Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Pooja Rani

Solution
Assignment 06 — 24/10/2018 – v1.0a
Software Metrics and Problem Detection

Please submit this exercise by mail to sma@list.inf.unibe.ch before 31 October 2018, 10:15am.

Note: For the following exercises you should use the pre-configured Moose 6.1 environments available in 32 bit flavor for Linux, Windows, and macOS. Please choose the correct version for download in accordance with your current platform.

Exercise 1: Metrics (8 Points)

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:

Large companies cannot take the risk of missing major issues in code due to neglected static analyses of code. That's why they are employed at different levels of project, e.g. *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: Evaluation of metrics (2 Points)

a) Write a query to find all classes that have more than 42 methods. **Answer:**

```
self allModelClasses select:  [ :aClass | aClass methods size > 42 ].
```

b) Write a query to find all methods that have cyclomatic complexity more than 84. **Answer:**

```
self allMethods
  select:  [ :aMethod | aMethod cyclomaticComplexity > 84 ].
```

c) What kinds of methods have a cyclomatic complexity of more than 84?
Answer:

None, but there exist classes with a CYCLO value of more than 40. Those classes cover complex problems, for example, parsing C# files, or performing synchronization tests.

d) Is 84 a large value for the cyclomatic metric?
Answer:

Yes, it is very high and an indicator of bad design. These classes must be evaluated and refactured accordingly, e.g. *splitting functionality into different classes.*

Exercise 3: More evaluation of metrics (6 BONUS Points)

a) Write a query to obtain the list of classes from any package that begins with `org.argouml.core` or `org.apache.solr` that call deprecated methods. The packages can be downloaded here and here. **Answer:**

```
targetClasses := self allModelClasses select: [ :aClass |
  aClass mooseNameWithDots beginsWith: 'org.argouml.core.' ].
targetClasses select: [ :aClass |
  aClass providerTypes anySatisfy: [ :aProviderClass |
    aProviderClass isAnnotatedWith: 'Deprecated' ]].
```

b) Write a query to obtain all attributes that are public and camel case with capital letters, but are not declared final. **Answer:**

```
((self allAttributes select: [ :each | each isPublic ])
  select: [ :each | each name
    allSatisfy: [ :c | c isUppercase or: [c = $_]]])
    reject: [ :each | each modifiers includes: 'final' ].
```

c) *Advanced:* Write a query to obtain the list of methods that make more than one call to methods from deprecated classes. **Answer:**

```
deprecatedMethods := (self allModelClasses select:  [ :aClass |
  aClass isAnnotatedWith:  'Deprecated']) flatCollect:  #methods.

methodsUsingDeprecatedMethods := (deprecatedMethods
  flatCollect:  [ :aMethod | aMethod clientMethods]) asSet.

methodsUsingDeprecatedMethods select:  [ :aMethod |
  | calledDeprecatedMethods |
  calledDeprecatedMethods := aMethod clientMethods
    select:  [ :anotherMethod | anotherMethod parentType
    isAnnotatedWith:  'Deprecated' ].

  calledDeprecatedMethods size > 1 ].
```