



SOEN 6611

DELIVERY 2

Team J

Quoc Phong Ngo
SanVinoth Pacham
Preet Angad Singh
Siddharth Bharat Oza
Nomesh Palakaluri

November 23, 2023

[Github Link](#)

Contents

1	Background Information	4
2	Problem 3: Use Case Points(UCP)	5
2.1	Using the use case points (UCP) approach (or one of its extensions), provide an estimate of the effort towards the project.	5
2.1.1	Introduction	5
2.1.2	Effort Estimate using Use Case Point approach	5
2.2	Using Basic COCOMO 81, provide an estimate of the effort towards the project.	9
2.3	Comment on the difference in estimates using the UCP approach and CO-COMO 81, and the actual effort towards the project.	9
2.4	Actual Project Effort:	11
3	Problem 4: Implementation of Project METRICSTICS	12
4	Problem 5: Cyclomatic number	15
4.1	Cyclomatic number	15
4.2	Conclusions for each classes:	18
5	Problem 6: WMC, LF, LSOC	19
5.1	WMC:	19
5.2	Comments of WMC:	20
5.3	CF:	21
5.4	LCOM:	21
6	Problem 7: Physical And Logical SLOC	26
6.1	Goal:	26
6.2	Physical SLOC:	26
6.3	Logical SLOC:	26
6.3.1	Rules for counting Physical SLOC and Logical SLOC	28
7	Problem 8	29
8	Work Distribution	32
9	References	32

Symbols and Abbreviations

GQM	Goal Metric Question
UC	Use Case
HRM	Human Resource Management
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

3.1 METRICSTICS Application.	11
3.2 Code Sample	12
3.3 Python Structure Classes.	13
4.1 Cyclometric Complexity of Metristics clas	16
4.2 Cyclometric Complexity of MetristicsFrame class.	16
4.3 Cyclometric Complexity of DataGenerator class	17
4.4 Cyclometric Complexity of LoginWindow clas.	17
4.5 Cyclometric Complexity of App class	17
5.1 Overall dependencies of the System.	24
6.1 Physical and Logical SLOC	26

1 Background Information

The development of the Healthcare Workers Analytics Platform, referred to as METRICSTICS, will entail the creation of a pivotal system aimed at providing a holistic analysis of healthcare workforce performance. This system is envisioned to conduct robust statistical examinations of various workforce metrics, thus equipping healthcare management with valuable tools to effectively oversee trends in staffing and optimize workforce strategies. Access to METRICSTICS will be extended to healthcare personnel, including both frontline healthcare workers and administrative staff, fostering collaborative data input processes for the former and seamless access to statistical insights for the latter. Furthermore, METRICSTICS will facilitate the generation of comprehensive reports, which will be presented periodically to the governing board, offering critical insights for informed decision-making.

Remark: The primary stakeholders for METRICSTICS encompass healthcare workers, including frontline staff and administrative personnel, as well as healthcare management and governing authorities.

2 Problem 3: Use Case Points(UCP)

2.1 Using the use case points (UCP) approach (or one of its extensions), provide an estimate of the effort towards the project.

2.1.1 Introduction

The relationship between use cases and effort in software development is that use cases help define the functional requirements of a system and serve as a basis for estimating the effort (time, resources) required to develop the software. The complexity and granularity of use cases can influence the accuracy of effort estimation, and they also assist in prioritizing tasks, communication, and risk management throughout the project.

2.1.2 Effort Estimate using Use Case Point approach

These are a few steps required for the effort estimation process

1. Determine and Calculate Unadjusted Use Case Points (UUCP).
2. Determine and Calculate Technical Complexity Factor (TCF).
3. Determine and Calculate Environment Complexity Factor (ECF).
4. Determine Productivity Factor (PF).
5. Calculate Use Case Points (UCP).
6. Calculate the Estimated Number of Person-Hours.

The **effort estimate** is determined by multiplying the Use Case Points by the Productivity Factor.

$$\text{Effort Estimate} = \text{UCP} * \text{PF} \quad (3.1)$$

The **UCP** equation can be expressed as follows:

$$\text{UCP} = \text{UUCP} * \text{TCF} * \text{ECF} \quad (3.2)$$

UUCP is determined by adding the Unadjusted Actor Weight (UAW), reflecting actor complexity, and the Unadjusted Use Case Weight (UUCW), representing the total scenario steps.

The formula for UUCP is:

$$\text{UUCP} = \text{UAW} + \text{UUCW} \quad (3.3)$$

The Unified Function Point (UCP) calculation recognizes that not all actors in a use case model are equal in terms of complexity. There are three types of actors:

- Simple Actor
- Average Actor
- Complex Actor

In summary, a simple actor is a machine, an average actor can be either a human or a machine, and a complex actor is a human. These distinctions help in assessing actor complexity in UCP calculations.

In our use case model, we have a single actor, the user, who interacts with the system through a graphical user interface. Consequently, this actor is classified as a **complex actor**, and it is assigned a weight of 3 points, resulting in:

$$UAW = 3$$

We have x classes in the system.so, it comes under average use case.

$$UUCW=10$$

From 3.3:

$$UUCP=UAW + UUCW$$

$$UUCP = 3 + 10$$

$$UUCP = 13$$

TCF is a multiplier that assesses the impact of technical and environmental factors on a software project's complexity. It's used to adjust function points and estimate development effort, aiding in project planning. It consists of 13 different factors, each with there own weight.

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

Table 2.1 The Technical Complexity Factors in the UCP approach.

The equation is given by:

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

C1=0.6 and C2=0.01, W_{Ti} is Technical Complexity Factor Weight, F_i is Perceived Impact Factor.

