IJCSE International Journal of Computer Sciences and Engineering Open Access

Survey Paper Vol.-7, Issue-3, March 2019 E-ISSN: 2347-2693

Software Comprehension Using Open Source Tools: A Study

Jyoti Yadav

Department of Computer Science, Savitribai Phule Pune University, Maharashtra, India

*Corresponding Author: jyo_yadav@yahoo.co.in, Tel.: 9881252910

DOI: https://doi.org/10.26438/ijcse/v7i3.657668 | Available online at: www.ijcseonline.org

Accepted: 12/Mar/2019, Published: 31/Mar/2019

Abstract: Software applications developed in recent times are written in lakhs of lines of code and have become increasingly complex in terms of structure, behaviour and functionality. At the same time, development life cycles of such applications reveal a tendency of becoming increasingly shorter, due to factors such as rapid evolution of supporting and enabling technologies. As a consequence, an increasing portion of software development cost shifts from the creation of new artefacts to the adaptation of existing ones. A key component of this activity is the understanding of the design, operation, and behaviour of existing artefacts of the code. For instance, in the software industry, it is estimated that maintenance costs exceed 80% of the total costs of a software product's lifecycle, and *software understanding* accounts for as much as half of these maintenance costs. *Software Comprehension* is a key subtask of software maintenance and evolution phase, which is driven by the need to change software. This paper will help in enhancing the ability of the developers to read and comprehend large pieces of software in an organized manner, even in the absence of adequate documentation by using existing open source tools. It highlights the program elements, components, its analytical solutions for understanding, comprehensions and extension.

Keywords—beautifiers, profilers, slicers, top-down, version control systems

I. INTRODUCTION

One of the features of working in software evolution is the constant need to modify and update software systems that are unaccustomed, or only partly understood. However, evidence suggests that programmers who have a better understanding of the system as a whole are able to introduce changes into the system with fewer defects than those with a weaker understanding. The number of existing software to be maintained is much greater than the number of software newly developed each year. Their maintenance is thus both costly and extremely important. The ease with which a software can be modified and updated after it has been delivered to the customer depends in part upon how well it was constructed in the first place. Many of the programming techniques like code reusability, object oriented features, modular design etc. are aimed at making software easy to maintain once it is deployed. But, no matter how well we design and develop software nor how carefully requirement specifications are noted, we can never predict exactly what modifications the users will demand in the long run, nor what technological advances will affect the software. Nor can we predict the amount of cost that will be incurred in making the required changes. Software change is therefore an inexorable reality, regardless of how skillful the initial programmer was. To obfuscate matters further, modifying an active software is a very different and a difficult task from developing a new software from scratch. For one thing, modifications must be made without causing any negative impact on the rest of functionality of the software. Hence, making changes to an existing piece of software under these conditions is risky and indeed very challenging. Thus understanding or comprehending the software plays a very crucial role during software maintenance phase.

The software needed to run all of Google's Internet services right from Google Search, Google Maps, Google Docs, Google+, Google Calendar, Gmail, YouTube, and every other Google Internet service, spans around two billion lines of code, all sitting in a single repository. Google is an extreme case. But its example shows how complex software has grown in the Internet age—and how we've changed our coding tools and philosophies to accommodate this added complexity. It is estimated that the maintenance including minor modifications, enhancements, test hypotheses, code navigation etc. of existing system consume 50-80% of the resources in the total software budget. While traditional software engineering methods focuses on improving the productivity of the software development process and the quality of systems under development or being planned, software comprehension addresses the complementary issues. Programmers do not use sophisticated and dedicated software comprehension tools. In fact, they are un-known to the software comprehension functionality and features that exists in the Integrated Development Environments they use regularly.

Software Comprehension (SC) is a domain of computer science concerned with the ways developers/programmers maintain huge source code and identify cognitive processes involved. SC is a sub-branch of software engineering maintenance phase that consumes about half of the time spent by the programmers who have to explore a systems' source code to find and understand the sub-set of the code which is relevant to their current job. The term software comprehension means understanding large sources codes written by someone else. There has been a constant flow of work into techniques, tools and methods for improving developers/programmers ability to understand software. Many researchers are of the opinion that developers follow pragmatic comprehension strategies depending on context and try to avoid comprehension whenever possible. Developers are of the view that standards, experience, and personal communication facilitate comprehension. The group size, its division, and open-source experience influence their knowledge sharing and access behavior giving importance to face-to-face communication.

Rest of the paper is organized as follows, Section I contains the introduction to Software Comprehension, Section II lists the related work, Section III details a broad survey of open source software comprehension tools used for comprehending large code which is the main crux of the paper, Section IV concludes the proposed research work with future directions and Section V lists down the references.

II. LITERATURE SURVEY

With the increasing demand, the amount of software that is being used and developed is growing. The size of software is becoming enormously larger. After the software is been put into use, software maintenance activities such as correction, expansion and improvement begin. Software maintenance is a process to analyze, understand, modify and re-confirm the software. The software maintenance problems are important issues the current software industry faces. Accurate, rapid and comprehensive understanding of program is the key to successful software maintenance. Thus, analyzing and understanding the program is the first step to maintain software. Program comprehension is to make clear "what a program does" and "how a program does" through certain facilities and methods. Software comprehension generally has four tasks, which are identifying program unit, tracing control flow, tracing data flow and integrating program logic. Research of software comprehension is very important for software development, management and maintenance, especially for system upgrades of legacy software.

