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 */

One 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 usually code smell except in rare complex mathematical calculations (of which there are none in either project). For this assignment, we will assume code should not have this. 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 handle this more appropriately.

Some additional things to consider are:

- Our program was run on Windows - any variations on Linux and Mac may be due to differences in how the OS handles text.

- There was no style normalization, the project was analysed 'as is'.

We count the import of 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. We have decided that they are, as they are essential for the code to function.

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.

The pseudocode can be found in the appendix at B.1.

We also count each of these as a line of code:

- Test code (as long as it is in Java)

- Single-line curly brackets

1.1.4   Unit Count

SIG defines a unit as 'the smallest named piece of executable code' Every implemented method within the project is counted towards the total unit count. As such, we can define the unit count as the amount of implemented methods within the project. Methods without an implementation are discarded, as they are not executable.

The pseudo-code for determining unit size is as follows:

1. Get the project AST

2. For each node

    Check if it is a method with an implementation (so that it is executable)

    If so, increment the unit count

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 and that we also count single-line curly brackets.

1.1.6   Unit Complexity

Each unit within the project has a cyclomatic complexity. This is equal to the amount of possible execution paths a unit has [1]. The complexity of unit is equal to the sum of the complexity of each statement within the unit plus 1. Each statement within table 6 increases the unit complexity by a value of 1.

We do not count the complexity of invoked methods towards the complexity of a method. This would not make sense, as the metric is supposed to be used to estimate which methods could be broken up into smaller methods.

1.1.7 Duplication

For code to be considered duplicate, it has to check the following requirements:

- The node tree should have at least 25 nodes (see below)

- The location of the code cannot be unknown (see below)

- It has to be at least 6 lines long (see 1.1.2)

We insist the node tree has to be at least 25 nodes long (declarations, statements or expressions all count towards this number). This helps us weed out any code that we deem too small to be an issue, such as simple calculations or getters/setters. While most of these wouldn't check the 6-line-minimum, it's still possible to spread very simple code over many lines (see caveat at 1.1.2)

Code with an unknown location is discarded, as it is impossible for us to verify that we are not checking the same code twice if our algorithm finds it identical to other code.

There are some additional rules we have established within our metric:

- By ignoring method and argument types, code may be marked as duplicate where it does in fact make sense to have similar implementations. We are aware of these possible false-positives and accept this risk.

- Only exact matches (after discarding types) are considered. It is possible to check for near-misses and mark them as well, but we have decided not to do that for this project.

- Duplicate code is counted on a unique occurrence basis, meaning if there are 3 chunks of duplicate code we count 3 of them as duplicate.

- Code marked as duplicate that is part of a larger chunk of code marked as duplicate is discarded. This prevents us from counting so-called 'subclones' towards the total amount of duplicate code, as this would effectively count some code twice. More on that below.

The pseudo-code for our duplication metric is available in the appendix at B.2.

By default, we also prune what we call 'subclones'. These are bits of code marked as duplicate that are also part of a larger bit of code that is also marked as duplicate. If we didn't perform this, we would be counting these twice. This can happen because the AST is evaluated per-node instead of per-method. While this allows us to pick up duplicate code inside of methods as well as duplicate method declarations, it does mean we treat code inside a method as a separate thing to check.

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, refer to the pseudocode in the appendix at B.3

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 every asserting method's name starts with 'assert' (e.g. assert, assertEquals, assertNull, etc)

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

Lastly, we do not perform a final check to ensure 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 3 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) [2] 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 3.

