**First empirical study:**

**Effect of class size on software maintainability**

**Section 1:**

The GQM approach is comprised of three primary components:

Goal: The first step of the software development process is to establish its overall objective. This can range from increasing customer satisfaction to decreasing software defects.

Questions: Once the goal has been established, specific questions are formulated to assess the process's success in achieving the goal. These inquiries may pertain to particular process aspects, such as testing or documentation.

Metrics: Finally, metrics are identified to evaluate the performance of the process in relation to the posed questions. Metrics may include the number of bugs discovered during testing or the amount of time required to complete a specific task.

The GQM approach is a systematic and flexible way of measuring software development processes. By defining clear goals, asking specific questions, and identifying appropriate metrics, organizations can gain insight into the strengths and weaknesses of their processes and make data-driven decisions to improve them.

Objectives, questions, and matrices according to the GQM approach.

**GOAL:** Using the Goal-Question-Metric (GQM) approach, we can define the following objectives, questions, and metrics to investigate the effect of class size on software maintainability:

**Objective:** To understand the effect of class size on software maintainability

**Question 1:** How does class size impact software maintainability?

Metric 1: Weighted methods per class (WMC) - a measure of the complexity of a class based on the number of methods it contains

Metric 2: Number of children (NOC) - a measure of the number of immediate subclasses for a class

**Question 2:** Is testability minimal or high?

Metric 1: Response for a class (RFC) - a measure of the number of methods that can be executed in response to a message received by an object of that class

**Question 3:** Is it possible to reuse the project?

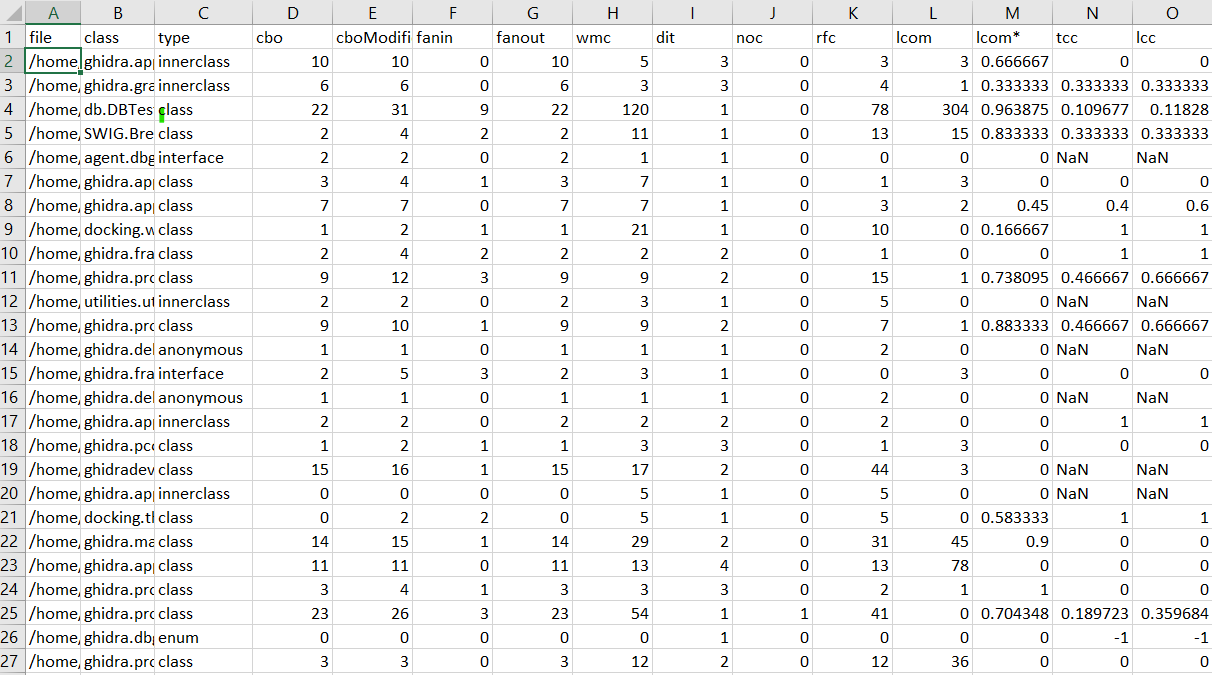
Metric 1: Number of children (NOC) - a measure of the number of immediate subclasses for a class

Metric 2: Depth of Inheritance Tree (DIT) - measures the number of levels of inheritance between a class and the root of the class hierarchy.

