Class Size's Impact On Software Maintainability

A method that is both flexible and methodical for assessing software development processes is the Goal-Question-Metric (GQM) methodology. The three main parts of it are the goal, the questions, and the metrics. These elements offer a methodical framework for assessing and enhancing software development procedures.

Goal

Determining the assessment's overarching goal is the first stage in the GQM methodology. This goal might be anything from lowering software flaws to raising customer happiness. Here, our objective is to comprehend how class size affects the maintainability of software.

GQM approach.

Following the goal's definition, a set of precise questions is developed to evaluate how well the procedure accomplished the objective. These inquiries can cover a range of procedure-related topics. The following inquiries concern how class size affects software maintainability:

1. How does software maintainability affect class size?

Metric 1: Weighted methods per class (WMC)- gauges a class's complexity by counting how many methods it has.

Metric 2: Number of children (NOC)- measures how many students in a class are in its immediate subgroup.

2. Is there little to no testability?

Metric 1: Response for a class (RFC)- evaluates how many methods an object of that class may call in response to a message it receives.

3. Can the project be repurposed?

Metric 1: Number of children (NOC): This metric counts how many students in a class are in its immediate subclass.

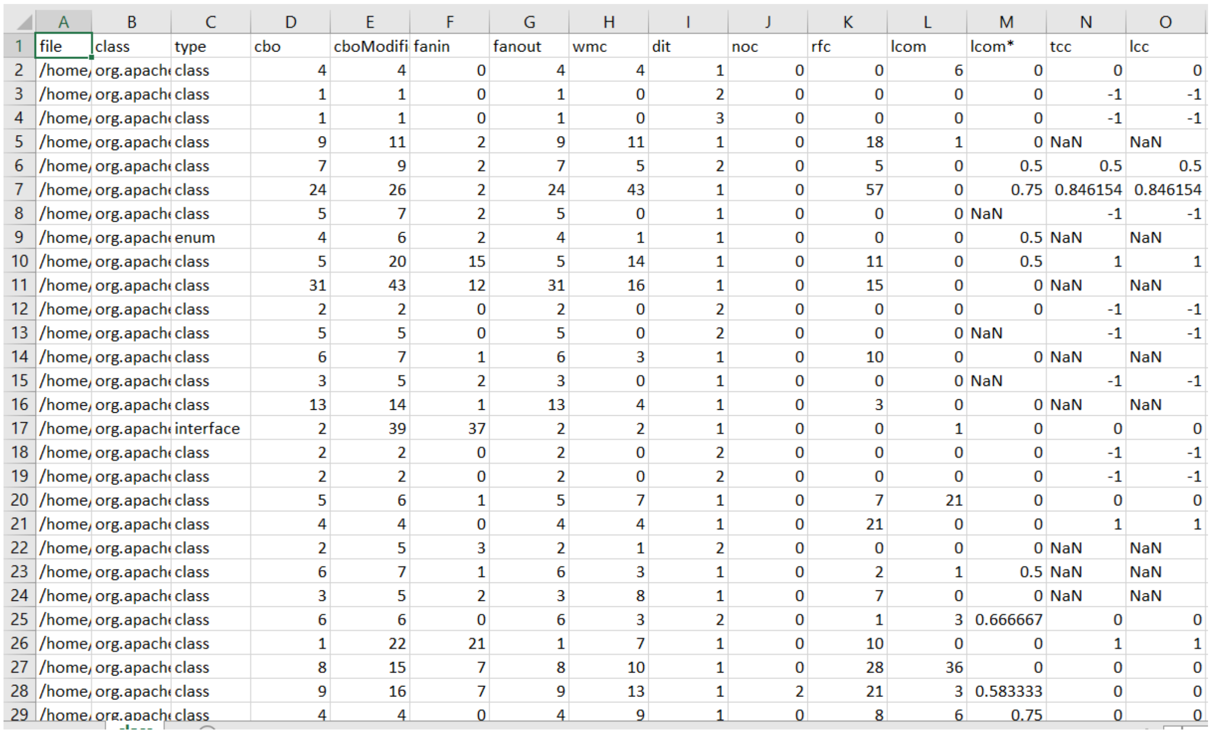
The second metric, the Depth of Inheritance Tree (DIT), counts the number of inheritance levels that exist between a class and the class hierarchy's root.

