Journal of Computer and Communications, 2014, 2, 1-16
Published Online December 2014 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2014.214001

Scientific Research

On the Relationship between Software Complexity and Maintenance Costs

Edward E. Ogheneovo

Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria
Email: edward_ogheneovo@yahoo.com

Received 29 September 2014; revised 20 October 2014; accepted 5 November 2014

Abstract

As software becomes more and more complex due to increased number of module size, procedure size, and branching complexity, software maintenance costs are often on the increase. Consider a software such as Windows 2000 operating systems with over 29 million lines of code (LOC), 480,000 pages if printed, a stack of paper 161 feet high, estimate of 63,000 bugs in the software when it was first released [1] and with over 1000 developers, there is no doubt that such a large and complex software will require large amount of money (in US Dollars), social and environmental factors to maintain it. It has been estimated that over 70% of the total costs of software development process is expended on maintenance after the software has been delivered. This paper studies the relationship between software complexity and maintenance cost, the factors responsible for software complexity and why maintenance costs increase with software complexity. Some data collected on Windows, Debian Linux, and Linux Kernel operating systems were used. The results of our findings show that there is a strong correlation between software complexity and maintenance costs. That is, as lines of code increase, the software becomes more complex and more bugs may be introduced, and hence the cost of maintaining software increases.

Keywords

Software, Software Maintenance, Software Evolution, Maintenance Costs

1. Introduction

There is no doubt that software is becoming increasingly complex due to technological development, organizational demand, need for ease of use, etc. As noted in [2], the demand for complex hardware/software system has increased more rapidly than the ability to design, implement, test, and maintain them. It involves the integration of potential of software that has allowed designers to contemplate more ambitious systems encompassing a

How to cite this paper: Ogheneovo, E.E. (2014) On the Relationship between Software Complexity and Maintenance Costs. *Journal of Computer and Communications*, **2**, 1-16. http://dx.doi.org/10.4236/jcc.2014.214001

broader and multi-disciplinary scope, and it is the growth in utilization of software components that is largely responsible for high overall complexity of many system designs. Humphrey [3] concurs with Lyu when he said and we quote "while technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years. It is also why, unless you do something, the situation won't improve by itself. In fact, current trends suggest that your products will use more software and become more complex than those of today. This means that more of your people will work on software and that their work will be harder to track and more difficult to manage. Unless you make some changes in the way your software work is done, your current problems will likely get much worse." Ever since Lyu made this assertion, the situation has not improved. Rather, the costs of developing and maintaining software are on the increase due to software complexity.

The first operating system that was produced is the batch operating systems in the late 1940s. These operating systems could only support a limited jobs being processed in groups. Ever since then, operating systems have continued to become more and more complex supporting networking, real-time processing, multi-processing, multi-programming, and a lot of other activities. It is a well known fact that after the first release of Windows operating system in the 1990s, Windows operating system has become more and more complex, evolving from Windows 3.0 to Windows 8. The lines of code (LOC) have also increased due to the fact that developers of Windows operating system are reacting to customer demands, environmental factors, etc. As a result, the costs of maintaining the Windows operating system have also increased due to the increase in human effort needed to develop and maintain it as it evolves from one version to the next. This case is not peculiar to only Windows operating system but to all other software that are in use especially those in high demand such as application software since software must evolve from time to time in order to meet customer's needs [4].

Normally, the software systems become more and more complex as the software evolve and updates are made to software systems. Such increasing complexity confronts much more challenges in system robustness and adaptability, which depends on predetermined factors to ensure long-term safety and reduce the cost of maintenance. Maintenance plays an important role in the life cycle of a software product. It is estimated that there are more than one billion lines of code in production world-wide. As much as 80% of it is unstructured, patched and not well documented. It is through the process of maintenance that these problems are alleviated [5]. It is therefore necessary to study the relationship between software complexity and maintenance costs so as to know the factors responsible for software complexity and why complexity increases the costs of maintenance. Therefore, there is need to estimate the costs of maintaining software. Software engineering is the applicability of engineering to software development. Software maintenance is an integral part of software life cycle [6]. The software life cycle includes requirements, design, construction, testing, and maintenance. Software development efforts result in delivery of a software product that satisfies user requirements. However, the software product must evolve in order to be able to meet user's requirements.

As noted in [7], complexity is a measure of understandability, and lack of understandability leads to errors. A system that is more complex may be harder to specify, harder to design, harder to implement, harder to verify, harder to operate, risky to change, and/or harder to predict its behavior. Complexity affects not only human understandability but also "machine understandability". Larger and complex software projects require significant management control. They also introduce challenges as complex software systems are a crucial part of the organization. Also, the maintenance of large software systems requires a large number of employees. The estimated costs of software maintenance are high enough to justify strong efforts on the part of software managers to monitor and control complexity. Therefore, management must find ways to reduce the costs of software maintenance by ensuring that the right people are employed to maintain them to avoid more complication of the software.

2. Background of Software Maintenance

Software maintenance as defined by IEEE Standard for Software Maintenance, IEEE 1219, as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. The standard also addresses maintenance activities prior to delivery of the software product. However, the Software Engineering Body of Knowledge (SWEBOK) definition which is the definition generally accepted by software researches and practitioners is defined as the totality of activities required to provide cost-effective support to a software system. Activities are performed both at the pre-delivery

stage and post-delivery stage. Pre-delivery activities include planning for post-delivery operations, supportability, and logistics determination while post-delivery activities include software modification, training, and operating a help desk.

Software maintenance is an essential component of software development process [8]. Software maintenance is a process of modifying existing operational software by way of correcting errors, migration of the software to new technologies and platforms, and adapting it to deal with new environmental requirements for efficiency and productivity. As software grows in size, it becomes necessary to determine the complexity of such software [9]. The size of software increases as the complexity increases [10]. Software maintenance is the general process of changing a system after it has been delivered to the organization or user requesting the software. Therefore, software maintenance and evolution are important concepts in the software life cycle because organizations are now completely dependent on their software systems and have invested millions of dollars in these systems. Their systems are critical business assets and they must invest in system change to maintain the value of these assets.

Software maintenance and evolution of systems was first proposed by Lehman in 1969 [11]. Lehman notes that systems continue to evolve over time. As a result, they become more complex unless some action such as code refactoring is adopted to reduce the complexity that may arise as a result of maintenance. Software maintenance is a very broad activity that includes error correction, enhancements of capabilities, removal of obsolete functions, and optimization. Because changes are inevitable, certain mechanisms must be developed to evaluate, control, and modify the software. Thus changes to software are done in order to preserve the value of the software over time. The software value can be enhanced by expanding the customer base, meeting additional requirements, thus making the software more easier to use, more efficient, more reliable, and employing new technology to cater for the new features that may be introduced to these technologies. Software evolution plays an important role in software maintenance process. Most development effort and expenditure is allocated to the evolution and update of existing versions of software. Software is ceaselessly changed—maintained, evolved and updated—more often than it is written, and changing software is extremely costly.

