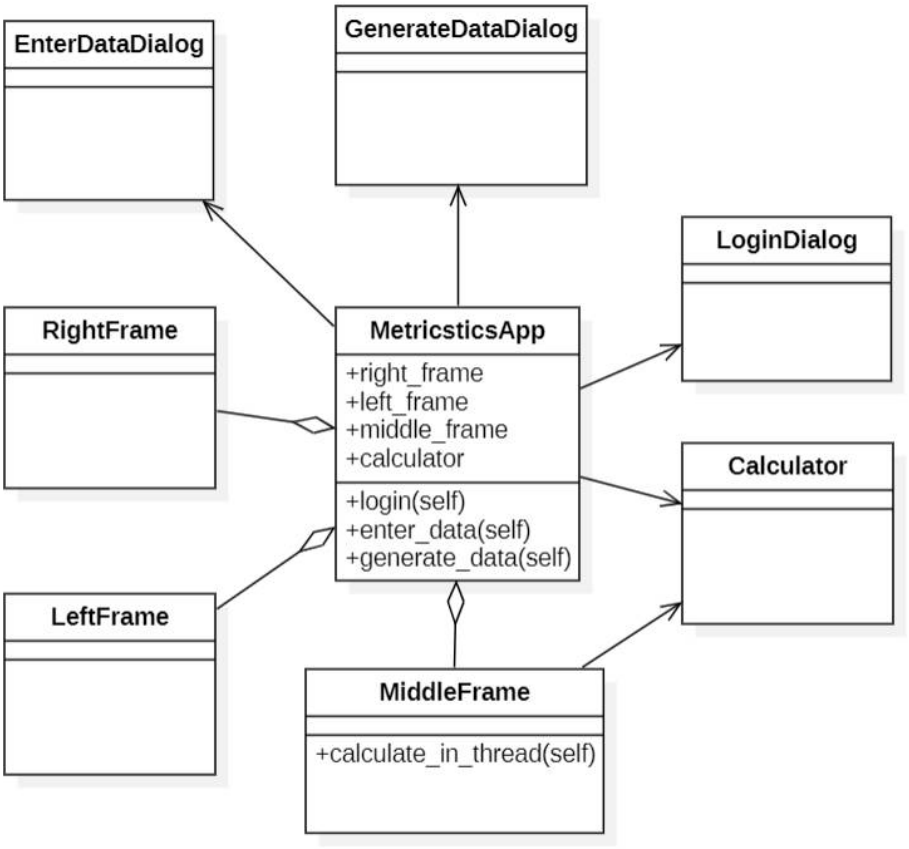


Figure 5.1: System's Class Diagram

5.3 LCOM* Calculations

LCOM*, or Lack of Cohesion in Methods, is a metric used to evaluate the cohesion (or lack thereof) within a class in object-oriented programming. It measures how closely the methods of a class are related to each other based on their interactions with the class's attributes.

A low LCOM* value is desirable because it suggests that the methods of the class are more cohesive, meaning they work closely together and share common data. On the other hand, a high LCOM* value indicates a lack of cohesion, implying that the methods are relatively independent and not closely tied to the same set of attributes.

Steps to Calculate LCOM*:

1. Identify Attributes and Methods:
    - List all attributes (A_i) and methods (M_i) of the class.
2. Determine Attribute Access:
    - For each method (M_i), determine which attributes (A_i) it accesses.
3. Calculate $\mu(A_i)$:
    - Count the number of methods that access each attribute (A_i). This count is denoted as $\mu(A_i)$.
4. Calculate LCOM*:
    - Calculate using the below mentioned formula:

$$LCOM^* = \frac{\frac{1}{a}\left[\sum_{i=1}^{a} \mu(A_i)\right] - m}{1 - m} \qquad \text{(Equation 20)}$$

5. Interpret the Result:
    - A lower LCOM* value is preferable, indicating higher cohesion. The range of LCOM* is $[0, 1]$, where 0 signifies maximum cohesion.

