Software Evolution - Software Metrics

Anandan Krishnsamy (14874121), Nefeli Tavoulari (15043347)

November 2023

Universiteit van Amsterdam

Contents

1 Introduction

Software evolution is a critical aspect of the software development lifecycle. As software systems grow, they undergo continuous changes, updates, and improvements to meet evolving user needs and address issues. Understanding and evaluating the quality and maintainability of software is crucial for ensuring its long-term viability and effectiveness.[3] Metrics used by organizations like the Software Improvement Group (SIG) play a significant role in assessing software quality and maintainability. These metrics often encompass various aspects, including code complexity, code duplication, code churn, code smells, architectural issues, test coverage, and more. Analyzing these metrics provides insights into the health of the software and helps in identifying areas that require attention or improvement.

In this assignment we examine some metrics that are used by the Software Improvement Group [2] to examine the quality and maintainability of software. More specifically, we implement these metrics and question if they are useful and how they could be improved to get more accurate results.

2 Project structure

```
/ src
├── Metrics
│   ├── Volume.rsc
│   ├── UnitSize.rsc
│   ├── UnitTesting.rsc
│   ├── UnitComplexity.rsc
│   ├── Duplication.rsc
│   ├── Maintainability.rsc
│   ├── Comments.rsc
│   └── UnitInterfacing.rsc
├── Tests
│   ├── VolumeTests.rsc
│   ├── UnitSizeTests.rsc
│   ├── UnitTestingTests.rsc
│   ├── UnitComplexityTests.rsc
│   ├── DuplicationTests.rsc
│   ├── MaintainabilityTests.rsc
│   ├── CommentsTests.rsc
│   └── UnitInterfacingTests.rsc
├── Lib
│   └── Utilities.rsc
└── Main.rsc
```

Metrics folder: contains all the metrics implementations based on the SIG paper. Maintainability.rsc contains the System level metrics.
Tests folder: contains all the tests on the metrics, using the smallsql project,

and a test project that we created, and that is included in our submission.
Lib folder: contains some commonly used functions.
Main file: script to run everything on smallsql or hsqldb, both Source code metrics and System level metrics.

3 Instructions

To run the project from the Rascal terminal:

- import Main;
- main(<projectLoc>);

To run tests:

- import Main;
- import <testsFile>;
- :test

4 Metrics

4.1 Source code metrics

4.1.1 Volume

The volume of the source code influences the analysability and therefore the maintainability of a system. A larger system is harder to maintain, it takes more time and effort to understand it. The way SIG takes into account the volume of a system is using these metrics:

- Lines of code: all lines of source code that are not comments (block/line comments) or blank lines.
- Man years via backfiring function points: ranking on lines of code based on the programming language. For Java, using the following table:

rank	man years	Java (KLOC)
++	0-8	0-66
+	8-30	66-246
o	30-80	246-665
-	80-160	655-1,310
–	> 160	> 1, 310

Table 1: Programming Languages Table of Software Productivity Research LLC

- Other volume measures: some estimate of functional size, by counting database tables and fields, screens or input fields, logical and physical files, and such.

To calculate the volume of a project we summed up the lines of code of all files of the project. To get the the lines of code we iterated over the lines of each file to get the count of block comments, the count of line comments and the count of blank lines and subtracted these from the file lines. We made sure that we do not take into consideration the line comments that are inside block comments, or the line or block comments that are in the same line with some actual code. e.g

```
/*
    // line comment
*/

int main() /* block comment*/
```

We basically created some functions for processing a file and some for processing a project, that only call the former ones iterating over the files. In this way, we made our code more maintainable, since we have units that are reusable and small. Also, based on the KLOC we return the corresponding rank in string.

4.1.2 Unit Size

The size of units (methods) affects analysability and testability, since large units are harder to comprehend, reuse and test. SIG measures unit size calculating the lines of code per unit and uses risk categories and scoring guidelines to classify a system.