F_i is decided based on the values of:

- No influence = 0 • Average influence = 3 • Strong influence = 5

In our case,

- No Influence - T1,T10,T11,T12,T13
- Average Influence - T2,T3,T4
- Strong Influence - T5,T6,T7,T8,T9

$$TCF=0.6+(0.01*((2*0)+(1*3)+(1*3)+(1*3)+(1*5)+(0.5*5)+(0.5*5)+(2*5)+(1*5)+(1*0)+(1*0)+(1*0)+(1*0))))$$

$$\text{TCF} = 0.6 + (0.01 \times 44)$$

$$\text{TCF} = 1.04$$

The Environmental Complexity Factor (**ECF**) is a multiplier that quantifies how external and environmental factors affect the complexity of a software project. It's used to adjust function points and estimate development effort, facilitating project planning.

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

Table 2.1 The Environmental Complexity Factors in the UCP approach.

The equation is given by:

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

C1=1.4 and C2=-0.03, WEi is ECF Weight, Fi is Perceived Impact Factor.

For Case: E1, E3, E4, E5, E6, and E8 -

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

For Case: E2 and E7-

- 0-No influence
- 1-Strong Favourable influence
- 3-Average influence

- 5-Strong, Unfavourable influence.

From 2.5:

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

$$ECF = 1.4 - 0.03(9 + 1.5 + 5 + 5 - 1 + 10)$$

$$ECF = 1.4 - 0.03(29.5)$$

$$ECF = 0.515$$

From 2.2:

$$UCP = UUCP * TCF * ECF$$

$$UCP = 13 * 1.04 * 0.515$$

$$UCP = 6.9628$$

From 2.1:

$$\text{Effort Estimate} = UCP * PF$$

$$\text{Effort Estimate} = 6.9628 * 20$$

$$\text{Effort Estimate} = 139.25$$

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

A method for estimating the amount of work needed to produce a software project based on the size of the software is the Basic COCOMO (Constructive Cost Model) 81.

$$E = a \times KLOC^b$$

Let $a = 2.4$ and $b = 1.05$ as we are taking an estimate of an Organic Project.

The estimate number of lines in the code is 500, so $KLOC = 0.5$.

$$E = 2.4 \times (0.5)^{1.05}$$

$$E = 1.15 \text{ PM(Person - Months)}$$

Now counting per hour effort,

$$\text{Hour-Effort} = 1.15 * 22 * 4$$

$$\text{Hour-Effort} = 102.002$$

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

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

UCP Approach Estimate (139.25) Methodology: UCP is a method for estimating software development effort based on the complexity and size of systems, considering factors like use cases and actors.

Calculation Process: UCP involves determining Unadjusted Use Case Points (UUCP) by considering factors such as Technical Complexity Factor (TCF) and Environmental Complexity Factor (ECF).

Effort Estimation: The estimated effort is obtained by multiplying UCP by the Productivity Factor (PF), resulting in a measure expressed in Person-Hours.

Specifics of Estimate (139.25 Person-Hours): The UCP approach for the METRICSTICS application yielded an estimate of 139.25 Person-Hours, representing anticipated effort based on system characteristics, including user interactions and system complexity.

User Interaction and System Functionality: UCP tends to consider user interactions and system functionality in detail, potentially leading to a higher estimated effort in certain cases.

Baseline for Comparison: The UCP estimate serves as a baseline value for comparing different software estimation approaches.

COCOMO 81 Approach Estimate Methodology: COCOMO 81 is a model for estimating software development effort based on project size (KLOC) and project type (Organic, Semi-Detached, or Embedded).

Calculation Process: The COCOMO 81 approach involves estimating project size in Kilo Lines of Code (KLOC) and selecting the project type, with effort calculated using the Basic COCOMO 81 formula.

Effort Estimation: The estimated effort, in the context of COCOMO 81, is determined by considering project size and complexity, resulting in a measure expressed in Person-Hours.

Specifics of Estimate (1.15 Person-Month): The COCOMO 81 approach for the METRICSTICS application yielded an estimate of 1.15 Person-Month, representing anticipated effort based on project size and complexity.

Focus on Project Size and Complexity: COCOMO 81 primarily focuses on project size and complexity as key factors influencing effort estimation.

Baseline for Comparison: The COCOMO 81 estimate serves as a baseline value for comparing different software estimation approaches.

Estimate Difference: The UCP approach, giving detailed consideration to user interactions and system functionality, projected a higher effort of 139.25 Person-Hours for the METRICSTICS application.

Conversely, the COCOMO 81 approach, with its emphasis on project size and complexity, provided a lower estimate of 1.15 Person-Month.

The substantial difference in estimates implies that the UCP approach encompassed additional factors or intricacies not addressed by the COCOMO 81 model in the specific context of the project.

2.4 Actual Project Effort:

Real-Time Development Duration: The METRICSTICS application was developed within a total timeframe of approximately 60-70 hours, a notable reduction compared to both UCP and COCOMO 81 estimates.

Factors Influencing Discrepancy: Various factors contribute to this variance, including the efficiency of the development team, the adoption of contemporary development tools and practices, and a more focused project scope than initially envisioned in theoretical models.

Team Proficiency in Python: The team's expertise in Python and familiarity with the utilized tools and libraries (such as Tkinter) played a significant role in streamlining development and reducing the actual time invested.

Contributions of Effective Project Management: Effective project management, clear requirements, and a streamlined development process further contributed significantly to minimizing the time and effort required for the project.

3 Problem 4: Implementation of Project METRICSTICS

Python is a high-level, popular programming language in today's world. It is known for readability and simplicity. Python syntax is also simple in fewer lines of code compared to other programming languages, making it an ideal choice for beginners. Another strength of Python is that it supports object-oriented programming.

