Report Rascal Assignment

Bas van de Louw, Brent Van Handenhove

15-Jan-2022

Table 1: Team

Bas van de Louw	852361065
Brent Van Handenhove	852152619

Table 2: Work division

Bas van de Louw	Design, code, report
Brent Van Handenhove	Design, code, report

Contents

1 Design

1.1 Assumptions made

1.1.1 Only considering Java files

We will only be counting Java source files. Any compiled Java code or code written in another language is ignored.

1.1.2 What is a 'Line of Code'

In order to calculate the total amount of code or unit size, we have to first define what a 'line of code' is.

For the purpose of this project, we decided that a line should check all the below rules for it to be considered a line of code:

- Not blank: Trimmed length of line is 0
- Not a comment: Starts with //, /*, * or ends with */

A small caveat here is that in Java, lines of legitimate code may actually start with * as seen below:

```
int x=10;
int y=25;
int z=x
*
y;
```

However, this is code smell and quality code should not have this as it is difficult to read. For the purpose of this assignment, we assume that no non-comment lines will start with *. In a real life scenario, as illustrated above, it may happen. As such, a more mature analysis tool should mark this kind of code as a problem to be fixed.

This also means we count import other modules as lines of code. We also count the module declaration itself as a line of code. Whether or not these are really lines of code can be debated. For the purpose of this assignment we have decided that they are, as they are necessary for the code to work.

1.1.3 Lines of Code

In addition to the assumptions and caveats in 1.1.2, we have noticed that Eclipse tends to generate some additional files for the SmallSQL / HSQLDB projects that may affect our metrics. For example: The amount of lines of code for SmallSQL increases by roughly 2000 when this happens. For the rest of this report, assume that we have discarded any generated files and are looking solely at the base project.

1.1.4 Unit count

No explicit assumptions were made about what counts as a unit. Every method within the project is counted towards the total unit count. As such, we can define the unit count as the amount of methods within the project. This includes any abstract methods without an implementation.

1.1.5 Unit Size

Each unit within the project has a size. We calculate the size as the amount of lines of code within the unit that check the requirements set in 1.1.2. Note that the caveats apply here as well.

1.1.6 Unit Complexity

Each unit within the project has a cyclomatic complexity. The complexity of unit is equal to the sum of the complexity of each statement within the unit plus 1.

Each statement within table 3 increases the unit complexity by a value of 1.

Table 3: Cyclomatic Complexity Statements

if(_, _)	if statement
case(_)	case within a switch
do(_, _)	do loop
while(_,_)	while loop
for(_, _, _)	for loop
foreach(_, _, _)	foreach loop
catch(_, _)	catch statement
conditional(_, _, _)	conditional
infix(_, "&&", _)	infix AND
infix(_, "||", _)	infix OR

1.1.7 Duplication

By ignoring method and argument types, code may be marked as duplicate where it does in fact make sense to have similar implementations.

1.1.8 Test Coverage

In order to calculate test coverage, some likely inaccurate assumptions were made. Primarily, we do not check for any 'real' unit testing that may be present - such as JUnit - because we can neither rely on assuming one test library is used nor can we account for any unknown or proprietary test libraries. In addition, any Rascal locations that point to an outside library do not appear to be tracked.

To explain our process for calculating test coverage, we go through the following steps in pseudo code:

1. Create a list of every method (=unit) and its location in the project
2. Create two empty lists for 'real' methods and 'test' methods
3. For every method
   (a) If it contains at least 1 assert, add it to the list of test methods
   (b) If it has no asserts and the access modifier is public, add it to the list of real methods
4. Create an empty list for 'covered' methods
5. For every method in test methods
   (a) Get every method called within the test method
   (b) If they are not yet in the list for covered methods, add them
6. The amount of covered methods divided by the amount of 'real' methods is the test coverage percentage

You may have noticed that we are only counting public methods. This is because we assume that private methods are either:

- Used within a public method (and thus indirectly tested)
- Only used internally (and thus not of any interest to us)

We make an additional assumption in that we count every method call that follows the pattern `'/^assert/i'` as an assert. In doing this, we implicitly assume the following:

- That any test method's name starts with 'assert'
- That any other method's name does not start with 'assert'.

Lastly, we do not do a final check that every covered method is public. This is because we assume that unit tests are not within the same file or package, so they should not be able to access anything other than public methods.

1.2 Visualizing the data

Table 4 shows the computed metrics, which provide us a solid overall picture of the maintainability state of the evaluated projects. When visualizing this data we wanted to give an insight on how to fix these maintainability issues. For this reason we chose a fine-grained view, so that issues can be traced back to their source.