It has been estimated that the majority of the software budget in large companies is devoted to maintaining existing systems. According to [12], about 90% of software costs are evolution costs. Although this percentage may not be exactly correct, but the fact remains that the large percentage of software costs is expended on software maintenance. When we talk about software maintenance, it is not just about fixing bugs, it involves much more [13]. Software maintenance takes more effort than all the other phases of software life cycle. Barry et al. [14] notes that about 60 to 70% effort is expended on maintenance phase of software development life cycle. It therefore suffix to say that software maintenance activities span a system's productive life cycle and consume a major part of the total life cycle costs of the system. The software maintenance process can last for years or even decades after development process [15]. Therefore, there is need for effective planning in order to address the scope of software maintenance, the tailoring of the post delivery/deployment, the designation of who will provide maintenance, and an estimation of the life-cycle costs. Software maintenance takes more effort than all the other phases of software life cycle.

2.1. Software Complexity

In order to discuss software complexity, there is the need to first discuss the term complexity. Complexity is a measure of resources that must be expended in developing, implementing and maintaining an algorithm or a system [16]. Basili [17] defines complexity as a measure of resources expended by a system while interacting with a piece of software to perform a given task. The IEEE Standard Computer Dictionary defines complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify [18]. As seen from these definitions, complexity is not necessarily measured on an abstract scale, complexity is relative to the observer, what appears complex to one person might appear simple to the other person. Dvorak is right when complexity is considered in general term but this may not be true for software. Complexity in software is not entirely subjective because it can be measured and since it can be measured, it has some determined values.

Perhaps the best definition of complexity is the one provided by [19]. He defines complexity as "the label we give to the existence of many independent variables in a given system. The more there are variables and the greater their interdependence, the greater that system's complexity. Great complexity places high demands on a

planner's capacities to gather information, integrate findings, and design effective actions. The links between the variables oblige us to attend a great many feature simultaneously, and that, concomitantly, makes it impossible for us to undertake only one action in a complex system. A system of variables is 'interrelated' if an action that affects or is meant to affect one part of the system will also affect other parts of it. Interrelatedness guarantees that an action aimed at one variable will have side effects and long-term repercussions on the other variables". This definition is similar to that provided by Dvorak. Dörner defines complexity using the phrase "the label we give", that is to say complexity is a state of the mind, and it is subjective in nature. However, the definition differs from that of Dvorak in that the planner has to gather information and combined findings before conclusion can be drawn. That is to say, complexity is not entirely subjective. Dörner's book provides an insight about the nature of complexity and the difficulties that people have with it. Therefore, the growth in complexity of a system is the growth in risk. Dörner further posit that complex interactions are those of unfamiliar, unplanned, or unexpected sequences, and either not visible or not immediately comprehensible.

Basili note that if the interacting system is a computer, then complexity is defined in terms of the execution time and amount of storage needed to perform the computation. These are often referred to as time complexity and space complexity. On the other hand, if the interacting system is a program, then complexity is defined in terms of the difficulty of performing the tasks such as coding, debugging, testing, or modifying the software [20]. As noted in Akanmu *et al.*, well-designed software exhibits a minimum of unnecessary complexity. Complexity must be managed. Unmanaged complexity can lead to a number of problems especially in software. These problems include difficulty to use, maintain and modify the system. However, it must be noted that managing complexity is a very arduous and challenging task not only in software but also in other fields such as computer science as a whole, engineering, management and generally in all research problems. Fortunately, various metrics have been developed for measuring various aspects of complexity such as size, control flow, data structure and intermediate structure.

Figure 1 shows an example of a complex system. The figure contains a number of objects such as object 1 (0_1), object 2 (0_2), …, object 8 (0_8) and various classes such as class 1 (c_1), class 2 (c_2), …, class 7 (c_7). Just as the objects are related to one another, so also the classes are related to one another and each object is also related to a particular class. It therefore means that both the objects and the classes are interrelated to form a complex system. Complexity is not only found in engineering, sciences, or management, it is also found in every aspect of life including society. From a smaller society such as the family, we then have a collection of families to form a community and so on and so forth to form larger societies that are interrelated which is then called a complex system. Therefore, the interrelationship of program modules, functions and procedures, methods, classes and objects in a program forms complex software. This is so because one program module or procedure or function or method may call another one to execute an action and so they are interdependent on each other. Thus the

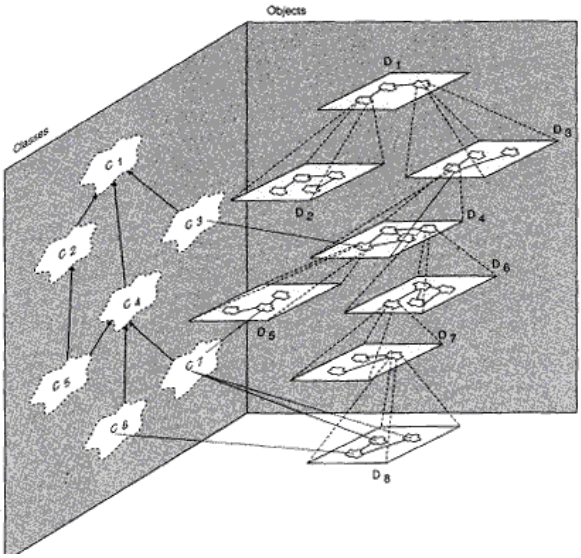

Figure 1. Canonical form of a complex system.

more these modules are in a program, the lager the program will be and the more complex it will be. Also, the more complex software system is, the more it will be difficult to maintain because it will take time for a maintainer to understand the software. Complexity measures have a number of advantages. First, complexity measures can be used to predict critical information about reliability and maintainability of software system from automatic analysis of source code. Complexity measures also provide continuous feedback during software projects to help control the development process. Also, during testing and maintenance phases, they help to provide detail information about software modules that help identify areas of potential instability in the software.

Software complexity refers to the extent to which a system is difficult to comprehend, modify and test, not to the complexity of the task which the system is meant to perform since two systems equivalent in functionality can differ greatly in their software complexity due to the design of the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task. Software complexity is a branch of software metrics that focused on direct measurement of software attributes. Software complexity is widely regarded as an important factor in software maintenance costs [21] [22]. The complexity of software increases as the size increases. That is, as software grow in size, it becomes necessary to determine the complexity. Therefore, increased software complexity means that maintenance and enhancement projects will take longer time to completion, will costs more, and will result in more errors.