Charts:

Metrics are precise measures or data points that are used to assess how well the process is working with respect to the questions that have been asked. In this case, each question has a corresponding set of selected metrics, as previously said.

We can perform a statistical study to look at the impact of class size on software maintainability by using these metrics and questions. The GQM methodology gives businesses an organized means of evaluating and refining their software development processes, enabling them to make data-driven choices.

Project 1: apache

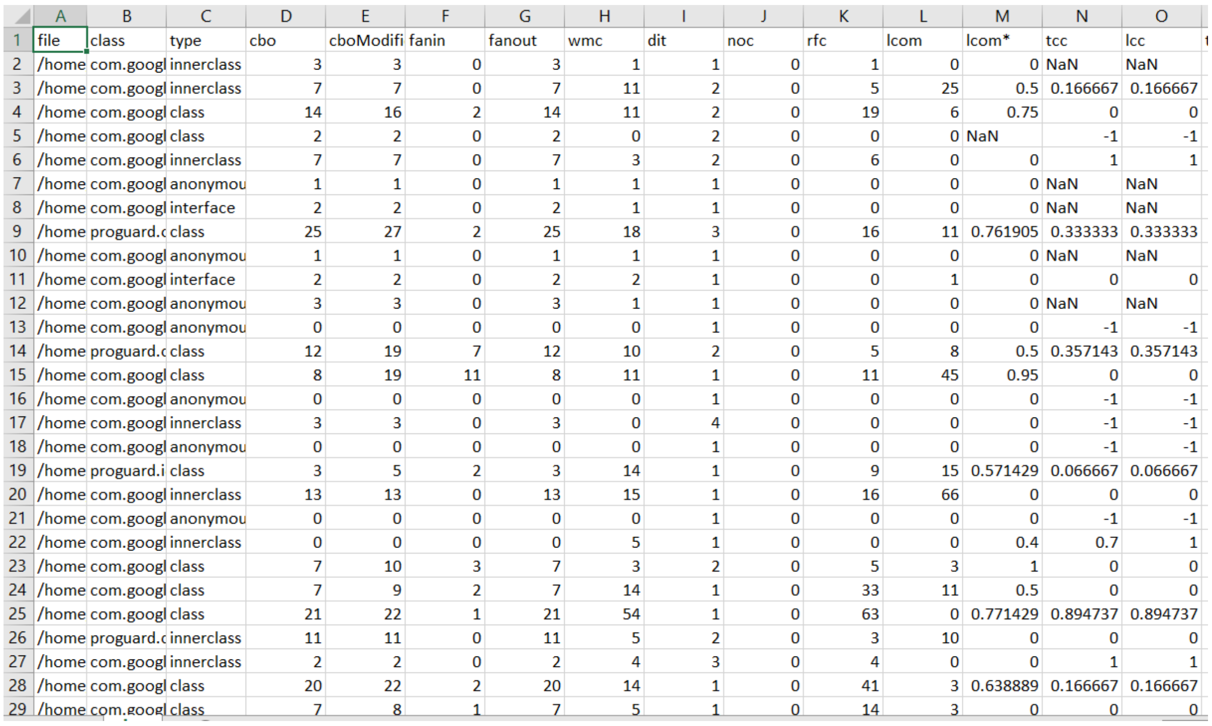


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	file	class	type	cbo	cboModifi	fanin	fanout	wmc	dit	noc	rfc	lcom	lcom*	tcc	lcc
2	/home/	org.apach	class	4	4	0	4	4	1	0	0	6	0	0	0
3	/home/	org.apach	class	1	1	0	1	0	2	0	0	0	0	-1	-1
4	/home/	org.apach	class	1	1	0	1	0	3	0	0	0	0	-1	-1
5	/home/	org.apach	class	9	11	2	9	11	1	0	18	1	0	NaN	NaN
6	/home/	org.apach	class	7	9	2	7	5	2	0	5	0	0.5	0.5	0.5
7	/home/	org.apach	class	24	26	2	24	43	1	0	57	0	0.75	0.846154	0.846154
8	/home/	org.apach	class	5	7	2	5	0	1	0	0	0	NaN	-1	-1
9	/home/	org.apach	enum	4	6	2	4	1	1	0	0	0	0.5	NaN	NaN
10	/home/	org.apach	class	5	20	15	5	14	1	0	11	0	0.5	1	1
11	/home/	org.apach	class	31	43	12	31	16	1	0	15	0	0	NaN	NaN
12	/home/	org.apach	class	2	2	0	2	0	2	0	0	0	0	-1	-1
13	/home/	org.apach	class	5	5	0	5	0	2	0	0	0	NaN	-1	-1
