Abstract

of the

Thesis

Reengineering Application Software: A source code perspective

Software reengineering can be defined as the examination, analysis and alteration of any existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. In broad terms this process typically encompasses a combination of reverse engineering, re-documentation, restructuring, and forward engineering [94]. All software projects and products constitute of source code. Now, it is also around this source code that the cost, complexity or simplicity of the software lies. One’s ability to read, understand and above all draw the futurity of present knowledge from the first two would ultimately lead to easily reengineerable software.

Several books such as Software Engineering by Ian Sommerville [102], Software Engineering by Roger Pressman [103], Software Engineering by Lawrence S. Pfleeger [104], Code complete by Steve McConnell [105], examine source code understanding concepts and techniques and other important concepts relating to source code. An analytical, honest and consistent observation of source code and its business logic (i.e. how it achieves the desired output) will show that source code can comfortably be divided into 3 parts namely:

1. The reserved words,
2. Programming language statements and
3. The user-named-variable names
Now of the three it is the user-named-variable names that are of notable importance if knowledge of source code is to be achieved. Daily as human beings we use variable naming theories that we neither aware of nor understand but by the end of the day our communication either forwards or backwards continues with outstanding progress. However, several variable naming conventions have since been coined like the Hungarian Approach, pascalCase, SREAMINGCAPS, camelToe, Sigils, Twigils etc and in turn they have been supported by Object Oriented Approaches (OOA) namely encapsulation, data abstraction, objects, classes, polymorphism, and inheritance. These conventions and approaches either in singular or collectively are bound by a universal naming theory [7].

It should be noted that the fact that they are built on a particular “Naming Theory” makes them more than fundamentally theoretical. To make a ‘named variable’ meaningful in a particular environment the variables just have to be named “domain specifically”, right from the first instance. An intimate understanding of words and their use will show that using a “Naming Theory” to name the user-variables will make the source code readable, understandable and above all will disseminate the right and useful knowledge to the developer, hence making it easier to reengineer the software, since it is known that even if a man comes up with a new word still he has to use old words to explain it.

Only the source code holds the key to successful application software reengineering with regards to cost, time to market, flexibility and portability (i.e. in the absence of documentation).

This thesis proposes a Framework for extracting these variable names, from available source code, and cleaning them and then storing them in a database. The framework allows domain experts, assisted by the framework, to then conclude whether the names used are leading us towards or away from the actual domain or
even that the words used are giving us enough knowledge and control of the subject under reengineering through a variable name scoring process.

We propose to apply a comprehensive Framework for extracting variable names from the source code, clean them through. We would be satisfied to show merely that the conditions within the framework can be used to help us achieve a mechanism for understanding at least some essential properties of the variable names within the system and how they are evolving over a given period of time.

The question we seek to address is whether it is possible to have a framework for arriving at a critical decision as to how we can use variable names in software reengineering. These modes of determination need play an important part in the explanation of mutual understanding, of communication. Or whether, as in the case of a name, it is sufficient for two people to understand one another that they know they both associate with a predicate of the same reference. This being achieved would mean it would be very possible for any person to read and understand source code that one has not written (for example in team work or when reengineering legacy software) without problems and in the least possible time-frame.