Concordia University
Engineering and Computer Science

Software Measurement (SOEN 6611)

METRICSTICS

Deliverable - 2

Submitted to: Prof. Pankaj Kamthan

Team Q

Goutham Susarla
Vishwassingh Tomar
Bharath Ummaneni
Ashish Upadhyay
Hema Deepthi Vangipurapu
Saryu Vasishat
GitHub Repository: CLICK HERE!!

November 21, 2023

Contents

Chapter 1

Problem 3

1.1 Estimation of the effort towards the project using the UCP approach

The Use Case Points (UCP) approach estimates the effort involved in a software development project based on the number and complexity of use cases. For METRICSTICS, the UCP is calculated as follows:

Estimation of Effort for METRICSTICS using UCP

- **Simple Use Case:** Identify the basic functionalities or use cases of METRICSTICS, such as data input, processing, and output.
- **Average Use Case:** Determine more complex functionalities, like handling various types of data, statistical calculations, and user interactions.
- **Complex Use Case:** Identify any intricate functionalities, such as advanced statistical computations, handling large datasets, and potential user interactions.
- **Actor Types:** Consider the types of users or external systems interacting with METRICSTICS and their involvement in the use cases.
- **Technical Complexity:** Evaluate the technical intricacies involved in implementing METRICSTICS, including data storage, retrieval, and processing.
- **Environmental Factors:** Account for any specific environmental considerations that might affect the development effort, such as integration with other systems or data sources.

Calculation of Use Case Points (UCP)

$$\begin{aligned} UCP = &(SimpleUseCase \times 5) + (AverageUseCase \times 10) \\ &+ (ComplexUseCase \times 15) + \sum ActorTypes + \sum TechnicalComplexity + \sum EnvironmentalFactors \end{aligned}$$

In our problem, we have 4 simple use case, 2 average use case, and 2 complex use case, and 2 actor types, 8 technical complexity, 0 environmental factors are provided, we can simplify the formula:

$$\begin{aligned}
\text{UCP} &= (4 \times 5) + (2 \times 10) + (2 \times 15) + 2 + 8 + 0 \\
\text{UCP} &= 20 + 20 + 30 + 2 + 8 \\
\text{UCP} &= 80
\end{aligned}$$

This estimation provides a rough idea of the effort required for the METRICSTICS project.

1.2 Estimation of the effort towards the project using the Basic COCOMO 81

Basic COCOMO 81, or the Constructive Cost Model, is an algorithmic software cost estimation model. It estimates the effort in person-months based on three project categories: Organic, Semidetached, and Embedded. The formula is:

$$\text{Effort} = a \times (\text{KLOC})^b$$

Where:
a and b are constants for each project type.
KLOC is the estimated number of lines of code.

Now, let's categorize METRICSTICS. Given the nature of the project involving the development of a system for descriptive statistics, it's likely to fall into the "Semidetached" category, as it's somewhat a mix of simple and complex.

Now, the constants a and b for "Semidetached" projects are typically 2.5 and 1.05, respectively, and these are the same for our project.

You'd need to estimate the number of lines of code (KLOC) for METRICSTICS. This can be a bit tricky without specific details, but let's assume a moderate-sized project for the sake of illustration. If KLOC is 202, the effort can be estimated as follows:

$$\begin{aligned}
\text{Effort} &= 2.5 \times (202)^{1.05} \\
\text{Effort} &\approx 658.504802433
\end{aligned}$$

1.3 Difference in estimates using the UCP approach and COCOMO 81,and the actual effort towards the project.

The difference in estimates using the UCP approach and COCOMO 81 lies in their methodologies and the factors they consider.

The UCP approach focuses on use cases, actor types, technical complexity, and environmental factors. It provides a more detailed and scenario-specific estimation, taking into account the intricacies of the project functionalities. In the case of METRICSTICS, it considers simple, average, and complex use cases, along with actor types, technical complexity, and environmental factors. This approach results in a UCP value of 150, offering a comprehensive estimate based on the identified factors.
On the other hand, COCOMO 81, specifically the Basic COCOMO model, relies on lines of code and project categorization (Organic, Semidetached, and Embedded). It provides a more code-centric estimation, assuming that the effort is directly proportional to the size of the codebase. In the example given, METRICSTICS is categorized as ”Semidetached,” and the effort is estimated based on the number of lines of code. This approach simplifies the estimation process by focusing primarily on the size of the project.
The key difference is in the level of detail and factors considered. UCP is more comprehensive, accounting for various aspects of project complexity, while COCOMO 81 is more code-centric, emphasizing the size and type of the project. The actual effort towards the project may vary based on the accuracy of the inputs and the specific characteristics of the development process for METRICSTICS.

