1. **Which metrics are used?**

   1.1 **Volume**
   The overall volume of the source code influences the analysability of the system.

   1.2 **Complexity per unit**
   The complexity of source code units influences the system's changeability and its testability.

   1.3 **Duplication**
   The degree of source code duplication (also called code cloning) influences analysability and changeability.

   1.4 **Unit size**
   The size of units influences their analysability and testability and therefore affects the system as a whole.

   1.5 **Unit testing**
   The degree of unit testing influences the analysability, stability, and testability of the system.

2. **How are these metrics computed?**

   2.1 **Volume**
   A simple line of code metric (LOC) is used, which counts all lines of source code excluding comment or blank lines. The Programming Languages Table of Software Productivity Research LLC is used to account for language differences. LOC is then ranked as follows(values are in the 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

   2.2 **Complexity per unit**
   Cyclomatic complexity (CC) is calculated for each unit and mapped to one of four risk levels:

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

LOCs of each unit are then mapped to their respective risk evaluation groups. After that is done the rank is assigned based on the % of the lines of code with a given risk evaluation.

Example: System with 26% of lines of code(without comments or whitespaces) with *moderate* risk and 0% *high* and *very high* risk will get a +

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

2.3 **Duplication**
A moving window of 6 lines is used to detect duplicate blocks. The percentage of duplicated lines determines the rank:

Rank	Duplication
++	0 – 3%
+	3 – 5%
o	5 – 10%
-	10 – 20%
–	20 – 100%

Comments and whitespaces are ignored.

2.4 **Unit size**
Approach is similiar to calculating **Unit complexity**.

Method Size Guidelines

Since there were no clear guidelines on the size of methods, we created our own based on the following resource:
About Rule 30.

This paper suggest using 24 lines of code as the optimal number. We would argue that it is very much dependent on the usecase and libraries used. Some add boilerplate, others are quite concise.

The best we can do is propose values that in our proffesional experience were the tell signs and allow for their override.

Citing Bob Martin:

> "This is not an assertion that I can justify. I can't produce any references to research that shows that very small functions are better."

Guidelines for Risk Evaluation

Method Size	Risk Evaluation
0-30	Simple, without much risk
31-50	More complex, moderate risk
51-100	Complex, high risk
>100	Untestable, very high risk

Rank Classification

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

2.5 **Unit testing**
Code coverage is measured using language-specific tools. We used JacoCo, the results and instruction can be found in ./test_coverage/README.md

Ranks are assigned as follows:

Rank	Unit Test Coverage
++	95 – 100%
+	80 – 95%
o	60 – 80%
-	20 – 60%
–	0 – 20%

Additionally, we measure the amount of asserts and present the statistics.

3. **How well do these metrics indicate what we really want to know about these systems and how can we judge that?** These metrics are a solid starting point for evaluating the systems maintability aspects. They can help the organisations prioritise the most critical pieces of the system and guide the initial exploration. While better then flipping a coin, they are to be approached with a grain of salt.

In our opinion the biggest strength of them is the actionability and transparency of methods, which enables comprehensive analysis and building on top of whats already there. Overall the metrics are helpful and informative - they have scientific backing and while people may argue about the details of grading, there is a shared understanding in the industry, that the proposed metrics speak of the quality of a system.

4. **How can we improve any of the above?**

   There is room for improvement in the duplication metric. Currently it is calculated in a crude manner, omitting the duplication of less then 6 lines and exaggerating the risk for the fragments with more then 6 lines duplicated. We could improve it by expanding the duplication window beyond 6, as long as there are any duplicates of that size matching, counting duplicates found on the way. That would complicate the algorithm and most likely increase the time of the measurement. Also in some cases the duplication could be acceptable - there should be a way to exclude those classes/files from measuring.

   For unit size, the better - more fitting set of criteria regarding the desired size and scales for the grading could be introduced.

   Unit testing metric could be enchanced with mutation testing. Code coverage only says, that the project may be tested enough - not if it is. While counting asserts gives some insights it is not informative enough.

Coupling - new non-SIG metric: Apart from implementing the metrics suggested by SIG we implemented *Coupling metric* that describes the coupling between objects. High coupling can decreese *Changability* as change in one file will impact all the files that use it as a dependency.