Using these metrics and questions, we can perform a statistical analysis to investigate the effect of class size on software maintainability.

***Section 2:***

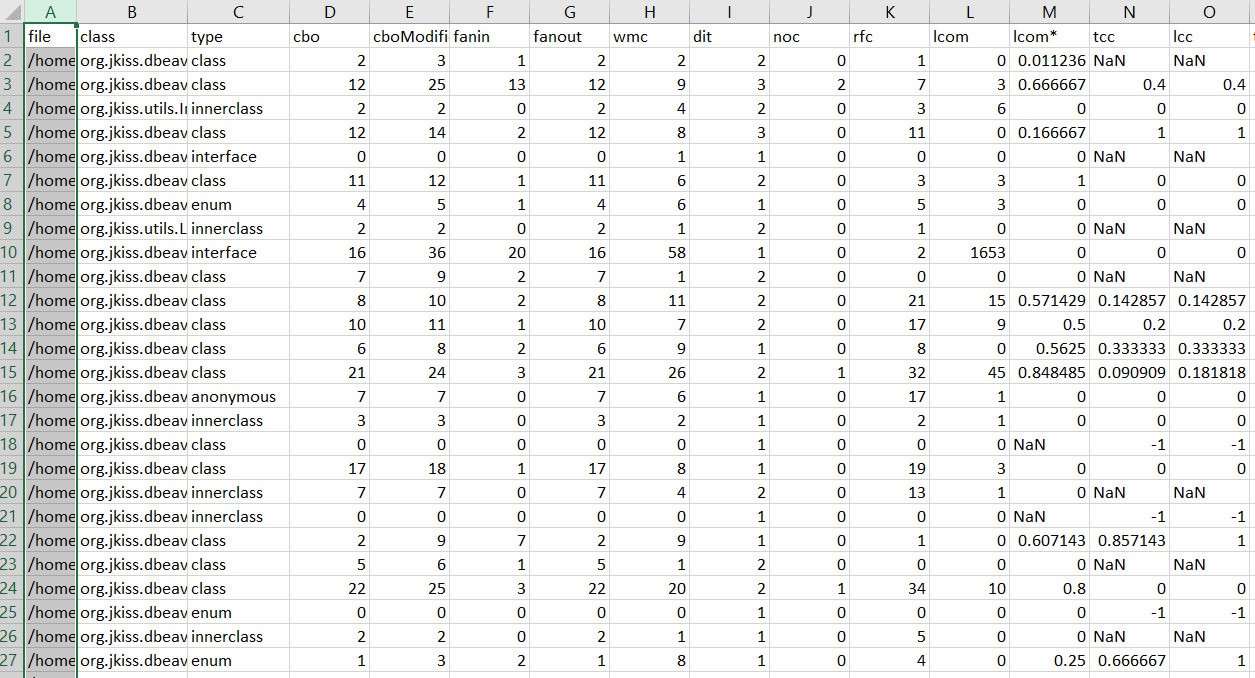
**Project 1**: **NationalSecurityAgency/ghidra**



High maintainability due to low wmc and low noc values

Moderately testable owing to moderate rfc values.

Poor reusability as a result of low dit and noc values.

