

Software evolution report series 1

- Simon Schneider
- Laurance Saess

Lines of code

Lines of code is one of the simplest or even the simplest metric that one can think of.

Heitlager defines it as `all lines of source code that are not comment or blank lines`. [Heitlager, 2007]

Java has two types of comments:

1. single line comments that start with `//`
2. multiline comments that are surrounded by `/*` and `*/`

After reading all files from the project that we are going to analyze we have to remove these comments and blank lines.

We can do this by replacing matches that follow these two regular expressions with an empty string:

1. `/^(\\s*\\/\\/)/` for single line comments (arbitrary white space followed by `//`)
2. `/\\/*[\\s\\S]*?*\\/` strings that go over multiple lines and start with `/*` and end with `*/`, between them is any character (nonwhitespace, or whitespace (including new lines))

Threat to validity: Comments inside strings

Before applying these two regexes we have to consider edge cases like the following:

```
System.out.println("Hello world /*"  
    + "asd*/asdsdasd"  
);
```

Fortunately Java only supports strings over a single line. They have to be concatenated by a `+` to reach over multiple lines. Applying the regex solution from above will result in a wrong solution (LOC = 2 instead of 3):

```
System.out.println("Hello world asdsdasd"  
);
```

To tackle this we replace every comment start and end inside a string before passing it to the mentioned

regex with this function:

```
return visit(stringContent){
    case /<stringstart:.*><commentstart:\\.*><stringend:.*>/ => "\"<string
start><COMMENT_START_TOKEN><stringend>\\""
    case /<stringstart:.*><commentend:*\\/><stringend:.*>/ => "\"<stringst
art><COMMENT_END_TOKEN><stringend>\\""
}
```

The expected result is then produced (LOC = 3)

```
System.out.println("Hello world %%% || RASCAL_COMMENT_START || %%%"
    + "asd%% || RASCAL_COMMENT_END || %%%asdsasd"
);
```

Fortunately, we were able to find this behaviour with an automated test: [LinesOfCodeTest.rsc](#) via the analyzation of [CommentsInStrings.java](#).

Thread to validity: Curly Brackets

The original paper of Heitlager gives a very broad definition of "lines of code" since its supposed to be language agnostic. Languages like python do not use curly brackets, languages like PHP favor brackets in new lines and Java users usually prefer the opening bracket in the same line as the method definition:

PHP: (4 LOC ?) - formatted according to [PSR-2](#)

```
public function main()
{
    echo "Hello, World";
}
```

Java: (3 LOC ?)

```
public static void main(String[] args) {
    System.out.println("Hello, World");
}
```

Pyhton: (2 LOC !)

```
def happyBirthdayEmily():
    print("Hello, World")
```

The question arises if we should count bracket lines in the first place since the LOC greatly depend on the style of the programmer or the current conventions of the community.

There are also other practitioners that do not count bracket lines as lines of code [Klint, 2009, p. 9].

We decided to let the user of our tool decide which approach he wants to use, even though we suggest to count bracket lines to stay as close to [Heitlager, 2007] as we can. (See [Configuration.rsc](#))

Cyclomatic complexity

With this metric, it is possible to give an indication of the complexity of a program. It measures the number of paths through the source code of a code section. In theory, the lower the cyclomatic complexity, the easier the code should be to understand. However, in practice this is not always the case.

The following method gets a lower complexity:

```
public int complexity1(bool a) {
    int asd = 234;
    asd *= 2342;
    .....
    int asv = asd * 24234 + (a > 234 ? 234 : 3) + 234 / asd;
    asd += asv - 234;
    return asd + 123;
}
```

Then the following method:

```
public int complexity1(bool a) {
    switch(a) {
        case 1:
            return 123523;
        case 2:
            return 12353;
    }
    return 23423344;
}
```

However, the last method is easier to understand. The complexity can be used as an indication. But it should not be used as the only measure.