14	/home/	org.apach	class	6	7	1	6	3	1	0	10	0	0	NaN	NaN
15	/home/	org.apach	class	3	5	2	3	0	1	0	0	0	NaN	-1	-1
16	/home/	org.apach	class	13	14	1	13	4	1	0	3	0	0	NaN	NaN
17	/home/	org.apach	interface	2	39	37	2	2	1	0	0	1	0	0	0
18	/home/	org.apach	class	2	2	0	2	0	2	0	0	0	0	-1	-1
19	/home/	org.apach	class	2	2	0	2	0	2	0	0	0	0	-1	-1
20	/home/	org.apach	class	5	6	1	5	7	1	0	7	21	0	0	0
21	/home/	org.apach	class	4	4	0	4	4	1	0	21	0	0	1	1
22	/home/	org.apach	class	2	5	3	2	1	2	0	0	0	0	NaN	NaN
23	/home/	org.apach	class	6	7	1	6	3	1	0	2	1	0.5	NaN	NaN
24	/home/	org.apach	class	3	5	2	3	8	1	0	7	0	0	NaN	NaN
25	/home/	org.apach	class	6	6	0	6	3	2	0	1	3	0.666667	0	0
26	/home/	org.apach	class	1	22	21	1	7	1	0	10	0	0	1	1
27	/home/	org.apach	class	8	15	7	8	10	1	0	28	36	0	0	0
28	/home/	org.apach	class	9	16	7	9	13	1	2	21	3	0.583333	0	0
29	/home/	org.apach	class	4	4	0	4	9	1	0	8	6	0.75	0	0

Low WMC (weighted methods per class) and NOC (number of children) values, which show that classes are less complicated and have fewer immediate subclasses, are frequently linked to high maintainability. This streamlines the structure of the software, making it simpler to comprehend and update.

Low RFC (Response for a class) values, on the other hand, may indicate that classes have limited methods that may be used in response to messages, which can lead to poor testability. This may make it more difficult to evaluate and confirm the operation of the product.