Tkinter is Python's standard GUI toolkit. Tkinter provides a wide range of tools and widgets to create interfaces for the application, including text box, button, menus, and other elements. Furthermore, Tkinter can be used on various operating systems, from MacOS, Windows to Linux. To make GUI simpler, Tkinter allows developers to customize the appearance of widgets using styles. Below image depicts the usage of tkinter in our system

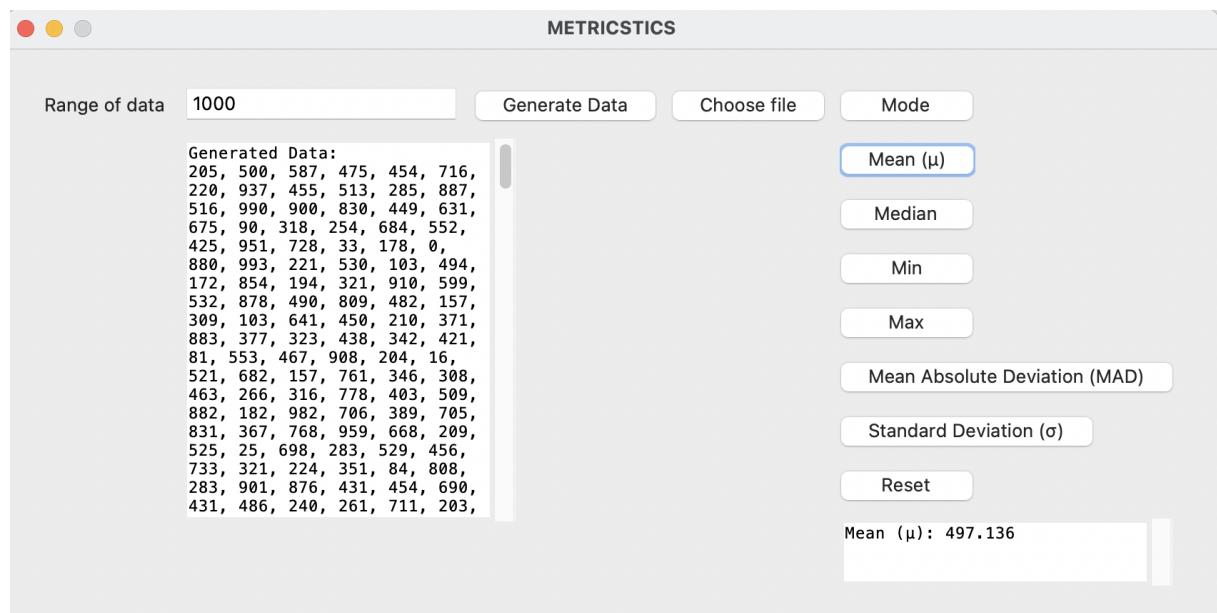
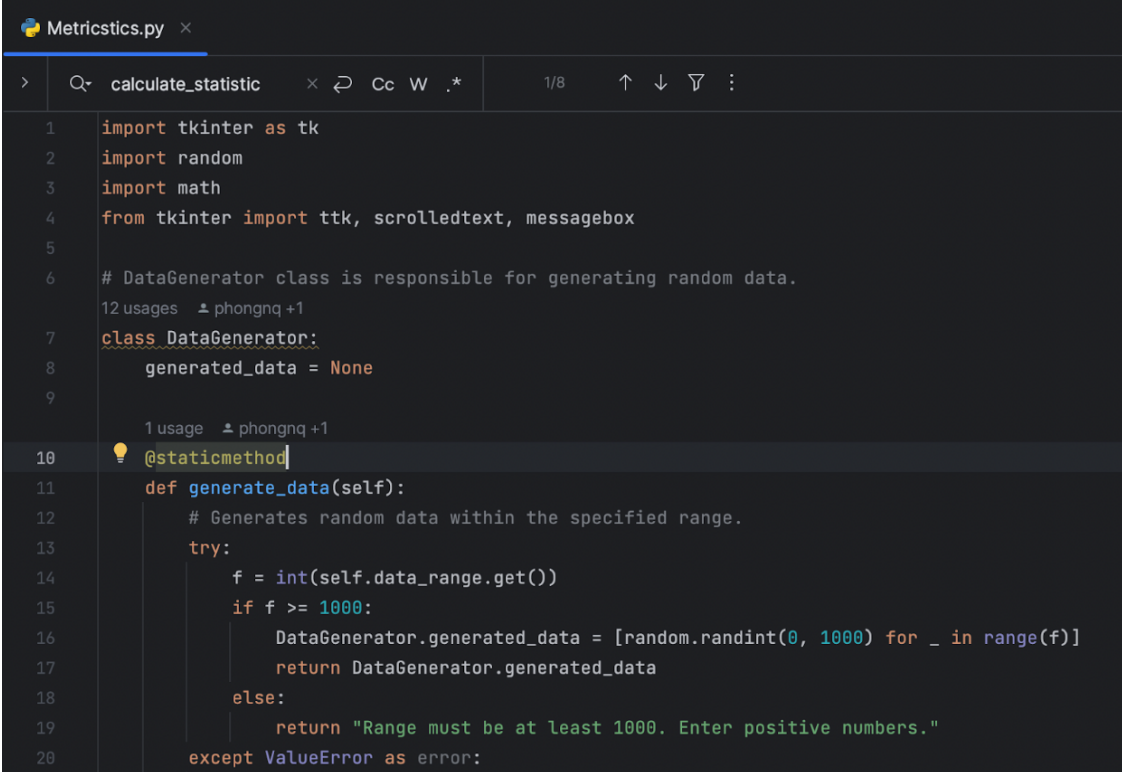


Fig 3.1 METRICSTICS Application

GUI includes these components:

- Range of data input: data that consists of (at least) 1000 values, randomly distributed between 0 and 1000. If users enter the wrong value, an error message will be displayed.
- "Generate data" button: press to generate sample data.
- "Choose file" button: press to upload an external file containing the dataset.
- Mode, Mean, Median, Min, Max, MAD, and Standard Deviation buttons.
- "Reset" button: reset all values in both result text and generated data text.
- Result text: display metric value from selection of users.



```
1 import tkinter as tk
2 import random
3 import math
4 from tkinter import ttk, scrolledtext, messagebox
5
6 # DataGenerator class is responsible for generating random data.
7 12 usages  ↑ phongnq +1
8 class DataGenerator:
9     generated_data = None
10
11     1 usage  ↑ phongnq +1
12     @staticmethod
13     def generate_data(self):
14         # Generates random data within the specified range.
15         try:
16             f = int(self.data_range.get())
17             if f >= 1000:
18                 DataGenerator.generated_data = [random.randint(0, 1000) for _ in range(f)]
19                 return DataGenerator.generated_data
20             else:
21                 return "Range must be at least 1000. Enter positive numbers."
22         except ValueError as error:
```