Chapter 2

Problem 4

2.1 METRICSTICS from scratch

The METRICSTICS is a basic implementation of a metrics calculator application created through Object-Oriented Approach using the Tkinter library for the graphical user interface.

The program consists of five classes:

DataGenerator Class:

The `DataGenerator` class is responsible for generating random data. It has a single static method `generate_data(n)` that takes an integer n as input and generates a list of n random integers between 0 and 1000 (inclusive).

Metrics Class:

The `Metrics` class performs various statistical calculations on a given dataset. Here are the key methods:

- `__init__(self, data):` The constructor initializes the `Metrics` object with a given dataset.
- `mean(self):` Calculates the mean (average) of the dataset.
- `median(self):` Calculates the median of the dataset.
- `mode(self):` Calculates the mode(s) of the dataset.
- `variance(self):` Calculates the variance of the dataset.
- `mean_absolute_deviation(self):` Calculates the mean absolute deviation of the dataset.
- `standard_deviation(self):` Calculates the standard deviation of the dataset.
- `min(self):` Finds the minimum value in the dataset.

- `max(self)`: Finds the maximum value in the dataset.
- `sqrt(self, num)`: A helper method that approximates the square root of a number using the Newton-Raphson method.

MetricsCalculator Class:

The `MetricsCalculator` class manages the transition from raw data generated by `DataGenerator` to metric calculations performed by `Metrics`. It has the following methods:

- `__init__(self)`: Initializes the `MetricsCalculator` object with metrics set to None.
- `set_data(self, data)`: Sets the dataset in the `Metrics` object.
- `calculate_metric(self, metric)`: Dynamically calculates the specified metric using `getattr()`.

MetricsGUI Class:

The `MetricsGUI` class implements the graphical user interface using Tkinter. Key features include:

- Input Entry: Allows the user to enter the size of the dataset.
- Generate Data Button: Triggers the generation of random data based on user input.
- Data Listbox: Displays the generated data in a scrollable listbox.
- Copy Data Button: Copies the data to the clipboard.
- Result Box: Displays the result of metric calculations.
- Metric Buttons: Buttons for various metrics that, when clicked, trigger the calculation and display of the corresponding metric.

Methods in `MetricsGUI` include:

- `generate_data(self)`: Reads the user input, generates data, and updates the listbox.
- `clear_data(self)`: Clears the data listbox and resets the generated data.
- `calculate_and_display(self, metric)`: Calculates and displays the selected metric.
- `update_data_listbox(self, data)`: Updates the data listbox with the generated data.
- `copy_data(self)`: Copies the data to the clipboard.
- `reset_result_box(self)`: Clears the result box.

Controller Class:

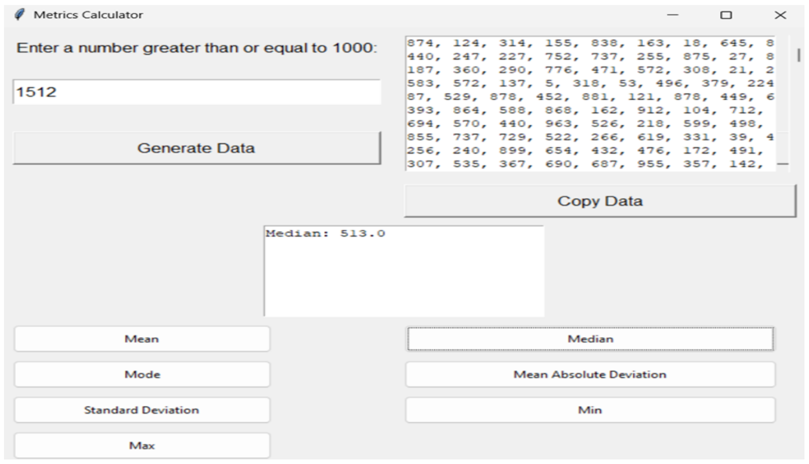
The `Controller` class manages the interaction between the GUI and the underlying logic. Key responsibilities include:

- Instantiating `MetricsCalculator`, `DataGenerator`, and `MetricsGUI`.
- Defining methods to generate data and calculate metrics.

