SoftwareEvolution18

Martin Schvarcbacher and Jelle Manders

Changes since grading session:

1. Modified the way we compute and rank unit size to incorporate relative percentages, not just the average. This enables us to penalize for codebases with unmaintaiably long methods.

2. Changed the duplicate detection algorithm. With a window size of 6, the same file copy/pasted twice results in 100% not 50% duplication. Re-ran the tool.

3. Added extra metric outside the required ones: test quality (see below) and the ISO stability metric. We incorporated test quality into the other derived ratings.

4. Added pretty printing of SIG ratings to console (`PLUS_PLUS()` is now printed as `++`)

Volume

The volume metric is by far the simplest of the analyzed four; We simply go over all of the Java Files, strip them of all comment and blank lines, and count whatever is left. This is done by asking RASCAL for all of the JAVA files in the project, parsing each of them with our Comment and Whiteline stripper, and counting the amount of lines left.

Project	volume	Rating
SmallSQL	24050	`++`
HSQL	172119	`+`

Unit Size

Units are "the smallest piece of code that can be executed and tested individually" (Heitlager). For JAVA, these units are functions. This makes our life very easy, since RASCAL has a builtin method (`MR.methods()`) that returns the strings encompassing all of the functions in a project. We simply call this function over the given project, parse all of the chunks of code given to us by this function, strip the chunk of all empty and comment lines, including docblocks at the top of the function, and count whatever is left. We classify each function into 4 distinct bins according to method SLOC.

For the resulting SIG rating, we used the model proposed in [1] to classify method length. The maximum maintainable size in [1] is stated to be 100 to 200 SLOC. We then count the number of methods falling into the four categories: `20<SLOC` , `20<=SLOC<50` , `50<=SLOC<100` , `SLOC>=100` . These categories are then converted to percentages and compared to a table similar to the one of complexity as described in (Heitlager). From this we derive the final SIG rating for unit size.

[1] Code Complete: A Practical Handbook of Software Construction, Second Edition 2nd Edition

Results

Project	Low Risk	Medium Risk	High Risk	Very High Risk	Rating
SmallSQL	89.7%	8.1%	1.7%	0.46%	`-`
HSQLDB	83.0%	12.2%	3.4%	1.4%	`-`

Duplicate Lines of Code (LOC)

Version 2, changed since grading (NEW):

We no longer consider a single block of lines to be unique, if there are two files with the same content in a project, the duplication ration will be 100%. We now go through all of the files and generate 6 line long chunks and store them in a set with their file location and line number range. On the second pass we create a boolean index for each file and line. If a given chunk is contained in the set more than once, we iterate over all of the files containing this chunk and set the boolean index for the file to `true` to mark a duplicate. Finally we go through the file boolean index to count the number of duplicated lines and lines in total and return this as the result.

Version 1 from first grading (OBSOLETE):

Duplicate lines of code are defined as chunks of at least six lines, if we encounter six lines of code and encounter these exact same lines (excluding comments and whitelines) later, the second (and third, fourth, and so on) will be marked as duplicate. Note that the original will not be marked as a duplicate, hence the ratio of duplicate code will never quite reach 1.
Since the Duplicate chunks can also be bigger than 6 lines, we devised a sort of sliding window approach. The algorrithm works as follows:
All of the file locations in the current project are loaded into memory.
For each file, we do a first pass over the lines and remove all whitespace and comment lines.
Then, for each line, we take the next five lines after it and count this as a chunk. If it is an unknown chunk, we store it in a set.
If the chunk is already in the set, this means we have encountered this chunk before. We mark this section of six lines as duplicate.
If the last chunk that we checked already was a duplicate chunk, this means we haven't found a completely new duplicate chunk. We just found a duplicate chunk that is more than 6 lines long. Instead of marking all of the code as duplicate, we only add 1 LOC to the duplicate counter.
After parsing all of the files like this we have counted how many LOC we encountered, and how many of these are encountered more than once.
This gives us all the necessary information to calculate the Duplication Rate.

Results (version 2)

Project	Duplicates	Total LOC	Percentage	Rating
SmallSQL	2560	24015	10.66%	`-`
HSQL	34289	171470	19.99%	`-`

Cyclomatic Complexity

We use M3 to convert the Java source code to an AST. For each encountered unit (method for Java), we traverse the AST declaration.
For each branch in a method, we count the number of branching statements (`if`, `else if`, `else`, `case` inside `switch`, `ternary operator`) and inside each if statement we add a new branch for each `&&` and `||` operator.

The resulting cyclomatic complexity is aggregated into 5 separate "bins" according to the SIG cyclomatic complexity rating methodology.
From this we derive the final SIG rating.

Results:

Project	Low Risk	Medium Risk	High Risk	Very High Risk	Rating
SmallSQL	74%	7%	12%	6%	`--`
HSQLDB	65%	13%	11.6%	10%	`--`

Unit Testing

We look at each **test** method under the `*tests` directory in the project source and investigate the test quality. The test quaility metric as proposed by SIG is to either look at the unit test coverage percentage or to look at the number of assert statements. Due to the fact that neither SmallSQL or HSQLDB use Maven for their tests, it would be difficult to use standardized Maven plugins (e.g.: JaCoCo) to get the resulting code coverage. We picked the second language independent metric using the number of assert statements per test method. We count the number of `assert*`, `expect\.*` and `@Test(expected = .*\.class)` code statements in the method's source code. From this we classify test methods as either having zero assert (or related) statements or having at least one. For the ones having at least one assert statement, we count their number of occurences per the test to calculate the density.

For `--` rating there are more than 10% no assert test cases, for `-` rating this must be between 5% and 10%. For ratings above `0` we classify according to assertion density (average of assert statements per single test method).

Results

Project	No assert methods	Assert methods	Assert density	Rating
SmallSQL	47	162	5.5	`--`
HSQLDB	120	204	5.3	`--`

Aggregated Metrics

For each atomic metric described above we can derive aggregated metrics using at least two or more atomic metrics.
We assign a numeric value to each SIG rating (5 = `++`, 1 = `--`) and use arithmentic average of the individual ratings to derive the aggregated metric.
The numeric rating is then converted in reverse to the SIG rating.

The SIG ratings can be converted to ISO 9126 maintainability ratings by taking the average of multiple SIG metrics.

Results

Key: Metric: SIG rating without unit testing / SIG rating with unit testing

SmallSQL

Analysability: `+` / `-`

Changeability: `-` / `--`

Testability: `0` / `--`

Stability: `NA` / `--`

Maintainability: `0` / `--`

HSQL

Analysability: `0` / `-`

Changeability: `--` / `--`

Testability: `0` / `--`

Stability: `NA` / `--`

Maintainability: `-` / `--`