Fig 3.2 Code sample

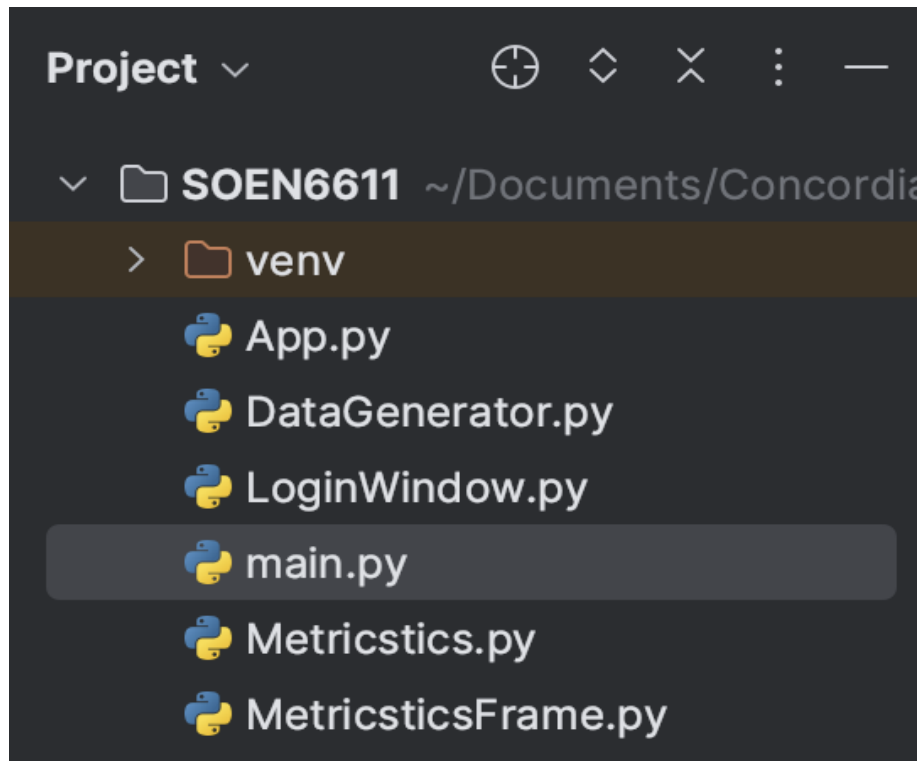


Fig 3.3 Class structure of METRICSTICS

4 Problem 5: Cyclomatic number

4.1 Cyclomatic number

Datagenarator.py

This file contains the **DataGenerator** class, which is responsible for generating random data. A static function, **generate_data**, is included in the class. It receives a data range as input and generates random data within that range.

Metrics.py

The **Metrics** class is included in the **Metrics.py** file and conducts numerous statistical calculations without the need for built-in functions. Minimum, maximum, mean, median, mode, mean absolute deviation, and standard deviation are all calculated.

MetricsFrame.py

This file contains the **MetricsFrame** class, which is the application's main user interface. It generates data and performs statistical calculations using the **DataGenerator** and **Metrics** classes. Based on the tkinter library, the class contains methods for changing UI components and conducting calculations.

LoginWindow.py

Represents a user login pop-up window in a tkinter-based Python application. Here, users are asked to enter credentials based on their role.

App.py

This file defines the **App** class, which represents the primary application window. It extends the **tkinter.Tk** class and provides a **LoginWindow** class that handles user login. The **App** class handles the transition from the login screen to the main program window. This file also defines the application's entry point and primary event loop.

To obtain the cyclomatic complexity for each class and its corresponding values, I utilized a tool known as Radon from python library. This tool aids in analysing the complexity of software code by calculating the cyclomatic number, which provides valuable insights into the code's structural intricacies and potential points of concern.

Here is the Cyclomatic number for each method in the provided Metristics.py:

- Metricstics:

```
PS C:\Users\NOMESH\Desktop\Fall_2023\SOEN6611-main> radon cc Metricstics.py
Metricstics.py
M 45:4 Metricstics.calculate_mode - B
C 4:0 Metricstics - A
M 5:4 Metricstics.calculate_min - A
M 15:4 Metricstics.calculate_max - A
M 25:4 Metricstics.calculate_mean - A
M 34:4 Metricstics.calculate_median - A
M 64:4 Metricstics.calculate_mad - A
M 74:4 Metricstics.calculate_standard_deviation - A
M 84:4 Metricstics.absolute - A
```

Figure 4.1: Cyclometric complexity of Metricstics class

- MetricsticsFrame:

```
PS C:\Users\NOMESH\Desktop\Fall_2023\SOEN6611-main> radon cc MetricsticsFrame.py
MetricsticsFrame.py
M 181:4 MetricsticsFrame.choose_file - A
C 7:0 MetricsticsFrame - A
M 96:4 MetricsticsFrame.calculate_mode - A
M 108:4 MetricsticsFrame.calculate_mean - A
M 120:4 MetricsticsFrame.calculate_median - A
M 132:4 MetricsticsFrame.calculate_min - A
M 144:4 MetricsticsFrame.calculate_max - A
M 156:4 MetricsticsFrame.calculate_mad - A
M 168:4 MetricsticsFrame.calculate_standard_deviation - A
M 234:4 MetricsticsFrame.calculate_statistic - A
M 201:4 MetricsticsFrame.read_and_display_file - A
M 209:4 MetricsticsFrame.generate_data - A
M 218:4 MetricsticsFrame.update_data_text - A
M 226:4 MetricsticsFrame.update_result_text - A
M 8:4 MetricsticsFrame.__init__ - A
M 85:4 MetricsticsFrame.reset_program - A
M 91:4 MetricsticsFrame.clear_text - A
M 246:4 MetricsticsFrame.calculate_mode - A
M 250:4 MetricsticsFrame.calculate_mean - A
M 254:4 MetricsticsFrame.calculate_median - A
M 258:4 MetricsticsFrame.calculate_min - A
M 262:4 MetricsticsFrame.calculate_max - A
M 266:4 MetricsticsFrame.calculate_mad - A
M 270:4 MetricsticsFrame.calculate_standard_deviation - A
```