For this metric, we calculated the size of every unit/method, using the same method as for file/project size, subtracting the count of comments and blank lines from the lines of each method. At first, we iterated over the methods like this:

```
for(method <- methods(model)) {
    methodsLoc[method] = (linesOfCodeFile(method) -
       blankLinesFile(method) - commentsFile(method));
}
```

But since we needed to have the same keys in our methodsLoc map as in the methodsComplexity map for the Unit Complexity metric, we changed the iteration to this:

```
list[Declaration] asts = getASTs(projectLoc);
visit(asts) {
        case Declaration decl: \method(_, _, _, _, _):
            methodsLoc[decl.src] = (linesOfCodeFile(
           decl.src) - blankLinesFile(decl.src) -
           commentsFile(decl.src));
        case Declaration decl: \method(_, _, _, _):
           methodsLoc[decl.src] = (linesOfCodeFile(
```

```
            decl.src) - blankLinesFile(decl.src) -
            commentsFile(decl.src));
        case Declaration decl: \constructor(_, _, _, _
            ): methodsLoc[decl.src] = (linesOfCodeFile(
            decl.src) - blankLinesFile(decl.src) -
            commentsFile(decl.src));
}
```

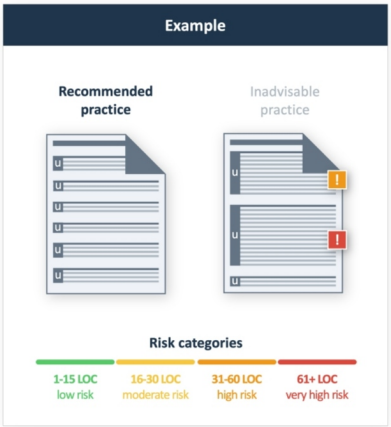
This decreases slightly the results but due to lack of documentation, we could not find what is missing from the second implementation or if maybe this is the correct one. However, with testing, we came to the conclusion that are results are accurate and the difference is so small that it does not affect the final results.

Then, we found the risk profile for every method and normalized our data to 100. Then, according to the results, we returned the corresponding rank in string. The thresholds we used for the risk profiles and the ranking are the following:[1]

risk profile	**size range**
low risk	0-15
moderate risk	16-30
high risk	31-60
very high risk	> 60

Table 2: Risk profiles

Based on this:



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

Table 3: Unit size ranks

4.1.3 Unit Testing

Unit tests raise testability, since with a single push of a button, tests can be executed. Unit tests raise stability, because they provide a regression suite as safety net to help prevent introducing errors when modifications are made. Unit tests also have a strong documentative nature, which is good for analysability. SIG measures test quality using the following metrics:

- Unit test coverage: using tools such as Clover4.
- Number of assert statements.

For this metric, we calculated the number of assert statements of every unit/method. Specifically, iterating over the methods and using method ast, we are looking for method calls that contain the substring "assert" in their name.

Another design choice we made here is that we only consider methods that have at least one assert, so that they are test functions. Otherwise, our results would be spoiled by normal source code functions, that would have no assert functions, as expected. Also, we use only a specific folder for this processing, passed to the function when we run it. For example, for smallsql, we should use the directory "smallsql0.21_src/src/smallsql/junit" and for our TestProject we used this "TestProject/src/test". In this way, we ensure that our results will not get spoiled by functions that are not test functions, but simple code.

Then, we categorized methods based on their strength and normalized our data to 100. Then, according to the results, we returned the corresponding rank in string. The thresholds we used for the categories and the ranking are the following:

test coverage profile	number of assert statements
low	0-2
moderate	3-5
high	6-10
very high	> 10

Table 4: Test coverage profiles

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

Table 5: Unit testing ranks

4.1.4 Duplication

An excessive duplication can harm a system's maintainability and unnecessarily increase its size. Measuring code duplication provides a straightforward estimate of how much larger a system becomes due to redundancy.

[2]The method we used here involves identifying duplicated code blocks that consist of at least 6 lines of code. Code lines are compared without considering leading spaces. If a group of 6 lines appears unchanged in more than one place, it is classified as duplicated code.

```
// Iterate over units
for (str line <- readFileLines(fileLoc)) {

    // Clean line and Ignore comments block
    line = trim(line);
    if (line != "" && !startsWith(line, "//")) {
        if (startsWith(line, "/*") || (
            insideBlockComment == true)) {
            // inside the block comment
            insideBlockComment = true;
            if (endsWith(line, "*/")) {
                // outside the block comment
                insideBlockComment = false;
            }
        } else {
            lines += line;
            // if the lines are 6 it considered as a
               block and added to the map
            if (size(lines) == 6) {
                // clean lines
                str block = trim(lines[0]) + trim(
                   lines[1]) + trim(lines[2]) + trim(
                   lines[3]) + trim(lines[4]) + trim(
                   lines[5]);
                updatedVars = addToMap(blocks, block,
                   consecutive, duplicateLOC);
                blocks = updatedVars[0];
                consecutive = updatedVars[1];
```

```
                duplicateLOC = updatedVars[2];
                lines = lines[1..6]; // remove the
                    first line and make it ready to
                    form next block
            }
        }
    }
}
```