Table 3: Metric results

Table 4: SmallSQL		Table 5: HSQLDB	
General Data		General Data	
Lines of Code	24 850	Lines of Code	160 929
Number of Units	2 358	Number of Units	9 424

Unit Size		Unit Size	
Simple	38.1 %	Simple	19.8 %
Moderate	15.0 %	Moderate	16.8 %
High	20.5 %	High	20.6 %
Very High	26.3 %	Very High	42.9 %

Unit Complexity		Unit Complexity	
Simple	63.0 %	Simple	59.9 %
Moderate	10.6 %	Moderate	16.4 %
High	16.9 %	High	13.4 %
Very High	9.5 %	Very High	10.4 %

Duplication		Duplication	
Duplication	15.5 %	Duplication	19.6 %

Unit Testing		Unit Testing	
Coverage	29.6 %	Coverage	6.0 %
Asserts	961	Asserts	691

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

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

2.2 Interpreting the results

2.2.1 Interpretation of metrics

Lines of code within a project do not say much about the quality of that project, merely its scope. It could provide some idea of how complex it may be, but even then it is entirely possible for large software to have no high risk units in either size or complexity.

Our other metrics indicate that:

1. Both projects have poor scores for unit size and complexity. These often go hand-in-hand, as larger units tend to be more complex. Both projects are intended to be small in filesize, and for SmallSQL especially it may not make sense to split the project to improve these stats as that would increase filesize.

2. There is quite some duplicate code in both projects. False positives aside, we think this is mainly due to trying to decrease footprint and in part due to the nature of working with SQL.

We recognize that not counting comments, while more accurate for the unit size metric, may lead to situations where methods are overly documented. Whilst poorly documented code is harder to understand, overly documented code may also be harder to understand. This is something that is not taken into account when calculating these metrics, but it also hard to quantify.

3 Evaluation and Reflection

3.1 Validity

3.1.1 Lines of Code

We added unit testing for our project volume calculation metric, which also affects 3.1.2.These verify that our metric is accurate by our own defined rules in 1.1.2. Our unit test in listing 1 checks for our defined requirements and returns '18', which is the expected amount.

An additional, extremely rudimentary way of checking if our result is realistic is by taking the total file size of all source files and dividing it by the average size of a line of code. If we put the average line of code at 50 bytes (or 50 characters), we come to a result of 26 662 for SmallSQL, which matches our metrics.

To get to our average, we took the file 'SSResultSet.java' within SmallSQL, calculated the amount of characters and divided it by the amount of lines. This was a total of 36 759 characters over 1 280 lines (including blanks and comments). This resulted in an average characters/line of 28.7. Rounding up to 30 brings us to 44 437 lines. If we discard comments (5 105) and blank lines (658) we are at 30 996 characters over 903 lines, or roughly 34 characters/line. This brings us to 38 000 lines of code. This margin of almost 50% above our

estimated amount seems excessive, but given that some files are almost 40% comments (such as LongTreeList.java) it is reasonable to assume that this is plausible.

3.1.2   Unit Size

We wrote unit tests that are simple enough to verify manually. Our metric has reported the expected results, confirming the accuracy of our implementation given our rules and assumptions set in 1.1.2.

The unit test applied for lines of code (see 3.1.1) is also relevant here. In addition, manual counting can and was done for a few random units in SmallSQL and HSQLDB. These matched what our metric calculated.

3.1.3   Unit Complexity

We wrote unit tests to ensure our unit complexity metric is accurate. The expected results for the unit tests below are 2 and 4 respectively, which is what our metric reports. This indicates that we implemented it correctly. See listing 2 in the appendix.

We also performed some manual checks on SmallSQL files by cross-referencing the reported complexity by our metric with a manual calculation of the expected result. Whenever we did this, the expected result would match our output.

3.1.4   Duplication

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

Take a look at an example from SmallSQL in listing 3 in the appendix.

Because our duplication-check ignores method return types, it will mark these methods as duplicate, as they check the requirements in 1.1.7. There are likely cleaner ways to handle code such as this (because admittedly, it is almost identical), but we realize that it is not always pragmatic to do so.

For the example method, though, the getValue() and null-check segment could have been split into a different method - so in this case, we are fine marking it as duplicate.

More false-positives happen especially when dealing with localization. Small-SQL, for example, contains a localization template and 3 language files, all as java files. We ignore the contents of strings, so all language files are marked as duplicate. This makes sense, as they will all use the same keys for localization purposes. This means these are false-positives and will skew results towards a higher duplication estimate. Having said that, we propose that having localization files as java files is bad practice and should have been handled using files specifically made to handle data (such as json or csv).