Figure 4.2: Cyclometric complexity of MetricsticsFrame class

- DataGenerator:

```
PS C:\Users\NOMESH\Desktop\Fall 2023\SOEN6611-main> radon cc DataGenerator.py
DataGenerator.py
  C 5:0 DataGenerator - A
  M 9:4 DataGenerator.generate_data - A
```

Figure 4.3: Cyclometric complexity of DataGenerator class

- LoginWindow:

```
PS C:\Users\NOMESH\Desktop\Fall 2023\SOEN6611-main> radon cc LoginWindow.py
LoginWindow.py
  C 5:0 LoginWindow - A
  M 28:4 LoginWindow.login - A
  M 6:4 LoginWindow.__init__ - A
```

Figure 4.4: Cyclometric complexity of LoginWindow class

- App:

```
PS C:\Users\NOMESH\Desktop\Fall 2023\SOEN6611-main> radon cc App.py
App.py
  C 5:0 App - A
  M 6:4 App.__init__ - A
  M 15:4 App.show_main_window - A
```

Figure 4.5: Cyclometric complexity of App class

Note:

- B indicates that the method contains decision points.
- A indicates that the method does not contain decision points.
- C indicates Class.
- M indicates methods of the class.

The Cyclomatic complexity is a measure of the complexity of a program. It is calculated based on the number of decision points in the control flow of the program. The higher the Cyclomatic complexity, the more complex the code is considered.

4.2 Conclusions for each classes:

LoginWindow Class: Cyclomatic Complexity is moderately high. The cyclomatic complexity (1-2) is low, indicating a straightforward control flow at initialization.

App Class: Overall, the App class has low Cyclomatic complexity, with simple and straightforward control flow in its `__init__` method.

MetricsticsFrame Class:

- The MetricsticsFrame class exhibits a mix of Cyclomatic complexities.
- Several methods have low complexity (1-2), indicating simple control flow.
- Some methods, such as `calculate_mode`, `calculate_mean`, etc., have moderate complexity (3-4), suggesting multiple decision points and branching paths.
- Overall, the class has a varied level of control flow complexities.

Metrics Class:

- The Metrics class contains methods related to statistical calculations.
- Several methods, like `calculate_mode` and `calculate_mean`, have moderate Cyclomatic complexity (3-7), indicating multiple decision points.
- The complexity is generally moderate, reflecting the nature of statistical calculations.

DataGenerator Class:

- The DataGenerator class, responsible for generating random data, has a moderate Cyclomatic complexity of 5 in its `generate_data` method.
- The complexity is associated with control flow during data generation.

Summary:

- Each class has a mix of low to moderate Cyclomatic complexities, reflecting the diversity in the control flow within the methods.
- Methods with simple control flow typically have a Cyclomatic complexity of 1-2.
- Methods involving calculations or decision points have a moderate complexity (3-7).
- Consideration for refactoring may be given to methods with higher complexities to enhance code maintainability.

5 Problem 6: WMC, LF, LSOC

5.1 WMC:

The Weighted Methods per Class (WMC) is calculated by obtaining the sum of the cyclomatic complexities for each method within a class. The non-normalized cyclomatic complexity values are used in this scenario. Therefore, the Weighted Methods per Class (WMC) for each class is as follows:

- $WMC(\text{DataGenerator}) = 4$
- $WMC(\text{Metricstics}) = 33$
- $WMC(\text{MetricsticsFrame}) = 50$
- $WMC(\text{App}) = 4$
- $WMC(\text{LoginWindow}) = 7$

These values are calculated using the Radon library with the help of the following Python code.

```
import radon
from radon.complexity import cc_visit, sorted_results
import LoginWindow, Metricstics, MetricsticsFrame, App
import inspect

# Function to calculate WMC for a class
def calculate_wmc(class_instance):
    # Get class name
    class_name = class_instance.__name__

    # Get the code of the class
    class_code = inspect.getsource(class_instance)

    # Analyze the code and get the results
    results = cc_visit(class_code)
    complexities = [result.complexity for result in results]

    # Calculate WMC
    wmc = sum(complexities)

    return wmc
```

```
# Example usage:
my_class_instance = App
wmc_value = calculate_wmc(my_class_instance)
print("Weighted Method Per Class (WMC) for MyClass:", wmc_value)
```

Where **myclassinstanc**e will use Each class and generate WMC for that class

5.2 Comments of WMC:

- **DataGenerator:**

- $WMC(\text{DataGenerator}) = 4$
- The **DataGenerator** class has minimal complexity, indicating simple and linear control flow. It is easy to understand and maintain.

- **Metricstics:**

- $WMC(\text{Metricstics}) = 33$
- The **Metricstics** class is somewhat complex, suggesting a fair number of decision points and control flow in its methods.

- **MetricsticsFrame:**

- $WMC(\text{MetricsticsFrame}) = 50$
- The **MetricsticsFrame** class is more complicated, with methods involving more intricate control flow, decision points, and possibly nested structures.

- **App:**

- $WMC(\text{App}) = 4$
- The **App** class has low complexity, indicating simple and linear control flow. It is easy to understand and maintain.