As noted in [23], since the 1990s, many of the ailments plaguing software could be traced to one principal cause—complexity engendered by software's abstract nature. According to him, this latter characteristic not only increases the complexity of software artifacts but also severely vitiates the usefulness of traditional engineering techniques oriented toward analog systems. Although computer hardware, most especially integrated circuits, also involves great complexity (due to both scale factors), this tends to be highly patterned complexity that is much more amenable to the use of automated tools. Software, in contrast, is characterized by what [24] called arbitrary complexity. This was further corroborated in [25] who states that, "in some way software products are similar to other engineering products, and in some ways they are very different. The characteristic that perhaps sets software apart from other engineering products the most is that software is malleable. We can modify the product itself—as opposed to its design—rather easily. This makes software quite different from other products such as cars or ovens."

According to Lehman's second law of program evolution [26], as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it. What this implies is that with increased complexity, maintenance will become more difficult and will definitely result in more efforts being expended on the software. The implication is that more maintainer staff will need to be recruited, more time need to be spent in understanding the design and the coding especially if the code is "spaghetti" in nature and it was probably done by a team of inexperienced programmers or it was done using "quick fix" method just to meet up with user's demand or to meet up the stipulated time. In this case, the software may have a lot of errors or bugs that will even make it more difficult to understand and maintain. Thus the estimated costs of software maintenance are high enough to justify strong efforts on the part of software managers to monitor and control complexity [27].

2.2. Causes of Software Complexity

Several factors can be traced to the causes of software complexity. These include:

- Lack of documentation
- Presence of dead code
- Presence of cloned code
- Presence of bugs/faults

Lack of documentation: Most software projects are lacking in documentation. Where there are, they are often incomplete or inadequate for maintainers to fully understand the software. Most often, the people that understand the software usually leave or retire from the organization without being replaced. Also, sometimes, they die and their knowledge dies with them. Thus software becomes increasingly complex due to maintenance from long usage. The implication is that maintainers spend most of their efforts and time trying to analyze the design of the software which tends to make the system more difficult, very costly and time consuming in maintaining it.

The concept of software documentation is very important in software development and maintenance processes. According to Omnext white paper on "How to save on software maintenance costs" [28], "without documenta-

tion, differences in skill levels between good and bad programmers disappeared". The paper further notes that "with proper documentation, good programmers often do better works than poor programmers. Without documentation, they will both do equally bad work. Thus study recommended that spending money on the best programmers without documentation is a waste of resources". What this simply means is that documentation is an essential activity in software development process as it helps programmers to better understand the software during analysis and design stages of maintenance process. Therefore, having documentation of code, especially dispersed code along a user scenario would help programmer(s) understand software better and faster.

Presence of dead code: The term dead code means unnecessary, inoperative code that can be removed without affecting program's functionality. These include functions and sub-programs that are never called, properties that are never read or written, and variables, constants and enumerators that are never referenced, user-defined types that are never used, API declarations that are redundant, and even entire modules and classes that are redundant. Dead codes account for about 30% of legacy software. These dead codes often increase the size of software by increasing the Executable or (EXE or DLL) file size by hundreds of kilobytes thereby making it to be complex and therefore difficult to maintain. Thus the older and larger the system is, the more dead code there is in the system.

Dead code could cause hidden bugs in the software and these could become active or operational later when the program is being used by the user or organization that will eventually make use of the program. Dead code in a program often cause increased memory utilization, slower execution of programs, more code to read and maintain, increased effort and time in trying to comprehend the code. The implication of all these is that there will be increased in complexity and costs of maintenance. Thus there is need for programmers and software developers to remove dead code from their software as much as possible before releasing it to customers to aid future maintainability. As noted by Engelbertink and Vogt, the costs of maintaining both dead and active code are estimated to be at least $1.0 (US Dollar) per source line per year.

Presence of cloned code: A cloned code is a redundant code. By redundant code we mean the process of replicating (cloning) existing code and then customizing them to handle new demands on an application. A cloned code is a code portion in source files that is identical or similar to another code fragment [29]. Cloned are introduced because of various reasons such as reusing code by "copy-and-paste" and so on. Code cloned make the source file very hard to modify consistently. Cloned code constitutes a large proportion of modern software code bases [30]. Code cloned is a convenient way for developers to reuse existing code. It is one of the factors that make software maintenance become more difficult. Maintaining cloned code requires extra effort, because developers need to consider the consistency among the clone segments when trying to make a revision to one clone segment. Failure to maintain consistency between clone segments may cause bugs [31]. However, recent studies show that code clones are not always harmful to software development and evolution [32] [33].

Presence of bugs/faults: A software fault is an abnormal condition caused by the violation of a required property at a program point [34]-[36]. Faults in a software program are often called bugs. A software bug is failure or flaw in a program that produces undesired or incorrect results. It is an error that prevents the application from functioning as it should. Software faults are caused by hidden design flaws rather than wear-and-tear or physical failure since software does not wear out. Therefore, software faults are static: they exist from the day the software was written (or revised) until the day they get fixed. They also tend to be unique for each type of software regarding places of occurrence, mode and severity. Software faults manifest themselves only under particular conditions. For example, when the fault is in the "if clause" of a statement, the error can be manifested only when the conditions stated by the clause are true. However, a single software fault can give rise to system errors or failures. This may happen until the bug has been identified and corrected. Between the failures the fault induces, it will probably give no sign of its existence. So, there is a time lag from the failure to the correction of the underlying faults. This is stochastic in nature, and depends on the nature of the fault, the maintainability characteristics of the program, the abilities of program developer(s) tasked with the repairs, etc.

Software is developed manually and as such human beings are bound to make mistakes. Due to the increasing complexities of software, human understanding of software is not enough to cope with the rapid growth of sizes and complexities. Thus faults are unavoidably introduced to software development. According to a study conducted by the US Department of Commerce, in a typical software development project, 80% of the cost of software development is spent on identifying and correcting software faults [37]. There are many types of software faults such as buffer overflow, integer faults, null-pointer deference, empty catch block in code, empty "if statement", and resource leak. A fault (bug) in a software artifact is an incorrect step, process, or data definition.

Under suitable circumstances, a fault may cause a failure, or "inability of a system or component to perform its required functions within specified performance requirements". That is, a deviation from the stated or implied requirements. A huge time and money is often spent to verify that a system is fault free. However, same faults cannot be found due to either lack of time or simply overlooked. **Figure 2** shows the costs of fixing bugs in a typical software life cycle. As seen in the figure, the cost of fixing bugs in the maintenance stage appears to be much higher than all the other stages put together. Therefore there is need to properly develop software by having good design and ensuring that the software is well tested to remove bugs as much as possible before the software is released to the market or customers so that the costs of maintaining the software is reduced.