To calculate it we counted the number of dependencies by class and presented some statistics.

We didn't provide the table for ranking, as we believe it is very much dependent on the framewoks used. It can however give insight into the project and high number of dependencies per class should be something to investigate.

Scalibility of the solution: It is hard to determine it in the terms of O(n) complexity, as the complexity depends on the various factors such as: - Number of lines, - number of classes, - number of methods, - internal complexity of Rascal methods

We belive that for our needs a sufficient proof of the scalibility is a fact that hsqldb finishes within approximetly 10sec. We achieved that after a few iterations of the solution.

Result of running on smallsql:

```
--------------VOLUME--------------
Total lines of code: 22210 KLOC
Volume rank: ++
---------------VOLUME END--------------
---------------DUPLICATION--------------
Duplicated lines number: 804
Duplicated lines percentage: 3.818026403%
Total lines: 21058.0
Duplication rank: +
---------------DUPLICATION END--------------
---------------UNIT SIZE--------------
Moderate: 11.15965429
High:11.67252351
Very high: 9.079684680
Unit size rank: --
---------------UNIT SIZE END--------------
---------------UNIT COMPLEXITY--------------
Moderate: 7.804966797
High:13.72691713
Very high: 5.908305285
Unit complexity rank: --
---------------UNIT COMPLEXITY END--------------
---------------MAINTAINABILITY--------------
Maintainability rank: -
---------------MAINTAINABILITY END--------------
---------------ANALISABILITY--------------
Analisability rank: o
---------------ANALISABILITY END--------------
---------------CHANGEABILITY--------------
Changeability rank: -
---------------CHANGEABILITY END--------------
---------------TESTABILITY--------------
Testability rank: --
---------------TESTABILITY END--------------
---------------COUPLING--------------
Coupling between objects (CBO) statistics:
Average number of dependencies per class: 2
Max number of dependencies in a class: 29
Most coupled classes: ["/smallsql/database/SQLParser"]
---------------COUPLING END--------------
---------------UNIT TESTING--------------
Mean asserts per test method: 4.441988950
Median asserts per test method: 3.
25% of test functions have <= 1 asserts
50% of test functions have <= 3 asserts
75% of test functions have <= 5 asserts
```

```
--------------UNIT TESTING END--------------
ok
```

Results for hsqldb:

```
---------------VOLUME---------------
Total lines of code: 158171 KLOC
Volume rank: +
---------------VOLUME END---------------
---------------DUPLICATION---------------
Duplicated lines number: 7368
Duplicated lines percentage: 4.910199593%
Total lines: 150055.0
Duplication rank: +
---------------DUPLICATION END---------------
---------------UNIT SIZE---------------
Moderate: 14.44537003
High:15.93015894
Very high: 19.90070308
Unit size rank: --
---------------UNIT SIZE END---------------
---------------UNIT COMPLEXITY---------------
Moderate: 12.96904441
High:10.80650171
Very high: 14.68474997
Unit complexity rank: --
---------------UNIT COMPLEXITY END---------------
---------------MAINTAINABILITY---------------
Maintainability rank: -
---------------MAINTAINABILITY END---------------
---------------ANALISABILITY---------------
Analisability rank: o
---------------ANALISABILITY END---------------
---------------CHANGEABILITY---------------
Changeability rank: -
---------------CHANGEABILITY END---------------
---------------TESTABILITY---------------
Testability rank: --
---------------TESTABILITY END---------------
---------------COUPLING---------------
Coupling between objects (CBO) statistics:
Average number of dependencies per class: 9
Max number of dependencies in a class: 68
Most coupled classes: ["/org/hsqldb/dbinfo/DatabaseInformationFull"]
---------------COUPLING END---------------
---------------UNIT TESTING---------------
```

```
Mean asserts per test method: 2.615384615
Median asserts per test method: 0.
25% of test functions have <= 0 asserts
50% of test functions have <= 0 asserts
75% of test functions have <= 4 asserts
---------------UNIT TESTING END---------------
ok
```

Steps to recreate:

```
import Main;
main(|your project location|);
```

How to run tests?

```
import Tests;
:test
```