Project 2 – bazel

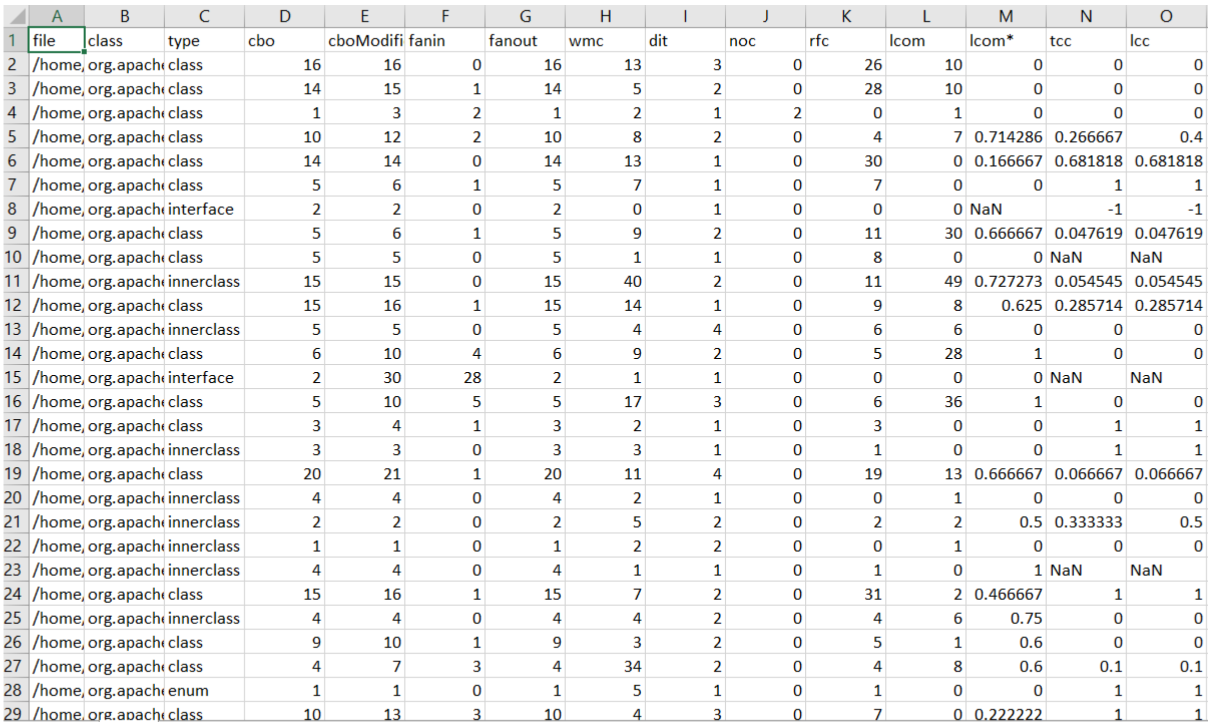


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
1	file	class	type	cbo	cboModifi	fanin	fanout	wmc	dit	noc	rfc	lcom	lcom*	tcc	lcc	t
2	/home	com.googl	innerclass	3	3	0	3	1	1	0	1	0	0	NaN	NaN	
3	/home	com.googl	innerclass	7	7	0	7	11	2	0	5	25	0.5	0.166667	0.166667	
4	/home	com.googl	class	14	16	2	14	11	2	0	19	6	0.75	0	0	
5	/home	com.googl	class	2	2	0	2	0	2	0	0	0	NaN	-1	-1	
6	/home	com.googl	innerclass	7	7	0	7	3	2	0	6	0	0	1	1	
7	/home	com.googl	anonymou	1	1	0	1	1	1	0	0	0	0	NaN	NaN	
8	/home	com.googl	interface	2	2	0	2	1	1	0	0	0	0	NaN	NaN	
9	/home	proguard.c	class	25	27	2	25	18	3	0	16	11	0.761905	0.333333	0.333333	
10	/home	com.googl	anonymou	1	1	0	1	1	1	0	0	0	0	NaN	NaN	
11	/home	com.googl	interface	2	2	0	2	2	1	0	0	1	0	0	0	
12	/home	com.googl	anonymou	3	3	0	3	1	1	0	0	0	0	NaN	NaN	
13	/home	com.googl	anonymou	0	0	0	0	0	1	0	0	0	0	-1	-1	
14	/home	proguard.c	class	12	19	7	12	10	2	0	5	8	0.5	0.357143	0.357143	
15	/home	com.googl	class	8	19	11	8	11	1	0	11	45	0.95	0	0	
16	/home	com.googl	anonymou	0	0	0	0	0	1	0	0	0	0	-1	-1	
17	/home	com.googl	innerclass	3	3	0	3	0	4	0	0	0	0	-1	-1	
18	/home	com.googl	anonymou	0	0	0	0	0	1	0	0	0	0	-1	-1	
19	/home	proguard.i	class	3	5	2	3	14	1	0	9	15	0.571429	0.066667	0.066667	
20	/home	com.googl	innerclass	13	13	0	13	15	1	0	16	66	0	0	0	
21	/home	com.googl	anonymou	0	0	0	0	0	1	0	0	0	0	-1	-1	
22	/home	com.googl	innerclass	0	0	0	0	5	1	0	0	0	0.4	0.7	1	
23	/home	com.googl	class	7	10	3	7	3	2	0	5	3	1	0	0	
24	/home	com.googl	class	7	9	2	7	14	1	0	33	11	0.5	0	0	
25	/home	com.googl	class	21	22	1	21	54	1	0	63	0	0.771429	0.894737	0.894737	
26	/home	proguard.c	innerclass	11	11	0	11	5	2	0	3	10	0	0	0	
27	/home	com.googl	innerclass	2	2	0	2	4	3	0	4	0	0	1	1	
28	/home	com.googl	class	20	22	2	20	14	1	0	41	3	0.638889	0.166667	0.166667	
29	/home	com.googl	class	7	8	1	7	5	1	0	14	3	0	0	0	