In a large system that is used by thousands or even millions of users on a daily basis, some faults may occur that the designers never anticipated. As said earlier, bugs in software are costly and difficult to find and fix. However, these days, several tools and techniques have been developed to help automatically find bugs by analyzing source code or intermediate code statically (at run time). These tools include: Bandera, ESC/Java2, FindBugs, JLint, and Pattern Matching Detector (PMD). These tools look for bug patterns which are code idioms that are often errors or potential problems.

Booch [39] also highlighted some other reasons for software complexity. These are grouped as:

- Complexity of problem domain
- The difficult of managing the development process
- The flexibility possible through software
- The problems of characterizing the behavior of discrete systems

Other causes include:

- User's changing requirements
- Bad design
- Environmental changes

Complexity of problem domain: The problem domain which the software will cover must be well defined during requirement phase of the software development process. This is very important in order to ensure that the size of the software is not too large as to create too much complexity especially during the maintenance phase. Once the domain of the problem is defined, the development team should work on that specification and should restrict attempt to introduce dead code into the software.

The difficulty of managing the development process: Often times, it is always difficult to manage the development team most especially when some members of the team feel that they are too much and cannot be corrected or giving advice simply because they know how to "write programs". Once the development team and the development process are poorly managed by the manager(s), it becomes very difficult to achieve their objective(s) of developing the software such that it could meet specification and the time frame set for the project.

The flexibility possible through software: Software is sometimes considered the most flexible part of any system. The flexibility, however, appears to exist only if the software was created "correctly". When software is not appropriately adaptable in relation to the problem domain, the costs of modification increases over the software's lifetime until such a time when it eventually becomes too costly to maintain. Software should be easily amenable. Software should be designed in such a way that it should be flexible to changes. Such changes include being able to be easily maintained without much difficulties arising from complexity. They should follow

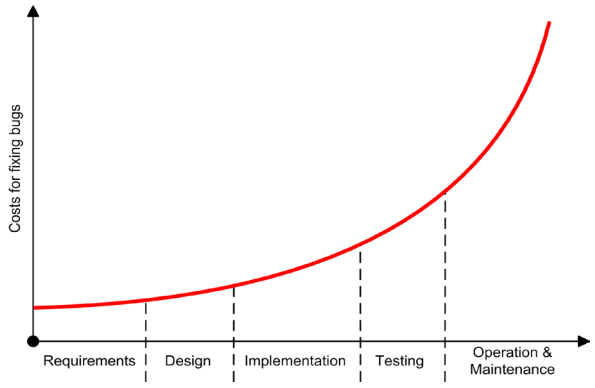

Figure 2. Costs of fixing bugs [38].

some pattern during development. That is, the software should be modular and structured. A situation where software is unstructured or anti-pattern and looks like "spaghetti" will definitely make the software complex and difficult to maintain.

The problems of characterizing the behavior of the discrete systems: The systems must be distinct in terms of the ability of the designers to effectively characterize their behaviors and functions so that these behaviors and functions are well spelt out. This way, the systems will become less complex and can easily be adaptable for the problems they are designed to solve.

User's changing requirements: Software users often demand for a host of other requirements when software has just been delivered to them. They often ask the developer to provider other functionalities that eventually make the software to become complex at the end. Such requirements include but not limited to the following: provide more features, fix some visible quality issues, increase performance, tighten up on security, integrate with a plethora of applications, connect to web services, internationalize, and run on other platforms. With user's increasing demands for new features to be added to the software, the software undergo a lot of modifications and patches and the resultant effect is the introduction of bugs such that the software becomes very complex and difficult to maintain.

Bad design: Bad design can also lead to software complexity. Software that is not properly designed can cause a lot of wastages during development and maintenance stages. Software must be designed in such a way that during the development of the code, the code will be meaningful and will solve the problem it was designed to solve. During the design phase, there should be a breakdown of the design model into smaller modules which is referred to as detailed design. Once there is comprehensive detailed design and there are pseudo codes or unified modeling language (UML) describing what each detailed design process does; then the software can easily be developed and will not be too complex for human comprehension.

Environmental changes: Most often, a particular environment might require that software should be modified. Software developed for a particular environment or organization when taken to another environment or organization might require that some modifications are made to the software before the software can be used or adapted to that environment. In trying to adapt the software to the new environment, maintenance must be carried out. However, this could lead to further complexity of the software, thereby making it difficult for future maintenance.

2.3. Factors Affecting Software Maintenance Costs

Banker identified the factors affecting maintenance costs. According to him, software maintenance costs are significantly affected by software complexity, measured in three dimensions: module size, procedure size, and branching size.

- Program size
- Modularity
- The use of branching

Program size: It is a general belief that the more the size of a program increases, the more the program becomes more complex. This is true of all large software. Large systems require more maintenance effort than do smaller systems. This is because there is a greater learning curve associated with larger systems, and larger systems are more complex in terms of the variety of functions they perform. Van Vliet [40] notes that less maintenance is needed when less code is written. According to him, the length of the source code is the main determinant of total cost during maintenance as well as initial development. As an example, a 20% change in module of 300 lines of code is more expensive than a 30% change in a module of 100 lines of code. Also, older systems are more difficult to maintain than newer systems because software systems tend to grow larger and more complex with age. This is because with usage and frequent changes to the older software, they become less organized and less understandable with staff turnover.

Modularity: Modularity is the degree to which system's components may be separated or recombined. In modular programming, modularity refers to the compartmentalization and inter-relation of the parts of a software system. It is the logical partitioning of software design that allows complex software to be manageable for the purpose of implementation and maintenance. This logic may be based on related functions, implementation considerations, data links etc. Baldwin and Clark [41] in their theory note that modular architectures add value to system designs by creating options to improve the system by substituting or experimenting on individual modules. The concept of modularity is very important in software development and maintenance. According to

[42], it took Microsoft more than three years of delay before they finally released Windows Vista. This was attributed to the presence of large modules which made the software to increase program complexity. Therefore, for large modules to be meaningful and useful in a program, they must be broken down into smaller modules called sub-modules. Guth [43] opined that the delay was attributed to complexity resulting from the system's lack of modularity. According to him, with each patch and enhancements, it became harder to strap new features into the software, since new code could affect everything else in unpredictable ways. This is why Walter [44] concluded that Microsoft Windows Vista is a 60-m-lines-of-code mess of spaghetti. This is because the code lacks pattern, is unstructured, and is not well modularized. Thus Microsoft suffered from unanticipated dependencies, modularity decay, and delays in bringing software to market. As seen from above, it is clear that there is no firm or organization that has been able to solve the problem of software complexity. According to Fried, even a firm with deep expertise in software development, like Microsoft, can still suffer from a complexity disaster resulting from a system's lack of modularity.