To conclude, we believe that it is possible to have acceptable duplication. Sometimes it is more pragmatic to solve an issue by copying some code than it is to design complex architecture, especially when considering the cost and scope of a project.

3.1.5   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 check we wanted to do was not possible at this time. This means any 'assert'-named method that is not an actual assert/test will violate our assumption in 1.1.8 but will not be caught. This means it will count as a test method and could 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.

To conclude, we believe our test coverage metric to be an acceptable estimation.

3.2   Evaluating the visualisation

3.2.1   The use of colors and data

In our visualisations, we have used colors to our advantage. Risk is represented by color, with green being low risk, orange medium risk and red high risk. This allows us to gather critical information about a unit at a glance. We also include the value of the metric inside the unit boxes on the grid. For example: When visualising unit size, each box will contain text with the size value inside of it. This combined with the color presents easy-to-access and valuable information when reading our metrics.

3.2.2   Unit Size

For an image of size visualisation, please refer to figure A.2.

We visualize unit sizes in a grid where each block represents a unit. The size of the block is directly dependent on the unit size. This allows for an easy to read visualisation where it is easy to pinpoint high risk classes right away. Hovering over a block will provide information in the bottom of the screen, such as location of the code and how many lines of code are in the unit. Clicking on a block will open the code it represents.

3.2.3   Unit Complexity

For an image of Cyclomatic Complexity visualisation, please refer to figure A.2.

We visualize unit complexities in a grid where each block represents a unit. The size of the block is directly dependent on the unit complexity. We use the

same visualisation as for unit sizes as we strongly believe it is the most readable for quick access to what's important: What classes should be investigated.

Hovering over a block will provide information in the bottom of the screen, such as location of the code and how many lines of code are in the unit. Clicking on a block will open the code it represents.

3.2.4   Duplication

For an image of duplication visualisation, please refer to figure A.2.

When looking at our duplication visualisation, it is immediately clear which chunks of code are considered duplicates of each other. This method of grouping them provides an immediate, visual clue of just problematic a set of clones may be. Having many clones could indicate that the code should be brought into its own class, or that information should be made available for re-use somehow.

However, what is not clearly identifiable in this view is just how many lines of code are duplicate. It is entirely possible to have a screen full of potentially problematic duplicates, only for all of them to be 6 duplicate lines. This could still be problematic, but there is no visual distinction between these types of clones and, for example, entirely copied classes of 200 lines of code. However, it is possible to click the offending chunks to quickly be taken right to the code in question.

3.2.5   Navigating the visualisation

When first starting up our visualization tool, a menu will appear. On the left-hand side, you can select a project. A coarse-grained view of the metrics of the selected project is presented on the right-hand side. To go to the fine-grained view of a metric, you can click on the metric.

3.2.6   Comments on developing the visualisation

We really liked the many examples we got of advanced and attractive visualizations such as D3.js[1]. Unfortunately, we weren't able to make it as sophisticated as we would have liked due to our inexperience with Rascal and its inherent limitations as well as a time constraints. Taking inspiration from some of the things we learned about software quality, we decided to focus on a single, clear view to convey what we believe is important as fast as possible. We would have liked to create the visualisation in a language more suited for UI design.

Some of the difficulties we faced trying to visualize in rascal is that registering callbacks within a loop does not seem to work correctly. This has led us to have to implement some pragmatic fixes.

We feel that if we had been able to leverage a more specialized language for UI, the process of creating a visualization would have been much quicker and faster, with a better end result. That said, we believe we convey the information we believe is most interesting.

---

[1] https://d3js.org/

3.3 Cooperation

3.3.1 Working as a team

Having worked together before, we know 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 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.

3.3.2 Version control

To ease collaboration, we used git for this project.

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 code reviews, mostly informal due to the small team size.