- **LoginWindow:**

- $WMC(\text{LoginWindow}) = 7$
- The **LoginWindow** class has moderate complexity, suggesting methods with some decision points and control flow.

Overall, the complexity levels vary among the classes. Some classes have low to moderate complexity (e.g., App, DataGenerator), while others (e.g., Metricstics, MetricsticsFrame) exhibit higher complexity.

5.3 CF:

The coupling factor (CF) [Abreu, 1995b; Harrison, Counsell, Nithi, 1998] measures the average coupling between classes excluding coupling due to inheritance.

Let C_1, \dots, C_n be classes in an OOD, where n greater than 1. Let $\text{IsClient}(C_i, C_j) = 1$, if class C_i has a relationship with class C_j ; otherwise, it is 0. (In other words, $\text{IsClient}(C_i, C_j)$ is a characteristic function.)

The semantics of the (is-client) relationship might be that an object of C_i calls a method in C_j , or has a reference to C_j , or to an attribute in C_j . However, the requirement is that this relationship cannot be inheritance.

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

- App has dependencies on LoginWindow and MetricsticsFrame.
- Main has dependencies on App, Metricstics, and MetricsticsFrame.
- MetricsticsFrame has a dependency on DataGenerator.

here $n=6$ classes including main as a separate $CF=(2+3+1+0+0+0)/(6^2-6)$

$CF=6/30$

$CF = 0.20$

The Coupling Factor is 0.20, indicating strong coupling among the classes. Each class has direct dependencies on the others, contributing to a higher coupling factor.

5.4 LCOM:

LCOM (Lack of Cohesion of Methods) is a metric used to measure the degree to which methods in a class are related to each other. It ranges from 0 to 1, where a lower value indicates better cohesion. The formula for LCOM is given by:

$$LCOM = 1 - \frac{P}{Q}$$

Where:

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

App Class:

- **Methods Using Similar Attributes:** show_main_window, init
- **Total Methods:** show_main_window, init, metrics_calculator, login_window

LCOM Calculation: $1 - \frac{P}{Q} = 1 - \frac{2}{6} = 0.67$

MetricsticsFrame Class:

- **Methods Using Similar Attributes:** calculate_mode, calculate_mean, calculate_median, calculate_min, calculate_max, calculate_mad, calculate_standard_deviation
- **Total Methods:** calculate_mode, calculate_mean, calculate_median, calculate_min, calculate_max, calculate_mad, calculate_standard_deviation, reset_program, clear_text, update_data_text, update_result_text

LCOM Calculation: $1 - \frac{P}{Q} = 1 - \frac{7}{66} = 0.89$

DataGenerator Class:

- **Methods Using Similar Attributes:** generate_data
- **Total Methods:** generate_data

LCOM Calculation: $1 - \frac{P}{Q} = 1 - \frac{0}{1} = 1$

Metricstics Class:

- **Methods Using Similar Attributes:** calculate_min, calculate_max, calculate_mean, calculate_median, calculate_mode, calculate_mad, calculate_standard_deviation, absolute
- **Total Methods:** calculate_min, calculate_max, calculate_mean, calculate_median, calculate_mode, calculate_mad, calculate_standard_deviation, absolute

LCOM Calculation $= 1 - \frac{P}{Q} = 1 - \frac{0}{28} = 1$

LoginWindow Class:

- **Methods Using Similar Attributes:** login
- **Total Methods:** login

$$\text{LCOM Calculation} = 1 - \frac{P}{Q} = 1 - \frac{0}{1} = 1$$

Conclusion:

- The App class exhibits moderate cohesion, suggesting that its methods are somewhat related.
- The MetricsticsFrame class shows moderate cohesion, indicating that its methods are somewhat related.
- The DataGenerator class demonstrates excellent cohesion with only one method and no disjoint sets.
- The Metricstics class displays excellent cohesion, featuring multiple methods with no disjoint sets.
- The LoginWindow class exhibits excellent cohesion with only one method and no disjoint sets.

In summary, most of the classes exhibit high to excellent cohesion, suggesting that the methods within each class are well-organized and work effectively together. Overall, the codebase demonstrates good design principles with well-structured and cohesive classes.