The use of branching: Branching plays a major role in the development process of large software. A branch is a virtual workspace created from a particular state of the source code that a developer or team of developers can make changes to without affecting others working outside the branch [45]. Branches provide isolation so that multiple pieces of the software system can be modified in parallel without affecting each other during times of instability. The use of branching in software development can have some negative effects on the code by making the program more complex. For one, branching can lead to errors and post-release failures. The need to move code across branches introduces additional overhead. Also, the use of branching can lead to integration failures due to conflicts or unseen dependencies. Furthermore, a build break on a branch often affects the team working on that branch and not on the entire development team. Branches are meant to provide a level of isolation for development teams to work on parts of the code base without having to worry about affecting others.

The use of branches within a project has a profound effect on the processes used during development, from the build processes to release management [46]. As noted in [47], branching needs to be leveraged correctly in order to be most effective especially where the development teams are not in the same place. Teams may choose to work in branches to avoid dealing with the work of other teams, but some coordination is needed. Branches may introduce a false sense of safety, as changes made in different branches will eventually be merged together. This may introduce bugs into the software if these changes are syntactically or semantically incompatible. The process of moving code between branches represents additional error-prone work for developers. A complex branching structure may hinder the development process, making it hard to track code changes, causing build failures (due to unexpected dependencies), increasing the chances of introducing regression failures and making it difficult to maintain the code base [48].

2.4. Software Maintenance Cost Estimation

When software maintenance is mentioned, what readily comes to mind among non-software practitioners is removing or fixing software bugs. Software cost estimation is basically the estimation of effort, time of development and this effort depends on the size of software and level of complexity. As the size of software increases, so is the complexity and as such the level of effort required to maintain the software increases [49] [50]. Mall further notes that in addition to the size and complexity, many other factors contribute to software estimation effort such as development environment, expertise of development team, and availability of tools. Thus the complexity of software increases as the size increases. Software maintenance accounts for a large majority of maintenance costs. During maintenance, accumulation of poorly managed changes almost always generates software instability and significant increase in the cost of software maintenance.

Software cost estimation is an important but difficult task since the beginning of the computer era in the 1940s. As software applications have grown in size and importance, there is therefore the need for accuracy in software costs estimation. In the early days of software, computer programs were typically less than 1000 machine instructions in size (or less than 30 function points), required only one programmer to write, and seldom took more than a month or so to complete. The entire development costs were often less than $5000 US dollars. Although costs estimate was still difficult at the time but the economic consequences of cost-estimation errors were not very serious. These days, however, software have grown so large to the extent that some large software systems exceed 50 million source lines of code (SLOC), or more than 500,000 function points, requiring well over 1500 technical staff; and may take more than three calendar years to completion. Therefore, errors in costs estimation can be a very serious issue. It is even more serious today because a significant percentage of large

software systems run late, exceed their budgets, or are canceled outright due to severe underestimation during the requirements phase. In fact, excessive optimism in software costs estimation is a major source of overruns, failures, and litigations.

Creating accurate software maintenance and enhancement estimation cost are more complex than new project cost estimation. Estimating maintenance costs requires knowledge of the probable number of users of the application, in addition to the knowledge of the probable number of bugs or defects in the product at the time of release. Estimating enhancement costs requires good historical data on the rate of change of similar projects once they enter production and start being used. For example, new software projects can add 10% or more in total volume of new features with each release for a number of releases, but slows down for a period of two to three years before another major release. However, there is no actual limit on the number of years that can be estimated, but because long-range projections of user numbers and possible new features are highly questionable, the useful life of maintenance and enhancement estimates run from three to five years. Estimating maintenance costs ten years into the future is possible, but no estimate of this range is unreliable because too many uncontrollable business variables can crop up especially in application software.

3. Case Study of Some Operating Systems Software

In this section, we provided estimated costs for development and maintenance. Although our estimation may not be exact, however, we used a generally accepted costing of $100.00 per hour for a typical project member. In this case, we assumed that the development costs for a large project such as Microsoft Windows operating system, Debian, or Linux is $100.00 per hour for a project member. Also, using the general rule that people/salary costs about 50% of a typical team of developers working on a large project, we estimated certain percentage of the costs for each version of the software we considered in this research. Finally, we estimated certain percentage of the costs for the various business-professionals for their direct project effort and costs. Based on all these, we came out with the data in **Table 1**, **Table 2**, and **Table 3**.

Again, we estimated the maintenance costs based on our findings on the development costs. It has been argued by various authors that over 80% of software costs go into maintenance and evolution. We therefore hinged our maintenance costs estimation on these facts. Based on these facts, we computed our maintenance costs estimation for Microsoft Windows, Debian, and Linux as shown in **Table 1**, **Table 2**, **Table 3**.

Wheeler [51] studied the distribution of the Linux operating system. According to Wheeler, Red Hat Linux version 7.1 released in 2001 contained over 30 million physical SLOC. In his report, about 8000 man-years of development effort would have been required and would have cost over $1 billion US dollars if developed in a conventional proprietary way. **Table 1**, **Table 2**, and **Table 3** show the information we were able to gather based on our study research. As seen in the table some of the information especially the development estimated costs and maintenance costs were based on our own estimations. However, we used them to drive home the point we aretrying to stress in this paper that as software complexity increases, maintenance costs also increases.

Table 1. Versions of Microsoft Windows operating system.

Year Released	Product Version	Dev. Team Size	SLOC (Million)	Dev. Estimated Costs (Billion US Dollars)	Maint Estimated Costs (Billion US Dollars)
July 1993	Windows NT 3.0	200	4 - 5	10	28
Aug 1995	Windows ‘95	450	7 - 8	18	47
June 1998	Windows ‘98	800	11 - 12	23	58
Feb 2002	Windows 2000 Professional	1400	29^{+}	32	72
Oct 2001	Windows XP	1800	45	40	86
April 2003	Windows Server 2003	2000	50	48	102
Nov 2006	Windows Vista	2500	56	60	128
Oct 2009	Windows 7	2700	67	75	161
Oct 2012	Windows 8	3000	80	83	192

Table 2. Versions of Debian Linux operating system.