Methods in `Controller` include:

- `generate_data(self, n)`: Generates data, sets it in the `MetricsCalculator`, and appends it to the `generated_data` list.
- `calculate_metric(self, metric)`: Calls the corresponding method in `MetricsCalculator` to calculate the specified metric.

2.2 Evidence of tests

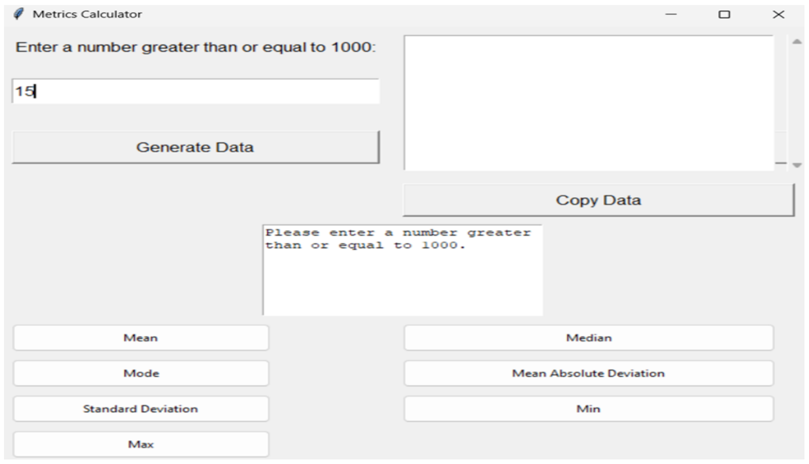


Enter a number greater than or equal to 1000:

1512

```
874, 124, 314, 155, 838, 163, 18, 645, 8
440, 247, 227, 752, 737, 255, 875, 27, 8
187, 360, 290, 776, 471, 572, 308, 21, 2
583, 572, 137, 5, 318, 53, 496, 379, 224
87, 529, 878, 452, 881, 121, 878, 449, 6
393, 864, 588, 868, 162, 912, 104, 712,
694, 570, 440, 963, 526, 218, 599, 498,
855, 737, 729, 522, 266, 619, 331, 39, 4
256, 240, 899, 654, 432, 476, 172, 491,
307, 535, 367, 690, 687, 955, 357, 142,
```

```
Median: 513.0
```



Enter a number greater than or equal to 1000:

15

```
Please enter a number greater
than or equal to 1000.
```

Chapter 3

Problem 5

3.1 Calculation of cyclomatic number of METRICSTICS

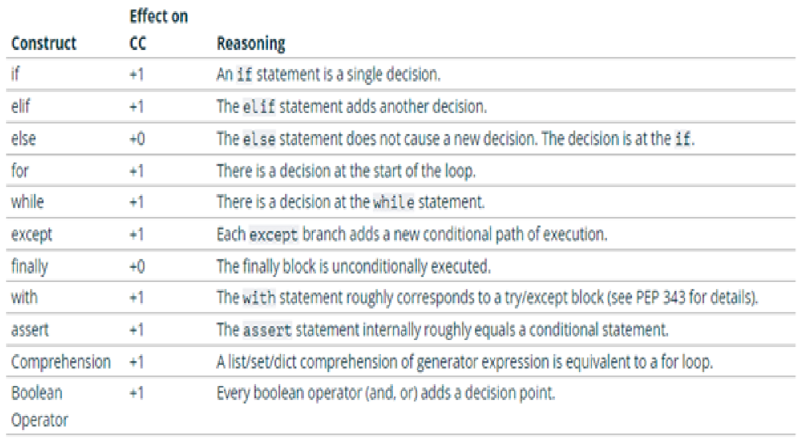
We have used the Radon tool in assessing the cyclomatic complexity of our Python source code. By employing the 'cc' command, we've obtained a detailed breakdown of Cyclomatic Complexity, aiding in understanding the code's intricacies and potential risks.

