Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Pooja Rani

Solution

Assignment 07 — 28.10.2020 – v1.0
Software Metrics and Problem Detection

Please submit this exercise by mail to pascal.gadient@inf.unibe.ch before 04 November 2020, 10:15am.

In this assignment, we will work with the GT image available from here. If you already have it installed because of previous exercises you can use that one as well. However, please clean the image to avoid spurious results when counting classes, *etc.*

First, we have to download and extract two datasets, and second, we need to import them into GT. We can perform both tasks using GT's Playground. Be warned: this process will take several minutes depending on your device's CPU and internet connection. We strongly advise you to save the image when the process succeeded to avoid redoing these steps.

The datasets can be downloaded and extracted with the following script:

```
targetFolder := (FileLocator imageDirectory asFileReference / 'models')
ensureCreateDirectory.
archiveFileName := 'ArgoUML-0-34.zip'.
archiveUrl := 'https://dl.feenk.com/moose-tutorial/argouml/'.
ZnClient new
  url: archiveUrl, archiveFileName;
  signalProgress: true;
  downloadTo: targetFolder.
(ZipArchive new
  readFrom: targetFolder / archiveFileName)
  extractAllTo: targetFolder.

targetFolder := (FileLocator imageDirectory asFileReference / 'models')
ensureCreateDirectory.
archiveFileName := 'lucene-solr-52f2a77.zip'.
archiveUrl := 'https://dl.feenk.com/moose-tutorial/lucene-solr/'.
ZnClient new
  url: archiveUrl, archiveFileName;
  signalProgress: true;
  downloadTo: targetFolder.
(ZipArchive new
  readFrom: targetFolder / archiveFileName)
  extractAllTo: targetFolder.
```

The sample dataset can be imported with the following script:

```
modelFile := (FileLocator imageDirectory asFileReference / 'models')
  / 'ArgoUML-0-34'
  / 'ArgoUML-0-34.mse'.
modelArgo := MooseModel new
  importMSEFromFile:  modelFile.

modelFile := (FileLocator imageDirectory asFileReference / 'models')
  / 'lucene-solr-52f2a77'
  / 'lucene-solr-52f2a77.mse'.
modelSolr := MooseModel new
  importMSEFromFile:  modelFile.
```

Exercise 1: Code metrics in theory (4 pts)

a) What is the cyclomatic complexity? Explain the term and use the words *benefit* and *drawback* in your answer.

Answer:

The cyclomatic complexity is a software metric based on the software's control-flow graph (CFG) that represents a quantitative measure about the complexity of source code. It is quantitative (and not qualitative), because it does not consider any semantics, nor it performs any reasoning about the content. The formula is defined as
$M = E - N + 2P$, *whereas*
M *represents the resulting metric,* i.e., *the complexity value*
E *represents the number of edges of the CFG (possible different instruction flows)*
N *represents the number of nodes in the CFG (instructions), and*
P *represents the number of connected components, (P = 1 for analysis of a single program)*

That said, benefits of this metric are (i) M represents an upper bound for the number of test cases that are necessary to achieve a complete branch coverage of, and (ii) M is a lower bound for the number of paths through the CFG. Assuming each test case takes one path, the number of cases needed to achieve path coverage is equal to the number of paths that can actually be taken. However, some paths may be impossible, so although the number of paths through the CFG is clearly an upper bound on the number of test cases needed for path coverage, this latter number (of possible paths) is sometimes less than M.
In simpler terms, the CYCLO value provides fast and easy to obtain information about the complexity of code, which is relevant for (almost) every static code analysis framework.
As mentioned before, the drawback is that it does not consider any context-specific peculiarities of code. Consequently, the results must be treated with caution.

b) Which other metrics do you know? List at least four and provide a short description for each.
Answer:

- *LOC. The lines of code in a system. More complex than you might think,* e.g., *are comments considered as code, or is code being compactified beforehand?*
- *BUGS. The number of reported bugs per project, class, or method. Specific bug properties must be considered in order to gather desired results,* e.g., *the bug classification.*
- *TIME. The execution time required to run code. Side effects introduced by the evaluation system must be avoided (*e.g., *concurrently running other tasks, varying network load).*
- *SIZE. The size of an executable program. Depends on the target platform and bitness.*

c) Do metrics always express problems? In other words, is, for example, the lack of cohesion always a property to optimize?
Answer:

No, since many metrics are quantitative and not qualitative, they should not be mindlessly interpreted. The values must always be put in context to a specific (code) component. They are rather indicators for interesting / uncommon behaviors. Hence, a major lack of cohesion can occur for value classes that contain a plethora of different configuration parameters, although this is in general a good practice which should be preferred over spreading configuration values all over the project.

d) How and when are nowadays checks for those metrics integrated into development processes?
Answer:

Static analyses are rather cheap compared to the losses the missed issues would generate. Hence, software publishers usually cannot take the risk of missing major issues due to neglected static analyses of code. As a result, they continuously perform checks during the whole development cycle. For example, they perform checks during development in the IDE with help of plug-ins, during automated builds in the build system, or even after release with systems that analyse user feedback.

Exercise 2: Simple code metrics in practice (1 pt)

a) Write a query to find all classes having more than 100 methods in `modelArgo`. **Answer:**

```
modelArgo allModelClasses select:  [ :each |
  each numberOfMethods > 100 ].
```

Exercise 3: Advanced code metrics in practice (3 pts)

a) Write a query to find all methods in `modelArgo` that have more than 150 lines of code, and a cyclomatic complexity of less than 4. (2 pts) **Answer:**

```
modelArgo allModelMethods select:  [ :each |
  (each numberOfLinesOfCode > 150) and:  [
    each cyclomaticComplexity < 4 ] ]
```

b) Apply your implementation from 3a) to `modelSolr` and compare the results between `modelArgo` and `modelSolr`. How do the results differ? (0.5 pts) **Answer:**

The ArgoUML model contains many factory methods which configure the corresponding object, whereas the Solr model contains many complex test setups. Those test setups are similar to factory methods, because they configure a complex test environment. Both projects contain many methods that fall into the specified category.

c) Is it appropriate to use the same threshold values (150, 4) for any model? Justify! (0.5 pts) **Answer:**

Yes, because the threshold is legitimate for simple and complex projects regardless of the programming language with only few exceptions, e.g., *generated code.*

Exercise 4: Expert code metrics in practice (2 pts)

Add a method to the class `FAMIXType` that computes the ATFD metric for its objects. Because `FAMIXClass` is a subtype thereof, it will also automatically be available for all `FAMIXClass` objects. ATFD counts the attributes from other classes used by a class. Since in Java most data is accessed through accessors, we only care for setters and getters, *i.e.,* methods that begin with "set" or "get". (2 pts)

NB: `queryAllOutgoingInvocations` *and* `parentType` *are two useful methods for this exercise.* **Answer:**

```
atfd
^ ( (self queryAllOutgoingInvocations opposites
  reject:  [ :each | each parentType = self ])
  select:  [:each |
     (each name beginsWith:  'set') or:
     [ each name beginsWith:  'get'] ] ) size.
```