The following code adds blocks to the hash map and calculates the duplicateLOC. If a block already exists in the map, the number of occurences is incremented, otherwise it is added to the map. If a block is duplicated, it implies that six lines are duplicated and subsequently added to the duplicateLOC. Moreover, if a block is duplicated in consecutive lines, only one new line is added to the duplicateLOC. To check if the duplicated code is in consecutive lines we use a boolean flag, which turned to false when seeing a block that is different from the ones in the map. We handle consecutive lines as a special case because otherwise, if we had eg 8 lines of the same thing, we would get that 12 lines are duplicated instead of 7, which is obviously wrong, since the total lines are less than 12. Also, the first time we find a block, we do not consider it as duplicated lines, because it is just the original code.

```
tuple[map[str, int], bool, int] addToMap(map[str, int]
    blocks, str block, bool consecutive, int
   duplicateLOC) {
    if (block in blocks) {
        // if consecutive
        if (consecutive) {
            duplicateLOC += 1;
        } else {
            // if not consecutive
            duplicateLOC += 6;
        }
        consecutive = true;
        blocks[block] += 1;
    } else {
        // if new block
        blocks[block] = 1;
        consecutive = false;
    }
    return <blocks, consecutive, duplicateLOC>;
}
```

The duplication percentage for the project is calculated by summing up the individual unit sizes. A well-structured system should aim to maintain code duplication at or below 5%. Exceptionally efficient systems may achieve even lower levels, with duplication rates below 3%. However, when code duplication

exceeds 20%, it is a clear indication that source code erosion has reached a critical point, and the situation is no longer under control. The ranking is calculated based on the duplication percentage.

rank	duplication
++	0-3%
+	3-5%
o	5-10%
-	10-20%
–	20-100%

Table 6: Rating schema for Code Duplication

4.1.5 Unit Complexity

Source code complexity pertains to the intricacy of code unit structures. Complex units are challenging to understand and test, negatively impacting the system's analyzability and testability. To assess complexity at the smallest executable and testable level, it's best to calculate **cyclomatic complexity**(CC) for each unit.[2]

Method of Calculation:

1. Calculate unit complexity by visiting each method.

```
visit(asts) {
    case Declaration decl: \method(_, _, _, _, _):
        {unitComplexities[decl.src] =
       unitComplexity(decl);}
    case Declaration decl: \method(_, _, _, _): {
       unitComplexities[decl.src] = unitComplexity
       (decl);}
    case Declaration decl: \constructor(_, _, _, _
       ): unitComplexities[decl.src] =
       unitComplexity(decl);
}
```

2. (a) Initial complexity for all methods is 1.
   (b) Increment the complexity by 1 for each of the followings execution branches (for, foreach, do, while, if, ?, case, default case, catch, throw, infix, return statements)[4][7]

```
visit(ast) {
    case \if(_, _): complexity += 1;
    case \if(_, _, _): complexity += 1;
    case \foreach(_, _, _): complexity += 1;
    case \for(_, _, _): complexity += 1;
    case \for(_, _, _, _): complexity += 1;
```

```
    case \do(_, _): complexity += 1;
    case \while(_, _): complexity += 1;
    case \case(_): complexity += 1;
    case \defaultCase(): complexity += 1;
    case \conditional(_, _, _): complexity += 1;
    case \catch(_, _): complexity += 1;
    case \throw(_): complexity += 1;
    case \infix(_, op, _): {
        if (op == "&&" || op == "||") complexity
           += 1;}
    case \return(_): {
        if (!firstReturn) complexity += 1;
        else firstReturn = false;}
}
```

3. Calculate unit size for each method.
4. Calculate Risk profile of the code based on the unit complexity.

Unit Complexity	Risk evaluation
1-10	Low risk
11-20	Moderate risk
21-50	High risk
> 50	Very high risk

Table 7: Risk evaluation based on unit complexity

5. Averaging these complexities helps smooth out extreme cases which is calculated by determining the percentage of lines of code within units categorized at various risk levels.
6. Calculate ranking based on the risk profile.

