Series 1 Report

Dertje Roggeveen and Emily Perry

November 2025

1 Volume

The SIG maintainability model uses a system-level volume metric. Volume aims to capture the overall size of a software system, typically measured in lines of code (LOC). In our implementation, we follow this definition and compute volume by summing the physical LOC of every Java compilation unit in the project. Using Rascal's M3 model, we iterate over all locations that are identified as Java compilation units and read their contents. We then count the number of newline characters to obtain the physical LOC for each file and aggregate them into a system-wide total.

Because the SIG papers do not provide explicit public thresholds for mapping LOC to the five-point rating scale ("++" to "- -"), we defined our own categories, inspired by commonly used size ranges in software measurement (small, medium, large, very large systems). Concretely, we use the following thresholds in LOC:

- `++` if LOC $\leq$ 10,000
- `+` if 10,000 $<$ LOC $\leq$ 50,000
- `o` if 50,000 $<$ LOC $\leq$ 200,000
- `-` if 200,000 $<$ LOC $\leq$ 1,000,000
- `--` if LOC $>$ 1,000,000

These thresholds are configurable in the code but fixed for all experiments in this assignment. When running our tool on the `smallsql` project, the total LOC falls into the 10–50 kLOC band, yielding a volume rating of `+`. This indicates a moderately sized system: large enough that maintenance effort matters, but not so large that size alone dominates maintainability concerns.

2 Unit Size Metric

The unit size metric measures the length of individual units and maps this to a risk profile and an overall rating on the SIG scale ("++" to "- -") [1]. In our implementation, a *unit* is a Java method. Using Rascal's M3 model, we obtain

all method locations in the project and read their full source text. The size of a unit is defined as its physical number of lines of code (including comments and blank lines), computed by counting newline characters.

Each method is first classified into a risk category based on its LOC. The SIG documentation provides qualitative guidance but no exact public thresholds. We therefore chose a set of intuitive cut-offs that align with the spirit of the SIG examples, and that clearly distinguish short utility methods from long, potentially problematic ones. See figure 1.

Figure 1: Unit Size Risk Criteria

After assigning a risk category to each method, we compute a system-level risk profile by counting how many methods fall into each category and converting these counts into percentages. To derive a single unit-size rating for the project, we follow the same percentage-based thresholds that SIG uses for complexity per unit [1] (see also figure 2):

- `++` if at most 25% of the methods are at moderate risk and none are high or very high risk;
- `+` if at most 30% are moderate risk, at most 5% are high risk, and none are very high risk;
- `o` if at most 40% are moderate risk, at most 10% are high risk, and none are very high risk;
- `-` if at most 50% are moderate risk, at most 15% are high risk, and at most 5% are very high risk;
- `--` otherwise.

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

Figure 2: Unit Size Risk Thresholds

On `smallsql`, our risk profile lies in the "`-`" band: a substantial fraction of methods are longer than 30 lines, and some methods exceed 60 lines. This suggests that unit size is a potential maintainability concern, since larger units are harder to understand and test in isolation.

We also encountered unpublished alternative thresholds (for example, 1–20 LOC as "no risk", 21–50 as "low risk", etc.), but we deliberately chose to base our implementation only on publicly available and reproducible criteria.

3 Unit Complexity Metric

The unit complexity metric assesses how complex individual units are, again resulting in both a risk profile and a rating on the "++" to "- -" scale. We measure complexity using cyclomatic complexity, a well-established metric that counts the number of independent paths through the code.

In Rascal we compute cyclomatic complexity by traversing the abstract syntax tree (AST) of each unit and incrementing a counter whenever we encounter a control-flow construct that introduces a decision point:

- `if` (with or without `else`),
- `switch`–`case`,
- loops (`for`, enhanced `for`, `while`, `do-while`),
- `catch` blocks,
- conditional expressions (the ternary `?:` operator).

Each unit starts with a base complexity of 1, and each of the above constructs adds 1 to the cyclomatic complexity.

Per unit, we then classify the resulting cyclomatic complexity value into risk categories using the thresholds from the SIG paper, which in turn are based on guidelines from the Software Engineering Institute [2]:

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
> 50	untestable, very high risk

Figure 3: Cyclomatic Complexity Risk Categories

As with unit size, we convert these counts into percentages and apply the same percentage thresholds to obtain an overall unit-complexity rating for the project.

For `smallsql` this leads to a rating of `--`, which means that the share of units with at least moderate complexity is relatively high (e.g., more than half of the units are at least moderate risk, or a notable fraction are high or very high risk). This indicates that, from a complexity perspective, many units may be difficult to understand and modify safely. A threat to validity here is that we currently approximate units using the available AST declarations in Rascal; this may over- or under-approximate the exact method set compared to SIG's tooling, but it still captures the general distribution of control-flow complexity in the system.