LCOM* calculations for every class:

I. LCOM* for the provided "Calculator" class:

1. Attributes(a): data(A_1) = 1
2. Methods(m): __init__, input_data, minimum, maximum, mode, median, mean, mean_absolute_deviation, standard_deviation, descriptive_statistics, round_number, absolute_value = 12

3. Methods Accessing the Attribute(s):

    - data(μA_1): input_data, minimum, maximum, mode, median, mean, mean_absolute_deviation, standard_deviation, descriptive_statistics = 9

4. $\Sigma\mu(A_i)$: $A_1 = 9$

5. Substituting values in LCOM* formula = (9 - 12)/(1-12) = 0.27

LCOM*(Calculator) = 0.27

II. LCOM* for the provided "MetricsticsApp" class

1. Attributes(1): username(A_1), right_frame(A_2), csv_file(A_3), calculator(A_4), logo_image(A_5) = 5

2. Methods(2): __init__, login, update_frames, enter_data, generate_data, load_csv, load_pre_data, export_data = 8

3. Methods Accessing the Attribute(s):

    - username(μA_1): login, load_pre_data = 2
    - right_frame(μA_2): __init__, update_frames = 2
    - csv_file(μA_3): load_csv = 1
    - calculator(μA_4): __init__ = 1
    - logo_image(μA_5): __init__ = 1

4. $\Sigma\mu(A_i)$: $\mu A_1 + \mu A_2 + \mu A_3 + \mu A_4 + \mu A_5 = 7$

5. Substituting values in LCOM* formula = (7 - 40)/(5 * -7) = 0.94

LCOM*(MetricsticsApp) = 0.94

III. LCOM* for the provided "EnterDataDialog" class

1. Attributes(1): result(A_1) = 1

2. Methods(2): __init__, body, validate = 3

3. Methods Accessing the Attribute(s):
    - result(μA_1): validate = 1
4. $\Sigma\mu(A_i)$: $\mu A_1 = 1$
5. Substituting values in LCOM* formula = (1 - 3)/(1 - 3) = 1

LCOM*(EnterDataDialog) = 1

IV. LCOM* for the provided "GenerateDataDialog" class

1. Attributes(1): decimal_places(A_1), result(A_2) = 2
2. Methods(2): __init__, body, validate, apply = 4
3. Methods Accessing the Attribute(s):
    - decimal_places(μA_1): __init__, apply = 2
    - result(μA_2): apply = 1
4. $\Sigma\mu(A_i)$: $\mu A_1 + \mu A_2 = 3$
5. Substituting values in LCOM* formula = (3 - 8)/(2 * (1 - 4)) = 0.83

LCOM*(GenerateDataDialog) = 0.83

V. LCOM* for the provided "LeftFrame" class

1. Attributes(1): loaded_data_label(A_1), data_table(A_2), scrollbar(A_3) = 3
2. Methods(2): __init__, update_width, get_data, set_data = 4
3. Methods Accessing the Attribute(s):
    - loaded_data_label(μA_1): __init__ = 1
    - data_table(μA_2): __init__, update_width, get_data, set_data = 4
    - scrollbar(μA_3): __init__, set_data = 2
4. $\Sigma\mu(A_i)$: $\mu A_1 + \mu A_2 + \mu A_3 = 7$
5. Substituting values in LCOM* formula = (7 - 12)/(3 * -3) = 0.55

LCOM*(LeftFrame) = 0.55

VI. LCOM* for the provided "RightFrame" class

1. Attributes(1): figure(A_1), ax(A_2), canvas(A_3), canvas_widget(A_4), label(A_5) = 5
2. Methods(2): __init__, set_background, clear, draw_bar_chart, draw_histogram, draw_box_plot, draw_dot_plot, draw_line_chart, draw_violin_plot = 9
3. Methods Accessing the Attribute(s):
   - figure(μA_1): __init__, draw_bar_chart, draw_histogram, draw_box_plot, draw_dot_plot, draw_line_chart, draw_violin_plot = 7
   - ax(μA_2): __init__, draw_bar_chart, draw_histogram, draw_box_plot, draw_dot_plot, draw_line_chart, draw_violin_plot = 7
   - canvas(μA_3): __init__, set_background, draw_bar_chart, draw_histogram, draw_box_plot, draw_dot_plot, draw_line_chart, draw_violin_plot = 8
   - canvas_widget(μA_4): __init__ = 1
   - label(μA_5): __init__ = 1
4. $\Sigma\mu(A_i)$: $\mu A_1 + \mu A_2 + \mu A_3 + \mu A_4 + \mu A_5 = 24$
5. Substituting values in LCOM* formula = (24 - 45)/(5 * -7) = 0.525

