Software Maintainability

Part 1: Tool Demonstration

Tool Demonstration

For this project, I used the Resource Standard Metrics (RSM) tool to analyze a simple Java program that generates a Fibonacci sequence. The purpose of this tool demonstration is to evaluate basic software quality metrics that provide insight into the design, complexity, and maintainability of the program. The Java file (Main.java) defines a single class with one method — the main() method — which prints the first 10 numbers of the Fibonacci series using a for loop.

To begin the analysis, I saved the code in a properly formatted .java file and used the RSM Wizard to add the file to the input list. I selected the report titled "Functional Complexity Detail", which includes the default metric Cyclomatic Complexity and allows deeper insight into function-level complexity and structure. The tool successfully processed the file and generated a detailed report showing metrics such as Lines of Code (LOC), Executable Lines of Code (eLOC), Logical Lines of Code (lLOC), total function complexity, and Interface Complexity [1].

Metrics Interpretation

In addition to the default Cyclomatic Complexity metric, I chose Interface Complexity as the non-default metric to analyze. Interface Complexity is a measure of the number of parameters a function takes and the number of return points it has. This metric is useful because it reflects how tightly coupled the function is to its inputs and outputs, which directly affects readability and maintainability [3][5].

The RSM report showed that the main() function has a Cyclomatic Complexity (Vg) of 2, which is low and indicates a very simple control structure — in this case, the loop used to generate Fibonacci numbers. According to McCabe’s complexity thresholds, any value below 10 is considered low risk and easy to test and understand. This confirms that the function has minimal branching and is not likely to contain hidden logic errors.

The total function complexity was calculated as 4, accounting for the cyclomatic value (2), one function parameter (String[] args), and one return point. This metric gives a more comprehensive understanding of the overall cognitive load needed to comprehend and maintain the function. The Interface Complexity of 2 indicates that the function has one parameter and one return point. A low interface complexity value like this suggests that the function has minimal coupling with external components. This simplicity enhances modularity and testability, as the function can be easily isolated and reused in other contexts without introducing dependencies or side effects.

Together, these metrics show that the program is well-structured for its purpose. The function is compact, with 10 physical lines of code, 8 executable lines (eLOC), and 7 logical lines (lLOC). This indicates that the function is not bloated with unnecessary statements, and its purpose is focused and well-defined.

Overall, the results suggest that the code is of high quality in terms of maintainability and design. The low complexity and low interface overhead make it easy to read, test, and refactor if needed. Although this is a simple program, the metrics demonstrate how RSM can be used effectively to quantify software quality attributes [1].

Metrics Report and Source Code

Source Code:

```
1   class Main {
2     public static void main(String[] args) {
3       int n = 10, firstTerm = 0, secondTerm = 1;
4       System.out.println("Fibonacci Series till " + n + " terms:");
5       for (int i = 1; i <= n; ++i) {
6         System.out.print(firstTerm + ", ");
7         int nextTerm = firstTerm + secondTerm;
8         firstTerm = secondTerm;
9         secondTerm = nextTerm;
10      }
11    }
12  }
13
```

Metric Report:

```
                    ~~ Project Functional Metrics ~~

Function: Main.main
Parameters: (String[] args)
Complexity   Param 1      Return 1      Cyclo Vg 2       Total        4
LOC 10       eLOC 8       lLOC 7        Comment 0        Lines       10

Total: Functions
LOC 10       eLOC 8       lLOC 7        InCmp 2        CycloCmp       2

Function Points         FP(LOC) 0.2     FP(eLOC) 0.2    FP(lLOC)     0.1

-------------------------------------------------------------------------

                    ~~ Project Functional Analysis ~~

Total Functions .......:          1  Total Physical Lines ..:         10
Total LOC .............:         10  Total Function Pts LOC :        0.2
Total eLOC ............:          8  Total Function Pts eLOC:        0.2
Total lLOC.............:          7  Total Function Pts lLOC:        0.1
Total Cyclomatic Comp. :          2  Total Interface Comp. .:          2
Total Parameters ......:          1  Total Return Points ...:          1
Total Comment Lines ...:          0  Total Blank Lines .....:          0
       ------     -----     -----     ------     ------     -----
Avg Physical Lines ....:      10.00
Avg LOC ...............:      10.00  Avg eLOC ..............:       8.00
Avg lLOC ..............:       7.00  Avg Cyclomatic Comp. ..:       2.00
Avg Interface Comp. ...:       2.00  Avg Parameters ........:       1.00
Avg Return Points .....:       1.00  Avg Comment Lines .....:       0.00
       ------     -----     -----     ------     ------     -----
Max LOC ...............:         10
Max eLOC ..............:          8  Max lLOC ..............:          7
Max Cyclomatic Comp. ..:          2  Max Interface Comp. ...:          2
Max Parameters ........:          1  Max Return Points .....:          1
Max Comment Lines .....:          0  Max Total Lines .......:         10
       ------     -----     -----     ------     ------     -----
Min LOC ...............:         10
Min eLOC ..............:          8  Min lLOC ..............:          7
Min Cyclomatic Comp. ..:          2  Min Interface Comp. ...:          2
Min Parameters ........:          1  Min Return Points .....:          1
Min Comment Lines .....:          0  Min Total Lines .......:         10

-------------------------------------------------------------------------

                         ~~ File Summary ~~

C Source Files *.c ....:          0  C/C++ Include Files *.h:          0
C++ Source Files *.c* .:          0  C++ Include Files *.h* :         0
C# Source Files *.cs ..:          0  Java Source File *.jav*:          2
Other Source Files ....:          0
Total File Count ......:          2


        Shareware evaluation licenses process only 20 files.
        Paid licenses enable processing for an unlimited number of files.

_________________________________________________________________________

          Report Banner - Edit rsm.cfg File
```