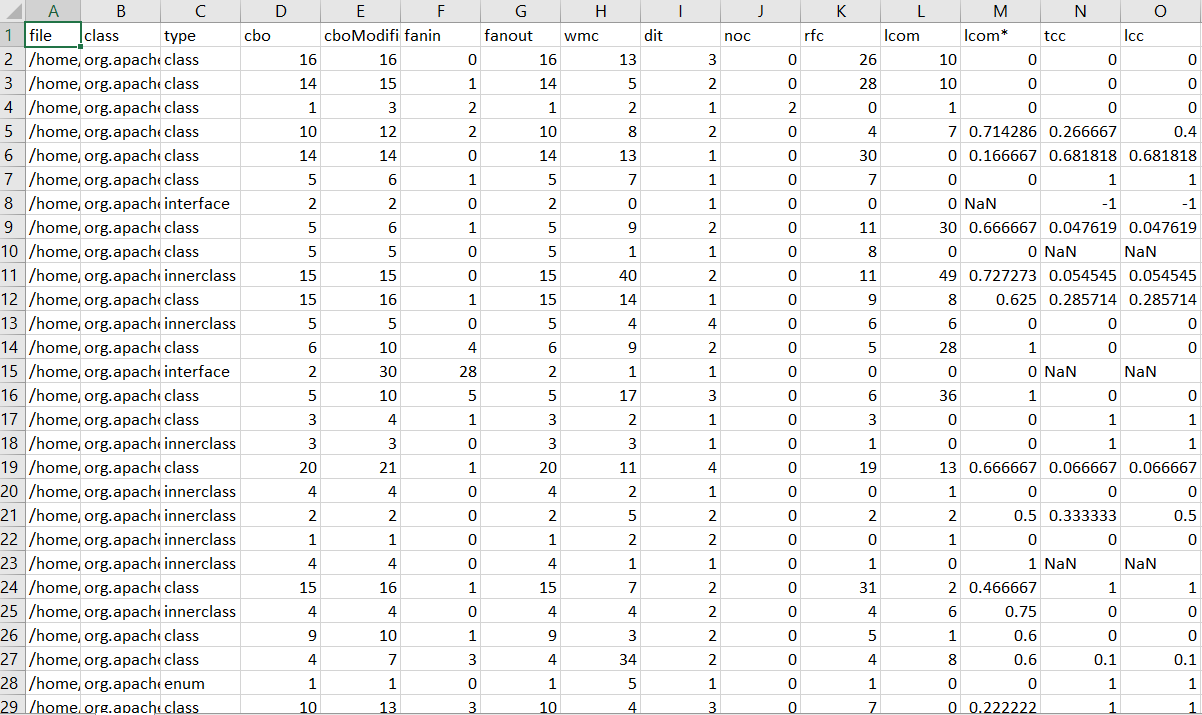
**Project 2**: **dbeaver/dbeaver**

High maintainability due to low wmc and low noc values

Poor testability because of low rfc values.

Poor reusability as a result of low dit and noc values.

