Practical Lab Series 1 - Software Metrics

Jovan Maric - 10443762
Hector Stenger - 10398872

Introduction **2**

Project layout **2**

Volume **2**

Unit size **4**

Duplication **4**

Complexity per unit **5**

Unit testing **6**

Testing **6**

Introduction

Below is our result for the first practical lab series done in Rascal. Both of the runtimes done for the different projects are acceptable. We could have done a couple of things differently / better but in the end time was as always of the essence. We learned a lot though.

Project layout

The project layout has been structured according to the following layout:

- Assignment
    - Helpers
    - Metrics
        - Name of metric
    - Projects
- Specs
    - Fixtures
    - Helpers
    - Name of metric
- Runner

We use the Runner as a front loader file which loads all the resources automatically for us by including files, this ranges from test files to actual metric calculation files. All the helper directories are used to store specific helper methods we consider general purpose functionality for that directory. The helper methods we store in the specs directory are helper methods used in the specs and the helper directory in the assignment share functionality we use in the files in the assignment directory.

The biggest thing to note about our project structure is that we have two different folders for tests and actual code. The specs directory contains all the test related code and fixtures. For every metric we wrote there is a specific test file and fixture file. Sometimes we use multiple fixture files to test one specific metric, for instance the ‘volume’ metric.

Volume

In order to calculate the total lines of code we perform the following steps.

1. We read the files so we get an array of strings.
2. We filter out single line comments.
3. We filter out multi line comments.
4. We filter out remaining whitespace.

There are a couple of tricks we are applying to get the job done.

For the single line comments there are two cases we must check for:

1. Lines starting with a comment, double dash (‘//’) in Java.
2. Multi line comments starting and ending in the same line.

The first case is the easiest and involves just a simple pattern check using regex. The Second one is also not that hard since we check if both the starting and ending tag for multiline comments are present in the same line. If only the starting tag is present this means that the ending tag is on a different line and therefore the method won’t do anything.

Double dashed single line comments are just replaced by an empty string, which is okay cause in the end we filter empty lines from the source code. We only do this if the line begins with it, if it doesn’t start with a single line comments than we don’t do anything. If it starts with code it’s a valid sentence, so removing the comment from that line doesn’t have any effect on the total lines of code. It only adds clock cycles. Removing multi line comments on the same line is as simply as replacing the multi line string by using the index from the start and ending tag.

Multi line comments are a bit harder since they could contain actual code before the start and after the end tag. We simply can’t remove the whole line like we did at lines that start with a double dashed comment.

We use two functions to achieve this result. We first scan a line for a starting tag, ‘/*’. If it’s present we then start searching for the corresponding closing tag. We do the searching in a lazy manner. We scan line for line in the given array and return the index of the first pattern we match. The only edge case for this logic is strings that contain starting tags. To counter this fact we check of the given index of a found starting tag is surrounded in string tags.

To delete the multi line comments from the array we pass the indices, starting and closing tag index, and the array to a different function that cleans it and returns it. We then continue to move on to the rest of the array. If no starting tags are found the lazy find functions returns an exception which signals the calling function that it has completed its work and returns the array.

Finally we filters out all the lines that only contain whitespace. We understand that a couple of optimizations could be made. For instance we could reduce the amount of traversals from 3 to 1. We check first for single line comments then for multi line comments and finally we filter out the empty rows from the result.

The second thing we could have improved is the orderBy function we implemented but don’t currently use. For large arrays it has a slow performance time. Instead of using a proven algorithm like ‘quicksort’ we use recursion to find the right place to insert itself.

Unit size

Calculating the unit size is basically using the same logic we applied to the calculation of the total volume of the project. We used the same risk levels for the unit size metric, as were used for the risk levels of the cyclomatic complexity. The reason is mostly because of our previous work experience. We are supporters of the “Keep it simple, stupid” principle and find that more than 10 lines of code have a negative impact on the maintainability of the class.

We started off by loading in all methods from the m3 directories. Each method gets trimmed by removing single- and multiline comments and trimming the whitespaces. Then by means of iteration we traverse each method and calculate the risk level of it.

The risk level method consists of the risk map, which contains the keys: “low”, “moderate”, “high” and “veryHigh” which correspond with the same risk levels of the cyclomatic complexity implementation in the SIG paper. With the above iteration this map gets filled and the values are calculated into a percentage. The output of the risk levels is then used to find the appropriate score for the unit size of the project.

Duplication

Calculating duplication was a bit of a hassle in the start. For instance we didn’t know that hashing is automagically done for you by rascal itself. In the end it greatly increased our readability and performance.