Developers/programmers spend most of their time constructing huge and complex software applications, an activity termed as programming, which is according to [1] "a kind of writing". Writing code is a human and mental activity. The more familiar we are with a program, the easier it is to understand the impact of any modification we may want to perform, i.e., familiarity has an important influence on software comprehension strategies [2]. Software Comprehension can be defined as "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program [3]." Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in life to understand its construction and intentions and to change it comfortably and confidently. Developers should feel at home, and be able to place their hands on any item without having to think deeply about its location [4]. The comprehensive understanding of a large software system is a tedious job because of the enormous lines of code and complexity that such software systems exhibit. The problems encountered while comprehending the source code directly influences the time spent on software comprehension. The 80-20 rule applies to program creation and maintenance. Wherein 80% of time is spent on maintenance activity and 20% of the time spent creating the software. Of the maintenance time 20% is spent changing while 80% of the time is spent just trying to understand what the code is written for in the first place.

Though majority of the time is spent in reading code, there been very little efforts and research put into tools that help software comprehension. When a novice maintenance programmer is given the task to maintain, debug or enhance a complex and sophisticated code where the documentation is scant the programmer indeed gets disconcerted. Following are the list off few questions that shall be taken into consideration while focusing on software comprehension:

- What really do the maintenance programmer wants?
- What would make comprehending complex software simpler?
- How can we display complex software in an understandable way?
- What part(s) of the software need to be focused?
- Will a change to module 'A' affect the functionality of module 'B'?
- And, most critically, "where to start from?"

Majority of software development effort is spent on maintaining existing systems rather than developing new ones. It is estimated that around 50-75% of time is spent on comprehending or understanding the program code. Estimates of the proportion of resources and time devoted to maintenance range from 50% to 75%. Majority of the time and effort of the software maintenance process, in turn, is devoted to understanding the system being maintained. 47-62% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities which involve reading the documentation, scanning the source code, and understanding the changes to be made [5]. The implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of understanding existing programs using modularizations [6]. Integrated development environments provide a number of facilities to support software development, such as source code browsers, refactoring engines, test runners etc.

Despite common belief, developers do not spend most time coding. Approximately 50 to 90% of development time is spent on code orientation that is navigation and understanding of source code which includes reading of local source code and documentation, searching the internet for code examples, but also seeking help of other developers. This gives rise to the need for development tools that provide support for the code orientation hints that developers require. Furthermore the development tools need to tap unconventional information from the source code in order to help the developers that would be out of their reach without tool support.

The process of learning about the domain from the code which is a prerequisite of code reengineering is discussed in [7]. The paper details the similarities and overlaps between software comprehension and human learning. Authors of [8] present a framework to support researchers in planning and conducting experiments regarding software comprehension. The authors compare two software comprehension tools viz. PARAMAT and the PAP recognizer targeted to support automatic parallelization, discussing the trade-off between the generality of one and speed of the other [9]. A study [10] leverages interaction data that is collected across many applications by the developers and find that the developers spend ~58% of their time on software comprehension activities, taking into consideration the impact of the programming language, developers experience and the project phase. It finally highlights the importance of several research directions needed to reduce software comprehension time. Integrative levels of software comprehension such as the element level, the percept level, the concept level, the subject level and the domain level have been demonstrated using object oriented testing framework jUnit as a case study [11].

The work of [12] aims at identifying tasks that can improve novices' software comprehension. The study is to validate the classification of cognitive tasks as in Bloom taxonomy. An online survey was conducted on a number of programming instructors as well as developers. The respondents were asked to place each of the fourteen identified tasks into one of the six cognitive categories of revised Bloom taxonomy. The findings showed that most of the respondents agreed with the classification. 3D visualization of software system is applied on large systems as it is useful for software comprehension that leads to clarity about the overall structure of the system [13]. Various types of slicing techniques like static, dynamic, quasi, amorphous, forward and backward which comes under the broad category of syntactic and semantic slicing is discussed along with its application in areas like debugging, cohesion measurement, comprehension, maintenance and re-engineering and testing are highlighted in the paper [14].

The slicing results obtained using Frama-C and WP are compared and discussed in [15]. The author of [16] explore whether software comprehension tools enhance the way that programmers understand programs. Experimental observations from three tools viz. Rigi, SHrimMP, and SNiFF+ are compared for browsing program source code and exploring software structure. Some authors have used program summarization techniques to support software comprehension [17]. The maintenance and evolution of large-scale code is considered based on questions that relate to scalability of existing experimental results with small programs, validity and credibility of results based on experimental procedure and challenges of data available in work done by [18]. A systematic literature survey on software comprehension through dynamic analysis is carried out by [19]. The authors also present an attribute framework that can be used to characterize papers in the area of software comprehension through dynamic analysis. Two empirical studies have been performed to report that programmers have two types of programing knowledge: programming plans and rules of programming discourse by [20]. In the schema based approach the role of semantic structures is emphasized whereas in the control flow approach the role of syntactic structures is emphasized by the author of [21]. Studies show that according to the understanding the knowledge used may be related to different kinds of information: data-flow relations, functional relations or control-flow relations. Programming plans formalize information on data flow and functions whereas syntactic constructs reflect the structure of the program as described with control flow relations. Research efforts aiming at addressing challenges in software comprehension can be characterized by both the tools that are used to assist a comprehension task as well as the cognitive theories that provide explanations on how developers understand software. Although research in the field of software comprehension has considerably evolved over the past 20 years and many theories have been proposed to explain how programmers may comprehend software, there are still open issues that require further exploration.