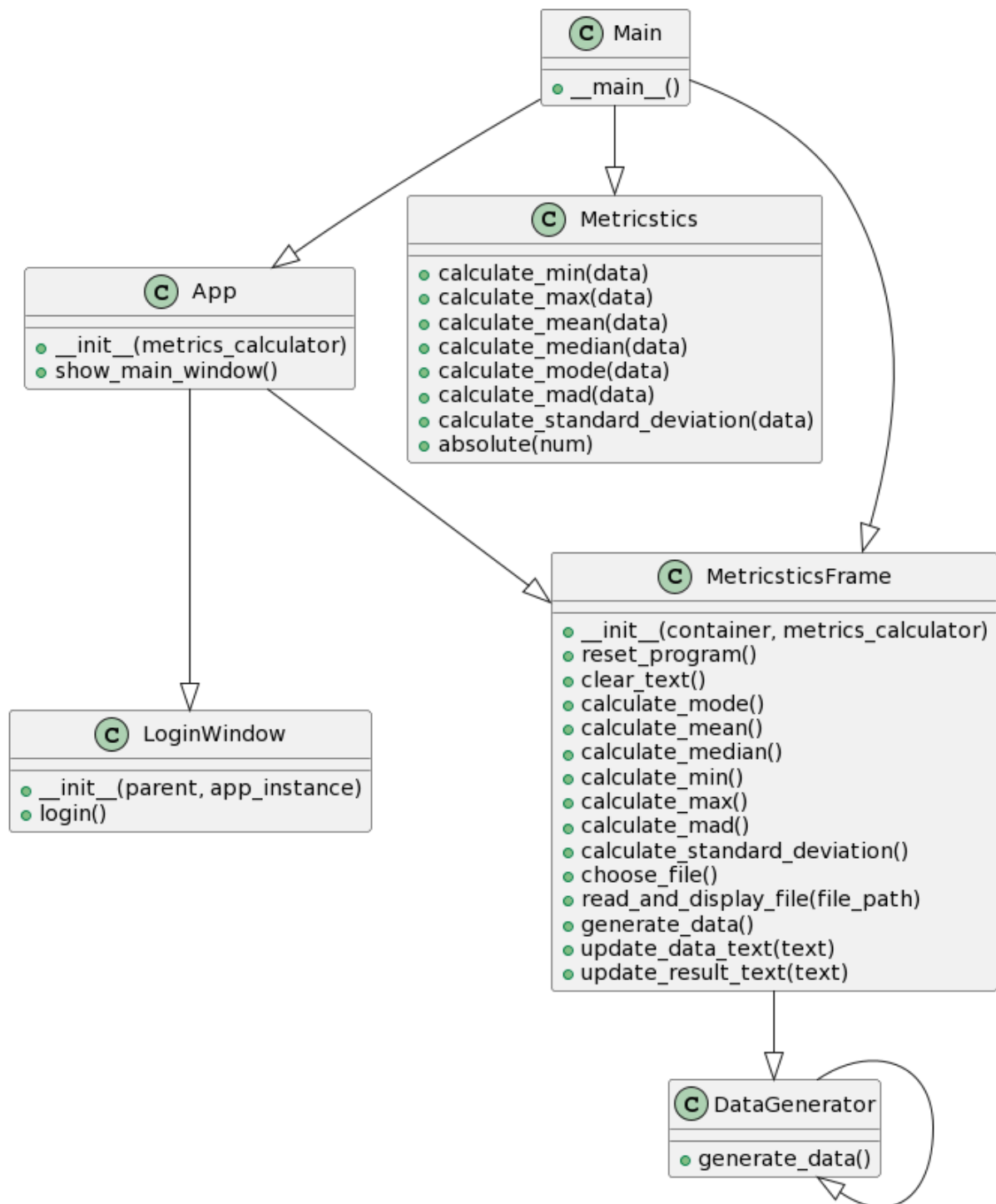


Figure 5.1: Overall dependencies of the System

6 Problem 7: Physical And Logical SLOC

6.1 Goal:

Goal Calculating the Physical SLOC and Logical SLOC for METRICSTICS. For Physical SLOC and Logical SLOC a tool named Radon is used.

6.2 Physical SLOC:

Physical SLOC refers to the count of lines in the source code containing executable statements, excluding blank lines and comments. In this project, the RADON tool was employed to calculate the Physical Source Lines of Code (Physical SLOC).

RADON, a tool for performing static code analysis in Python, offers the capability to assess various code metrics, including Physical SLOC. The process involved utilizing RADON's functionalities to analyze the main directory. The tool provides insights into code complexity, maintainability, and other metrics.

One advantage of using RADON is its suitability for Python projects, allowing for in-depth analysis and metrics calculation. By executing appropriate commands in the terminal or script, RADON aids in understanding the codebase's structure and complexity.

In this specific project, the calculated Physical SLOC value of **445** lines indicates a smaller-sized system. This suggests lower complexity, making the codebase more manageable and easier to maintain due to its compact size. The utilization of RADON for code analysis aligns with the project's Python-centric nature, providing valuable metrics and facilitating informed decision-making in code maintenance and development.

6.3 Logical SLOC:

It is a measure of the size or complexity of a software project based on the number of logical statements or instructions in the source code that affect the program's control flow or behavior.

Logical SLOC (Source Lines of Code) is a measure of the number of executable lines in code that contribute to the logical structure of the program. Here's a general guide on how logical SLOC is typically counted in Python code:

Advantages of Logical SLOC: Measures essential code which indicates functionality, maintainability, and performance of the code

Below fig represents the both Physical and Logical SLOC calculated using python:

```
PS C:\Users\NOMESH\Desktop\Fall 2023\SOEN6611-main> & C:/Users/NOMESH/AppData/Local/Programs/Python/Python38-64/python.exe "c:/Users/NOMESH/Desktop/Fall 2023/SOEN6611-main/SLOC.py"
```

Class	Physical SLOC	Logical SLOC
DataGenerator.py	18	15
LoginWindow.py	39	28
Metricstics.py	88	69
MetricsticsFrame.py	272	197
App.py	17	14
main.py	11	9
Total	445	332

```
PS C:\Users\NOMESH\Desktop\Fall 2023\SOEN6611-main>
```