1.2.1 Using fine-grained views

We began by designing a menu to let users traverse the available projects. Then, for unit sizes, cyclomatic complexity, and duplication, we introduced three fine-grained views. Risk factors can be used to filter these views to an even more detailed view. We can, for example, limit the cyclomatic complexity visualization to units with a cyclomatic complexity greater or equal to 50. Software Improvement Group (SIG) [1] considers this code to be untestable and a very high risk.

Because the goal is to eventually fix these issues, we made these views interactive. Clicking on a unit in any fine-grained view will cause the visualization to open a new tab with the source code location. This makes locating the root of major maintainability problems a simple task. Appendix A contains further information on the fine-grained views.

2 Results

2.1 Metrics

The calculated metrics for SmallSQL and HSQLDB can be found in table 4.

2.2 Interpreting the results

2.2.1 Lines of Code

2.2.2 Unit Size

We recognize that not counting comments, while more accurate for the metric, may lead to situations where methods are overly documented. We think this may be something that should either be included in the unit size calculation somehow, or that a separate metric should be created to indicate overly documented code.

2.2.3 Unit Complexity

2.2.4 Duplication

2.2.5 Test Coverage

3 Evaluation and Reflection

3.1 Validity

3.1.1 Unit Test Coverage

Something we wanted to do but couldn't was to verify that the assert decl was inside of a test library. We could do this by checking if, for example, 'junit.org' was inside the decl path. However, it appears Rascal does not correctly handle a decl to an outside library. Instead, they are 'unknown', meaning this additional

Table 4: Metric results

Table 5: SmallSQL

General Data	
Lines of Code	24 850
Number of Units	2 358

Unit Size	
Simple	38.1 %
Moderate	15.0 %
High	20.5 %
Very High	26.3 %

Unit Complexity	
Simple	63.0 %
Moderate	10.6 %
High	16.9 %
Very High	9.5 %

Duplication	
Duplication	15.5 %

Unit Testing	
Coverage	29.6 %
Asserts	961

Metric Scoring	
Unit Size Score	- -
Volume Score	+ +
Unit Complexity Score	- -
Duplication Score	-
Unit Testing Score	-

SIG Scoring	
Analyzability	o
Changeability	-
Stability	-
Testability	- -
Overall Maintainability	-

Table 6: HSQLDB

General Data	
Lines of Code	160 929
Number of Units	9 424

Unit Size	
Simple	19.8 %
Moderate	16.8 %
High	20.6 %
Very High	42.9 %

Unit Complexity	
Simple	59.9 %
Moderate	16.4 %
High	13.4 %
Very High	10.4 %

Duplication	
Duplication	19.6 %

Unit Testing	
Coverage	6.0 %
Asserts	691

Metric Scoring	
Unit Size Score	- -
Volume Score	+
Unit Complexity Score	- -
Duplication Score	-
Unit Testing Score	- -

SIG Scoring	
Analyzability	-
Changeability	-
Stability	- -
Testability	- -
Overall Maintainability	-

check we wanted to do was not possible at this time. This means any violations of our assumption in 1.1.8 are not double-checked and will skew results towards a higher coverage.

Another limitation to using our method of estimating test coverage is that

'partial tests' are not properly handled. What we mean with this is that test methods that by themselves do not contain an assert are not evaluated, even if they are called by another test method that does contain asserts. Likewise, a test method with an assert that does not itself invoke the methods it tests are not properly handled. In other words, we do not support nested testing. This may skew results towards a lower coverage.

We can conclude that while we believe our method of calculating test coverage is at worst an educated guess and at best an acceptable estimation.

3.2 Evaluating the visualisation

3.3 Cooperation

To ease collaboration, we used git for this project.

Having worked together before, we knew each other's strengths and weaknesses. Even though we both work full-time, we constantly communicated about ideas, questions, and specific code implementations. In the evenings, we would each work on different parts of the code - often sending over what we were currently working on for quick feedback. This allowed us to implement features quickly while having a form of code reviews.

After we had a good foundation, we started using pull requests for further features and bug fixes. We did not do this right away as we do not believe it fruitful to complicate the pipeline so early into development. We found that this works well for us.

There were constant informal code reviews.

No specific work division was done, as neither of us had any experience with Rascal. It is our belief that on a project of smaller scope with a small team, working on a bit of everything leads to a better result.

References

[1] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," 10 2007, pp. 30 – 39.