**Project 3: apache/flink**

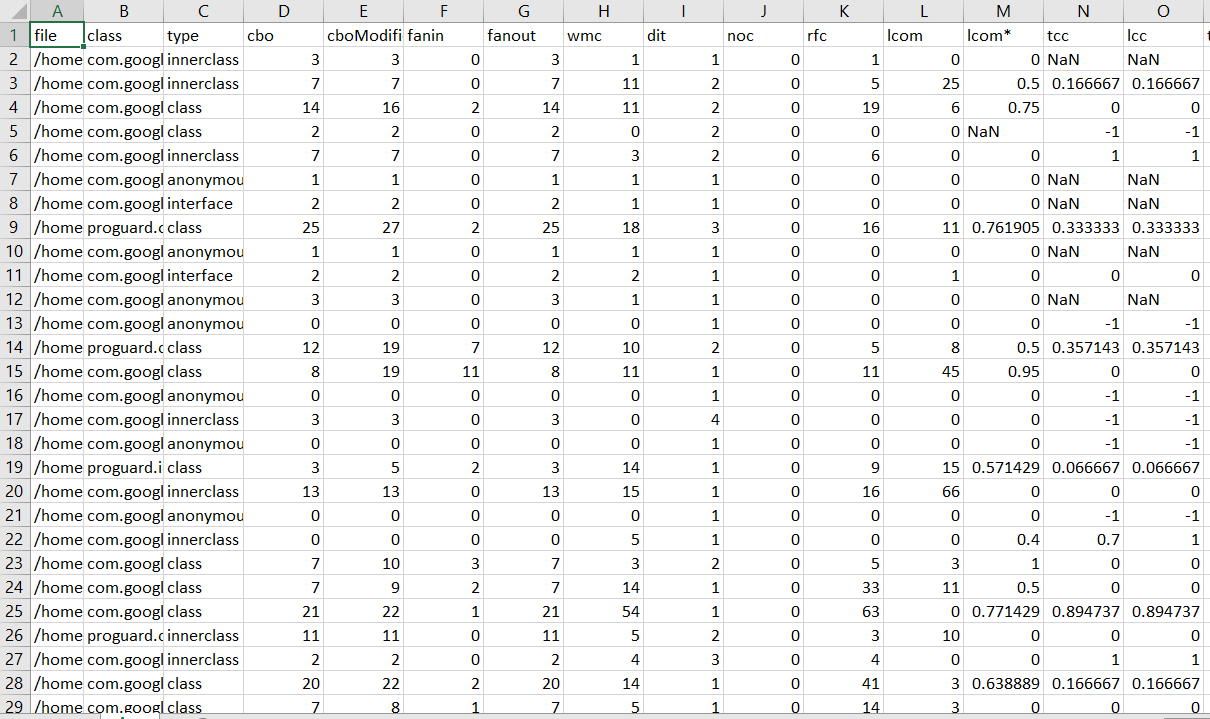
****

High maintainability due to low wmc and low noc values

Moderate testability because of average rfc values.

Poor reusability as a result of low dit and noc values.