Rank	Moderate	High	Very High
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
–	-	-	-

Table 8: Complexity Ranking

4.1.6 Unit Interfacing

Software products where more source code resides in units with large interfaces are deemed to be harder to maintain. Units with a large interface often indicate a unit with multiple responsibilities which is harder to modify. Large interfaces are

also error prone, especially if multiple parameters have the same type and can thus be accidentally supplied in the wrong order. This also impedes the speed of development. To maximize the rating of a product for the unit interfacing property, the software producer should avoid units with large interfaces. The size of the interface of a unit can be quantified as the number of parameters (also known as formal arguments) that are defined in the signature or declaration of a unit.

The thresholds we used for the risk profiles and the ranking are the following[1]:

risk profile	**number of params**
low	0-2
moderate	3-4
high	5-6
very high	> 6

Table 9: Unit interfacing risk profiles

rank	**moderate**	**high**	**very high**
–	25%	0%	0%
-	30%	5%	0%
o	40%	10%	0%
+	50%	15%	5%
++	-	-	-

Table 10: Unit interfacing ranks

4.1.7 Comments

Comments enhance the readability and understanding of the source code. We use comments for documenting the source code, clarifying why specific statements are written, and specifying instructions that aid in debugging the code. Comments should focus on enhancing the understandability and maintainability of the source code and should not explain every single statement in the code.

It's important to note that there is no standard code-to-comment ratio[7][6]. However, the absence of comments can result in poor code readability and maintenance. Therefore, we consider one comment per ten lines to be adequate. Units with one comment per hundred lines or no comments are categorized as very high risk. Those with five or fewer comments are considered high risk, while units with five to ten comments fall into the moderate risk category. Units with more than ten comments are categorized as low-risk. Based on this ranking, we calculate the ranking according to the table below. Users can take action based on this ranking, similar to other metrics.

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

Table 11: Comments ranks

4.2 System level metrics

4.2.1 Analysability

How easy it is to diagnose the system for deficiencies or to identify the parts that need to be modified. It depends on analysability, duplication, unit size and unit testing. So, we took the average of them.

4.2.2 Changeability

How easy it is to make adaptations to the system. It depends on complexity per unit and duplication. So, we took the average of them.

4.2.3 Testability

How easy it is to test the system after modification. It depends on complexity per unit, unit size and unit testing. So, we took the average of them.

4.2.4 Stability

How easy it is to keep the system in a consistent state during modification. It depends on unit testing. So, it has the same rank as the unit testing metric.

4.2.5 Maintainability

It depends on all of the above. So, we took the average of Analysability, Testability, Stability and Changeability, as a very simplified metric.

5 Design Decisions

We utilized a Hash Map to determine duplications, enabling us to swiftly verify whether a block is duplicated or not. If we had employed a List for the same purpose, we would have needed to traverse through the entire List and compare multiple elements, leading to increased time consumption and complexity. The Unit Complexity and Unit Size are stored in a Hashmap using a common key. This method has facilitated the calculation of the Unit Complexity score. [5]

Our modular approach not only streamlines metric calculations but also fosters scalability within our project. By compartmentalizing functionalities

into reusable modules, we simplify the integration of new features and enhance the codebase's maintainability. This structured organization has significantly improved collaboration and promoted a more efficient workflow.

6 Results

6.1 Test Project

metric	rank	score
Volume	++	Lines of code: 569
Unit Size	–	Risk - Low:38%, Moderate:13%, High:13%, Very high:38%
Unit Testing	++	Coverage - Low:50%, Moderate:0%, High:25%, Very high:25%
Duplication	–	Duplicate lines: 480, Duplication percentage: 84%
Unit Complexity	–	Complexity - Low:12%, Moderate:11%, High:22%, Very high:55%
Unit Interfacing	++	Complexity - Low:100%, Moderate:0%, High:0%, Very high:0%
Comments	–	Risk - Low:15%, Moderate:0%, High:0%, Very high:85%
Analysability	o	
Changeability	–	
Testability	o	
Stability	++	
Maintainability	o	
Scalability		Execution Time: 15 Seconds