LCOM*(RightFrame) = 0.525

VII. LCOM* for the provided "MiddleFrame" class

1. Attributes(1): app(A_1), calculator(A_2), right_frame(A_3), last_saved_data(A_4), update_queue(A_5), task_running(A_6), calculate_button(A_7), graph_label(A_8), selected_graph_type(A_9), graph_frame(A_{10}), graph_types(A_{11}), results_frame(A_{12}), results_label(A_{13}), results_text(A_{14}), bg_image(A_{15}), results_scroll(A_{16}) = 17
2. Methods(2): __init__, set_background, clear, draw_bar_chart, draw_histogram, draw_box_plot, draw_dot_plot, draw_line_chart, draw_violin_plot, calculate, calculate_in_thread, show_notification,

process_queue, save_results_to_excel, generate_graphs, set_results, get_results_data = 17

3. Methods Accessing the Attribute(s):

   - app(μA_1): __init__, calculate, calculate_in_thread, process_queue, save_results_to_excel, generate_graphs, set_results, get_results_data = 8
   - calculator(μA_2): __init__, calculate, calculate_in_thread, process_queue, save_results_to_excel, generate_graphs = 6
   - right_frame(μA_3): __init__, calculate, calculate_in_thread, process_queue, generate_graphs = 5
   - last_saved_data(μA_4): __init__, save_results_to_excel = 2
   - update_queue(μA_5): __init__, process_queue = 2
   - task_running(μA_6): __init__, calculate, calculate_in_thread, show_notification, process_queue = 5
   - calculate_button(μA_7): __init__, calculate = 2
   - graph_label(μA_8): __init__ = 1
   - selected_graph_type(μA_9): __init__, calculate, generate_graphs = 3
   - graph_frame(μA_{10}): __init__, generate_graphs = 2
   - graph_types(μA_{11}): __init__, generate_graphs = 2
   - results_frame(μA_{12}): __init__, process_queue, set_results = 3
   - results_label(μA_{13}): __init__ = 1
   - results_text(μA_{14}): __init__, set_results, get_results_data = 3
   - bg_image(μA_{15}): __init__ = 1
   - results_scroll(μA_{16}): __init__ = 1

4. $\Sigma\mu(A_i)$: Sum of all the attributes and the methods that access them = 50

5. Substituting values in LCOM* formula = (50 - 17)/(1 - 17) = 0.805

LCOM*(MiddleFrame) = 0.805

VIII. LCOM* for the provided "LoginDialog" class

1. Attributes(1): username(A_1), password(A_2), authenticated(A_3), user_data(A_4), e1(A_5), e2(A_6) = 6
2. Methods(2): __init__, body, apply, ok, get_username = 5
3. Methods Accessing the Attribute(s):
   - username(μA_1): __init__, apply, ok, get_username = 4
   - password(μA_2): __init__, apply, ok = 3
   - authenticated(μA_3): __init__, apply, ok = 3
   - user_data(μA_4): __init__, apply = 2
   - e1(μA_5): body, apply, ok = 3
   - e2(μA_6): body, apply, ok = 3
4. $\Sigma\mu(A_i)$: $\mu A_1 + \mu A_2 + \mu A_3 + \mu A_4 + \mu A_5 + \mu A_6 = 26$
5. Substituting values in LCOM* formula = (26 - 5)/(6 * -1) = 0.16

LCOM*(LoginDialog) = 0.16

5.4 Conclusions

I. Qualitative Conclusions - WMC

1. **Calculator Class (WMC: 44):** Moderately complex class due to a relatively high WMC. It involves various mathematical operations.
2. **MetricsticsApp Class (WMC: 30):** Moderate complexity, indicating that it handles several functionalities related to metrics in the application.
3. **EnterDataDialog Class (WMC: 10):** Simple class with low complexity. It likely deals with data entry dialog functionality, making it easily understandable.
4. **GenerateDataDialog Class (WMC: 9):** Simple class with low complexity. It handles data generation with limited complexity.

5. **LeftFrame Class (WMC: 14):** Moderately complex class, probably managing the left frame of the application. It may involve data presentation or manipulation.

6. **MiddleFrame Class (WMC: 38):** Relatively high complexity. This class likely plays a critical role, involving calculations, data management, and results generation.

7. **RightFrame Class (WMC: 12):** Moderately complex class, responsible for graphical representation and exception handling on the right side of the application.

8. **LoginDialog Class (WMC: 8):** Low to moderate complexity. Handles the login dialog functionality, likely with straightforward operations.

In summary, the complexity distribution of the classes suggests a modular design where each class is responsible for specific tasks. This distribution has maintenance implications, as classes with higher complexity, such as Calculator and MiddleFrame, may demand more attention during maintenance, testing, and debugging. On the other hand, classes with lower complexity, like EnterDataDialog and GenerateDataDialog, tend to be more readable and easier to maintain.