The goals of the research is to investigate the automated tools a developer can use for understanding the software. Therefore the research question being explored in this paper is: How and which open source tools can a programmer/developer use to ease the task of software comprehension?

III. A Broad Survey of Open Source Tools Used For Software Compehension

Drupal web content management open source software code is used as a case study to explain software comprehension in detail since it has a strong architecture, is reliable, secure, flexible, and scalable and can be integrated with third party applications. The paper reports output of various practical experiments carried out to comprehend programs. Scripts used for comprehending a program have been compiled and executed on Linux (Ubuntu 16.4 LTS) operating system. Drupal (Version 5.x, 6.x, 7.x and 8.x) was considered for understanding the working of various tools.

3.1. A Commentary on Software Comprehension

This section briefly discusses the highlights and importance of software comprehension. Software comprehension plays a crucial role during the software maintenance phase. The programmers spent most of their time with comprehending source code, and maintenance is the main cost factor in software development. Thus, if we improve software comprehension, we can save considerable amount of time and cost. To improve, software comprehension, we have to measure it first. However, software comprehension is a complex, internal cognitive process that we cannot observe directly though empirical research is applied in software engineering. To bridge this gap, we support researchers in planning and conducting experiments regarding software comprehension. This paper lists down various strategies and preferred tools that a developer can use to comprehend programs. Program understanding tools should enhance and ease the developer's job of reading and understanding massive source codes.

Programmers use Integrated Development Environments (IDEs) to read, understand, and write source code. IDEs provide a number of facilities to support software development, such as source code browsers, profilers, slicers, test runners etc. While using an IDE, developers generate a large number of events, for example, browsing the source code of a method, editing the body of a method, or inspecting an object at runtime. Following is a list of various models and analysis techniques that support soft-ware comprehension:

3.2. Overview of Software Comprehension Models

Basically, to understand the source code, the developers typically use either top-down or bottom-up comprehension. Based on amount of domain knowledge, there are three different kinds of comprehension models like top-down models, bottom-up models, and integrated models. In top-down models, if the programmers are familiar with a program's domain (e.g., operating systems), they understand programs top-down. First, they state a general hypothesis about a program's purpose. To this end, programmers compare the current program with familiar programs and scheduling strategies of that domain. During that first step, they ignore details and only focus on relevant facets for building the hypothesis. Bottom-up models, start to understand a program by examining details of a program—the statements or control constructs that comprise the program. Statements that semantically belong together are grouped into higher level abstractions, called chunks. If enough chunks are created, programmers leave the statement level and integrate those chunks to further higher level abstractions. For example, if programmers recognize that a group of statements have a high level purpose, they create one chunk and then refer to that chunk for e.g. as "searching an element in a list", not the single statements.

When further examining the program, programmers combine these chunks into larger chunks like implementing scheduling strategies, until they have a high-level understanding of a program. Integrated models combine top-down and bottom-up comprehension. For example, if programmers have domain knowledge about a program, they form a hypothesis about its purpose. During the comprehension process, they encounter several fragments that they cannot explain using their domain knowledge. Hence, they start to examine the program statement by statement, and integrate the newly acquired knowledge in the hypotheses about the source code. Usually, programmers use top-down comprehension where possible and bottom-up comprehension only where necessary, because top-down comprehension is more efficient. One of the examples of integrated models divide software comprehension into four processes, where three of them are comprehension processes that construct an understanding of the code and the fourth provides the knowledge necessary for the current comprehension task.

3.3. Software Comprehension through Open Source Tools

Figure 1 list down various free and open source tools that can be used to ease software comprehension activity.

3.3.1. Document Generator

Code can be hard to understand, analyze and reverse engineer if no help is given. Such reverse engineering is required if the code is to be updated, corrected or parts of it is to be reused. Documentation makes it easier to come back to the code after some time. It makes sharing work with others much easier. The code should be sufficiently commented.

The idea of Doxygen is to use the comments and commenting capabilities in code to create the documentation. For the programmer this means that comments should be left in somewhat more organized manner, but for that really nice documentation can be compiled from the comments. Generated Document (Doxygen) is automatically updated when code changes assuming that the comments in the code changes as well. Doxygen is a program that can look into your source files, extract comments and generate useful documentation. It is especially great when used with object oriented languages such as C, C#, PHP, Java, Python etc. Doxygen can generate an on-line documentation browser in HTML and/or an off-line reference manual in LATEX from a set of documented source files. It also supports generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and UNIX man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. Doxygen can be configured to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically. You can also use doxygen for creating normal documentation.

3.3.2. Code Reading

Online accessible code can be processed with tools to enhance reading efficiency and understanding. Following are the list of tasks that could be performed on code:

- Identify the declaration of a particular entity to determine the type of function, variable, method, template, or interface.
- Locate where a particular entity is defined, for example, find the body of function or class.
- Go through the places where an entity is used.
- List deviations from coding standards.
- Discover code structures that might help you understand a given fragment.
- Find comments explaining a particular feature.
- Check for common errors.
- View the code structure.
- Understand how the code interacts with the environment.