4 Duplication

The SIG model includes duplication as a factor influencing both analysability and changeability. A system with high duplication requires developers to understand and modify the same logic in multiple places, which increases the chance of inconsistent changes and regressions. The SIG definition of duplication is based on the percentage of code that appears in clones of six lines or more.

Our implementation closely follows this definition using a simple textual clone-detection approach:

1. Using the M3 model, we collect all Java compilation units and read their contents.
2. Each file is split into lines.
3. For every file, we slide a window of six consecutive lines over the list of lines.
4. For each 6-line block, we construct a block string and store it in a map from block text to the list of positions at which it occurs.
5. Any 6-line block that occurs more than once is considered duplicated, and all lines in all its occurrences are marked as duplicated.
6. Finally, we compute the duplication percentage as:

$$\text{duplication} = \frac{\text{number of duplicated lines}}{\text{total LOC}} \times 100.$$

This implementation is intentionally simple: it does not normalise whitespace, ignore comments, or attempt to detect near-miss clones. Nevertheless, it is consistent with the SIG emphasis on textual similarity and is sufficient for the purposes of this assignment.

Because explicit SIG thresholds for duplication are not publicly available, we use widely referenced ranges for mapping the percentage to a rating:

- `++` if duplication $\leq 2\%$,
- `+` if duplication $\leq 5\%$,
- `o` if duplication $\leq 10\%$,
- `-` if duplication $\leq 20\%$,
- `--` otherwise.

On `smallsql`, our analysis finds that at most 2% of the lines are part of duplicated 6-line blocks, which results in a duplication rating of `++`. Compared to the size and complexity results, this suggests that duplication is not a major contributor to reduced maintainability in this system.

5 Maintainability Aspects

The SIG maintainability model defines several sub-characteristics—such as analysability, changeability and testability—that are derived from underlying source code metrics [1]. In our tool, we compute scores for these sub-characteristics by aggregating the metric ratings we described in the previous sections (volume, unit size, unit complexity and duplication).

For all maintainability aspects we use the same internal numeric scale for the five rating categories:

$$\texttt{++} \mapsto -2, \quad \texttt{+} \mapsto -1, \quad \texttt{o} \mapsto 0, \quad \texttt{-} \mapsto 1, \quad \texttt{--} \mapsto 2.$$

The relevant metric ratings are mapped to numbers, averaged, and then mapped back to a rating using fixed thresholds (e.g., $(-\infty, -1.5)$ yields `++`, $[-1.5, -0.5)$ yields `+`, $[-0.5, 0.5)$ yields `o`, etc.). This mirrors the SIG idea of aggregating source-level metrics into ISO 9126 sub-characteristics while keeping the scoring scheme consistent across aspects.

5.1 Analysability

Analysability represents how easily a developer can understand the system's structure and behaviour. In the SIG maintainability matrix, analysability depends on several underlying source code properties. Following that matrix, we model analysability as the (unweighted) average of the four basic metrics we implemented:

- Volume,
- Unit complexity,
- Unit size,
- Duplication.

Given the ratings for these metrics, our tool internally converts them to the numeric scale $\{-2, -1, 0, 1, 2\}$, computes their average, and maps the result back to a rating from "++" to "--". This design makes it straightforward to reuse the same aggregation logic for other sub-characteristics, and it reflects the SIG view that analysability is not driven by a single metric but by a combination of size, complexity and code duplication.

5.2 Changeability

Changeability describes how easily the system can be modified to implement new requirements or fix defects. The SIG model relates changeability primarily to complexity and duplication: complex units are harder to change correctly, and duplicated code multiplies the effort and risk of each change.

In our implementation, we therefore base the changeability score on two metrics:

- Unit complexity,
- Duplication.

We map both ratings to their numeric values in $\{-2, -1, 0, 1, 2\}$ and take the average. The outcome is mapped back to a rating using the same thresholds as for analysability. Equal weighting is consistent with SIG's guideline that equal weights are a reasonable default, and we did not identify a clear reason to favour either complexity or duplication more strongly in the context of this assignment.

5.3 Testability

Testability reflects how easily the system can be tested effectively. According to the SIG model, testability depends on unit complexity, unit size and the quality of the unit tests themselves [1]. Complex and very large units are harder to test thoroughly, while the presence of structured unit tests can partially compensate for this.

We follow this structure and base our testability score on:

- Unit complexity,
- Unit size,
- Test quality.

In principle, the test quality metric would be computed by analysing the unit tests in the project. However, for this assignment we did not implement a dedicated test-analysis step. To avoid silently ignoring test quality altogether, we conservatively treat the test-quality rating as "--" in our current implementation. This is pessimistic, but it makes our assumptions explicit and ensures that missing test analysis does not falsely inflate the testability score.

All three ratings are converted to numeric values, averaged, and mapped back to the five-point scale. Because test quality is fixed at "--", the resulting testability scores should be interpreted as lower bounds: actual testability might be better if real tests are present and of sufficient quality.

