SE series 1

Duong Vu Hai and Manou Keizer

November 2024

1 Introduction

In software engineering, maintainability is a critical aspect of software quality, reflecting how easily systems can be understood, modified, and extended.

This report examines the maintainability of two Java projects, smallsql and hsqldb, by computing several metrics that influence maintainability aspects as defined in the paper. These metrics include volume, unit size, unit complexity, and duplication, each providing different insights into the system's analysability, changeability, testability, and overall maintainability. We base our analysis on the methodology and categorization outlined in the paper, which contains (most) threshold values and scoring criteria for these metrics.

The primary objective of this report is to evaluate the maintainability of smallsql and hsqldb by assigning each project a rating across multiple maintainability aspects. We do this by calculating individual metric scores and combining them into broader maintainability aspect scores. We also apply risk category thresholds and scoring frameworks to determine the risk level of each aspect of the projects.

The report is organized as follows: Sections 2 and 3 provides an overview of the metrics used and the methods applied to calculate them. In this section, we also discuss potential improvements for maintainability analysis based on the metrics and their implications on maintainability. Section 4 presents the specific calculated scores of both projects.

2 Questions

2.1 Which metrics are used?

The answer to this question is presented in section **Results**.

2.2 How are these metrics computed?

The answer to this question is presented in section **Implementation**.

2.3 How well do these metrics indicate what we really want to know about these systems and how can we judge that?

According to the paper, in general, a larger and more complex system requires a larger effort to maintain. Particularly, a higher volume of the source code lowers the analysability of the system. The size of units influences the analysability and testability of systems. The complexity of units influences the system's changeability and its testability. The degree of source code duplication influences analysability and changeability. These sub-characteristics determine the overall maintainability aspect of systems.

We believe the effectiveness of these basic metrics in determining what we want to know about the systems heavily depends on the interpretation framework we apply. As noted in the paper, the value of these metrics becomes clear only when mapped to maintainability aspects, like analysability, changeability, stability, and testability, which are also metrics but in a broader sense. By interpretation framework we mean tools like tables and matrices that categorize metrics into specific risk levels or scores. Determining the right thresholds for each risk level is also crucial. Without reasonable and accurate thresholds we risk misclassifying the maintainability aspects of the system (and maintainability overall), which can lead to overlooking issues.

We also think it does make sense to group these different basic metrics together for determining broader metrics (maintainability aspects), as relying on a single metric often doesn't give a complete picture of the system. For instance, unit size alone doesn't necessarily indicate low analysability; a large unit isn't inherently problematic if it's well-organized and low in complexity. However, when a large unit size is combined with high complexity or high duplication, it's more likely to represent an analysability risk. As another example, duplication alone doesn't necessarily lead to low changeability. In some cases, duplicated code may be simple and well-isolated, thus making it easy to understand and change without introducing changeability risk. However, complex duplicated code is harder to change and therefore risky.

2.4 How can we improve any of the above?

2.4.1 Unit Size and Unit Complexity tables

We think the thresholds in the tables are often too strict, which can lead to an unfair assessment of maintainability aspects. The unit size and unit complexity tables require 0% high-risk code for the first 3 risk levels, which can be unrealistic. Having a small amount like 1% high-risk code due to one complex method shouldn't automatically label the entire system as having high complexity risk or unit size risk. Making these thresholds less strict within the tables would allow for a more balanced and practical evaluation.

2.4.2 Comments

As we will mention in the implementation section, we decided to exclude comments from our analysis, as outlined in the paper. However, we believe that including comments could improve our assessment of metrics like unit size, as they provide valuable context and clarity. Comments can make larger or more complex units easier to understand, potentially justifying more flexible thresholds. Similarly, for volume, we currently are not making a distinction between a large project containing 50% of LOC in comments or one containing 2%, when in practice this will make a difference in the maintainability.

3 Implementation

Comments and blank lines - Following the definitions provided in the paper, we decided to exclude comments and blank lines from the calculations of each basic metric. This approach also makes sense to us, as there can be edge cases where a unit contains many blank lines or comments, making it appear more complex and larger in size than it actually is, which could affect the scores of some maintainability aspects. Removing them effectively addresses this issue.

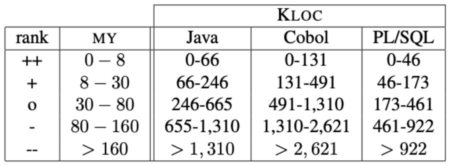
Unit - We follow the definition of a unit for Java provided in the paper, which is only methods.

Volume - Using AST and pattern matching, we traverse through each .java source file, get its content and location, remove comments and blank lines, and count the total number of lines containing code.

Unit size - Using AST and pattern matching, for each method, we get the method's source location, retrieve the method's lines, remove comments and blank lines, and calculate LOC. Then for each risk category, we calculate the number of units that fall into it, total LOC, and distribution (percentage).

Unit Complexity - AST and pattern matching are used to traverse the implementation of each method and compute its cyclomatic complexity. We walk over the source location of the method, counting all control flow constructs, namely conditionals and loops. It gives us essentially a way to measure how complex the method is in terms of its structure. For each method, we classify the complexity with regard to risk categories by computing the number of methods falling into each category, associated lines of code (LOC), and their distribution (percentage).