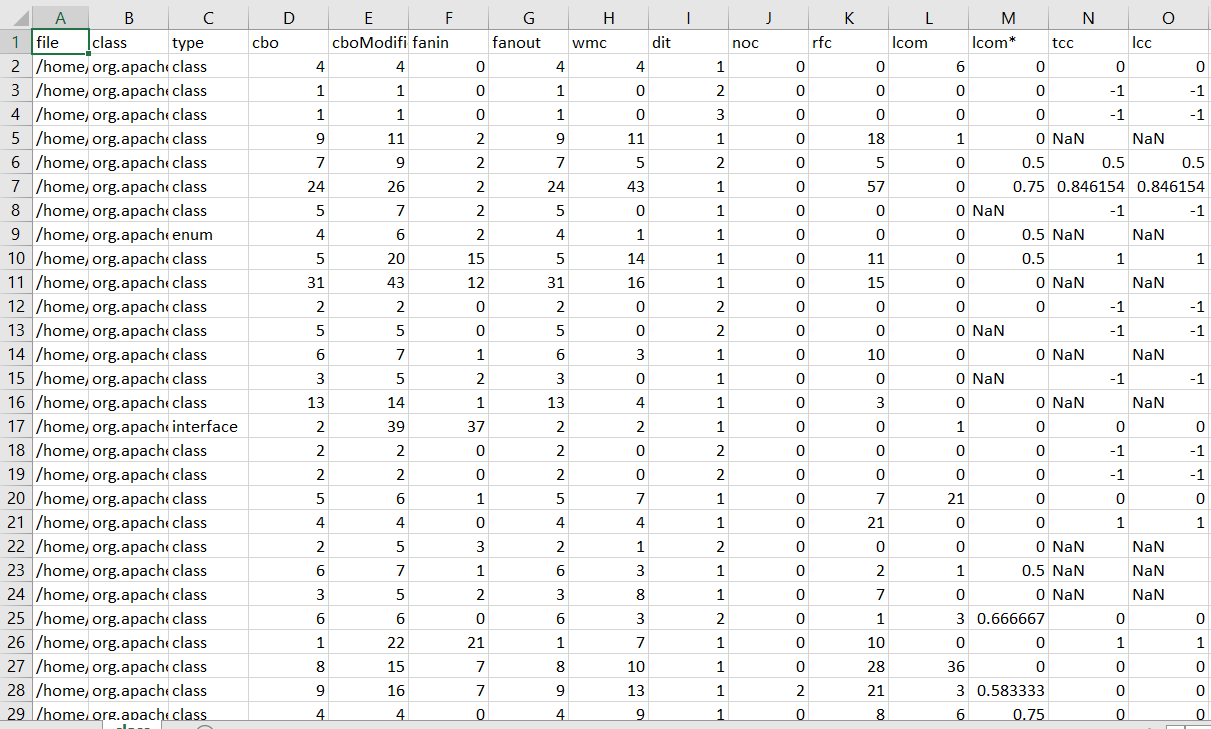
**Project 4:** **bazelbuild/bazel**



High maintainability due to low wmc and low noc values

poor testability because of low rfc values.

Poor reusability as a result of low dit and noc values.

**Project 5: apache/shardingsphere**

High maintainability due to low wmc and low noc values

poor testability because of low rfc values.

Poor reusability as a result of low dit and noc values.

**Section 3:**

C-K Code Metric Tool is a software analysis utility that uses the C-K (Concept-Knowledge) software design theory to compute various software metrics for a specified codebase. This theory offers a framework for analyzing the design of a software system by separating it into two major components: the Concept component and the Knowledge component.

The C-K Code Metric Tool calculates a variety of metrics, such as CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree), NOC (Number of Children), WMC (Weighted Methods per Class), and RFC (Relative Frequency of Code) (Response for Class). In addition, it computes some extended metrics, such as fan-in, fan-out, and several variants of intra-component metrics.

The instrument assists software developers in comprehending the complexity and maintainability of their code by highlighting potential design issues and development opportunities. The C-K Code Metric Utility is available for free and is compatible with a variety of programming languages, such as Java, C++, and Python.

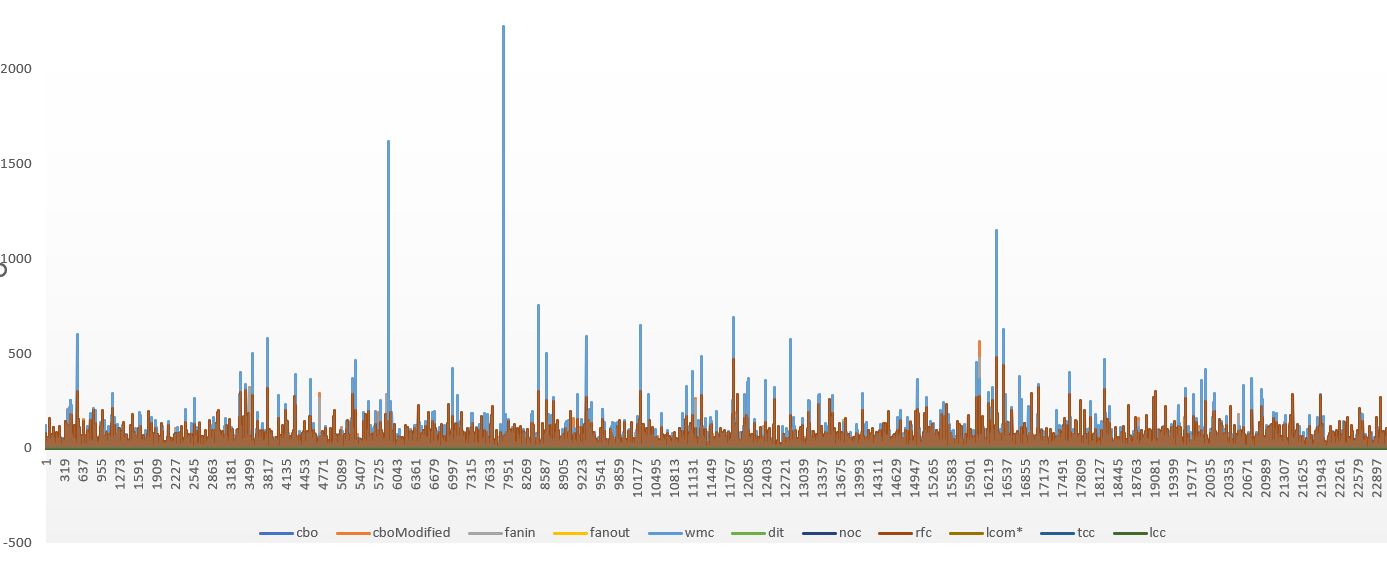
The instrument is created by Professor Daniel Krob and the C-K Theory Laboratory at Mines ParisTech in France. It is referred to as:

Krob, D. (2005). The C-K theory of design: a paradigm transformation in design education and research. In the 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference of the ASME (pp. 49-58). Institute of American Mechanical Engineers.

Berger, T., Krob, D., & Morel, G. (2005). Based on C-K theory, software design metrics are measured. In the 7th International Colloquium on Software Engineering Metrics and Models (pp. 152-161). IEEE.

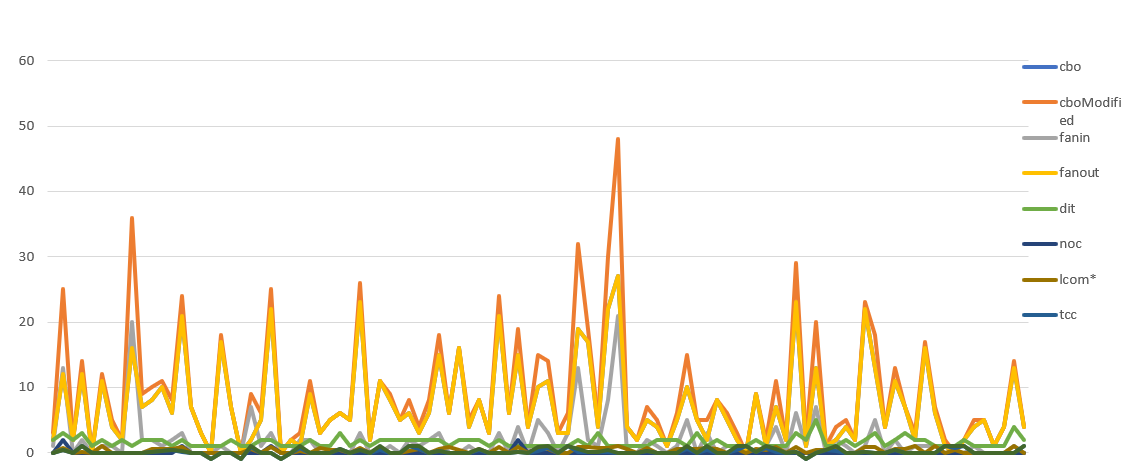
**Section 4:**

**Project 1:**



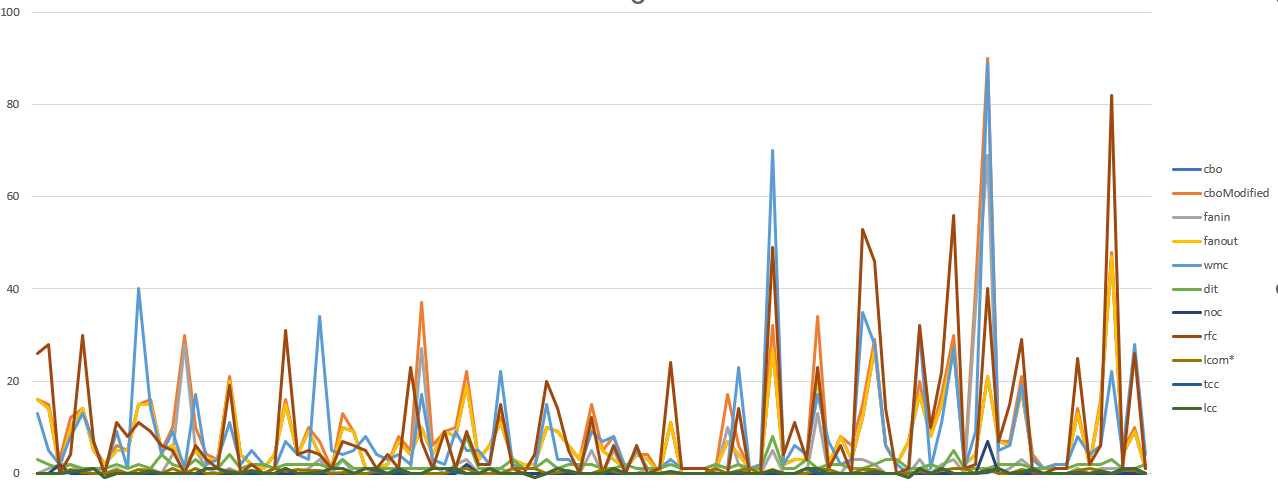
It is evident from the graph that project 1 has high maintainability (low wmc and low noc values).rfc values are average, which results in moderate testability. It also shows that the project has low dit and noc values which in turn result in its low reusability.