How do we calculate the complexity

We calculate the complexity over all units (methods, constructors and static constructors) in the project. We iterate over every statement in the unit and increment the complexity for every `do`, `foreach`, `for`, `if`, `case`, `catch`, `while`, conditional and `||` / `&&` infix operators, because all of these listed elements will result in a new branch.

How do you count a statement that always branches to one branch

There is the case that, for example, an `if` statement will only result into a single branch:

```
bool b = true;
if(b) {
    ...
}
```

This method will only result in one branch, but for the reader, it adds complexity and there are many cases when you cannot know that it only will result in one branch.

What do we call a unit in Java

Cyclomatic complexity is about the complexity of the smallest possible unit. In Java it those are methods, constructors and static constructors.

Do you calculate lambdas

For example, when you define a lambda inside a method, does it add up to the complexity? In our application, we add the complexity to the unit. It is part of the unit, and it adds complexity for the reader. So, we count it as complexity.

Other tools, like Checkstyle do not calculate these kind of things to the complexity.

Test

As a test, we compared the result of the complexity function with Checkstyle. We generated lists of the complexity per method and compared them:

Checkstyle output

```
/src/smallsql/database/Command.java:106:9: Cyclomatic Complexity is 3
/src/smallsql/database/Command.java:116:5: Cyclomatic Complexity is 2
/src/smallsql/database/Command.java:127:9: Cyclomatic Complexity is 2
/src/smallsql/database/Command.java:134:5: Cyclomatic Complexity is 3
/src/smallsql/database/Command.java:148:5: Cyclomatic Complexity is 2
/src/smallsql/database/Command.java:86:5: Cyclomatic Complexity is 3
/src/smallsql/database/Command.java:96:5: Cyclomatic Complexity is 2
```

Rascal output

```

/src/smallsql/database/Command.java| (2404,307,\<83,4\>,\<91,5\>): Cyclomatic Complexity is 3"
/src/smallsql/database/Command.java| (2719,207,\<93,4\>,\<100,5\>): Cyclomatic Complexity is 2"
/src/smallsql/database/Command.java| (2931,360,\<102,1\>,\<110,2\>): Cyclomatic Complexity is 3"
/src/smallsql/database/Command.java| (3300,313,\<112,4\>,\<121,5\>): Cyclomatic Complexity is 2"
/src/smallsql/database/Command.java| (3618,337,\<123,1\>,\<132,2\>): Cyclomatic Complexity is 2"
/src/smallsql/database/Command.java| (3963,383,\<134,4\>,\<144,5\>): Cyclomatic Complexity is 3"
/src/smallsql/database/Command.java| (4439,170,\<148,4\>,\<152,5\>): Cyclomatic Complexity is 2"

```

We compared every mismatch and reasoned if their approach is better than ours. One difference that we found was that Checkstyle does not count lambda's. In our opinion, they should be counted to the unit it is inside.

Score

We rank every method to a risk level

Risk	parameters
Low risk	0 - 10
Moderate risk	11 - 20
High risk	21 - 50
Very-high risk	> 50

And then we calculate the total percentage per risk, what results in a score.

Risk	C Low risk	C Moderate risk	C High risk	C Very-high risk
++	∞	max 25%	Max 0%	max 0 %
+	∞	max 30%	Max 5%	max 0 %
o	∞	max 40%	Max 10%	max 0 %
-	∞	max 50%	Max 25%	max 5 %
--	∞	∞	∞	∞

Code duplication

Code duplicates were the most painful metric to implement. Not because it is hard to find an efficient algorithm for type-0 clones, but because there are many misconceptions and different understandings that led to many discussions among students.

Threat to validity: Counting original lines

This is surprising since the paper of Heitlager gives a pretty clear definition on clones:

We calculate code duplication as the percentage of all code that occurs more than once in equal code blocks of at least 6 lines. When comparing code lines, we ignore leading spaces.