Year Released	Product Version	Dev. Team Size	SLOC (Million)	Dev. Estimated Costs (Billion US Dollars)	Maint Estimated Costs (Billion US Dollars)
Mar 1984	Debian 1.0	200	4 - 5	0.2	0.5
Apr 1994	Debian 1.1	450	7 - 8	0.3	0.7
Mar 1995	Debian 1.2	800	11 - 12	0.45	1.1
June 1996	Debian 1.3	1,400	29+	0.6	1.5
July 1998	Debian 2.0	1,800	45	0.8	1.9
March 1999	Debian 2.1	2,000	49	1.15	2.3
Jan 1999	Debian 2.2	2,500	55	1.4	4.0
July 2011	Debian 3.0	2,900	70	3.6	6.8
Oct 2011	Debian 3.1	3,100	215	4.3	10.4
April 2007	Debian 4.0	3,300	283	6.2	15.2
Feb 2009	Debian 5.0	3,500	324	8.1	20.1
Feb 2011	Debian 6.0	3,800	372	8.9	26.3
May 2013	Debian 7.0	4,200	419	10.2	28.1

Table 3. Versions of Linux Kernel operating system.

Year	Product Version	SLOC (Million)	Dev. Estimated Costs (Billion US Dollars)	Maint Estimated Costs (Billion US Dollars)
2003	Linux 2.6.0	5.2	0.61	1.5
2006	Linux 2.6.8	6.5	1.14	2.7
Oct 2008	Linux 2.6.25	8.2	1.3	3.1
2009	Linux 2.6.29	11.0	1.4	3.7
Dec 2009	Linux 2.6.32	12.6	1.8	4.0
Aug 2010	Linux 2.6.35	13.5	2.4	5.2
July 2011	Linux 3.6	15.9	3.0	6.8

Our estimation is based on a study conducted by Wheeler that examined software line of code count and the cost estimation (SLOCCount) and the work of [52], which estimated the total development costs of a Linux distribution using the same tools. We adopted and used the tools and methods used by Wheeler which were later used in our findings. We also used the Constructive Cost Model (COCOMO) II cost estimation model to estimate some of the development costs and the maintenance costs. Although we realize that one of our case studies, Microsoft Windows operating system is not an open source software, we did these estimations based on the SLOC and the development teams of each version. A similar study was also conducted on Debian Linux version 2.2 originally released in 2000. The study showed that Debian Linux 2.2 has over 55 million SLOC, and would have required 14,005 man-year and cost $1.9 billion US dollars if developed in a conventional proprietary way. In 2005, further studies showed that Debian Linux had 104 million SLOC.

As seen in **Table 1**, **Table 2**, and **Table 3**, as these operating systems evolve; source line of code (SLOC) increases. In would naturally be expected that the costs of development and maintenance through these versions will be on the rise. As noted by Erlikh, 80% of software costs are evolution costs. If this is anything to go by, then one would quite agree that as the development costs is on the increase so also is the maintenance and evolution costs. This is because new functionalities are added to the codes and thus making them to be more complex and in the process more bugs are introduced into software due to the fact that organizations employ the ser-

vices of non-technical people to handle these software maintenance projects. The table also shows that in each release of Windows version, the size of the development team is on the increase. This is because as more functionalities are needed by users, so there is pressure on the software development team to release new products that will meet user's demands and in the process of trying to meet deadlines, new maintainers will be employed and the more people are involved; the more bugs are introduced into the software. Thus when the software is eventually released to the user, there will be further need to maintain the software since those latent errors will start to manifest themselves with usage of the software. Although SLOC measures are somewhat controversial in the way they are misused, one thing that is clear is that effort is highly correlated with SLOC. That is to say, programs with larger SLOC values take more time to develop and hence more money in terms of dollar value. Again, SLOC is easily countable by automated means, in addition to apparently representing the amount of work required to build a system.

It has been argued by some authors that SLOC is a poor productivity of individuals because developers can develop only a few lines and be more productive in terms of functionality than a developer who creates more lines of code which will eventually results in more effort both during development and maintenance phases. This is quite true. This is because inexperience programmers often result in code duplication which often makes the program lengthier and increase the lines of code which may even introduce new errors into the program. It could also cause the addition of dead codes, which may result in certain variables, parameters, functions, procedures, or methods, etc., not being used or referenced or called during the lifetime of the program. They are only present in the program just to make the program look large. Therefore, having a large and complex program will eventually make the program more costly and difficult to maintain. We also considered the evolution of Linux Kernel (operating system) as show in **Table 3**. This is graphically shown in **Figure 5**.

4. Results and Discussions

Table 1, **Table 2**, and **Table 3**, shows that as newer versions of each of the software are released, the line of codes (LOC) increases, hence their complexity. The implication is that their maintenance costs also increases due to a number of factors such as increase in manpower needed to maintain the large software and the amount of effort required to maintain them. From these tables, it is also found that with each newer version of these software, the software becomes more larger thus resulting in more efforts and time are needed to maintain them. This is because the maintenance team needs to study the software and understand it most especially if they are not the team that developed the software. This will definitely increase costs of maintenance. Less maintenance is needed when less code is written. The length of the source code is the main determinant of total cost during maintenance as well as initial development.

Also, from **Table 1**, **Table 2**, and **Table 3**, and from graphs in **Figure 3**, **Figure 4**, and **Figure 5**, it is very obvious that as newer versions of these software are released, their line of codes increases, hence are their complexity. The result of these studies shows that as the line of code increases, so the software becomes more complex and hence, the maintenance costs also increases. Large-scale software products are much more difficult and

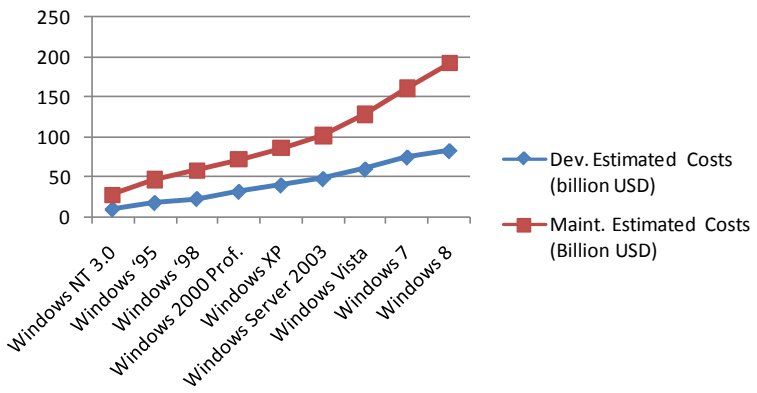

Figure 3. Evolution of Windows operating system and estimated development and maintenance costs.