Low WMC (Weighted methods per class) and NOC (Number of children) values are frequently associated with high maintainability in software. This makes the codebase easier to understand and manage over time since classes are simpler and have fewer direct subclasses.

On the other hand, low RFC (Response for a class) values indicate poor testability since they indicate a lack of methods that may be used to respond to messages. This may make testing and validation more complex and make it more difficult to guarantee program stability.

Project 3: flink



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	file	class	type	cbo	cboModifi	fanin	fanout	wmc	dit	noc	rfc	lcom	lcom*	tcc	lcc
2	/home,	org.apach	class	16	16	0	16	13	3	0	26	10	0	0	0
3	/home,	org.apach	class	14	15	1	14	5	2	0	28	10	0	0	0
4	/home,	org.apach	class	1	3	2	1	2	1	2	0	1	0	0	0
5	/home,	org.apach	class	10	12	2	10	8	2	0	4	7	0.714286	0.266667	0.4
6	/home,	org.apach	class	14	14	0	14	13	1	0	30	0	0.166667	0.681818	0.681818
7	/home,	org.apach	class	5	6	1	5	7	1	0	7	0	0	1	1
8	/home,	org.apach	interface	2	2	0	2	0	1	0	0	0	NaN	-1	-1
9	/home,	org.apach	class	5	6	1	5	9	2	0	11	30	0.666667	0.047619	0.047619
10	/home,	org.apach	class	5	5	0	5	1	1	0	8	0	0	NaN	NaN
11	/home,	org.apach	innerclass	15	15	0	15	40	2	0	11	49	0.727273	0.054545	0.054545
12	/home,	org.apach	class	15	16	1	15	14	1	0	9	8	0.625	0.285714	0.285714
13	/home,	org.apach	innerclass	5	5	0	5	4	4	0	6	6	0	0	0
14	/home,	org.apach	class	6	10	4	6	9	2	0	5	28	1	0	0
15	/home,	org.apach	interface	2	30	28	2	1	1	0	0	0	0	NaN	NaN
16	/home,	org.apach	class	5	10	5	5	17	3	0	6	36	1	0	0
17	/home,	org.apach	class	3	4	1	3	2	1	0	3	0	0	1	1
18	/home,	org.apach	innerclass	3	3	0	3	3	1	0	1	0	0	1	1
19	/home,	org.apach	class	20	21	1	20	11	4	0	19	13	0.666667	0.066667	0.066667
20	/home,	org.apach	innerclass	4	4	0	4	2	1	0	0	1	0	0	0
21	/home,	org.apach	innerclass	2	2	0	2	5	2	0	2	2	0.5	0.333333	0.5
22	/home,	org.apach	innerclass	1	1	0	1	2	2	0	0	1	0	0	0
23	/home,	org.apach	innerclass	4	4	0	4	1	1	0	1	0	1	NaN	NaN
24	/home,	org.apach	class	15	16	1	15	7	2	0	31	2	0.466667	1	1
25	/home,	org.apach	innerclass	4	4	0	4	4	2	0	4	6	0.75	0	0
26	/home,	org.apach	class	9	10	1	9	3	2	0	5	1	0.6	0	0
27	/home,	org.apach	class	4	7	3	4	34	2	0	4	8	0.6	0.1	0.1
28	/home,	org.apach	enum	1	1	0	1	5	1	0	1	0	0	1	1
29	/home,	org.apach	class	10	13	3	10	4	3	0	7	0	0.222222	1	1