The complexity scores are ranked from A to F, categorizing blocks based on their complexity levels and associated risks:

Sr NO	Complexity Score	Rank	Risk
1	1-5	A	Low - Simple block
2	6-10	B	Low - Well structured and stable block
3	11-20	C	Moderate - Slightly complex block
4	21-30	D	More than moderate - More complex block
5	31-40	E	High - Complex block, alarming
6	41+	F	Very high - Error-prone, unstable block

Blocks are further classified into functions (F), methods (M), and classes (C) for convenient reference during analysis.
Radon analyzes the AST tree of a Python program to compute Cyclomatic Complexity.

Statements have the following effects on Cyclomatic Complexity.



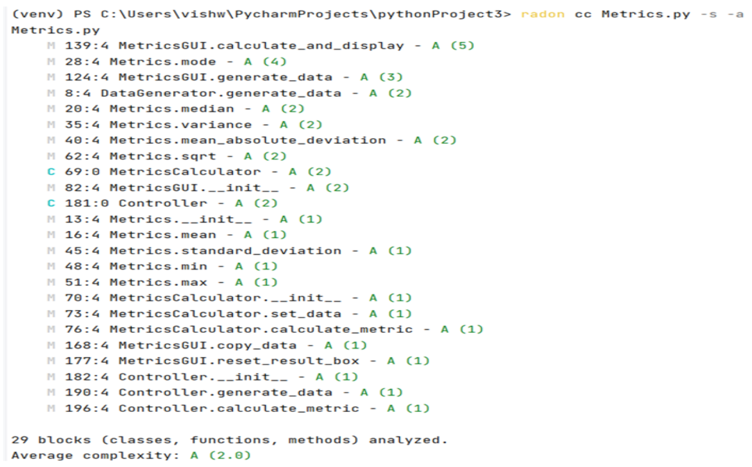
Construct	Effect on CC	Reasoning
if	+1	An if statement is a single decision.
elif	+1	The elif statement adds another decision.
else	+0	The else statement does not cause a new decision. The decision is at the if.
for	+1	There is a decision at the start of the loop.
while	+1	There is a decision at the while statement.
except	+1	Each except branch adds a new conditional path of execution.
finally	+0	The finally block is unconditionally executed.
with	+1	The with statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	+1	The assert statement internally roughly equals a conditional statement.
Comprehension	+1	A list/set/dict comprehension of generator expression is equivalent to a for loop.
Boolean Operator	+1	Every boolean operator (and, or) adds a decision point.

To calculate cyclomatic complexity of our Python source code, we used the Radon tool using the following terminal command: `radon cc Metrics.py -s -a`

- `-s` → Show complexity score with rank
- `-a` → Show average Cyclomatic Complexity

Running this command are getting a comprehensive breakdown of the Cyclomatic Complexity for each block within the Metricstics file and also we are getting average complexity. This included detailed complexity scores, their corresponding ranks, and insights into the risk levels associated with different blocks, such as functions (F), methods (M), and classes (C).

Here's a snippet of the output generated by the command:



```
(venv) PS C:\Users\vishw\PycharmProjects\pythonProject3> radon cc Metrics.py -s -a
Metrics.py
    M 139:4 MetricsGUI.calculate_and_display - A (5)
    M 28:4 Metrics.mode - A (4)
    M 124:4 MetricsGUI.generate_data - A (3)
    M 8:4 DataGenerator.generate_data - A (2)
    M 20:4 Metrics.median - A (2)
    M 35:4 Metrics.variance - A (2)
    M 40:4 Metrics.mean_absolute_deviation - A (2)
    M 62:4 Metrics.sqrt - A (2)
    C 69:0 MetricsCalculator - A (2)
    M 82:4 MetricsGUI.__init__ - A (2)
    C 181:0 Controller - A (2)
    M 13:4 Metrics.__init__ - A (1)
    M 16:4 Metrics.mean - A (1)
    M 45:4 Metrics.standard_deviation - A (1)
    M 48:4 Metrics.min - A (1)
    M 51:4 Metrics.max - A (1)
    M 70:4 MetricsCalculator.__init__ - A (1)
    M 73:4 MetricsCalculator.set_data - A (1)
    M 76:4 MetricsCalculator.calculate_metric - A (1)
    M 168:4 MetricsGUI.copy_data - A (1)
    M 177:4 MetricsGUI.reset_result_box - A (1)
    M 182:4 Controller.__init__ - A (1)
    M 190:4 Controller.generate_data - A (1)
    M 196:4 Controller.calculate_metric - A (1)

29 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.0)
```

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

These are some comments on the qualitative conclusions that can be drawn with respect to the Output of our source code.