Figure 6.1: Physical and Logical SLOC

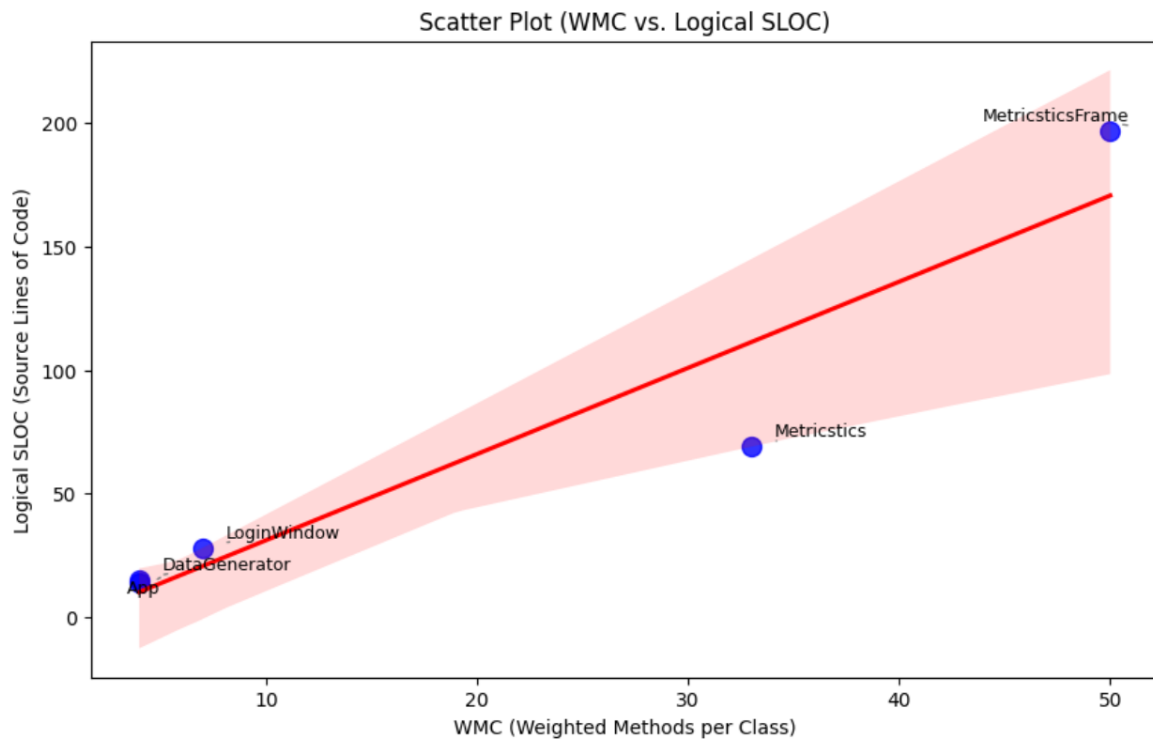
In summary, the logical SLOC count of 332 indicates a codebase of moderate size and complexity, which is likely to be well-structured and maintainable. It provides a foundation for further development and potential refactoring if necessary.

6.3.1 Rules for counting Physical SLOC and Logical SLOC

Rule	Physical SLOC	Logical SLOC
Blank Lines	Count each line that contains only spaces or tabs.	Ignore blank lines.
Comment Lines	Count each line that begins with a comment character (#).	Ignore lines that contain only comments.
Executable Lines	Count each line that contains executable code.	Count each line that contains executable code.
Multiline Strings	Count each line inside triple-quoted strings (""" or """).	Ignore lines inside triple-quoted strings.
Imports	Count each line with an import statement.	Ignore lines with import statements.
Docstrings	Count lines inside triple-quoted strings immediately after a class or function definition.	Ignore lines inside docstrings.
Empty Lines	Count lines with no visible characters.	Ignore lines with no visible characters.
Logical Continuation	Count each logical line as one line.	Count each logical line as one line.
Comments within Code	Count comments embedded within code lines.	Ignore comments embedded within code lines.
Single Line Comments	Count lines that contain a single-line comment.	Ignore lines that contain a single-line comment.
Inline Comments	Count lines that have inline comments after executable code.	Ignore lines with inline comments.

7 Problem 8

Class	WMC	SLOC(L)
App	4	14
DataGenerator	4	15
LoginWindow	7	28
MetristicsFrame	33	69
Metristics	50	197



We can see that there is a strong positive correlation between WMC and SLOC. This means that as the number of Weighted Methods in a Class (WMC) increases, the number of Source Lines of Code (SLOC) also tends to increase. This is likely because more complex classes tend to have more code.

In the context of this problem, Spearman's rank correlation coefficient was used to measure the strength and direction of the association between Weighted Methods in a

Class (WMC) and Source Lines of Code (SLOC). The coefficient was found to be 0.75, which indicates a strong positive correlation between the two variables. This means that as WMC increases, SLOC also tends to increase.

$$\rho = 1 - (6 * \Sigma d^2)/(n(n^2 - 1))$$

	WMC	SLOC	Rank(WMC)	Rank(SLOC)	d	d^2
1	4	14	1.5	1	0.5	0.25
2	4	15	1.5	2	-0.5	0.25
3	7	28	2	3	-1	1
4	33	69	4	4	-1	1
5	50	197	4	5	-1	1

$$\Sigma d^2 = 5$$

$$\rho = 1 - (6*0.5)/(5*(5^2 - 1)) = 0.9$$

The value of ρ is 0.9, which indicates a strong positive correlation between WMC and SLOC. This means that the two variables are positively related, and that as WMC increases, SLOC also tends to increase.

Conclusion:

The analysis of the correlations between the data for Logical SLOC and WMC obtained from METRICSTICS reveals a strong positive correlation between the two variables. As the number of Weighted Methods in a Class (WMC) increases, the number of Source Lines of Code (SLOC) also tends to increase. This finding aligns with the expectation that more complex classes, with a higher WMC, typically require more code to implement.

The calculation of Spearman's rank correlation coefficient, a non-parametric measure of correlation, further confirms the existence of a strong positive correlation between WMC and SLOC. The coefficient value of 0.9 indicates a significant positive relationship between the two variables. This finding reinforces the notion that as WMC increases, SLOC also tends to increase.

8 Work Distribution

Name	Contribution
Quoc Phong Ngo [40230574]	Problem 4, developing the code and testing with various test cases
SanVinoth Pacham [40198906]	Problem 4 & 8, modifications and updates in the code, problem 8
Preet Angad Singh [40234930]	Problem 3 Use case point approach
Siddharth Bharat Oza [40230155]	Problem 3 COCOMO81 model, documentation of report in the latex
Nomesh Palakaluri [40229979]	Problem 5,6,7 and documentation of the report

9 References

tkinter — Python interface to Tcl/Tk:

<https://docs.python.org/3/library/tkinter.html>

Developing a Full Tkinter Object-Oriented Application:

<https://www.pythontutorial.net/tkinter/tkinter-object-oriented-application/>

Radon - Introduction to Code Metrics

<https://radon.readthedocs.io/en/latest/intro.htmlcyclomatic-complexity>

For generating Palntuml code for class diagram based on our code

<https://chat.openai.com/>

Using the code generated from chatgpt we plot dependency diagram

<https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000>