II. Qualitative Conclusions - CF

1. **CF Value (0.125):** The CF value of 0.125 suggests that the average coupling between classes in the project is relatively low. This indicates that there are fewer interclass relationships, which can be considered a positive aspect of the project's design. Lower CF values generally imply reduced complexity and better modularity.

2. **Metric Thresholds:** While the specific metric thresholds may vary depending on the project's requirements and industry standards, a CF value of 0.125 is often considered favorable. It falls below typical thresholds associated with high coupling, which can lead to software that is difficult to maintain and understand.

3. **Low Coupling:** The low CF value indicates that classes in the project have limited dependencies on each other, reducing the risk of ripple effects when

making changes to one class. This is a desirable quality for software maintainability and scalability.

4. **Modular Design:** The project's design appears to be modular, with classes exhibiting a relatively low level of interdependence. This modular structure can facilitate code reuse and make it easier to isolate and fix issues when they arise.

5. **Efficient Maintenance:** The lower CF value implies that maintenance and updates to the codebase are likely to be more efficient, as changes in one class are less likely to impact other parts of the system.

In summary, the calculated CF value of 0.125 suggests that the project's class relationships exhibit low coupling, which aligns with good design principles. This is indicative of a modular and maintainable codebase. However, the specific interpretation may depend on the context and requirements of the project, and it's important to consider other metrics and qualitative aspects of the design for a comprehensive evaluation.

III. Qualitative Conclusions - LCOM*

1. **Calculator Class (LCOM*: 0.27):** The methods in the Calculator class are somewhat cohesive, with a reasonably low LCOM* value. This indicates that the methods tend to access a shared set of attributes, promoting cohesion.

2. **MetricsticsApp Class (LCOM*: 0.94):** The MetricsticsApp class has a high LCOM* value, suggesting a lack of cohesion among its methods. Methods appear to be relatively independent, potentially indicating a design issue with the class.

3. **EnterDataDialog Class (LCOM*: 1):** The EnterDataDialog class has a high LCOM* value, indicating a lack of cohesion among its methods. Methods are relatively independent, potentially making the class less maintainable.

4. **GenerateDataDialog Class (LCOM*: 0.83):** The GenerateDataDialog class exhibits moderate cohesion among its methods, with a relatively lower LCOM* value. Methods share some common attributes, promoting a degree of cohesion.

5. **LeftFrame Class (LCOM*: 0.55):** The LeftFrame class has a moderate LCOM* value, indicating a moderate level of cohesion. Methods seem to interact with a shared set of attributes, contributing to a cohesive design.

6. **RightFrame Class (LCOM*: 0.525):** The RightFrame class shows moderate cohesion, as indicated by its LCOM* value. Methods interact with a common set of attributes to some extent, contributing to a cohesive design.

7. **MiddleFrame Class (LCOM*: 0.805):** The MiddleFrame class has a high LCOM* value, suggesting a lack of cohesion among its methods. This may indicate that methods are relatively independent, potentially impacting maintainability.

8. **LoginDialog Class (LCOM*: 0.16):** The LoginDialog class exhibits low cohesion among its methods, as indicated by the low LCOM* value. Methods appear to be relatively independent, which may impact maintainability.

In conclusion, the LCOM* values reflect varying levels of cohesion among the classes. Classes with higher LCOM* values have less cohesive methods, while those with lower values tend to exhibit more cohesive designs. These insights can inform decisions regarding code organization and design enhancements to improve maintainability and readability.

Problem 7: Physical SLOC vs. Logical SLOC

In this section, we delve into the quantitative analysis of the source code developed for our project, employing the measures of physical and logical Source Lines of Code (SLOC). This metric not only serves as a reflection of the coding effort but also offers insights into the complexity and volume of the codebase. To ensure accuracy and objectivity in our assessment, we utilized two well-regarded tools in the field of software metrics: Lizard and Radon. These tools facilitated the computation of physical SLOC and logical SLOC. The application of these instruments enabled us to draw a comprehensive picture of the program's structure, providing a foundation for further discussions on code quality, maintainability, and efficiency.

6.1 Physical SLOC Calculations

In the calculation of physical SLOC, we adhered to the rule that a source line of code is defined as any line of program text that is not a comment or blank line, irrespective of the number of statements or fragments of statements present on that line.

