CSE593 – Capstone Project
Name
Biweekly Report #5

Intro

The primary goal of this milestone is the application of static software complexity analysis algorithms to the code base of the Mobile Agent and our SIF toolkit (ultra-core) and to elaborate on its overall meaningfulness.

Algorithm

For this exercise, the selected algorithm was McCabe's Cyclomatic complexity, which is the most widely accepted static software metric and has been applied to millions of lines of code in both academia and commercial applications (Edmond VanDoren)

Originally developed by Thomas McCabe in 1976, this algorithm has continued to evolve and it is based upon directly measuring the number of linearly independent paths through a program's source code (Wikipedia) and it is formally described as:

$$M = E - N + P$$

Where,

M = *the McCabe metric*
E = *the number of edges of the graph of the program*
N = *the number of nodes of the graph*
P = *the number of connect components*

Cyclomatic complexity can be used in several areas of software engineering including code development risk analysis, test planning as well as reengineering and its wide usage is primarily due to its ability to generate a discrete and simple numerical representation of the overall complexity of a software system. A generally accepted rule dictates that a Cyclomatic complexity score greater than 10 represents a moderate complexity level while a score greater than 20 represents a complex, high risk program.

Results

For sake of this report, the Cyclomatic analysis was performed using the Metrics plug-in of the Eclipse IDE and the results were (on a per method basis): Mobile 1.127 and ultra-core 1.667, as seen on the graph below.

Findings

At a first glance, it is very clear that based upon McCabe's algorithm neither one of the applications is inherently complex. However, at closer inspection, one can begin to perceive that it is crucial to understand that normalization can be playing a role in obscuring certain problematic classes, especially in the case of ultra-core due to its significant size of 36,194 LOC and 5,495 methods.

Moreover, because the overall complexity estimate is normalized on a per method basis, a code structure, which is unbalanced between simple and complex methods and possesses a method count significantly larger than the variance between simple and complex scores, would unequivocally drawn the overly complex methods and would, on the surface, appear to not need further decomposition.

A meaningful example of the scenario described above is method *addObjectData* of class *SIFResponse*[1](part of ultra-core) that carries – by itself – a Cyclomatic complexity of value 88 and is, consequently, an overly complex method that should be marked for further refactoring and decomposition but does not get brought to light under the overall Cyclomatic score.

Many ways to improve how system engineers utilize McCabe's complexity algorithm has been proposed in the last decades,including assessing scores on a per class or n lines of code basis as the means to circumvent an unbalanced class structure (Narayan). Unfortunately, all of those have found no overwhelming success.

Furthermore, as software systems become increasingly more distributed, and conspicuous, gauging code complexity based solely on the number of logical paths will fail to take into consideration the inherited complexity of a service oriented architecture and will, ultimately, drive computer scientists to research better ways to assess complexity for the next century.

Works Cited

Edmond VanDoren, Kaman Sciences, Colorado Springs. Carnegie Mellon Software Engineering Institute. 12 July 2000. 01 04 2008 <http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html>.

Narayan, Sriram. Using Cyclomatic Complexity effectively. 2 5 2006. 1 4 2008 <http://sriramnarayan.blogspot.com/2006/05/using-cyclomatic-complexity.html>.

Wikipedia, The Free Encyclopedia. Cyclomatic Complexity. 01 03 2008. 01 04 2008 <http://en.wikipedia.org/wiki/Cyclomatic_complexity>.