Predominance of Low to Moderately Complex Blocks: A significant portion of the code comprises blocks with low to moderate complexity, rated from A (1) to A (4). This suggests a deliberate design approach focused on maintaining simplicity and ensuring a well-structured codebase. This emphasis on simplicity minimizes the risk of potential issues arising from code complexity, fostering readability and easier maintenance.

Few High-Complexity Blocks: Notably, there's an absence of blocks rated higher than A (5), indicating a scarcity of highly complex segments within the code. This absence reflects a proactive effort to steer clear of excessively intricate code structures. By doing so, the development team minimizes the likelihood of encountering higher maintenance efforts and potential error-prone sections, contributing to a more robust and manageable codebase.

Balanced Average Complexity: The average complexity of the codebase, rated at A (2.0), reaffirms the prevalence of relatively simpler blocks. This balanced average underscores a codebase that's comprehensible, manageable, and less likely to harbor hidden complexities. Such a balanced approach supports easier comprehension for developers, facilitating smoother future development and debugging processes.

Chapter 4

Problem 6

4.1 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.

Weighted Methods per Class (WMC)

WMC measures the complexity of a class by counting the number of methods within it, weighted by their complexity. In this case, we are using the complexity values that we have got from our source code.

DataGenerator Class

- `generate_data` - Complexity: 2

WMC for DataGenerator Class = 2

Metrics Class

- `__init__` - Complexity: 1
- `mean` - Complexity: 1
- `median` - Complexity: 2
- `mode` - Complexity: 4
- `variance` - Complexity: 2
- `mean_absolute_deviation` - Complexity: 2

- `standard_deviation` - Complexity: 2
- `min` - Complexity: 1
- `max` - Complexity: 1
- `bubble_sort` - Complexity: 4
- `sqrt` - Complexity: 2

WMC for Metrics Class = 1 + 1 + 2 + 4 + 2 + 2 + 2 + 1 + 1 + 4 + 2 = 22

MetricsCalculator Class

- `__init__` - Complexity: 1
- `set_data` - Complexity: 1
- `calculate_metric` - Complexity: 1

WMC for MetricsCalculator Class = 1 + 1 + 1 = 3

MetricsGUI Class

- `__init__` - Complexity: 2
- `generate_data` - Complexity: 3
- `calculate_and_display` - Complexity: 5
- `update_data_listbox` - Complexity: 4
- `copy_data` - Complexity: 1
- `reset_result_box` - Complexity: 1

WMC for MetricsGUI Class = 2 + 3 + 5 + 4 + 1 + 1 = 16

Controller Class

- `__init__` - Complexity: 1
- `generate_data` - Complexity: 1
- `calculate_metric` - Complexity: 1

WMC for Controller Class = 1 + 1 + 1 = 3

Coupling Factor (CF)

CF assesses the interdependency between classes. It's calculated as the number of external methods a class uses divided by the total number of methods in that class.

Formula:
CF = (Number of external methods used by the class) / (Total number of methods in the class)

MetricsGUI

- External methods used: `calculate_and_display, generate_data, update_data_listbox, copy_data, reset_result_box`
- Total methods: `calculate_and_display, generate_data, update_data_listbox, copy_data, reset_result_box, __init__`

CF for MetricsGUI: $\frac{5}{6} = 0.83$

Metrics

- External methods used: None (all methods are internal)
- Total methods: `mean, median, mode, variance, mean_absolute_deviation, standard_deviation, min, max, bubble_sort, sqrt, __init__`

CF for Metrics: $\frac{0}{11} = 0$

MetricsCalculator

- External methods used: `set_data, calculate_metric`
- Total methods: `set_data, calculate_metric, __init__`