We conducted an analysis on eight class files within our codebase using the two aforementioned tools and have summarized our findings in a matrix, details of which can be observed from Table 6.1 to 6.8. Within this matrix, we have dissected each class file to determine the physical SLOC for package lines, class declarations, and the respective methods contained within. The final physical SLOC is computed by summing the number of lines of code for package declarations, class declarations, and all the methods. In alignment with the definition of physical SLOC, we have excluded comments and blank lines from this calculation.

The final physical SLOC is computed by summing the number of lines of code for package declarations, class declarations, and all the methods: Physical SLOC =

Lines of Package Declarations + Lines of Class Declarations + $\sum$(Lines of Each Method).

Based on the table 6.1 to table 6.8, the Physical SLOC for each class file is:

- Physical SLOC of class file Calculator = 126.
- Physical SLOC of class file EnterDataDialog = 30.
- Physical SLOC of class file GenerateDataDialog = 32.
- Physical SLOC of class file LeftFrame = 59.
- Physical SLOC of class file LoginDialog = 44.
- Physical SLOC of class file MetricsticsApp = 158.
- Physical SLOC of class file MiddleFrame = 159.
- Physical SLOC of class file RightFrame = 201.

Class File	**Package line**	**Class declaration**	**Method**	**Physical SLOC**
Calculator	0	1	__init__	2
			absolute_value	5
			descriptive_statistics	13
			input_data	12
			maximum	9
			mean	9
			mean_absolute_deviation	8
			median	17
			minimum	9
			mode	19
			round_number	12
			standard_deviation	10

Table 6.1: Physical SLOC for Calculator class file.

Class file	Package line	Class declaration	Method	Physical SLOC
EnterDataDialog	2	1	__init__	2
			body	11
			validate	14

Table 6.2: Physical SLOC for EnterDataDialog class file.

Class file	Package line	Class declaration	Method	Physical SLOC
GenerateDataDialog	3	1	__init__	3
			apply	13
			body	6
			validate	6

Table 6.3: Physical SLOC for GenerateDataDialog class methods.

Class file	Package line	Class declaration	Method	Physical SLOC
LeftFrame	2	1	__init__	17
			get_data	12
			set_data	16
			update_width	11

Table 6.4: Physical SLOC for LeftFrame class methods.

Class file	Package line	Class declaration	Method	Physical SLOC
LoginDialog	3	1	__init__	6
			apply	12
			body	16
			get_username	2
			ok	4

Table 6.5: Physical SLOC for LoginDialog class methods.

Class file	**Package line**	**Class declaration**	**Method**	**Physical SLOC**
MetricsticsApp	13	1	__init__	49
			enter_data	7
			export_data	17
			generate_data	7
			load_csv	19
			load_pre_data	31
			login	5
			update_frames	9

Table 6.6: Physical SLOC for MetricsticsApp class methods.

Class file	**Package line**	**Class declaration**	**Method**	**Physical SLOC**
MiddleFrame	8	1	__init__	38
			calculate	5
			calculate_in_thread	29
			generate_graphs	14
			get_results_data	17
			process_queue	9
			save_results_to_excel	25
			set_results	6
			show_error_msg	2
			show_notification	5

Table 6.7: Physical SLOC for MiddleFrame class methods.

Class file	Package line	Class declaration	Method	Physical SLOC
RightFrame	6	1	__init__	13
			clear	2
			draw_bar_chart	33
			draw_box_plot	23
			draw_dot_plot	27
			draw_histogram	23
			draw_line_chart	30
			draw_violin_plot	30
			set_background	6
			handle_exception	7

Table 6.8: Physical SLOC for RightFrame class methods.

6.2 Logical SLOC Calculations

In the context of software metrics, Logical Source Lines of Code (Logical SLOC) differs from Physical SLOC in that it measures the number of executable statements, which can be more indicative of the complexity and the amount of work involved in developing software. A logical statement may span multiple physical lines, but it is counted as one logical line of code.

Logical SLOC is defined as the count of executable statements, encompassing control structures, data declarations, and executable instructions. Our analytical review encompassed eight class files from our repository, employing tools designed to specifically calculate Logical SLOC. The results of this investigation are consolidated into a series of matrices, presented from Table 6.9 to 6.16. Within these matrices, each method is meticulously examined to ascertain the Logical SLOC .