[Heitlager, 2007]

Counting every line that occurs more than once means that we have to count the original lines and all of its duplicates. Since the original line is also a line that "occurs more than once".

On the other hand, there are other sources that give a more elaborate definition of duplicates and some of them state that the original line should not be considered part of the code duplication: [Roy, 2009]

- **Definition 1:** Code Fragment. A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple (CF.FileName, CF.BeginLine, CF.EndLine).
- **Definition 2:** Code Clone. A **code fragment CF2 is a clone of another code fragment CF1** if they are similar by some given definition of similarity, that is, $f(CF1) = f(CF2)$ where f is the similarity function. Two fragments that are similar to each other form a clone pair (CF1,CF2), and when many fragments are similar, they form a clone class or clone group.

To illustrate the difference we can take a look at the following Java code (assume a duplication threshold of 3):

```

00: package java;
01:     public class Duplicates {
02:     public void method1() {
03:         int a = 1;           //DUPLICATE?
04:         int b = 2;           //DUPLICATE?
05:         int c = 3;           //DUPLICATE?
06:         int d = 4;           //DUPLICATE?
07:     }                         //DUPLICATE?
08:     public void method2() {
09:         int a = 1;           //DUPLICATE!
10:         int b = 2;           //DUPLICATE!
11:         int c = 3;           //DUPLICATE!
12:     }
13:     public void method3() {
14:         int b = 2;           //DUPLICATE!
15:         int c = 3;           //DUPLICATE!
16:         int d = 4;           //DUPLICATE!
17:     }                         //DUPLICATE!
18: }

```

If we count the brackets (see problem above) we have 19 lines of code.

The duplicates differ depending on the definition that we use:

- Duplicate lines: **12** [Heitlager, 2007]
- Duplicate lines: **7** [Roy, 2009]

We again decided to stay to the paper of Heitlager by default, but leave it as an option to the user to count the originals or not. (See [Configuration.rsc](#))

Threat to validity: Duplicates across file borders

To calculate the code clones we concatenate the source code of all files in the project and analyze them. Depending on the approach we use for ignoring brackets, we could get false positives because of this.

Two files that would create a false positive like that can be found here: [File1.java](#) and [File2.java](#).

This cannot be desirable because the amount of code duplicates that depends on the order in which the files are read. We, therefore, added a "page break" token to tell the duplication algorithm that he cannot compare code duplicates across file borders.

Problem: Prune / preprocessing the code

To remove the number of lines that the relatively costly code duplication algorithm has to check we are using pruning as a filter before the actual line-search algorithm.

Example:

```
a
b
c
d
e
ZZ
f
g
1
2
3
a
b
c
d
e
f
g
...
```

Now we can iterate over the lines and count the occurrences:

```
"a" -> 3
"b" -> 2
"c" -> 2
"d" -> 2
"e" -> 2
"f" -> 2
"g" -> 1
"1" -> 1
"2" -> 1
"3" -> 1
"ZZ" -> 1
```

Can be replaced by


```
a
b
c
d
e
%UNIQUE_LINE%
f
%UNIQUE_LINE%
%UNIQUE_LINE%
%UNIQUE_LINE%
a
a
b
c
d
e
f
%UNIQUE_LINE%
...
```

This can be even further be reduced to:

```
%UNIQUE_LINES%
a
a
b
c
d
e
f
%UNIQUE_LINES%
...
```

Because we only care for 6 lines that are unique in a row. We then feed this reduced list of lines to an algorithm that knows how to handle the `%UNIQUE_LINES%` token that we introduced.

The first part of the algorithm has a complexity of $O(2n) \Rightarrow O(n)$, the actual search algorithm could be as bad as $O(n^2)$, but with a reduced number of lines. This preprocessing will only be beneficial for cases in which the code does not consist out of non-unique lines.