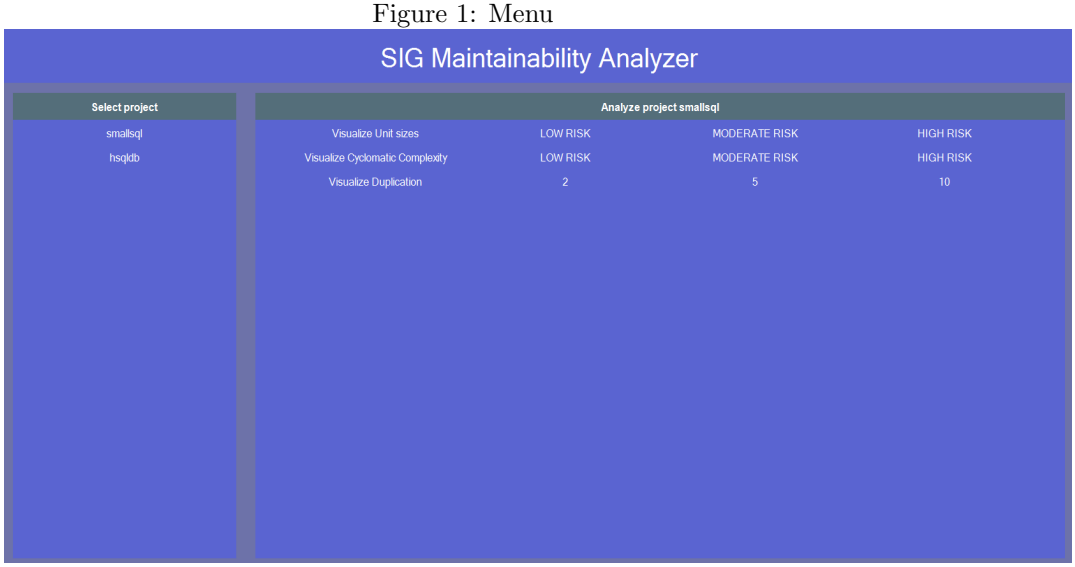
Appendix

A Visualization

A.1 Menu

Menu containing a project selector and analysis tool.

Figure 1: Menu



A.2 Fine-grained views

For every fine-grained view, the same concept applies. The user can hover over a unit and then the file, path and lines of code (loc) will be displayed on screen. Additionally, the user can click on the unit and a new window will be opened where the source code is located. In below fine-grained view of high risk units (loc 40+) from the HSQLDB project we can easily see that the biggest unit has a size of 872.