6.3 Conclusions

Utilizing the data provided in the tables for Physical and Logical SLOC, we can infer specific aspects of the codebase for each class file and make targeted observations:

Class file	**Method**	**Logical SLOC**
Calculator	__init__	2
	absolute_value	5
	descriptive_statistics	6
	input_data	12
	maximum	9
	mean	9
	mean_absolute_deviation	8
	median	17
	minimum	9
	mode	19
	round_number	12
	standard_deviation	10

Table 6.9: Logical SLOC for Calculator class methods.

Class file	**Method**	**Logical SLOC**
EnterDataDialog	__init__	2
	body	11
	validate	14

Table 6.10: Logical SLOC for EnterDataDialog class methods.

Class file	**Method**	**Logical SLOC**
GenerateDataDialog	__init__	3
	apply	13
	body	6
	validate	6

Table 6.11: Logical SLOC for GenerateDataDialog class methods.

Class file	**Method**	**Logical SLOC**
LeftFrame	__init__	17
	get_data	12
	set_data	16
	update_width	11

Table 6.12: Logical SLOC for LeftFrame class methods.

Class file	**Method**	**Logical SLOC**
LoginDialog	__init__	6
	apply	12
	body	17
	get_username	2
	ok	4

Table 6.13: Logical SLOC for LoginDialog class methods.

Class file	**Method**	**Logical SLOC**
MetricsticsApp	__init__	46
	enter_data	7
	export_data	16
	generate_data	7
	load_csv	15
	load_pre_data	32
	login	5
	update_frames	9

Table 6.14: Logical SLOC for MetricsticsApp class methods.

Class file	**Method**	**Logical SLOC**
MiddleFrame	__init__	35
	calculate	5
	calculate_in_thread	30
	generate_graphs	14
	get_results_data	17
	process_queue	9
	save_results_to_excel	25
	set_results	6
	show_error_msg	2
	show_notification	2

Table 6.15: Logical SLOC for MiddleFrame class methods.

Class file	**Method**	**Logical SLOC**
RightFrame	__init__	13
	clear	2
	draw_bar_chart	31
	draw_box_plot	23
	draw_dot_plot	27
	draw_histogram	23
	draw_line_chart	30
	draw_violin_plot	30
	set_background	6
	handle_exception	7

Table 6.16: Logical SLOC for RightFrame class methods.

1. **Calculator Class:** With a Physical SLOC of 126 and a Logical SLOC for methods ranging from 2 to 19, this class appears to have a moderate level of complexity. Methods like median and mode with higher Logical SLOC suggest more intricate logic that might need careful attention during maintenance.

2. **EnterDataDialog Class:** This class has a relatively low Physical SLOC of 30, indicating a smaller size. The Logical SLOC values suggest straightforward methods, which might indicate a well-structured and potentially easier to maintain dialog interface.

3. **GenerateDataDialog Class:** With a Physical SLOC of 32 and Logical SLOC values not exceeding 13, it shows a balance, possibly due to well-contained methods and a clear focus on functionality.

4. **LeftFrame Class:** The Logical SLOC peaks at 17 for the __init__ method, which might suggest that the initialization process for this class is quite complex and could benefit from refactoring if it impacts maintainability.

5. **LoginDialog Class:** The Logical SLOC for methods in this class are relatively low, with a maximum of 17, which implies that the methods might be well-defined with single responsibilities, making the class more maintainable.

6. **MetricsticsApp Class:** This class has the highest Physical SLOC of 158, indicating a significant amount of code. The Logical SLOC distribution shows several methods with higher values, such as load_pre_data, which may require optimization or refactoring to improve understandability and ease of changes.

7. **MiddleFrame Class:** With a Physical SLOC of 159 and a Logical SLOC range from 2 to 35, this class seems to have a mix of simple and complex methods. The __init__ method, having the highest Logical SLOC, might be overloaded and could be simplified.

8. **RightFrame Class:**The Physical SLOC is the highest at 201, and the Logical SLOC has several methods with values over 20, which suggests that this class is quite complex and may be doing too much. It could be a prime candidate for decomposition into smaller classes or methods.

From these observations, one can conclude that there are disparities in complexity and potential refactoring needs across the classes. Classes like RightFrame and MiddleFrame may benefit from a detailed review to simplify complex methods, whereas classes like EnterDataDialog and LoginDialog may be examples of concise and focused design. The data suggests that there may be an opportunity to standardize the approach to complexity across the codebase to enhance overall maintainability.