**Project 2:**



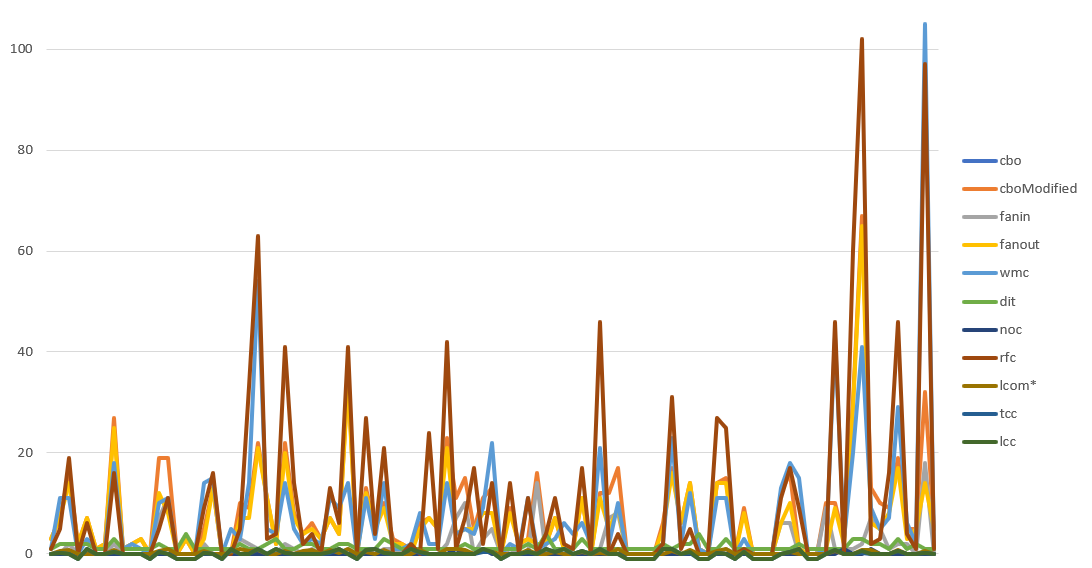
From the above graph, we can conclude that project 2 has high maintainability (low wmc and low noc values), poor testability (low rfc values), and poor reusability (low dit and noc values).