Duplication - Reusing the same method for Volume, we gather all the clean code lines and volume. Then we use a 2-pass algorithm. In the first pass, we



		KLOC		
rank	MY	Java	Cobol	PL/SQL
++	$0-8$	0-66	0-131	0-46
+	$8-30$	66-246	131-491	46-173
o	$30-80$	246-665	491-1,310	173-461
-	$80-160$	655-1,310	1,310-2,621	461-922
--	> 160	$> 1,310$	$> 2,621$	> 922

Figure 1: Volume table

map out 6-line blocks and their counts (number of occurrences). In the second pass, we iterate through each block with a count of more than 1 and calculate the number of duplicate lines. For example, if there are 2 identical blocks of 6 lines each, we count them as 12 duplicated lines. If there are 2 identical blocks of 7 lines each, we count 14 lines. In this case, there are actually 4 overlapping blocks of 6 lines, so we need to avoid double-counting these lines. Our algorithm addresses this in the second pass. Finally, we calculate the duplication percentage and the total number of duplicate lines.

Maintainability and its aspects - Each maintainability aspect score, including the overall maintainability score, was calculated by averaging the scores of the relevant basic metrics on which they are based. The results are assigned a value from 1 to 5 (rounded down), which are then translated to the ratings ++, +, o, -, - -.

4 Results

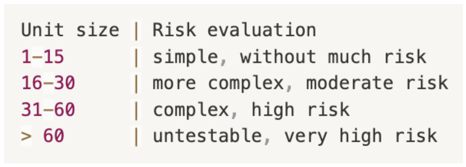
In this section, we present the calculated software metrics for the analyzed Java project smallsql and hsqldb. Below are the details of each metric calculated:

4.1 Volume

Volume is measured by counting the Lines of Code (LOC). We used the definition in the paper, which excludes comments and blank lines. The total LOC calculated is **24501 for smallsql and 175647 for hsqldb**. According to the risk profile table found in the paper shown in Figure 1, these results fall under the ++ and + categories, respectively.

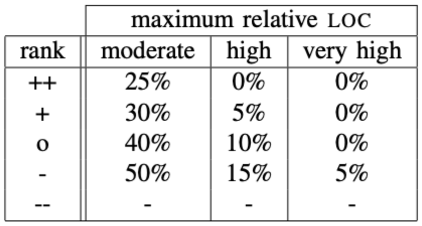
4.2 Unit size

Unit Size measures the size of individual methods without comments and blank lines, using the definition provided in the SIG paper. We calculated the unit sizes for smallsql and hsqldb and categorized them into risk levels in the table shown in Figure 2. The table was made using thresholds from the book "Building Maintainable Software" by SIG, page 25 [4] and the risk profiles of



```
Unit size | Risk evaluation
1-15      | simple, without much risk
16-30     | more complex, moderate risk
31-60     | complex, high risk
> 60      | untestable, very high risk
```

Figure 2: Unit size table



	maximum relative LOC		
rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
--	-	-	-

Figure 3: Unit size rating table

the complexity unit table in the paper, shown in Figure 2. We decided to use the book because it is authored by experts from the same organization responsible for the SIG model, therefore we think it is a reliable and consistent source for our analysis. Our findings are presented in Table 1. Figure 3 from the paper provides guidelines for the unit size rating of projects. We conclude that the projects fall under the **- -** and **- -** categories in unit size, respectively.

Project	**Low**	**Moderate**	**High**	**Very High**	**Total**
Unit Counts					
smallsql	1947	200	82	31	2260
hsqldb	7689	1294	679	403	10065
LOC					
smallsql	9158	4433	3527	3505	20623
hsqldb	42204	27542	28342	50232	148320
Unit Size Distribution					
smallsql	37.38%	18.09%	14.40%	14.31%	84.18%
hsqldb	24.03%	15.68%	16.14%	28.60%	84.45%

Table 1: Unit Counts, LOC, and Unit Size Distribution per Risk Category for smallsql and hsqldb. The percentage is calculated as LOC / Volume.

4.3 Unit complexity

The complexity of source code refers to how many branches or decision points are in the code. We did this by calculating the Cyclomatic Complexity per unit, where we counted conditional statements, loops, exception handling, and conditional expressions. For this we used the categorization provided by [2]. When categorizing the units by cc, we also counted the number of lines aligned with these units. Lastly, we computed the relative volumes of each system so that we could summarize the distribution of LOC over the various risk levels. We got the Cyclomatic Complexity definition and method from [3] and [1]. The results of this can be seen in table 2.

Project	**Low**	**Moderate**	**High**	**Very High**	**Total**
Unit Counts					
smallsql	2165	53	36	6	2260
hsqldb	9460	397	163	45	10065
LOC					
smallsql	14801	1875	2665	1282	20623
hsqldb	93458	22381	16918	15563	148320
Complexity Distribution					
smallsql	60.41%	7.65%	10.88%	5.23%	84.18%
hsqldb	53.21%	12.74%	9.63%	8.86%	84.45%

Table 2: Unit Counts, LOC, and Complexity Distribution per Risk Category for smallsql and hsqldb. Complexity Distribution is defined as LOC / Volume.