5.4 Overall Maintainability

The SIG paper notes that a single global maintainability score can be less informative than the individual sub-characteristics, but the assignment explicitly asks us to compute such an overall score. We therefore define overall maintainability in a way that is consistent with the SIG maintainability matrix: by aggregating the sub-characteristics that our tool currently supports:

- Analysability,
- Changeability,
- Testability.

Again, we map each sub-characteristic rating to a numeric value in $\{-2, -1, 0, 1, 2\}$, compute the (unweighted) average, and map the result back to the "++" to "--" scale. This produces a single maintainability indicator that matches the structure of the SIG model, while still allowing the individual sub-characteristics to be reported alongside it for more fine-grained analysis.

6 Results

In this section we report the metric and maintainability scores produced by our tool for the two subject systems, `smallsql` and `hsqldb`. All scores are on the SIG five-point rating scale from "++" (best) to "--" (worst).

6.1 Metric Ratings

Table 1 summarises the four primary source code metrics for each system.

System	Volume	Unit Complexity	Unit Size	Duplication
`smallsql`	+	--	-	++
`hsqldb`	-	--	--	++

Table 1: Metric ratings for the analysed systems.

For `smallsql`, the results indicate a moderately sized system (volume `+`) with very low duplication (`++`), but with relatively large and complex units (unit size `-`, unit complexity `--`). This combination already suggests that maintainability will be constrained more by internal structure (size and complexity of units) than by system volume or duplication.

6.2 Maintainability Sub-characteristics and Overall Score

Table 2 shows the derived maintainability sub-characteristics and the overall maintainability score for each system.

System	Analysability	Changeability	Testability	Overall
`smallsql`	`o`	`o`	`--`	`-`
`hsqldb`	`-`	`o`	`--`	`-`

Table 2: Maintainability sub-characteristics and overall scores.

For `smallsql`, the analysability and changeability scores are both `o`, which reflects a balance between helpful properties (moderate volume, low duplication) and harmful ones (many large and complex units). Testability is rated `--`, mainly because of the combination of high unit complexity, relatively large methods, and our conservative assumption that test quality is `--`. As a result, the overall maintainability score drops to `-`, indicating that while the system is not unmanageable, its structure and (assumed) test support will make long-term evolution harder.

7 Threats to Validity

Our implementation of the SIG maintainability model involves several design choices and simplifications that may influence the reported scores. We briefly discuss the most important threats to validity.

Metric Definition and Extraction

First, we measure volume and unit size using *physical* lines of code, counting newline characters in the source files and methods. This means that comments, blank lines and formatting differences all affect the LOC counts. The SIG model is generally robust to such variation, but it does introduce noise, especially when comparing systems that follow different formatting conventions.

Second, our notion of a "unit" is derived from the Rascal M3 model as Java methods. While this aligns with the SIG notion of units in most cases, differences in how constructors, anonymous classes or generated methods are represented could slightly change the unit set compared to SIG's proprietary tooling.

Cyclomatic Complexity Approximation

For unit complexity we compute cyclomatic complexity by counting a fixed set of control-flow constructs (e.g., `if`, `for`, `while`, `catch`, ternary conditionals). We deliberately do not count logical operators such as `&&` and `||` as separate decision points, even though some definitions of cyclomatic complexity do. This makes our numbers somewhat lower than in those variants, and may shift a few units across the risk thresholds.

Duplication Detection Heuristics

Our duplication analysis is purely textual and based on exact matches of six consecutive lines. We do not normalise whitespace, ignore comments, or detect near-miss clones. As a result, our duplication percentages are likely to be conservative: small edits, formatting differences, or comment changes can prevent otherwise similar fragments from being counted as clones. Conversely, boilerplate comments or license headers that are copied unchanged across files may inflate duplication if they exceed six lines.

Threshold Choices

The SIG papers describe the shape of the risk profiles and percentage thresholds, but not all exact numbers are publicly documented. Where thresholds were missing (e.g., for unit size and duplication), we selected values that are consistent with the examples and common practice, and documented them in our code. Different but still reasonable threshold choices could lead to slightly different ratings, especially near category boundaries.

Test Quality Assumption

We did not implement an automated analysis of test code. Instead, we conservatively fixed the test-quality rating to `--`. This design decision ensures that we do not overestimate testability, but it also means that the testability and overall maintainability scores for projects with good test suites will be underestimated. In that sense, our reported scores should be interpreted as lower bounds rather than precise estimates of actual testability.

Tooling and Model Limitations

Finally, our analysis depends on Rascal's M3 model and parsing infrastructure. Any limitations or bugs in the underlying Java front-end (e.g., inability to parse certain language features or build configurations) can influence which files and methods are included. For the systems we analysed, the M3 model appeared complete, but we cannot guarantee that it perfectly matches the build configuration that SIG's own tools use.

References

[1] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.

[2] C. S. E. Institute, *Cyclomatic complexity – software technology roadmap*. [Online]. Available: http://www.sei.cmu.edu/str/descriptions/cyclomatic.html.