This section discusses code reading tools that can be used to automate the above tasks and perform them in the most efficient manner. In addition, modelling tools can often help in reverse engineering a system's architecture, while a number of documentation tools can automatically create project documentation from especially formatted source code. We examine tools based on increasing levels of affinity between the source code and its execution. We start with tools that operate on source code at a lexical level (that is, they process characters without parsing the program structure), continue with tools based on code parsing and compilation, and finish with an overview of tools that depend on the code execution.

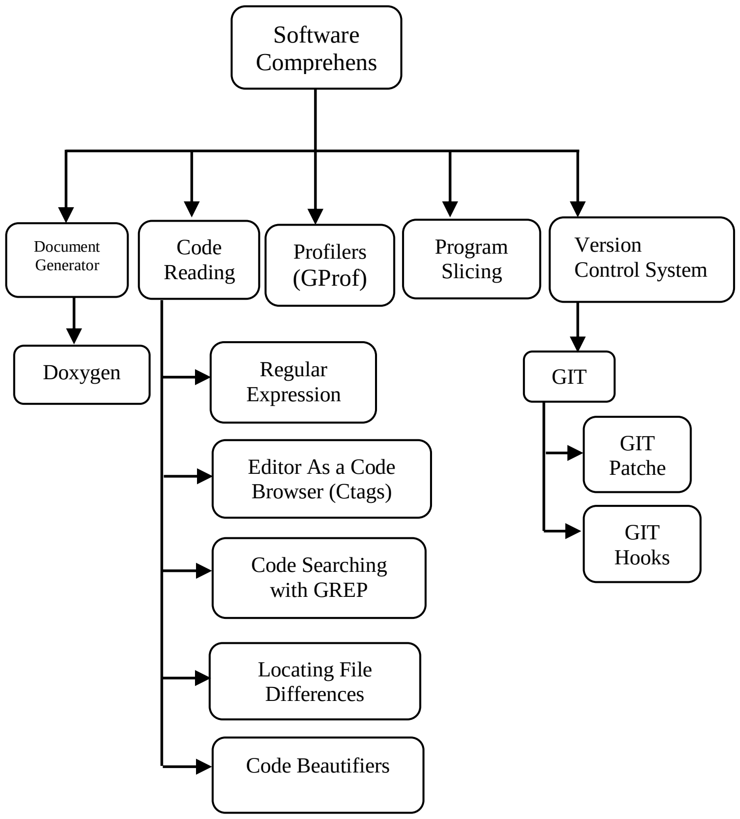


Figure 1. Open Source Tools to Enhance Software Comprehension

3.3.2.1. Regular Expression

The tools that operate on source code at a lexical level are powerful and can be used on any programming language or platform, operate without a need to pre-process or compile files, handle a wide variety of tasks, execute swiftly, and handle arbitrary amounts of program text. Using such tools, you can efficiently search for patterns in a large code file or across many files. These tools are by their nature imprecise; however, their use can save time and yield results that might escape casual manual browsing. The power and flexibility of many lexical tools come from the use of *regular expressions*. You can think of a regular expression as a recipe for matching character strings. A regular expression is composed from a sequence of characters. Most characters match only with themselves, but some characters, called *meta-characters*, have special meaning. You create regular expressions by using a combination of regular characters and meta-characters to specify a recipe for matching the exact code items you may be looking for.

3.3.2.2. The Editor as a Code Browser

Editors like *Emacs* and *vi* use regular expression searches in combination with an index file to efficiently locate various definitions in source code. Firstly create an index file using the *ctags or idutils* indexing tools. The index file, named tag, contains a sorted set of the definitions found in the source files. For C language the definitions recognized include functions, #defines, and, optionally, typedefs, structs, unions, and enums. Each line of the tags file contains the name of the entity recognized, the file it was found in, and a regular expression pattern for locating that entity within the file.

3.3.2.3. Code Searching with Grep

Large projects are typically split into multiple files, sometimes organized in a directory structure. Searching through each file with an editor is not always practical; fortunately, there are tools that can automate this task. The parent of all tools used for searching through large bodies of code is *grep*, which gets its name from the *ed/ex* editor command that prints all lines matching a pattern. *Grep* takes as an argument a regular expression to look for and a list of files to search in. The files are typically specified using a wildcard pattern, for example, *.c, *.h. Many of the characters used in regular expressions also have a special meaning for the command-line shell, so it is better to enclose the regular expression in quotes.

3.3.2.4. Locating File Differences

An easy way to reuse code is to create a copy of the respective source code and modify it as needed. There are many problems with this mode of code reuse; one of them is that two diverging versions of the code base are created. Another common situation where you will find two different versions of a source code file is when you are examining the evolution of a body of code. In both cases you will end up with two slightly different versions of a source code file. One way to compare them is to print both on fanfold paper and lay the listings side by side, looking for differences. A more efficient way is to use a tool. The *diff* program will compare two different files or directories and list lines that must be changed to bring the two files into agreement. The *diff* tool can output the file differences in a number of ways. Some output formats are terse and are optimized for use by other programs such as *ed*, Revision Control System (RCS), or Concurrent Versions System (CVS). One of the formats, the *context diff*, is particularly user-friendly: it displays the differences between the two files in the context of the lines that surround them. The -g option of *diff* specifies this output format; other tools such as CVS and RCS also support this option when comparing different source versions. Lines that are different between the two files are marked with a !, lines that are added are marked with a + , and deleted lines are predictably marked with a -.