We refer back to figure 3 to provide a rating, which gives that smallsql and hsqldb both fall under the risk category –.

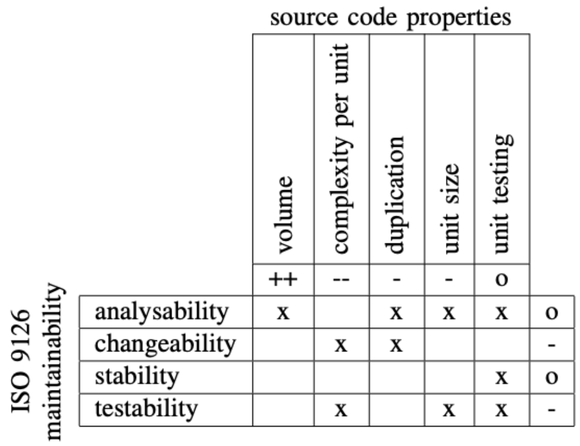
Note that adding up the LOC for both unit size and unit complexity adds up to the same number for both smallsql and hsqldb, but is not equal to the volume, as the volume also includes classes, constructors, and more.

4.4 Duplication

Code Duplication is measured to identify repeated code blocks in a project. For duplication, we used the definition provided in the paper, where code duplication is calculated as the percentage of all code that occurs more than once in equal code blocks of at least 6 lines. Our findings are presented in Table 2. We conclude that projects smallsql and hsqldb fall under the **- -** and **- -** categories in duplication, respectively.

Metric	**smallsql**	**hsqldb**
Duplication percentage	11.29%	16.28%
Total lines analyzed	24501	175647
Duplicate LOC	2765	28590

Table 3: Duplication Metrics for smallsql and hsqldb. Duplication percentage is defined as Duplicate LOC / Total lines analyzed (Volume).



ISO 9126 maintainability	source code properties					
	volume	complexity per unit	duplication	unit size	unit testing	
	++	--	-	-	o	
analysability	x		x	x	x	o
changeability		x	x			-
stability					x	o
testability		x		x	x	-

Figure 4: Maintainability table

4.5 Maintainability and its aspects

For maintainability aspects, we determined their scores by averaging the scores of relevant basic metrics, as outlined in Figure 4 (source code properties marked with an 'x'). The results are assigned a value from 1 to 5 (rounded down), which are then translated to the ratings ++, +, o, -, - -. All basic metrics are equally weighted, as suggested in the paper.

The overall maintainability score is calculated in the same way - by averaging the scores of all maintainability aspects.

We decided to round down our average score as this avoids inflation of scores. A system with a ++ score for both analysability and changeability, but a + score for stability, should in our opinion show an overall maintainability score of + that shows that improvements can still be made.

Below we present a table of scores of all metrics - both basic and broad ones.

Metric	**smallsql**	**hsqldb**
Volume	++	+
Unit size	- -	- -
Unit complexity	- -	- -
Duplication	-	-
Maintainability	- -	- -
Analysability	-	-
Changeability	- -	- -
Testability	- -	- -

Table 4: All metric scores for smallsql and hsqldb

5 Discussion

One of the challenges we faced during our analysis was how to handle duplicate method names. The existence of these duplicates made it difficult to identify methods uniquely as individual entities and caused discrepancies when our computed numbers would not add up correctly. We realized this was causing issues midway through our project and resolved it by modifying our code to treat every method as unique. Specifically, if a method name existed multiple times, we stored it as the method name combined with a unique suffix. This adjustment allowed us to differentiate the methods correctly, allowing us to compute metrics like unit size, complexity, and duplication in a reliable way.

Another challenge we came across was the fact that Rascal came out with the new 0.40.17 release, where they introduced new features and optimizations. However, by the time this update became available, we were nearly done with our project, which is why we decided to keep working with the older version.

Something else to note is the way in which we are currently computing the overall maintainability score. Currently we are averaging the analysability, changeability and stability score, but as mention by Heitlager this may not accurately reflect the true maintainability of our project. This is because averaging assumes equal importance among all metric, which may not hold true in all context. For instance, in projects requiring frequent changes, changeability might be more critical than analysability. This approach can also mask significant weaknesses; a low score in one area could be overlooked if the average remains acceptable. To improve our assessment, we should consider a weighted scoring system that reflects the relative importance of each metric based on project-specific needs. This would provide a better understanding of maintainability.

References

[1] C. M. Software Engineering Institute. Cyclomatic complexity – software technology roadmap. `http://www.sei.cmu.edu/str/descriptions/cyclomatic.html`, 2000.

[2] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability – a preliminary report, 2007.

[3] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[4] J. Visser, S. Rigal, G. Wijnholds, P. Van Eck, and R. van der Leek. *Building Maintainable Software: Ten Guidelines for Future-Proof Code.* O'Reilly Media, Inc., 2016.