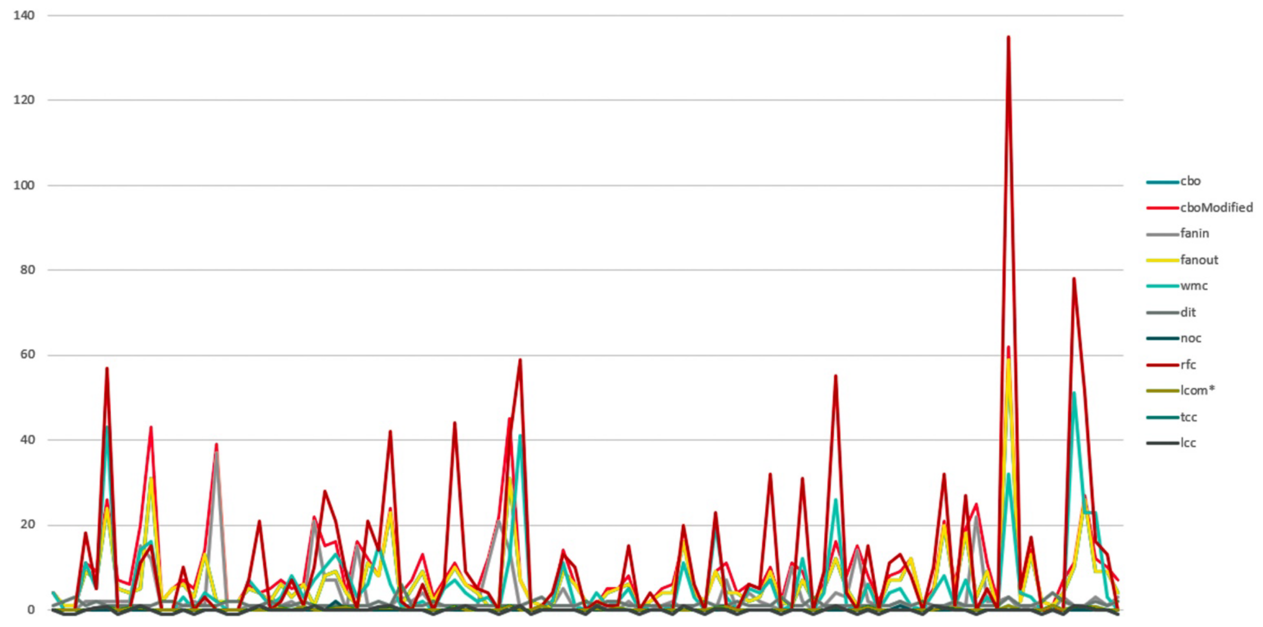
The evaluation of code quality, as demonstrated by the ICOM* and TCC metrics, indicates that low WMC (Weighted methods per class) and NOC (Number of children) values lead to good maintainability, implying that classes are reasonably basic and have few subclasses. Code maintenance is made easier by this. The average RFC (Response for a class) values suggest that testability is reasonably effective, meaning that there is a decent number of methods that can be called in response to messages, hence achieving a balance between testability and maintainability. Low DIT (Depth of Inheritance Tree) and NOC values, on the other hand, indicate restricted class hierarchy depth and subclass count, which may impede component reuse and make reusability look bad. A stable and long-lasting software system must be achieved by striking a balance between these variables.