3.3.2.5. Code Beautifiers/Formatter

Programmers often use tools to format programming language source code in a particular manner. Proper code formatting makes it easier to read and understand. Different programmers often prefer different styles of formatting, such as the use of code indentation and whitespace or positioning of brackets. A code formatter converts source code from one format style to another. This is relatively straightforward because of the unambiguous syntax of programming languages. Code beautification involves parsing the source code into component structures, such as assignment statements, if blocks, control flows etc. and formatting them in a manner specified by the user in a configuration file. Code beautifiers exist as standalone applications and built into text editors and Integrated Development Environments (IDE). For example, Notepad++ is widely used for writing Python programs as it can correctly indent blocks of code attractively.

3.3.3. Profilers

Program profiling or software profiling is a form of dynamic program analysis that measures the space or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. Profiling is achieved by instrumenting, either the program source code or its binary executable form using a tool called a profiler (or code profiler). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

Application performance is crucial to a software Engineering. When code executes quickly and efficiently, then the software is responsive and reliable. However, when code goes into unnecessary loops, calls extraneous functions, or trips over itself in some other way, then the code is a bad code. The software may be sluggish or unresponsive. If left unchecked, this ultimately may result in serious consequences. Very few lines of code run at peak performance when they're first written. Code must be analysed, debugged, and reviewed to determine the most effective way to make it run faster. How can software developers and quality engineers ensure that their code is quick, efficient and ultimately seen as valuable? The solution to this is in using a profiling tool to answer questions like how many times each method in your code is called and how long does each of those methods takes. It tracks things like memory allocations and garbage collection. Some profilers also track key methods in the code so you can understand how often SQL statements are called on web service calls. Some profilers can track web requests and the train those transactions to understand the performance of dependencies and transactions within your code. Profilers can track all the way down to each individual line of code or the CPU instructions, which is very slow. These tools can quickly diagnose how the software performs and enable programmers to zero in on areas of poor performance. The result is a streamlined code base that performs well. Optimizing code is a challenging task. It requires time, thought, and investigation from developers. Without the proper tools, programmers have to fall back on slower, less efficient ways of trying to optimize their applications. Some developers take to "pre-optimizing" code; they guess where performance problems will occur and refactor their code in an attempt to eliminate problems before they appear. This approach is problematic because a developer will often incorrectly diagnose the potential bottlenecks. The programmer may look only at his own code, instead of the full code base, thus missing integration issues. He may not have insight into his target users' expected behaviour, or he may focus on an area of code that's infrequently used.

There are three different types of profilers:

- *High level:* High level profilers track performance of key methods and typically do transaction timing, such as tracking how long a web request takes, while also giving visibility to errors and logs.
- *Low Level*: Low level code profiling can be very slow and has a lot of overhead, potentially making the software slower than it should be. This kind of profiler usually tracks performance of every single method in your code and potentially, every single line of code within each method. These types of profilers are also tracking memory allocations and garbage collection to help with memory leaks. They're very good at finding that hot path, figuring out every single method that's being called, and what's using the most CPU.
- *Hybrid*: A hybrid profiler gets some detail from server-based profiling and desktop based profiling thus giving a high level overview with low overhead.

Flat and Call-graph profilers are profilers based on output. Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context. Call-graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee.

3.3.4. Slicers

Slicing has served as the basis of numerous software comprehension tools. The slice of a program for a particular variable at a particular line of the program is just that part of the program responsible for giving a value to the variable at that spot. Obviously, while debugging you determine that the value of a variable at a particular line is incorrect, it is easier to search for the faulty code by looking at the appropriate slice than by examining the entire program. It is the computation of the set of programs statements, the program slice that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis and information flow control. At first, slicing was only static, i.e., applied on the source code with no other information than the source code. But later *dynamic slicing* was introduced which works on a specific execution of the program (for a given execution trace).

- *Static slicing:* A static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion C = (x, V), where x is a statement in program P and V is a subset of variables in P. A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

- *Dynamic Slicing:* Makes use of information about a particular execution of a program. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.

3.3.5. Version Control System

A tool that manages and tracks different versions of software or other content is referred to generically as a version control system (VCS), a source code manager (SCM), or a revision control system (RCS). These tools aim at developing and maintaining a repository of content, provide access to historical editions of each datum, and record all changes in a log. GIT is a powerful, flexible, and low-overhead version control tool that makes collaborative development a pleasure. It was invented by Linus Torvald to support the development of the Linux Kernel, but it has since proven valuable to a wide range of projects. A "patch" is a compact representation of the differences between two files, intended for use with line-oriented text files. It describes how to turn one file into another, and is asymmetric: the patch from *file1* to *file2* is not the same as the patch for the other direction (it would say to delete and add opposite lines, as we will see). The patch format uses context as well as line numbers to locate differing file regions, so that a patch can often be applied to a somewhat earlier or later version of the first file than the one from which it was derived, as long as the applying program can still locate the context of the change. The terms "patch" and "diff" are often used interchangeably, although there is a distinction, at least historically. A diff only need show the differences between two files, and can be quite minimal in doing so. A patch is an extension of a diff, augmented with further information such as context lines and filenames, which allow it to be applied more widely.