Unit interfacing

Unit interfacing is a metric for changeability and testability. The more parameters there are for a method, the harder it is to test and change the functionality.

The metric is calculated in a samiliar fashin as the cyclomatic complexity with the following risk divisions for the parameters:

Risk	parameters
Interfacing Low risk	0 - 2
Interfacing Moderate risk	2 - 3
Interfacing High risk	3 - 4
Interfacing Very-high risk	4 - ∞

As described in the following paper:

The risk devision for the whole project is calculated and converted to a risk with the following table:

Risk	I Low risk	I Moderate risk	I High risk	I Very-high risk
++	∞	max 25%	Max 0%	max 0 %
+	∞	max 30%	Max 5%	max 0 %
o	∞	max 40%	Max 10%	max 0 %
-	∞	max 50%	Max 25%	max 5 %
--	∞	∞	∞	∞

We use the same as they used for cyclomatic complexity

Why this metric

Unit interfacing is a metric for changeability and testability. The higher the number of parameters there are for a method, the harder it is to test and change the functionality.

Test

We compared some of our output to the smallsql source code:

```

"updateCharacterStream":3:
public void updateCharacterStream(int columnIndex, Reader x, int length) throws SQ
LException

"testNotInsertRow":0:
private void testNotInsertRow() throws SQLException

"calcMillis":7
static long calcMillis(int year, int month, final int day, final int hour, final i
nt minute, final int second, final int millis)

```

Unit size

The metric is again calculated like the cyclomatic complexity with the following risk deviations for the parameters [Alves, 2011]:

Risk	parameters
U Low risk	0 - 30
U Moderate risk	31 - 44
U High risk	45 - 74
U Very-high risk	75 - ∞

The risk deviation for the whole project is calculated and converted to a risk with the following table:

Risk	U Low risk	U Moderate risk	U High risk	U Very-high risk
++	∞	max 25%	Max 0%	max 0 %
+	∞	max 30%	Max 5%	max 0 %
o	∞	max 40%	Max 10%	max 0 %
-	∞	max 50%	Max 25%	max 5 %
--	∞	∞	∞	∞