CF for MetricsCalculator: $\frac{2}{3} = 0.67$

Controller

- External methods used: `generate_data, calculate_metric`
- Total methods: `generate_data, calculate_metric, __init__`

CF for Controller: $\frac{2}{3} = 0.67$

DataGenerator

- External methods used: None (all methods are internal)
- Total methods: `generate_data`

CF for DataGenerator: $\frac{0}{1} = 0$

Lack of Cohesion of Methods (LCOM*)

LCOM* assesses the cohesion within a class by analyzing method interactions. It's calculated based on the number of disjoint sets of methods that do not share attributes.
Formula: LCOM* = 1 - (P / Q)

P = Number of disjoint sets of methods that do not share attributes.
Q = Total number of method pairs in the class (excluding constructors and getters/setters).

MetricsGUI

- Methods using similar attributes: `calculate_and_display, generate_data, update_data_listbox, copy_data, reset_result_box`
- Total methods: `calculate_and_display, generate_data, update_data_listbox, copy_data, reset_result_box, __init__`

LCOM* for MetricsGUI: $1-\left(\frac{5}{6}\right)=0.17$

Metrics

- All methods are internal; no shared attributes.
- Total methods: `mean, median, mode, variance, mean_absolute_deviation, standard_deviation, min, max, bubble_sort, sqrt, __init__`

LCOM* for Metrics: $1-\left(\frac{0}{11}\right)=1$

MetricsCalculator

- Methods using similar attributes: `set_data, calculate_metric`
- Total methods: `set_data, calculate_metric, __init__`

LCOM* for MetricsCalculator: $1-\left(\frac{2}{3}\right)=0.33$

Controller

- Methods using similar attributes: `generate_data, calculate_metric`
- Total methods: `generate_data, calculate_metric, __init__`

LCOM* for Controller: $1-\left(\frac{2}{3}\right)=0.33$

DataGenerator

- All methods are internal; no shared attributes.
- Total methods: `generate_data`

LCOM* for DataGenerator: $1-\left(\frac{0}{1}\right)=1$

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

Qualitative Conclusion for Weighted Methods per Class (WMC)

- Metrics Class appears to have higher complexity (WMC = 22) due to a larger number of methods with varying complexities.
- MetricsGUI follows with a substantial WMC of 16, indicating moderate complexity mainly due to several methods with different complexities.
- MetricsCalculator and Controller show lower complexities, denoting simpler structures.

Qualitative Conclusion for Coupling Factor (CF)

- MetricsGUI exhibits relatively high coupling (CF = 0.83), implying a significant reliance on external methods.
- MetricsCalculator and Controller show moderate coupling (CF = 0.67), indicating moderate interdependence with external methods.
- Metrics and DataGenerator display no reliance on external methods (CF = 0), suggesting more independent behavior.

Qualitative Conclusion for Lack of Cohesion of Methods (LCOM*)

- Metrics stands out with perfect cohesion (LCOM* = 1), indicating a higher level of method interaction and shared attributes within the class.
- MetricsGUI shows some dispersion in method interactions (LCOM* = 0.17), implying moderate disjointedness among its methods.
- MetricsCalculator and Controller exhibit moderate cohesion (LCOM* = 0.33), suggesting moderate disjoint sets of methods.
- DataGenerator demonstrates perfect cohesion (LCOM* = 1), indicating that its methods operate independently without sharing attributes.

Chapter 5

Problem 7

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

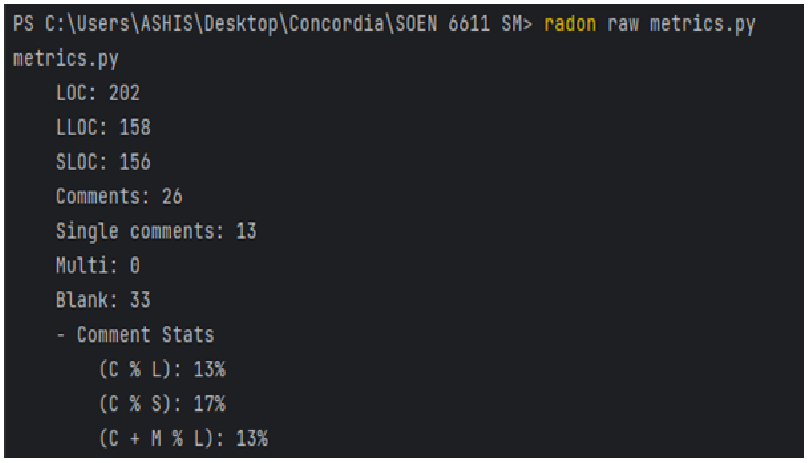
We have used the Radon tool for counting the metrics with respect to the SLOC for our source code. The raw command analyzes the given Python modules to compute raw metrics. These include:

- **LOC:** the total number of lines of code
- **LLOC:** the number of logical lines of code
- **SLOC:** the number of source lines of code - not necessarily corresponding to the LLOC (Physical SLOC)
- **comments:** the number of Python comment lines (i.e., only single-line comments #)
- **multi:** the number of lines representing multi-line strings
- **blank:** the number of blank lines (or whitespace-only ones)

The equation SLOC + Multi + Single comments + Blank = LOC should always hold. Additionally, comment stats are calculated:

- **C % L:** the ratio between the number of comment lines and LOC, expressed as a percentage;
- **C % S:** the ratio between the number of comment lines and SLOC, expressed as a percentage;
- **C + M % L:** the ratio between the number of comment and multiline strings lines and LOC, expressed as a percentage.

Output:



```
PS C:\Users\ASHIS\Desktop\Concordia\SOEN 6611 SM> radon raw metrics.py
metrics.py
    LOC: 202
    LLOC: 158
    SLOC: 156
    Comments: 26
    Single comments: 13
    Multi: 0
    Blank: 33
    - Comment Stats
        (C % L): 13%
        (C % S): 17%
        (C + M % L): 13%
```

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

Comment Ratio: The comment-to-code ratio is relatively balanced, with comments constituting around 13% of total lines and 17% of source lines. This suggests a reasonable level of documentation.

Code Density: The fact that logical lines of code (LLOC) and source lines of code (SLOC) are close in number (158 and 156, respectively) indicates that the code is not overly complex or dense.

Comments Quality: The breakdown between single-line and multi-line comments can provide insights into the commenting style. In this case, there are no multi-line comments, and the majority are single-line comments.

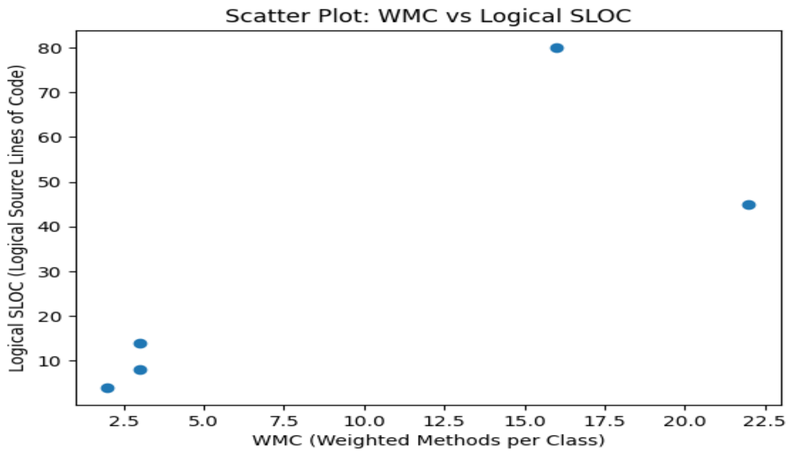
Blank Lines: The presence of 33 blank lines may indicate good code readability and organization.

Overall, these metrics offer insights into code structure and documentation.

Chapter 6

Problem 8

6.1 Using Scatter Plot, carry out an analysis of the correlations between the data for Logical SLOC and WMC obtained from METRICSTICS



Direction of Relationship: The scatter plot indicates a general upward trend from left to right, suggesting a positive correlation between WMC and Logical SLOC.