Figure 2: Fined grained view high-risk HSQLDB: Unit sizes

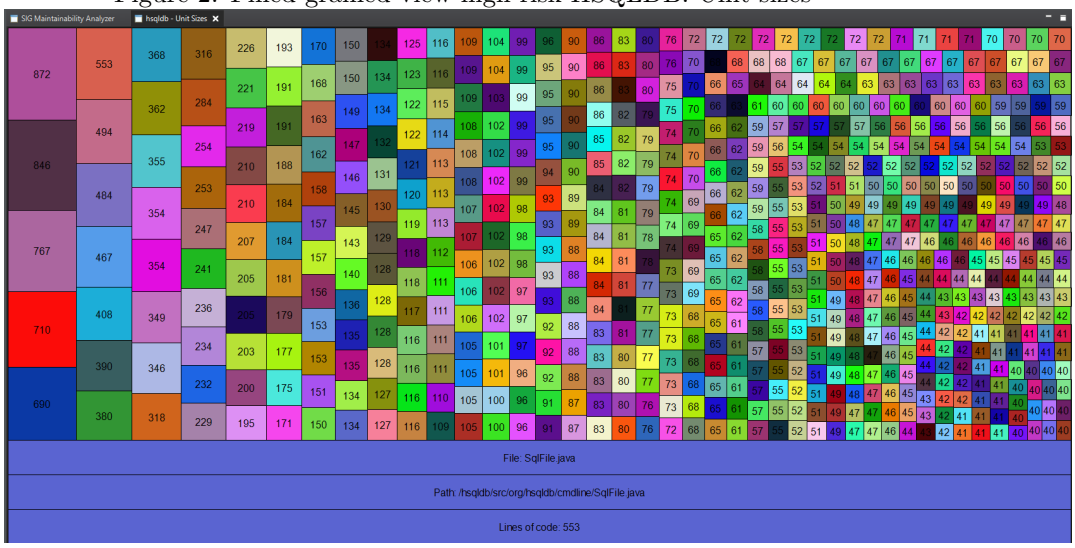
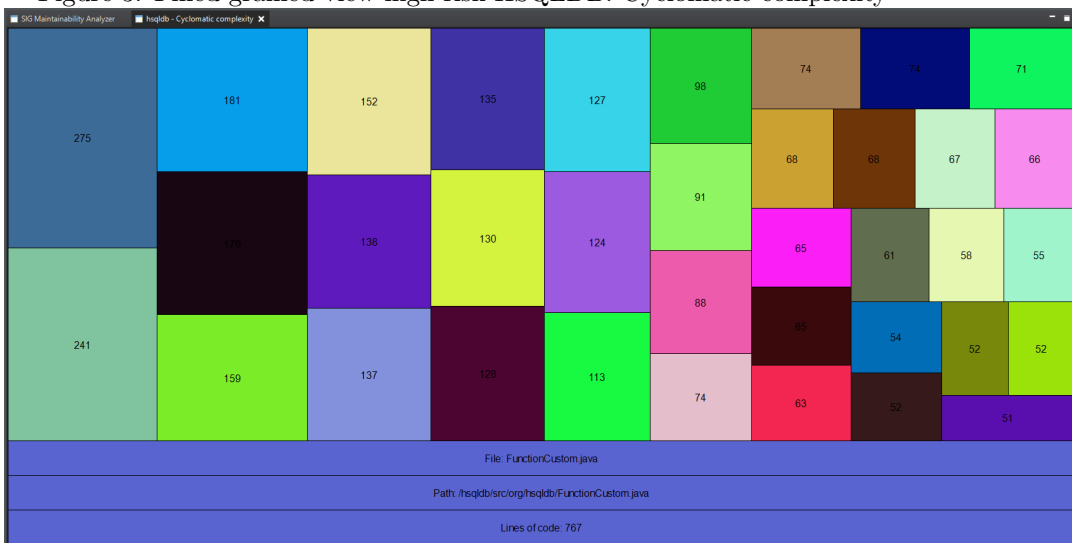
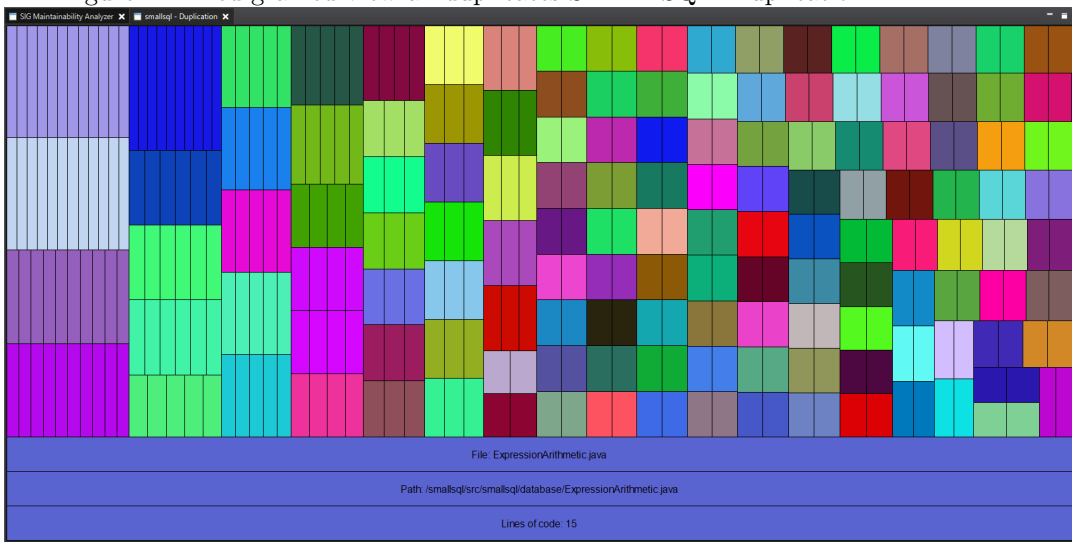


Figure 3: Fined grained view high-risk HSQLDB: Cyclomatic complexity



Visualizing duplicates is done by combining clones into a single entity. We can easily see the amount of clones by counting the amount of boxes per entity. Of course, we can also navigate to each occurrence of every clone by clicking on the instance.

Figure 4: Fined grained view all duplicates SMALLSQL: Duplication



Note that above screenshots are created using a specific configuration, we can generate different fine-grained views by applying different filters in the menu.