To start we concatenate all our source code from different files to each other to get one large array containing all the lines of code. The reasoning for doing this is the following. In order to filter out double lines that checked for duplicates we map the results to a set. In order to filter the duplicate lines we prepend the lines with the line number they are found on. If we wouldn’t concatenate all the files together, the line numbers might collide with each other potentially flawing the results.

We could have done it differently by adding the name of the file to the prefix. Not only would this still add a possibility for collisions when files share the same name it would also increase load times. Instead of individually loading every file for every method that uses these file we only load the files once and pass that array around to different methods.

Because every six sequentially lines could potentially be a duplicate we create windows. We start at an index of zero and get the first six lines from all the lines. We then shift this window by

one to get the next window. We keep doing this until we have all the possible windows. To calculate the amount of duplicates we reduce this list of windows over a map.

For every window, 6 lines of code, we check with rascal if there exists a value in the map. The following rules apply:

- No value is found at the key of the window in the map, which means that this is the first encounter of these 6 lines of code. We insert an empty array at that place in the map.
- A value is found, this is either an empty list or not. On both cases it adds the content of the window to the location of the key in the map.

In the end all the windows are mapped to a key in the map. However not all values in the map are filled arrays. A lot of keys still hold empty arrays which indicate that they don't have a duplicate. We first filter out the keys from the map that contain an empty array and then unpack these arrays into one big array so we can finally transform this array to a set. Transforming it to a set does all the heavy lifting for us, it filters all the duplicates. We end up with all the duplicates that are found in the source code with none of them being a duplicate as our prepended line numbers should avoid collisions .

One thing we could have done to improve the performance is adding the prefix and creating the windows in one go. Our current way of working is adding the line number prefix and then create the windows. We could have done it by adding the prefix when we create the windows.

Complexity per unit

For the calculation of the cyclomatic complexity per unit, we used a ParseTree parsing approach. The method locations get retrieved from the m3 first, as we want to calculate the complexity per method unit. Then for each method we count the complexity and append the line amount of that method to a map of risk levels. The risk levels from the SIG paper are used to define the risk levels.

The complexity is counted by parsing the method location into a tree. A method can either be of token type ConstrDec or MethodDec, respectively a constructor declaration or a method declaration. The method location is parsed in an AST, and then the visitor pattern integration of rascal (visit(ast)) is applied to traverse over the constructor or method bodies.

During the traversal, nodes like: if, do, while and for, increase the complexity value. The base complexity of a method is 1. The nodes which in our implementation increase the complexity, are nodes which in a graph representation create a choice flow in the execution flow.

When the risk levels are calculated, the levels get converted into percentages. Based on the percentages and the complexity rating of the sig paper, the score gets calculated.

The implementation works perfectly for MethodDecs and ConstrDec, but gives a ParseError on abstractmethod declarations. Therefore we couldn't calculate this metric.

Unit testing

Unfortunately we did not have time to implement this feature. However we did think a lot about it. There are two ways to test actual code coverage. The first one is shallow testing in which we check for testing functions that call other functions directly. Functions that aren't called by test methods are in this case not tested and should show up in the code coverage report. However this doesn't test which path is entered and which one isn't.

The best options are to either modify source code or object code. Both methods involve modifying the resulting code and adding 'breakpoints' to each possible path. With breakpoints we mean a call being made to a global environment flipping a boolean at a specific hash value. Executing the resulting code reveals in the global environment all the booleans that aren't switched and represent the paths that aren't visited.

Besides the fact we didn't have the time we had no idea on how to solve the execution of the modified source / object code part. We knew we had to modify the source code, which isn't a problem with rascal. But actually executing the resulting source / object code was a big question mark for us.

Testing

In order to test our metrics for correctness we added a simplified version of an 'e2e' testing framework. This made regression testing a lot easier and manageable. For every edge case we encountered we either added a method or a new Java file containing that edge case. We then defined a simple set of assertion functies that are called with the result, expected value and test name. We're not saying we got every edge case covered, we're saying that every edge case we encountered is added to one of the fixture files.

If the test failed the assertion function threw an error which halts the execution of the test suite. Before halting it would print the received and expected value so you could see what went wrong.

The testing library is really simple. We could have done a couple of things differently. For instance in one of the helpers we return the first found file or method that matches the given regex. In some cases this goes wrong as the library doesn't check for an exact match. Namespace collisions are therefore to be avoided because there exists a small chance the wrong fixture file will be loading resulting in a failing test.

We don’t have any functionality for checking function body lines for specific lines, we only check for specific length. This could potentially mean that extra whitespace is trimmed in the line.