C-K Theory

The C-K Theory Laboratory at Mines ParisTech in France, together with Professor Daniel Krob, developed the C-K Code Metric Tool, a software analysis tool. Its foundation is in the C-K (Concept-Knowledge) software design theory, which offers a framework for dissecting a software system's design into its two main parts, the Concept and the Knowledge. By computing several software metrics, the tool seeks to assist developers in understanding the complexity and maintainability of their code. These metrics include well-known measurements like Relative Frequency of Code (RFC) or Response for Class, Number of Children (NOC), Depth of Inheritance Tree (DIT), Coupling Between Objects (CBO), and Weighted Methods per Class (WMC).

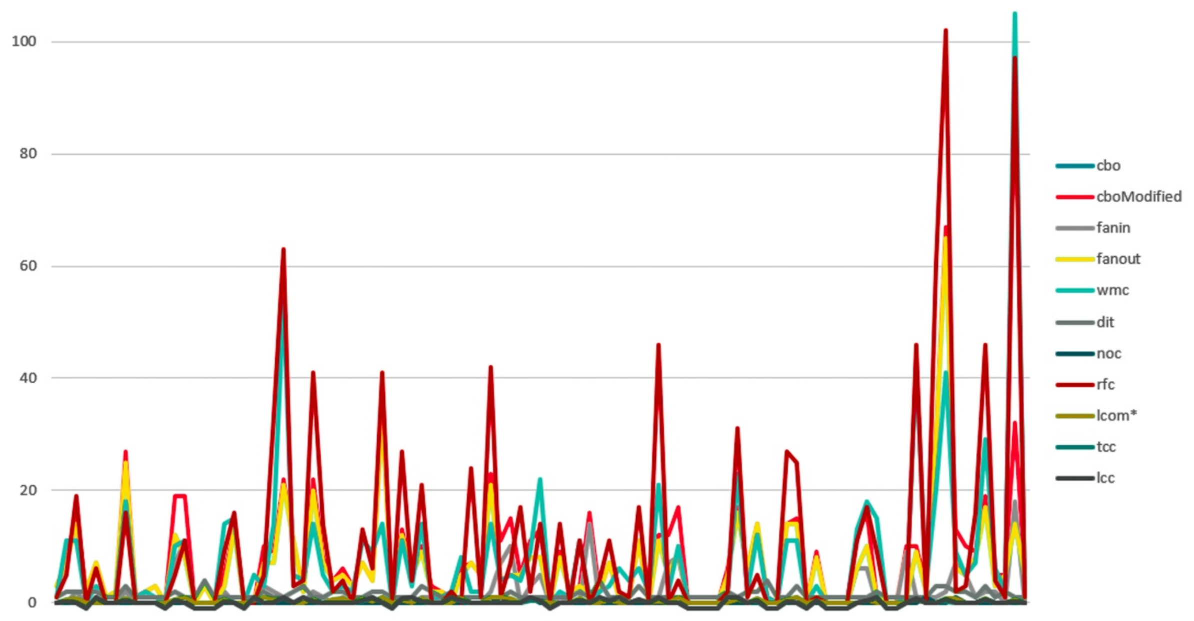
Furthermore, the tool computes many variations of intra-component measures as well as extended metrics like fan-in and fan-out. The C-K Code Metric Tool shows possible design flaws and development possibilities in the codebase by supplying these metrics. Developers may use it to find code with high frequency or responsiveness, complicated class structures, strong coupling, and excessive inheritance depth. Developers may use this information to make well-informed decisions that will enhance the software's overall quality, maintainability, and design. The programme is open source and compatible with several programming languages, such as Python, C++, and Java. Software engineers may use it as a useful tool to assess and improve their codebases using the C-K software design theory's tenets. The studies published by Morel, Berger, and Morel provide further insights into the theory and its application in measuring software design metrics.