A simple patch can be generated by git diff:

```
$ diff --git direct1/file1.c direct2/file2.c
```

This is the Git diff header; diff --git isn't a literal command, but rather just suggests the notion of a Git-specific diff in Linux command style. *direct1/file1.c* and *direct2/file2.c* are the files being compared, with added leading directory names *direct1* and *direct2* to distinguish them in case they are the same (this patch shows the changes from one version to another of the same file). To generate this patch, change the file *file1.c* and run git diff, which shows the unstaged changes between the working tree and the index.

In computer jargon, a "hook" is a general means of inserting custom actions at a certain point in a program's behaviour, without having to modify the source code of the program itself. For example, the text editor Emacs has many "hooks" that allow you to supply your own code to be run whenever Emacs opens a file, saves a buffer, begins writing an email message, etc. Similarly, Git provides hooks that let you add your own actions to be run at key points. Each repository has its own set of hooks, implemented as programs in *.git/hooks*; a hook is run if the corresponding program file exists and is executable. Hooks are often shell scripts, but they can be any executable file. Git Init automatically copies a number of sample hooks into the new repository it creates, which you can use as a starting point. These are named *hookname.sample*; rename one removing the *.sample* extension to enable it. The sample hooks themselves are part of your Git installation, typically under */usr/share/git-core/templates/hooks*.

The *templates* directory also contains a few other things copied into new repositories, such as the default *.git/info/exclude* file. For example, there is a hook named commit-msg, which is run by git commit after the user edits his commit message but before actually making the commit. The hook gets the commit message in a file as an argument, and can edit the file to alter the message. If the hook exits with a nonzero status, Git cancels the commit, which can be used to suggest a certain style of commit message. It's only a suggestion though, because the user can avoid hook with git commit --no-verify. You'd need a different kind of hook on the receiving end of a push to enforce style on a shared repository.

4. How to Use Open Source Tools on Large Code Using Shell Scripts?

Various Linux commands to install and use the aforementioned tools are given below:

4.1. To install Doxygen on Ubuntu use the following command:

```
$ sudo apt-get install doxygen
$ sudo apt-get install graphviz
```

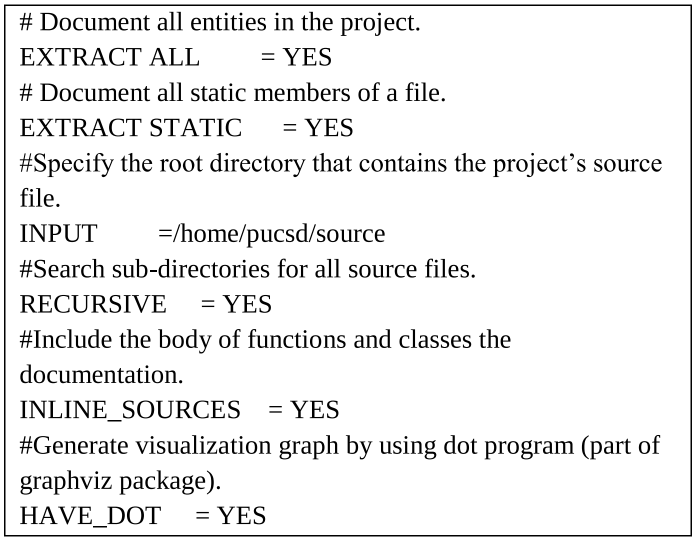
To generate documentation of source code, first generate a project specific doxygen configuration file using the following command:

```
$ doxygen –g myproject.conf
```

You can edit the following options in the configuration file as shown in figure 2.
Now run doxygen with the configuration file:

```
$ doxygen myproject.conf
```

Documentations are generated in both HTML and Latex formats, and stored in ./html and ./latex directories respectively.



```
# Document all entities in the project.
EXTRACT ALL        = YES
# Document all static members of a file.
EXTRACT STATIC     = YES
#Specify the root directory that contains the project's source file.
INPUT        =/home/pucsd/source
#Search sub-directories for all source files.
RECURSIVE    = YES
#Include the body of functions and classes the documentation.
INLINE_SOURCES    = YES
#Generate visualization graph by using dot program (part of graphviz package).
HAVE_DOT     = YES
```

Figure 2. Generated Configuration File

4.2. Using Profilers

The 'gprof' command produces an execution profile of C and many other programming languages. The effect of called routines is incorporated into the profile of each caller. The gprof command is useful in identifying how a program consumes processor resource. To find out which functions in the program are using the processor, you can profile the program with the gprof command. Gprof can be installed using the following command:

```
$ sudo apt-get install binutils
```

Compile the '.c' code with the -pg option as follows:

```
$ gcc -Wall -std=c99 -pg sorting.c -o sorting
```

Once compiled, run the program named 'sorting':

```
$ ./sorting
```

After successful execution, the program will produce a file named "gmon.out" that contains the profiling information, but in a raw form, which means that you cannot open the file and directly read the information. To generate a human readable file, run the following command:

```
$ gprof sorting gmon.out > prof_output
```

This command writes all the profiling information in human readable format to "prof_output" file. You can give a different name for the output file.

