Introduction
Chapter 1
1.0 Introduction

1.1 Background

Software engineering has been around for quite sometime now. It is defined as the schematic approach to the development, operation, maintenance, and retirement of software [19]. The use of the term systematic approach for the development of software implies that methodologies are used for developing software which are repeatable. That is, if the methodologies are applied by different groups of people, similar software will be produced [20]. In essence, the goal of software engineering is to take software development closer to science and engineering and away from ad-hoc approaches for development whose outcomes are not predictable but which have been heavily used in the past and still continue to be used for developing software [22].

Appreciating the above, it thus, follows that for us to be able to understand and solve any problems occurring in Software Engineering over time either due to

1.1.1 Hardware Malfunction;
1.1.2 Software malfunction;
1.1.3 Change in business rules;
1.1.4 Maintenance exercise routines;
1.1.5 Technological change

Bearing in mind that software engineering is anchored on a comprehensive systematic approach. It therefore follows that we have to follow an equally systematic approach; in this case it becomes Software Reengineering which is defined as the examination, analysis and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The process typically encompasses a combination of other processes such as:

1.1.6 Reverse engineering,
1.1.7 Redocumentation,
1.1.8 Restructuring,  
1.1.9 Translation, and  
1.1.10 Forward engineering.

The goal is to understand the existing software (specification, design, implementation) and then to re-implement it to improve the system's functionality, performance or implementation [12].

1.2 Motivation
The software reengineering exercise is initially a rigorous one, powered and driven by a very comprehensive Requirements Analysis [17], it surely encompasses all the potential class, package, variable names, their declarations and documentation et cetera. However, during the course of time several challenges emerge

1.2.1 Technological Innovations arise;
1.2.2 New programming paradigms arise;
1.2.3 Business rules change;
1.2.4 New methods of System documentation arise;
1.2.5 The original application software developers change jobs;
1.2.6 Small errors are solved without documentation overtime.

All these and more challenges constantly over a period of time. Either before or at the time the application software is classified as a legacy system [21], the application system will be having patches and duplications all over, but since we have business rules hidden in the source code it thus becomes mandatory that the system be resuscitated in an effort to maintain business operations. To this, Software Reengineering comes in handy as it seeks to address such issues. Several concepts, processes and methodologies [23] have been developed to address such issues are currently in use. These cover a wider range from Architectural Issues
[28], Structural issues [30], Grammar issues [26], Naming conventions [34], Process issues [38], et cetera.

1.3 Role of Naming in Software Reengineering

Since it has been agreed that the goal of software engineering is to take software development closer to science and engineering and away from ad-hoc approaches for development [42]. We therefore, draw from historical facts show that the meanings of words, in scientific as well as non-scientific language, though, are always flexible, never precisely precise, always somewhat vague, always changing [46]. It can be followed that words can never have meanings built into them to stay there, but meanings fluctuate and develop as our knowledge and experience pass from one time to time and from one person to another.

The basic process in the changes and in the communication of both language and experience can be called metaphor: a shift to enclose novelty within a framework already present. New ideas are understood and believed only if established modes of thought, applied to new facts or in a new way, permit comprehension and demand acceptance. With this in mind new activities therefore ought to occur only when old behavior patterns in new situations call for new responses. New meanings for words can develop quickly, especially in science, but only by metaphorically modifying old meanings. Words have precise meanings only from the limited viewpoint of a fixed narrow and a single user perspective.

It can thus be concluded that to take a word as having a precise meaning kills it as an instrument of scientific progress. The great scientists have recognized this in their practice, if not always in what they have said about their practice. Words don't file things into pigeon holes, to stay there, but are creative. Words help create, among other things, new and better meanings for themselves. If science is viewed statically, words seem to be precise labels.
On the other hand if science is viewed dynamically, as the moving thing it actually is, words are imprecise but constantly improving tools. Modern investigations have not yet taught us anything very definite about the effect of words on thought [50]. Most attention nowadays is paid to Whorf's thesis [51], that the general structure or syntax of a language affects its users' general modes of thought. It seems reasonable to suppose that one aspect of a culture, language, has a relationship to other aspects like thinking about experience. But the degree and the causal direction of the relationship cannot yet be proved [52]. The basic reason for this is that innovators so frequently use old words, or combinations of old words, to express their new ideas.

For one thing, language just doesn't have enough sounds and workable combinations to be able to keep up with the rapid and subtle changes in human conditions, in our responses to conditions, and in our knowledge about them. And even if it did, a man with a new word still has to use old words to explain it to the rest of society. New meanings for old words are constantly being forced into existence by changing knowledge and conditions. Apparently, we see this most easily in scientific language, for the world community of scientists is fairly small and coherent. Scientists publicize new meanings, as well as new words, along with new discoveries and interpretations, and useful novelties tend to be rather quickly accepted [53]. But every word in every language shows, over a period of time, some change in its meaning. We can see even in our present vocabularies how flexibly minds treat words. There are countless cases, for instance, where an original meaning retains little or no influence in a word's present meaning, just words tend to float rather freely. They can never be said to be attached to reality, and their attachments to aspects of experience are quite changeable. Hence they have a double function.
Their primary influence is conservative to facilitate present modes of thinking. But their secondary influence is to make change easier, maintaining necessary symbolic connections with the past while allowing new or modified associations to begin working. For change to be possible, meanings must be to a degree vague and open, rather than entirely precise and closed. Vagueness and metaphorical new meanings for old words form a crucial aspect of scientific development. This fact should be recognised as inevitable, and it should be welcomed for the advantages that go with it. Its disadvantages can be overcome only by a better and more widespread realization of how language works. Words are restrictive only when our attitudes are narrow. When we use words with understanding, they can help us to see more rather than fewer possibilities in things.