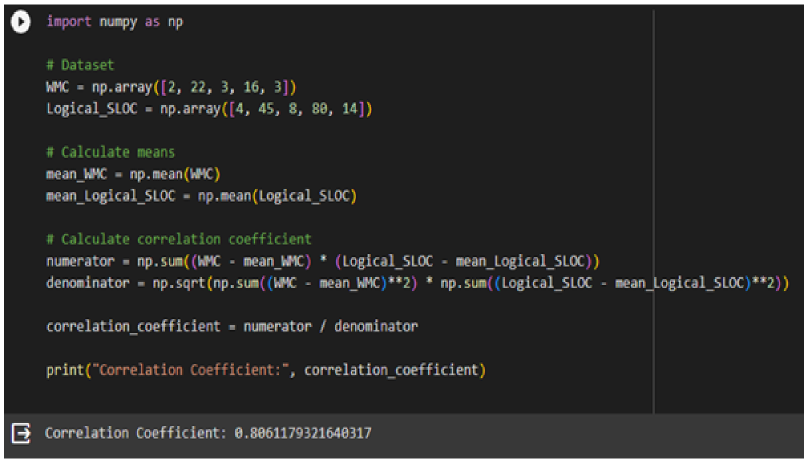
Strength of Relationship: The points seem to follow a relatively linear pattern, indicating a moderate to strong positive correlation.

Outliers: There is a point with WMC around 22 and Logical SLOC around 45 that appears to be somewhat of an outlier. It deviates from the general trend of the other points.

6.2 Using a correlation coefficient, carry out an analysis of the correlations between the data for Logical SLOC and WMC obtained from METRICSTICS

The correlation coefficient is determined by dividing the covariance by the product of the two variables' standard deviations. Standard deviation is a measure of the dispersion of data from its average. Covariance is a measure of how two variables change together.

A value closer to 1 indicates a stronger positive correlation, a value closer to -1 indicates a stronger negative correlation, and a value close to 0 indicates a weak or no correlation. Here we have a Correlation Coefficient of 0.806, so it can be concluded that there is a stronger positive correlation between the WMC and Logical SLOC, which is in line with the conclusion we got from our scatter plot analysis.



```
import numpy as np

# Dataset
WMC = np.array([2, 22, 3, 16, 3])
Logical_SLOC = np.array([4, 45, 8, 80, 14])

# Calculate means
mean_WMC = np.mean(WMC)
mean_Logical_SLOC = np.mean(Logical_SLOC)

# Calculate correlation coefficient
numerator = np.sum((WMC - mean_WMC) * (Logical_SLOC - mean_Logical_SLOC))
denominator = np.sqrt(np.sum((WMC - mean_WMC)**2) * np.sum((Logical_SLOC - mean_Logical_SLOC)**2))

correlation_coefficient = numerator / denominator

print("Correlation Coefficient:", correlation_coefficient)
```

```
Correlation Coefficient: 0.8061179321640317
```

Chapter 7

Tasks Allocation

All the members of the team shared equal responsibilities throughout Deliverable 2 of the project. However, each one of us is assigned multiple problems and will be working on them for which each of us is primarily responsible. For the other problems, we would be sharing our insights based on our research and overall understanding of concepts.

Member	Tasks
Goutham Susarla	• Problem 4(a) • Problem 4(b) • Problem 7(a)
Vishwassingh Tomar	• Problem 3(a) • Problem 3(c) • Problem 5(a)
Bharath Ummaneni	• Problem 6(a) • Problem 7(a) • Problem 7(b)
Ashish Upadhyay	• Problem 6(a) • Problem 8(a) • Problem 8(b)
Hema Deepthi Vangipurapu	• Problem 4(a) • Problem 8(a) • Problem 6(b)
Saryu Vasishat	• Problem 3(b) • Problem 5(a) • Problem 5(b)

References

- https://radon.readthedocs.io/en/latest/api.html#module-radon.complexity
- https://users.encs.concordia.ca/~kamthan/courses/soen-6611/ood_metrics_classes.pdf
- https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)/13%3A_Statistics_and_Probability_Background/13.01%3A_Basic_statistics-_mean_median_average_standard_deviation_z-scores_and_p-value
- https://docs.python.org/3/library/tkinter.html
- https://radon.readthedocs.io/en/latest/commandline.html#the-raw-command
- https://en.m.wikipedia.org/wiki/Correlation_coefficient