4.2.1. *Flat profile and Call graph*

The file contains profiling data, divided into two parts: Flat profile and Call graph. While the former contains details like function call counts, total execution time spent in a function, and more, the latter describes the call tree of the program, providing details about the parent and child functions of a particular function.

4.3. Program Slicing

Program slicing is a technique for aiding, debugging and program comprehension by reducing complexity. The essence of program slicing is to remove statements from a program that do not affect the values of variables at a point of interest. It is of great use in software comprehension, for example when software has to be maintained or evolved. It can be used to generate many automated software metrics such as a measure of cohesion in a program fragment. It can also aid testing, model checking and compiler tuning amongst others. In order to carryout program slicing we have to define slicing such that a slice is only equivalent to the original program when the original program terminates. Furthermore, a strictly minimal slice cannot be found and only an approximation can be computed, however, this is usually good enough and program slicing is still a useful technique.

Most forms of program slicing are syntax preserving. That is they leave the syntax of the original program largely untouched and simply remove statements to create a program slice. The exception to this rule is that if removing statements may cause a compilation error then statements may be altered, this is still considered syntax preserving. If this constraint is relaxed and the slicer is allowed to make syntactic changes as long as the relevant semantics are preserved then this is known as amorphous slicing. Consider the following program fragment to be sliced:

```
scanf("%d",&n);
s = 0;
p = 0;
while (n > 0)
{
s = s + n;
p = p * n;
n = n - 1;
}
printf ("%d%d", p, s);
}
/* the slice point is the end of the program */
```

Figure 3. A Typical 'C' Program

Different slices generated for the above program (Figure 3) is as follows:

Static Slice	Dynamic Slice	Condition Slice

scanf ("%d", &n);
p = 0;
while (n > 0)
{
p = p * n;
n = n - 1;
}
	p = 0;	scanf ("%d", &x);
		scanf ("%d", &y);
		if(x > y)
		z = 1;
		else
		z = 2;
		printf ("%d", z);

Figure 4. Slices for the Program in Figure 3

4.4. Code Reading

4.4.1. Regular Expression or Patterns

Most of the program editors provide a command for searching text string using regular expression or pattern. A regular expression allows the declarative specification of complex strings.

- Regular expressions consist of:
  - letters (that represent themselves) and
  - special symbols
  - backslash escapes special symbols

The following table contains the specific patterns

Table 1. Examples of Regular Expressions

Pattern	Description
.	Any character
[pqr]	Any of the characters p, q or r
a*	Zero or more occurrences of a
^	Beginning of a line
$	End of a line
\<	Beginning of a word
\>	End of a word
\d	Digit
\D	Non-Digit

Expr?	Expression 0 or one times
Expr*	Expression 0 or more times
Expr{n,m}	The *expression* at least n but no more than m times

Here is some sample examples of patterns:

1. Pattern that works as a trim function on strings:
   /^\s+ | \s+$/g
   where "^\s+" – means replace any first spaces of line, "^" - begin of line, "\s" - space character, "+" – matches one or more times, "|" – means or, "\s+$" – means replace any last spaces of line, "$"- end of line, "\g"- replace all matches. First and last "/" shows begin and end of the pattern, respectively
2. Pattern to find if a string has any substring matching a floating point number:
   [- +]?([0-9]+ \ . ?[0-9]* | \ .[0-9]+) ([eE] [- +] ? [0-9] +)?
3. Pattern to check whether a given value is a IP value:
   \b \d{1, 3} \ . \d{1,3}\. \d{1, 3}\. \d{1, 3} \b
4. Pattern to match all date formats:
   (\ d +) [- . \ /](\ d +) [- . / \] (\ d +)

4.4.2. Editor as a Code Browser (id-utils or Ctags)

Editor as a code browser provides the tag facility and use of regular expressions. The id-utils indexing tool creates a database of identifiers and their occurrences within a set of files. Separate command-line tools can then be used to query the database for files containing an identifier, edit a set of files that satisfy a query, list of tokens within a file. Compared to ctags, id-utils handle a wider range of identifiers including literals and names of include files. The exuberant ctags indexing tool enhance ctags by providing support for 23 different languages.

```
$ sudo apt-get update
$ sudo apt-get install id-utils
$ sudo apt-get install exuberant-ctags
```

You can ensure whether id-utils package is installed using the command given below:

```
$ sudo dpkg-query -l | grep id-utils *
```

Ctags allows to quickly jump to function call even if the function definition source code are from other directories. In order to use ctags, first run the command at destination directory where the source codes are located.

```
$ ctags –R *
```

-R is to recursively go across all the directories, and a 'tags' file will be created. Pressing 'ctrl]' allows you to jump from function call to function definition on the keyword. Let say when you discover a function call for which you want to see the definition, simply point the cursor to that function and press 'ctrl]' and it will brings you there. Use 'ctrl I' and 'ctrl o' to travel to forward and backward of the check points.