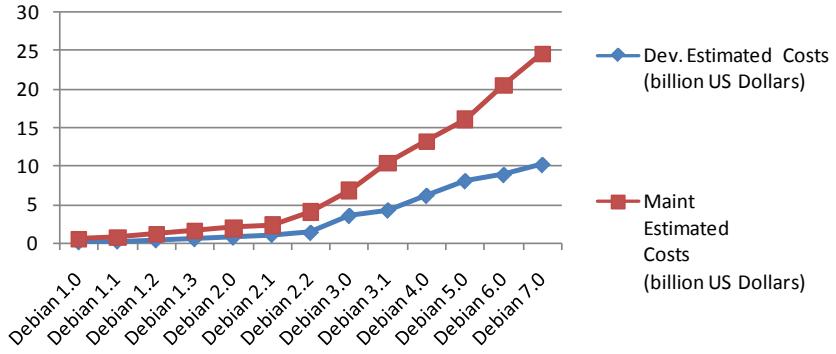

Figure 4. Evolution of Debian Linux operating system and estimated development and maintenance costs.

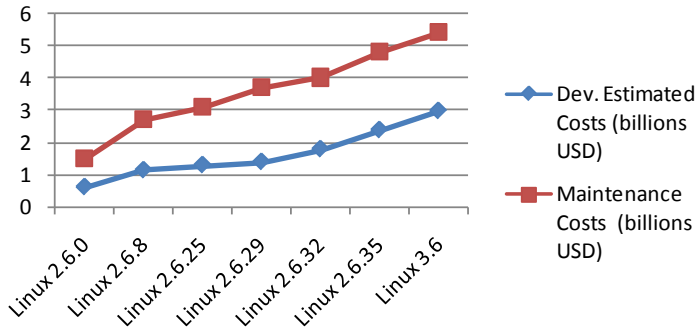

Figure 5. Evolution of Debian Linux operating system and estimated development and maintenance costs.

costlier to maintain than small-scale software products. It is a general belief that the more the size of a program increases, the more the program becomes more complex. This is true of all large software. Large systems require more maintenance effort than do smaller systems. This is because there is a greater learning curve associated with larger systems, and larger systems are more complex in terms of the variety of functions they performA-nother reason is because error might be introduced into the software which could even make it more difficult to maintain.

5. Conclusions

Complexity is a measure of understandability, and lack of understandability leads to errors. A system that is more complex may be harder to specify, harder to design, harder to implement, harder to verify, harder to operate, risky to change, and/or harder to predict its behavior. Complexity affects not only human understandability but also “machine understandability”. For example, static analysis tools for simple languages such as C are stronger than tools for more complex languages such as C++. Large and complex software projects require significant management control. They also introduce challenges as complex software systems are a crucial part of the organization. Also, the maintenance of large software systems requires a large number of employees. The estimated costs of software maintenance are high enough to justify strong efforts on the part of software managers to monitor and control complexity. Therefore, management must find ways to reduce the costs of software maintenance by ensuring that the right people are employed to maintain them to avoid more complication of the software.

In this paper, we studied the relationship between software complexity and maintenance costs so as to know the factors responsible for software complexity and why maintenance costs increase with software complexity. We succinctly provide background of software maintenance, software maintenance estimation and some estimation models used. We used some data collected on Windows, Debian, and Linux operating systems. The results of our findings show that there is a direct relationship between software complexity and maintenance costs. That

is, as lines of code increase, the software becomes more complex and more bugs may be introduced, and hence the cost of maintaining such software increases. We further proffer solutions to reduce software Maintenance costs. The estimated costs of software maintenance are high enough to justify strong efforts on the part of software managers to monitor and control complexity.

References

[1] Slonneger, K. (2004) Software Development: An Introduction. http://homepage.cs.uiowa.edu/~slonnegr/oosd/22Software.pdf

[2] Lyu, M.R. (1996) Handbook on Software Reliability Engineering. McGraw-Hill, USA and IEEE Computer Society Press, Los Alamitos, California, USA.

[3] Humphrey, W.S. (2001) Winning with Software: An Executive Strategy, Addison-Wesley Professional, Part of the SEI Series in Software Engineering, 2001.

[4] Ogheneovo, E.E. (2013) Software Maintenance and Evolution: The Implication for Software Development. *West Africa Journal of Industrial and Academic Research*, **7**, 34-42.

[5] Li, Z., Lu, S., Myagmar, S. and Zhou, Y. (2006) CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *Transaction on Software Engineering*, **32**, 176-192.

[6] Pigosky, T.M. (2001) Software Maintenance. IEEE—Trial Version 1.00—May 2001.

[7] Dvorak, D.L., Ed. (2009) NASA Study on Flight Complexity Final Report, Systems and Software Division, Jet Propulsion Laboratory, California Institute of Technology.

[8] Shukla, R. and Misra, A.K. (2008) Estimating Software Maintenance Effort—A Neural Network Approach. *Proceedings of the Int'l Software Engineering Conference*, 19-20 February 2008, Hyderabad, 107-112.

[9] Bhattacherje, V., Kumar, P. and Kumar, M.S. (2009) Complexity Estimation. *Journal of Theoretical and Applied Metric for Analogy Based Effort-Information Technology*, **6**, 1-8.

[10] Mishra, S., Tripathy, K.C. and Mishra, M.K. (2010) Effort Estimation Based on Complexity and Size of Relational Database System. *Int'l Journal of Computer Science and Application*, **1**, 419-422.

[11] Lehman, M.M. (1996) Laws of Software Evolution Revisited. *Proceedings of European Workshop on Software Process Technology* (*EWSPT*'96), Nancy, 9-11 October 1996, 108-124.

[12] Erlikh, I. (2000) Leveraging Legacy System Dollars for E-Business. *Information Technology Proceedings*, May/June 2000, 17-23.

[13] Ogheneovo, E.E. (2014) Software Dysfunction: Why Do Software Fail? *Journal of Computer and Communications*, **2**, 25-35.

[14] Barry, E.J., Kemerer, C.F. and Slaughter, S.A. (1999) Toward a Detailed Classification Scheme for Software Maintenance Activities. *Proceedings of the* 5*th Americas Conference on Information Systems*, Milwaukee, 13-15 August 1999, 126-128.

[15] Kemerer, C.F. and Slaughter, S.A. (1999) An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, **25**, 493-509. http://dx.doi.org/10.1109/32.799945

[16] Akanmu, T.A., Olabiyisi, T.A., Omidiora, E.O., Oyeleye, C.A., Mabayoje, M.A. and Babatunde, A.O. (2010) Comparative Study of Complexities of Breadth-First Search and Depth-First Search Algorithms Using Software Complexity Measures. *Proceedings of the World Congress on Engineering* (*WCE*'10), London, 30 June-2 July 2010, 203-208.