Testing

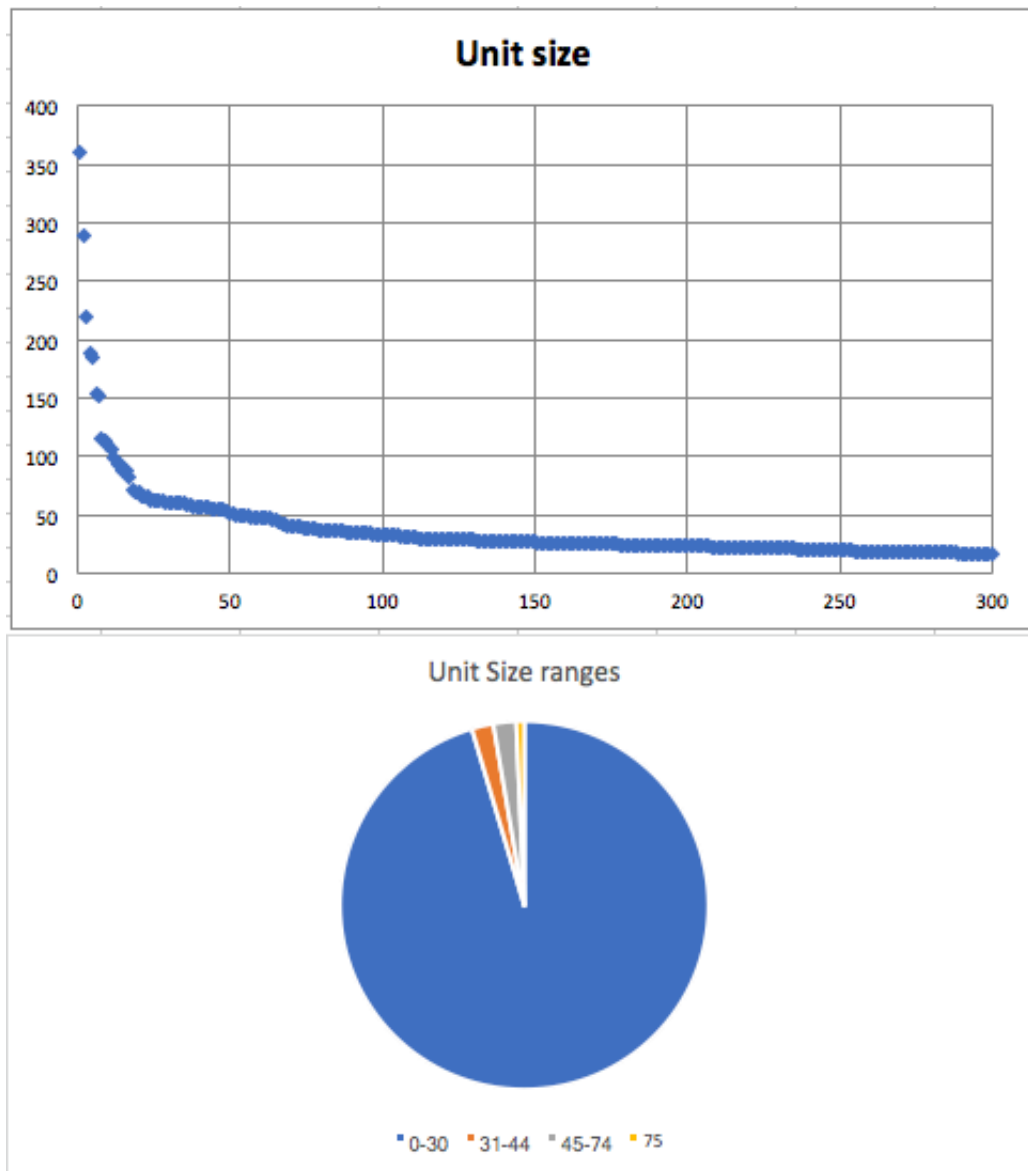
For testing, we calculate how many assertions there are for each test method. We calculate it in the following way:

We divide assertions by methods and multiply it by 100. The output 50 means that there is one assertion for every 2 methods. The output 100 means that there are as many test methods as methods. The output 200 means that there are 2 test methods per method.

Risk	Assertions per production method
++	3-10 assertions per method
+	2-3 assertions per method
O	1-2 assertions per method
-	0.5-1 assertions per method
--	0-0.5 assertions per method

An alternative approach would be to calculate how many assertions there are per test method. We used assertions per because then you get a better idea of how much of the functionality is actually tested. Normally you would do with a dynamic coverage test, but this is not possible in Rascal.

Unit size result for small project



Raw data: [unitsize.csv](#)

Since we already output our data as csv files it is easy to use tools like Excel or R to generate diagrams. We can see that the vast majority of the methods are below 30 lines of code.

Test

This implementation is tested manually. First, you have to specify what base class has to be extended to be called by the unit testing framework. Then you can generate a report with how many assertions there are made and how many methods there are.

Metric results

We executed the metric and got the following data:

- [output.tsv](#) - to view in github
- [output.csv](#) - to download

Bibliography

Alves, T. L., Correia, J. P., & Visser, J. (2011, November). Benchmark-based aggregation of metrics to ratings. In Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA) (pp. 20-29). IEEE.

Heitlager, Ilja, Tobias Kuipers, and Joost Visser. "A practical model for measuring maintainability." Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the. IEEE, 2007.

Klint, Paul, Tijs Van Der Storm, and Jurgen Vinju. "Rascal: A domain specific language for source code analysis and manipulation." Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on. IEEE, 2009.

Roy, Chanchal K., James R. Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach." Science of computer programming 74.7 (2009): 470-495.