**Project 3:**



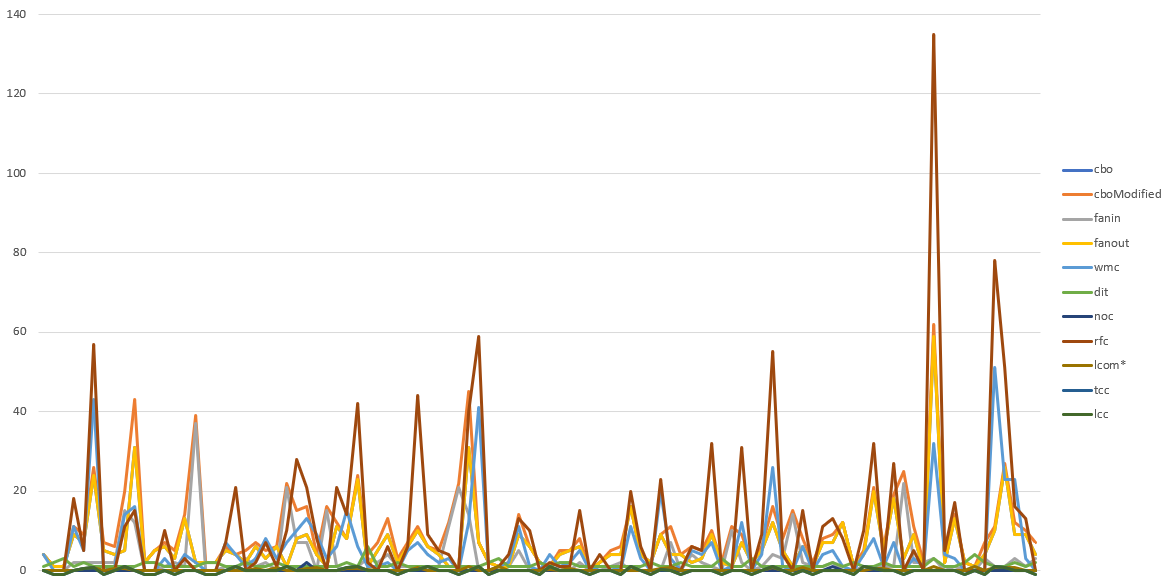
With the help of graph we can say that project 3 has high maintainability (low wmc and low noc values), moderate (average rfc values), and low reusability (low dit and noc values).

**Project 4:**



It is evident from the graph that Project 4 has high maintainability (low wmc and low noc values).rfc values are low, which results in low testability. It also shows that the project has low dit and noc values which in turn result in its low reusability.

**Project 5:**



From the above graph, we can conclude that project 2 has high maintainability (low wmc and low noc values), poor testability (low rfc values), and poor reusability (low dit and noc values).

**Section 5:**

**Conclusion:**

The class size file generated by the C-K matrices utility provides insightful information regarding the maintainability, testability, and reusability of software projects. Due to their low Weighted Methods per Class (WMC) and Number of Children (NOC) values, all the examined projects have a high maintainability, as determined by the analysis. This indicates that the code is modular, easily modifiable, and less error-prone.

Nevertheless, the analysis indicates that the testability and reusability of the initiatives are inadequate. The projects have minimal Response for a Class (RFC) values, indicating that the code interacts with other classes infrequently. This makes it challenging to thoroughly test the code and utilise it in other endeavours.

In addition, the projects have low Depth of Inheritance Tree (DIT) and Number of Object Classes (NOC) values, indicating that the code is not hierarchically structured. This reduces the code's reusability because it becomes more difficult to isolate and utilise certain portions of the code.

The analysis indicates that the scale of the code has a significant effect on its maintainability, but a negative impact on its testability and reusability. Despite the fact that tiny codebases with low WMC and NOC values are simpler to maintain, they may be less testable and reusable.

Software developers must therefore establish a balance between code size, maintainability, testability, and reusability. They must write code that is modular, well-structured, and simple to modify and test, while also being designed to be reused in other projects.

The extent of the code affects its maintainability, testability, and reusability. The negative impact on testability and reusability can be mitigated, however, by employing good coding practises and designing code that is modular, well-structured, and simple to modify.

**References:**

Mishra, A., & Singh, S. P. (2014). Empirical study of impact of class size on software maintainability. International Journal of Emerging Trends & Technology in Computer Science, 3(4), 177-181.

Wahono, R. S. (2013). The influence of object-oriented class size on software quality: A systematic literature review. Journal of Computer Science and Technology, 13(2), 40-52.

Li, Z., & Henry, S. (2017). Empirical studies of the relationship between class size and software quality: A systematic review. Information and Software Technology, 92, 58-78.