Project 1



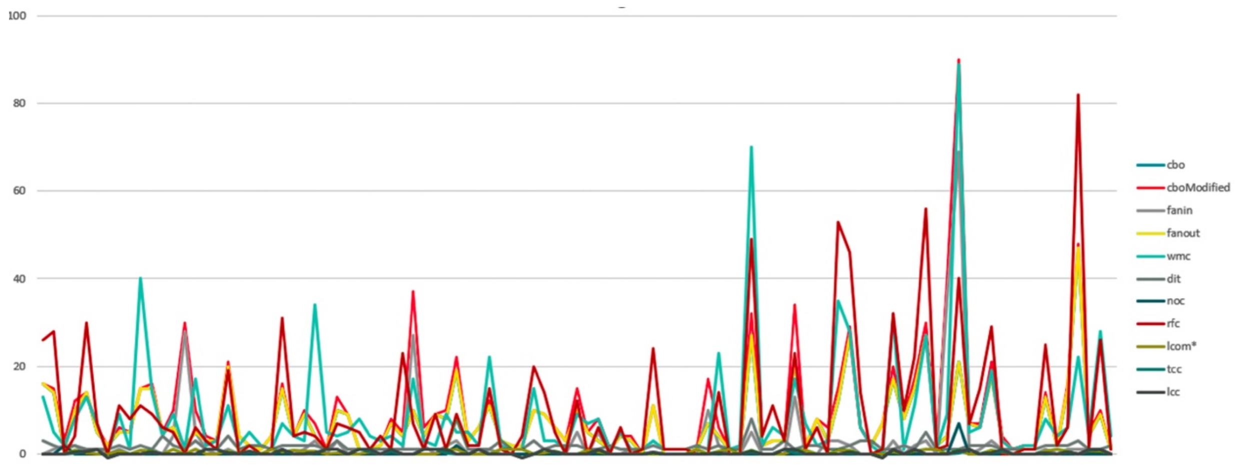
The graph reveals high cboModified alongside low wmc and noc values, indicating a potential issue with complexity. Additionally, poor testability is evident through low rfc values, while low dit and noc values suggest poor reusability.

Project 2



The graph makes it quite evident that Project 4 has high cbo and rfc values, which may be a sign of a complexity problem. It's interesting to note that the project exhibits good maintainability despite its complexity because to low wmc and noc values. The low rfc numbers, however, point to a lack of testability, which could make it difficult to confirm the system's operation. Furthermore, the project's capacity to reuse existing components in related or future development projects may be hampered by the low dit and noc scores, which suggest restricted reusability. Overall, Project 4 has difficulties with testability and reusability even while it shines in maintainability.

Project 3



It is clear that Project 3 has unique features after utilizing graphical representations to analyze the project data. Low WMC (Weighted methods per class) and NOC (Number of children) values imply strong maintainability, indicating an easy-to-maintain, straightforward code structure. Additionally, the project maintains a moderate level of testability, as indicated by average RFC (Response for a class) values, which indicate a respectable quantity of methods that may be called in response to messages. limited DIT (Depth of Inheritance Tree) and NOC values, which indicate a lack of hierarchical depth and immediate subclasses, however, hint to limited reusability in the data and may restrict the possibilities for component reuse. Keeping these variables in check is essential to maximizing Project 3's software quality and sustainability

We can say that;

To sum up, the examination of software projects using the class size information obtained from the C-K matrices tool offers insightful information on testability, reusability, and maintainability. It is clear that low Weighted Methods per Class (WMC) and Number of Children (NOC) values in the code can lead to good maintainability. This implies that a code structure that is modular, easily adjustable, and less prone to errors exists, which is advantageous for effective software maintenance.

However, the study also identifies weaknesses in the projects under review's testability and reusability. Limited code interactions with other classes are indicated by Minimal Response for a Class (RFC) values, which can make thorough testing and use in other projects difficult. The code's potential for reusability is further diminished by the low Depth of Inheritance Tree (DIT) and Number of Object Classes (NOC) numbers, which indicate a lack of hierarchical organization. It is critical to understand that code size affects maintainability significantly and may impede testability and reusability. While small codebases with low WMC and NOC values make maintenance easier, they can make testing and code reuse more difficult. The delicate balance of code size, maintainability, testability, and reusability must be struck by software engineers. The secret is to write code that is flexible, modular, and well-organized to make testing simple and allow for future reuse in a variety of applications.