References

[1] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[2] 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

We also give a button to visualize risk levels per category, as shown in the samples above. This allows us to look at the same data in a new light. For example, we can determine whether the project has a few high-risk units or whether it is a more general issue. In below HSQLDB example, we can easily see that there are (too) many units in the high risk category, and therefore refactoring this code will almost certainly take a long time.

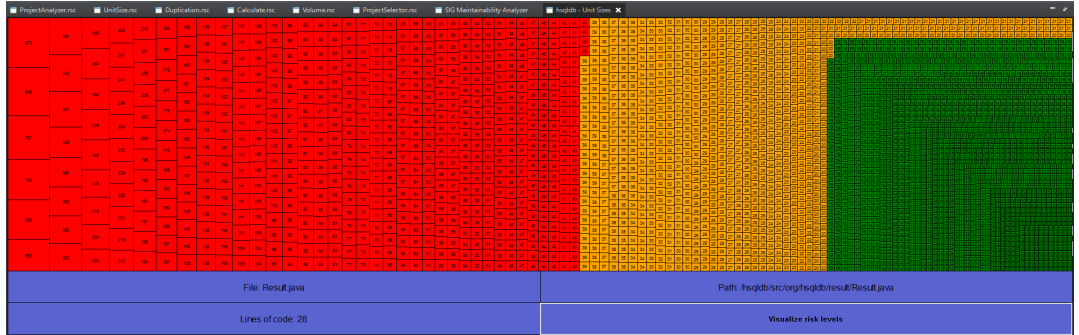Figure 4: Fined grained view HSQLDB categories: Unit sizes

Figure 5: Fined grained view HSQLDB categories: 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 6: Fined grained view high risk duplicates HSQLDB: Duplication
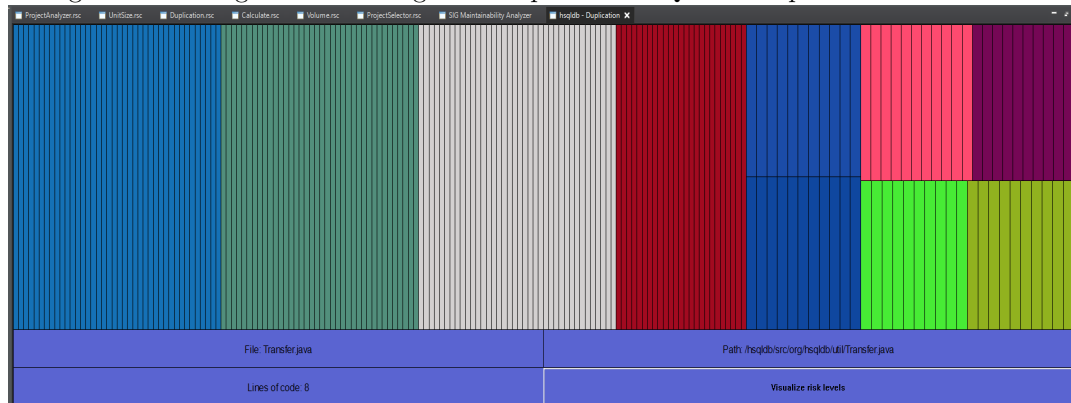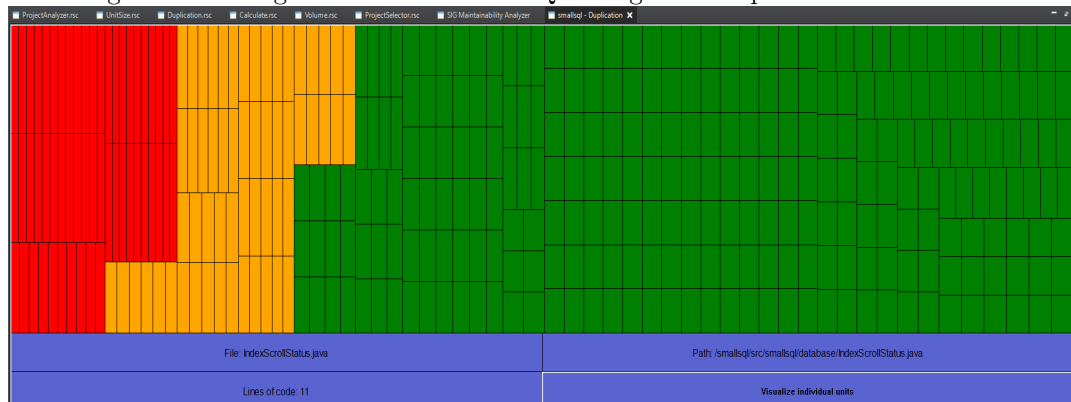