[17] Basili, V.R. (1980) Qualitative Software Complexity Models: A Summary. In: *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos.

[18] Delaune, S. and Jacquemard, F. (2004) A Theory of Dictionary Attacks and Its Complexity. *Proceedings of the* 17*th IEEE Computer Security Foundations Workshop*, Pacific Grove, 28-30 June 2004, 2-15. http://dx.doi.org/10.1109/CSFW.2004.1310728

[19] Dörne, D. (1996) The Logic of Failure: Recognizing and Avoiding Error in Complex Situations. Rowoholt Verlag GMBH, 1989. English Translation: Basic Books.

[20] Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Gray, M.A. and Adler, W.B. (1986) Software Complexity Measurement. *Communications of the ACM*, **29**, 1044-1050. http://dx.doi.org/10.1145/7538.7540

[21] Boehm, B.W. (1981) Software Engineering Economics. Prentice Hall, Englewood Cliff.

[22] Banker, R.D., Datar, S.M. and Zweig, D. (1989) Software Complexity and Maintainability. *Proceedings of the* 10*th International Conference on Information Systems*, Boston, 247-255.

[23] Shapiro, S. (1989) Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering. *IEEE An-*

nals of the History of Computing, **19**, 20-54. http://dx.doi.org/10.1109/85.560729

[24] Brooks Jr., F.P. (1996) No Silver Bullet: Essence and Accidents of Software Engineering. *Proceedings of the IFIP* 10*th World Computing Conference*, Amsterdam, 1069-1076.

[25] Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2002) Fundamentals of Software Engineering. Prentice Hall, Englewood Cliffs.

[26] Lehman, M.M. (1980) Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, **68**, 1060-1076.

[27] Banker, R.D., Datar, S.M., Kemerer, C.F. and Zweig, D. (1993) Software Complexity and Maintenance Costs. *Communications of the ACM*, **36**, 81-94. http://dx.doi.org/10.1145/163359.163375

[28] Engelbertink, F.P. and Vogt, H.H. (2010) How to Save on Software Maintenance Costs. Omnext White Paper. Accessed 10 July 2013.

[29] Biegel, B. and Diehl, S. (2010) JCCD: A Flexible and Extensible API for Implementing Custom Code Detectors. *Proceedings of the* 25*th IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, 20-24 September 2010, 167-168.

[30] Wang, X., Dang, Y., Zhang, L., Zhang, D., Lan, E. and Mei, H. (2012) Can I Clone This Piece of Code Here? *Proceedings of ACM International Conference on Automated Software Engineering* (*ASE*), Essen, 3-7 September 2012.

[31] Jiang, L., Su, Z. and Chiu, E. (2007) Context-Based Detection of Clone-Related Bugs. *FSE*'07, 3-7 September 2007, Cavtat, 55-64.

[32] Gode, N. and Koschke, R. (2011) Frequency and Risks of Change to Clones. *Proceedings of the International Conference on Software Engineering* (*ICSE*'11), Waikiki, 21-28 May 2011, 311-320.

[33] Kasper, C. and Godfrey, M.W. (2008) Cloning Considered Harmful, Considered Harmful: Pattern of Cloning in Software. *Empirical Software Engineering*, **13**, 645-692.

[34] Xie, Y. and Aiken, A. (2007) Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Transaction on Programming Language System*, **29**, Article No: 16.

[35] Le, W. and Soffa, M.L. (2008) Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. *Proceedings of the* 16*th ACM SCGSOFT, International Symposium of Foundations of Software Engineering*, *FSE*'08, 9-15 November 2008, Atlanta, 272-282.

[36] Le, W. (2010) Toward a Practical, Path-Based Framework for Detecting and Diagnosing Software Faults. Unpublished PhD Thesis, University of Virginia, Charlottesville.

[37] Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E. and Nagappan, M.L. (2013) Predicting Bugs Using Antipatterns. http://sail.cs.queensu.ca/publications/pubs/icsm2013-ehsan.pdf

[38] Grubb, P. and Takang, A.A. (2003) Software Maintenance: Concepts and Practice. 2nd Edition, World Scientific Publishing Company, Singapore.

[39] Booch, G. (2002) object-Oriented Analysis and Design with Applications. 2nd Edition, Pearson Education, Upper Saddle River.

[40] Vliet, H.V. (2000) Software Engineering: Principles and Practice. 2nd Edition, John Wiley & Sons, West Sussex, England.

[41] Baldwin, A. and Clark, J. (2006) Windows Vista Late—But Why? http://crave.cnet.co.uk/desktops/

[42] Fried, L. (2006) Vista Debut Hits a Delay. CNet News.com, 21 March 2006. http://news.com.com/2100_1016_3-6052270.html

[43] Guth, R.A. (2005) Code Red: battling Google, Microsoft Changes How It Builds Software. The Wall Journal, A1. Factiva, MIT Libraries, Cambridge. http://global.factiva.com

[44] Walters, R. (2005) Obstacles Stand in the Way of a Cultural Shift, Financial Times, November 17, 2005: 17. Factiva, MIT Libraries, Cambridge. http://global.factiva.com

[45] Shihab, E., Bird, C. and Zimmermann, T. (2012) The Effects of Branching Strategies on Software Quality. *Proceedings of ESEM*'12, Lund, 17-22 September 2012, 301-310.

[46] Walrad, C. and Strom, D. (2002) The Importance of Branching Models in SCM. *Computer Journal*, **35**, 31-38. http://dx.doi.org/10.1109/MC.2002.1033025

[47] Appleton, B., Berazuk, S., Cabrera, R. and Orenstein, R. (1998) Streamed Links: Branching Patterns for Parallel Software Development, 2002.

[48] Premraj, R., Tang, A., Linssen, N., Geraats, H. and Vliet, H. (2001) To Branch or Not to Branch? *Proceedings of the* 2*nd Workshop on Software Engineering for Sensor Network Applications*, 81-90.

[49] Mall, R. (2009) Fundamentals of Software Engineering. 3rd Edition, PHI Learning Private Ltd., New Delhi, 404-411.

[50] Chakraverti, S., Kumar, S., Agarwal, S.C. and Chakraverti, A.K. (2012) Modified COCOMO Model for Maintenance Cost Estimation of Real Time System Software. *International Journal of Computer Science and Network* (*IJCSN*), **1**, 9-15.

[51] Wheeler, D.A. (2002) More than a Gigabyte: Estimating GNU/Linux's Size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

[52] McPherson, A., Proffitt, B. and Hale-Evans, R. (2008) Estimating the Total Development Cost of a Linux Distribution. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html