Table 12: Results with TestProject

6.2 Project 'Smallsql'

Metric	Rank	Score
Volume	++	Lines of code: 24004
Unit Size	-	Risk - Low:87%, Moderate:9%, High:3%, Very high:1%
Unit Testing	++	Coverage - Low:43%, Moderate:35%, High:11%, Very high:10%
Duplication	o	Duplicate lines: 1510, Duplication percentage: 6%
Unit Complexity	–	Complexity - Low:70%, Moderate:9%, High:9%, Very high:12%
Unit Interfacing	+	Complexity - Low:89%, Moderate:10%, High:1%, Very high:0%
Comments	–	Risk - Low:28%, Moderate:6%, High:10%, Very high:57%
Analysability	+	
Changeability	-	
Testability	o	
Stability	++	
Maintainability	+	
Scalability		Execution Time: 1 Minute 14 Seconds

Table 13: Results with Smallsql

6.3 Project 'Hsqldb'

metric	rank	score
Volume	+	Lines of code: 168624
Unit Size	-	Risk - Low:78%, Moderate:13%, High:6%, Very high:4%
Unit Testing	++	Coverage - Low:44%, Moderate:27%, High:21%, Very high:8%
Duplication	-	Duplicate lines: 16776, Duplication percentage: 10%
Unit Complexity	–	Complexity - Low:58%, Moderate:16%, High:13%, Very high:12%
Unit Interfacing	-	Complexity - Low:82%, Moderate:14%, High:3%, Very high:1%
Comments	–	Risk - Low:45%, Moderate:6%, High:9%, Very high:40%
Analysability	+	
Changeability	-	
Testability	o	
Stability	++	
Maintainability	+	
Scalability		Execution Time: 6 Minutes 48 Seconds

Table 14: Results with Hsqldb

7 Accuracy and Improvement

Creating a test project, it was easier to ensure that our results are accurate. We also checked that we cover some edge cases, e.g. line comment inside block comment should be considered as one line of comment. What we could have done differently:

- Unit Size: we could use different thresholds for the risk profiles and the ranking. However, this is based on empirical knowledge and we could only know what is better reading more papers. But still, this would probably not be absolute.
- Unit Testing: we could have checked that the methods that are being called are builtin and not functions that exist in the project that simply contain the substring "assert" in their name. This could be misleading. Also, we could also have used different thresholds. In general, we consider this metric not a very smart one, since the number of asserts does not say much about the quality of the tests. Test coverage with Clover is probably a better metric.

8 Scalability

Scalability embodies an application's capacity to adeptly manage diverse workloads. Our solution has proven its versatility by seamlessly meeting the distinct requirements of the TestProject, smallSQL project, and HSQLDB, underscoring its robust scalability across diverse environments and databases. An intriguing

comparison emerges: the hsqldb is seven times larger than smallsql, yet our solution operates effectively in both scenarios. Notably, the execution time for TestProject is 17 seconds, 1 minute and 3 seconds for smallsql analysis, and 6 minutes and 48 seconds for hsqldb analysis.

9 Maintainability of our code

We tried to build some maintainable code, because it would be ironic if we did not :)
What we did:

- Clean and clear project structure, even for someone who does not know what the project is about. Seperate folder for tests, separate file for every metric.
- Naming conventions that make sense and describe the functionalities, using CamelCase for alignment through the project.
- Documentation in Latex, and comments whenever necessary to clarify something.
- Clean code, does not contain trash, has reusable functions, not too big and complex functions that do many things at once. Does not contain redundant imports. We avoided duplication.
- Automated unit tests, to find easily bugs.

References

[1] Sig/tÜvit evaluation criteria trusted product maintainability: Guidance for producers (v15.0).

[2] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[3] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE transactions on software engineering*, 25(4):493–509, 1999.

[4] Cameron McKenzie. How to calculate mccabe cyclomatic complexity in java, 2018.

[5] UseTheSource organization. The rascal meta programming language.

[6] Aditya Raj. What is a good comment code ratio?

[7] Adam Tornhill. The bumpy road code smell: Measuring code complexity by its shape and distribution.