Figure 7: Fined grained view SMALLSQL categories: Duplication

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

B Pseudocode

B.1 Lines of Code

The pseudocode for determining the total lines of code in a project is as follows:

1. Get all java files in a project

2. For each file

    Count the amount of lines that are not blank and not a comment

B.2 Duplication

Finding clones happens as follows:

1. For every node in the AST

    (a) If the location is valid (not unknown and at least 6 lines of code)

    (b) Normalize the node (remove all return types, method types, literal values) for comparison

    (c) Check if this exact node (loc) was not already tracked to prevent literal identical duplicates

    (d) If it is not an identical match, it is a clone - add it to the list with the node as key

2. Do a pass over the list of detections and remove any nodes where there are not at least 2 values in the list (which would indicate there are no duplicates)

By default, we also prune what we call 'subclones'. These are bits of code marked as duplicate that are also part of a larger bit of code that is also marked as duplicate. If we didn't perform this, we would be counting these twice. This can happen because the AST is evaluated per-node instead of per-method. While this allows us to pick up duplicate code inside of methods as well as duplicate method declarations, it does mean we treat code inside a method as a separate thing to check.

1. Do a runtime pass is done to check if duplicate code is not part of a larger chunk of duplicate code to avoid counting the same code twice

2. Do a second pass is done after all duplicate code has been checked in so that a 'subclone' added before its larger clone (which would not be picked up during our previous pass) is also not counted

B.3 Unit Test Coverage

To estimate the test coverage within a project, we:

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

C   Figures and samples

Table 6: 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

Listing 1: Lines of Code Example

```
package unit_tests.resources;

public class VolumeExampleTest {

        public static String test1() {
                return "test1";
```

```java
        }

        /**
         * This comment should not be counted
         *
         * @return A string
         */
        public static String test2() {
                return "test2";
        }

        /*
         * Ignore
         */
        public static int test3() {
                int a = 1;
                int b = 1;
                return a;
        }

        // Not a valid JavaDoc and should not be counted towards volume
        public static int test4() {
                int a = 1;
                return a;
        }
}
```

Listing 2: Unit Complexity Tests

```java
public static String test1() {
        int x = 0;
        if(x == 0) {
                x++;
        }
        return "test";
}

public static String test2() {
        int x = 0;
        int y = 1;
        if(x == 0) {
                x++;
        }
        while(x == 1) {
                if(y == 10)
                {
                        x++;
```

```
                }
                else {
                        y++;
                }
        }
        return "test";
}
```

Listing 3: Duplication Sample

```
\begin{lstlisting}
public Date getDate(int i) throws SQLException {
    try{
                Expression expr = getValue(i);
        wasNull = expr.isNull();
                if(wasNull) return null;
                return DateTime.getDate( expr.getLong() );
    }catch(Exception e){
        throw SmallSQLException.createFromException( e );
    }
}
public Time getTime(int i) throws SQLException {
    try{
                Expression expr = getValue(i);
        wasNull = expr.isNull();
                if(wasNull) return null;
                return DateTime.getTime( expr.getLong() );
    }catch(Exception e){
        throw SmallSQLException.createFromException( e );
    }
}
```

D   Project Development Stats

Table 7: Project stats

Files	49
Commits	130
Feature Branches	7
Bugfix Branches	1
Other Branches	3
Pull Requests	13
Dives into Rascal source	29
Rascal Documentation Visits	1 217
Google Searches with 'rascal'	78