Problem 8: Correlations - Logical SLOC vs. WMC

7.1 Scatter Plot

In Figure 7.1, we observe a scatter plot presenting the correlation between data for Logical Source Lines of Code (SLOC) and Weighted Methods per Class (WMC), as derived from METRICSTICS. By visually mapping out the data points on the scatter plot, discernible patterns and trends become apparent. WMC, denoting "Weighted Methods per Class," functions as a metric to gauge the complexity of a class, shedding light on both the quantity and intricacy of methods within a class. Meanwhile, Logical SLOC quantifies the size or complexity of a software project by considering the number of logical statements or instructions in the source code that impact the program's control flow or behavior. Scrutinizing these values on the scatter plot provides valuable insights into the relationship between SLOC and WMC within a given system.

Class	**WMC**	**SLOC (L)**
LoginDialog	8	41
GenerateDataDialog	9	28
EnterDataDialog	10	27
RightFrame	12	192
LeftFrame	14	56
MetricsticsApp	30	137
MiddleFrame	38	145
Calculator	44	118

Table 7.1: Class Metrics

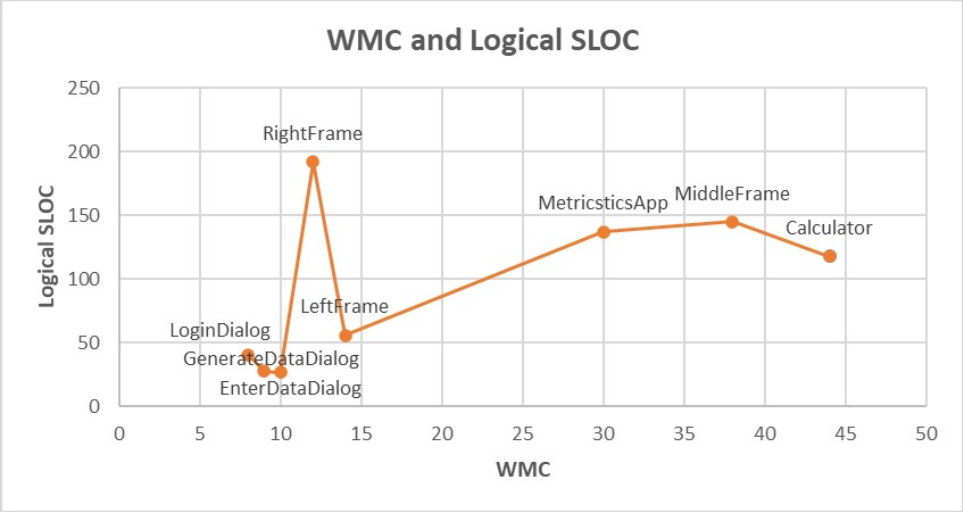


Figure 7.1: Scatter Plot (Logical SLOC and WMC

7.2 Correlation Coefficient

Within this section, we calculate the correlation coefficient between WMC and SLOC(L). Utilizing separate histograms for WMC values and SLOC(L) values, we observe that these values do not adhere to a normal distribution. The non-normal distribution of the x-values (WMC) and y-values becomes evident in Figure 7.2 and 7.3. Given the non-normal distribution of these values, the Spearman's Rank Correlation Coefficient (rs) is employed to determine the correlation coefficient. Spearman's Rank Correlation Coefficient (rs) serves as a measure of association for attribute values that deviate from a normal distribution, as indicated in the Lecture Slides. To compute rs, the data in each set of WMC (x) and SLOC(L) (y) are individually ranked in ascending order by rank (xi) and rank (yi). Let n represent the number of pairs of (x, y), and di denote the difference between the ranks (xi) and (yi). The formula for rs is then given by:

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

The value of n is 8 for this particular system. All calculations are done based on the values from table 7.2. 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. For the system of METRICSTICS the coefficient value is 0.378 which shows a strong correlation between WMC and SLOC(L). The value of coefficient suggests that in case the logical SLOC increases there is a high chance WMC will increase too. But conclusively we cannot say

$WMC(x_i)$	Rank(x_i)	SLOC(L)(y_i)	Rank(y_i)	d	d^2
8	1	41	3	-2	4
9	2	28	1	1	1
10	3	27	2	1	1
12	4	192	8	-4	16
14	5	56	4	1	1
30	6	137	6	0	0
38	7	145	7	0	0
44	8	118	5	3	9

Table 7.2: Data Table

only SLOC has an impact on WMC or vice versa but many other underlying factors may also have an effect on these values. It is important to take in account all metrics of software to make justification of change.