With the above in mind it therefore can be deduced that variable naming in software engineering is crucial, and more to it, how variable names evolve from an application over time holds the key to the direction in which the software is growing towards during its evolution. To successfully reengineer a software application it is mandatory to keep an inventory of the variable names and how they evolved over a given period of time, with respect to other environmental aspects. This being done religiously will help in identifying and easing the problem-solution curve.

1.4 Objectives

This Thesis is part of the greater reengineering process models and ultimately seeks to

1.4.1 Support existing application software reengineering tools;
1.4.2 Define a clear and precise name extraction point micro process models;
1.4.3 To map out a comprehensive Framework for the entire naming processes;
1.4.4 Gather all variable names into a database;
1.4.5 Monitor any changes applied to the variable names;
1.4.6 Apply relevant operations on the variable names as per need and scheduling.

1.5 Brief Review of Previous Work
We found it fit to breakdown previous Software Reengineering work into 3 distinct groups with the diagrammatic exclusion of the Universal set (as it, presumably, includes Software Engineering as well). It is the nature of these groups that also helps show how much ground has been covered and where this Thesis comes in to fill which gap.

1.6 Software Reengineering Concepts and Methodologies
Here the existing concepts are bound together and basically 2 comprehensive papers are used as a sample to represent the lesser detailed papers also. These papers are

1.6.1 Software re-engineering: concepts and methodology; Robert Behling et al

1.6.2 Forward and Reverse Engineering Concepts explained; Lawrence Shari Fleeger

1.7 Software Reengineering Literature
Here papers that have direct impact on our Thesis are highlighted and again as a measure of highlighting through sample size application we identified key papers which are:
1.7.1 An Interactive Method For Extracting Grammar From Programs, Pankaj Jalote et al;
1.7.2 Cooperative Software Development in GENESIS: Requirements, Conceptual Model and Architecture; Daniele Ballarini et al;
1.7.3 Issues in Conceptual Modeling; Stewart Robinson.

1.8 Naming Theories Literature

These are papers that also contribute closely to the Thesis, but not from a process based argument but rather with more theoretical one. These papers are critical in addressing the ultimate Thesis argument as they prove how much work has been done.

1.8.1 Scientific Naming, William Kent;
1.8.2 An Assessment of Name Matching Algorithms, A.J Lait et al
1.8.3 Descriptive and Causal Naming Theories; Gareth Evans
1.9 Overview of Our Work

After understanding what has been done, it became clear that a bridge between existing Software Reengineering Framework exists. Though more than enough work has been done but it has been confined to Grammar extraction, Refactoring, finite Automata, Naming Conventions and fundamentally naming Theories. To this weakness we have introduced a new Sub Software Reengineering Process which will dwell more on the original status of the Variable names from the original Requirements Engineering [39] in the Software Engineering Phase. But unlike in
Software Engineering we found it to be more effective to extract the variable names from the source code as opposed to gathering them from the Requirements Engineering deliverables (i.e. assuming they are available).

By extracting variable names from the source code and having them go through a variable name cleaning, standardization process, et cetera we can then have a database of all active variable names in the system, now their growth, be it vertical or horizontal (Positive or negative), can thus be monitored through an automated matching sub process [35] and a score can be generated based on an in-house developed scoring mechanism [37].

In this view, our ultimate objective is to have a variable name ‘harvesting’ system which can then be used to clean and manage them so that they become a source of information about the application software in use. These variable names combined with already existing Grammar Extraction Frameworks can, thus, help us understand Legacy systems better or even help manage maintenance of existing application software as to avoid ending up with legacy systems.

1.10 Organisation of Thesis

The rest of the Thesis is organized as follows:

Chapter 2 Presents in detail critical aspects of Software Engineering keeping a close track on the Evolution of Software as much as the laws governing it are concerned as well.

Chapter 3 Here we present the Software Maintenance process and its impact on the nature of software as it ages.
Chapter 4  Here we highlight the Taxonomy of Software with respect to software evolution. Critical points are highlighted leading to an understanding of the need and potential structure of software reengineering.

Chapter 5  Here we explore software reengineering as an absolute technology. Highlighting its potential properties.

Chapter 6  Here we explore two important concepts within software reengineering and these are

- Reverse Engineering and
- Forward Engineering

We highlight their properties and potential stages involved.

Chapter 7  We address the three basic categories under which application software reengineering in generally confined to. In this Literature survey we minimize the number of reference papers by dealing with them with respect to their categories as opposed to their individual capacity. This helps consolidates understanding of the technology gap that exists between the discipline of software reengineering as opposed to individual views.

Chapter 8  This being the heart of the Thesis, we show how the concept for bridging the software reengineering gap was grown and how it seeks to address the missing gap.

Chapter 9  Here we conclude by highlighting potential Future Work on the same and a summary of our contribution to the missing technology. We humbly highlight the weaknesses of our contribution and final close with bibliography.