Key Metrics from RSM Report:

- Function: Main.main
- LOC: 10
- eLOC: 8
- lLOC: 7
- Cyclomatic Complexity: 2
- Interface Complexity: 2
- Total Function Complexity: 4

Part 2: Software Maintainability Measure

Maintainability Measure Identification

One widely recognized software maintainability measure is the Maintainability Index (MI). The Maintainability Index is a composite metric that provides a numeric value to represent how maintainable a piece of software is [4][5]. It is especially useful for identifying problematic code sections that might become maintenance bottlenecks or sources of bugs over time. The MI is designed to be easy to interpret: the higher the score, the more maintainable the code is.

The original formula for the Maintainability Index, developed by Paul Oman and Jack Hagemeister, is [4]:

```
MI = 171 - 5.2 * ln(Halstead Volume) - 0.23 * Cyclomatic Complexity - 16.2 * ln(LOC)
```

In this formula:

- Halstead Volume measures the size of the implementation based on the number of operators and operands in the code [4].
- Cyclomatic Complexity (McCabe's complexity) quantifies the number of linearly independent paths through a function, essentially capturing its decision complexity [3].
- LOC (Lines of Code) provides a measure of code length and potential effort to understand or navigate it.

Some modern versions of the Maintainability Index adjust the constant weights, include the percentage of comment lines, or normalize the result to fall between 0 and 100. In these versions, an MI score:

- Above 85 indicates high maintainability,
- Between 65 and 85 suggests moderate maintainability,
- Below 65 warns of poor maintainability and high maintenance risk.

The Maintainability Index is particularly valuable in large codebases where manual code review is impractical. It helps software teams prioritize refactoring efforts and monitor trends in code quality over time [2]. One key advantage of the MI is that it synthesizes several core aspects of maintainability into a single, quantifiable metric. This makes it easy to compare functions or modules across a project and spot areas needing improvement without delving into each metric individually [3].

Relation to RSM Tool Metrics

The RSM (Resource Standard Metrics) tool does not calculate the Maintainability Index directly, but it does generate all of the necessary component metrics needed to compute it. This includes:

- Cyclomatic Complexity, under its complexity analysis reports [1],
- Halstead Metrics (available in certain report profiles) [4],
- Lines of Code (LOC, eLOC, and lLOC).

These component metrics directly align with the variables in the Maintainability Index formula. For example, the RSM report I generated for my Java Fibonacci program includes a Cyclomatic Complexity of 2 and a LOC of 10. If I had also enabled the Halstead metrics report in RSM, I could use those values to plug into the MI formula and estimate maintainability.

The Cyclomatic Complexity metric helps determine how complex the control flow is, which impacts testing and debugging difficulty. The LOC metric affects readability and effort, as longer code blocks are harder to navigate. If Halstead Volume were available, it would reflect the cognitive effort needed to understand the code's logic based on its operators and operands.

In this way, RSM acts as a feeder tool for computing or approximating the Maintainability Index, even if it does not output the MI value explicitly. More importantly, RSM allows developers to track changes over time and compare metrics across different files or modules. For example, a

function with increasing complexity and LOC over time may signal declining maintainability, even without computing a formal MI score.

Overall, the Maintainability Index is conceptually aligned with the insights provided by RSM, and using both in tandem can offer a rich view of code quality. RSM's role in generating the underlying data makes it a practical tool for anyone interested in measuring and improving maintainability using MI or other derived indicators [1].

References

[1] M Squared Technologies LLC (2009) *Resource Standard Metrics (RSM) User's Guide.* Available at: https://msquaredtechnologies.com/index.html (Accessed: 4 May 2025).

[2] P. Oman and J. Hagemeister (1992) *Metrics for Assessing a Software System's Maintainability*. Available at: https://ieeexplore.ieee.org/document/242525 (Accessed: 4 May 2025).

[3] T.J. McCabe (1976) *A Complexity Measure IEEE Transactions on Software Engineering.* Available at: https://ieeexplore.ieee.org/document/1702388 (Accessed: 4 May 2025).

[4] M. Halstead (1977) *Elements of Software Science (Operating and programming systems series)*. Available at: https://dl.acm.org/doi/10.5555/540137 (Accessed: 4 May 2025)

[5] S. R. Chidamber and C. F. Kemerer (1994) *A Metrics Suite for Object Oriented Design IEEE Transactions on Software Engineering*. Available at: https://ieeexplore.ieee.org/document/295895 (Accessed: May 4 2025)