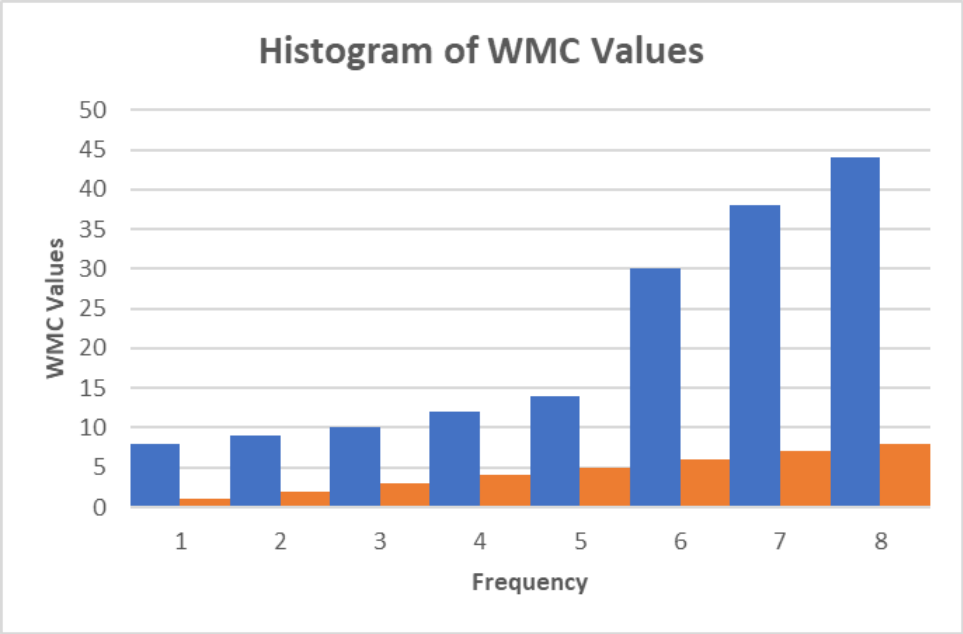


Figure 7.2: Correlation Coefficient (WMC)

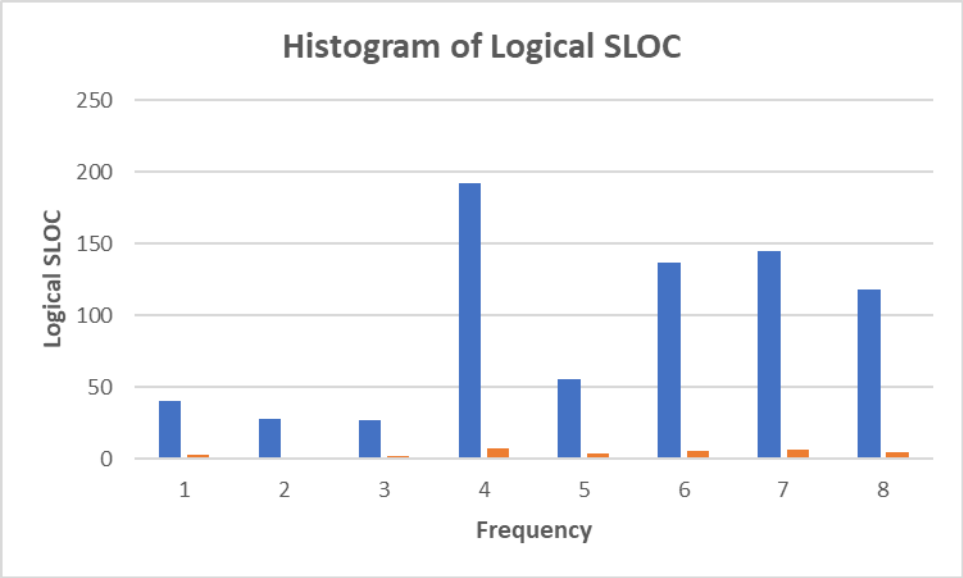


Figure 7.3: Correlation Coefficient (Logical SLOC)

References

[1] V. R. Basili, G. Caldiera, and H. D. Rombach, *The Goal Question Metric Approach*, in Encyclopedia of Software Engineering, vol. 2, pp. 528–532, 1994.

[2] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley Professional, 2000.

[3] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Co., 1996.

[4] ISO/IEC, *ISO/IEC 9126: Software engineering — Product quality — Part 1: Quality model*, International Organization for Standardization, 1999.