4.4.3. Code Searching with Grep

Grep comes already installed on every Linux system, so there is no need for manual installation. Simple wildcard search:

- Search for definitions
- Using a word's root
- Locating files: grep on the output of ls
- File list: grep -l
- Use the result of a file list vi `grep -l ...`
- Stream edit the grep result to create scripts
- Postprocess with grep (-v) to eliminate noise (malloc/xmalloc)
- Use sort -u to eliminate duplicates
- Use fgrep to search for a fixed list of strings
- Use find and xargs to obtain filename lists
- Simple wildcard search

Following is a summary of the *grep* command options:

- ***-i*** does case-insensitive character matching
- ***-r*** reads all files under each directory recursively
- ***-n*** shows the line number of each match
- ***-c*** shows the match count
- ***-v*** inverts the matching by selecting the non-matching lines
- ***-o*** prints only the matched parts of a matching line, with each part on a separate output line
- ***-w*** only matches on whole words

The following usage of grep command will search for the string 'Sample String' in two files viz. a.txt and b.txt.

```
$ grep 'Sample String' a.txt b.txt
```

The general syntax to use grep command with other commands is as follows:

```
$command|grep'search-pattern'
$ command1|command2|grep'search-pattern'
```

In this example, run ls command and search for the string/pattern called trial.pdf:

```
ls | grep trial.pdf
ls-l | grep trial.pdf
ls-l | *.mov|grep 'happy'
ls-l | *.mov|grep  -i 'happy'
```

4.4.4. Locating File Differences

- Uses:
  - Differences between versions
  - Examining code modifications
  - Verifying test results
- The diff program
- Obtaining a context diff (-c)
- Ignoring blanks (-w)

The command 'diff' analyses two files and prints the lines that are different. In the real sense it outputs a set of instructions for *how to change one file to make it identical to the second file.* Consider two files, **file1.txt** and **file2.txt**.

file1.txt	**file2.txt**	$ diff file1.txt file2.txt output will be: **2,4c2,4**
Saturn Jupiter Mercury Uranus	Saturn Venus Mars Neptune	< Jupiter < Mercury < Uranus --- >Venus > Mars > Neptune

Figure 5. Output of 'diff' Command

The diff output (Figure 5) means that it is describing these differences in a *prescriptive* context: it's telling how to change the first file to make it match the second file.

The first line of the **diff** output will contain:

- line numbers corresponding to the first file,
- a letter (**a** for *add*, **c** for *change*, or **d** for *delete*), and
- line numbers corresponding to the second file.

In the output above, "**2,4c2,4**" means: "Lines **2** through **4** in the first file need to be **c**hanged to match lines **2** through **4** in the second file." It then tells us what those lines are in each file:

- Lines preceded by a **<** are lines from the first file;
- Lines preceded by **>** are lines from the second file.

The three dashes ("**---**") merely separate the lines of file 1 and file 2.

4.4.5. Code Beautifiers

- Fix code that is written inconsistently without following any formatting standards.
- Adopt orphaned code for maintenance.
- Create a temporary version of the code to help you decipher it.
- Integrate code under the common roof of a larger project.

For printing use a pretty-printer, like listings (LaTeX), vgrind (troff).

4.4.6. Version Control System (GIT)

GIT Patch and Git-Hooks can be used as VCS thus helping in tracking project across times. Version control operations like a commit or a check-in adds a change into the repository, an update or check-out retrieves a version from the repository etc. The central repository keeps the historical data and the versions are tracked with version numbers. A patch is a small file that indicates what was changed in a repository. By using patch, you will get differences between one or more files. And later, you can apply the differences (patch) to get the changes on new files. There are many uses for a patch in Git.

$ cat file1.txt	$ cat file2.txt	$ diff -u file1.txt file2.txt > changes.patch $ cat changes.patch
This is first line. This is second line.	This is third line. This is second line.	--- file1.txt 2018-04-15 11:09:38.651010370 -0500 +++ file2.txt 2018-04-15 11:07:13.171010362 -0500 @@ -1,2 +1,2 @@ - This is first line. +This is third line. This is second line.

Figure 6. How to Create a Patch

After patching:

```
$ patch < changes.patch
patching file file1.txt

$ cat file1.txt
This is third line.
This is second line.
```

Git hooks are scripts that Git executes before or after events such as: commit, push, and receive. Git hooks are a built-in feature that run locally. Some example hook scripts include:

- pre-commit: Check the commit message for spelling errors.
- pre-receive: Enforce project coding standards.
- post-commit: Email/SMS team members of a new commit.
- post-receive: Push the code to production.

Git-hooks are simple scripts that run before or after certain actions. There are two types of hooks:

- Client-side – These run on the developer's system
- Server-side – These run on the server hosting the Git repository

Every Git repository has a ***.git/hooks*** folder with a script for each hook you can bind to. The scripts are editable as necessary, and Git will execute them when those events occur. Some of the Git-hooks you can attach scripts to are:

applypatch-msg, pre-applypatch, post-applypatch, pre-commit, prepare-commit-msg, commit-ms, post-commit, pre-rebase, post-checkout. post-merge, pre-receive, update, post-receive, post-update, pre-auto-gc, post-rewrite and pre-push. You can make the script executable by overwriting one of the scripts in ***.git/hooks***.

IV. Conclusions

Software Comprehension is the process of acquiring knowledge and understanding an existing computer program in order to be able to modify it. Increased knowledge enables activities such as bug correction, enhancement, reuse, and documentation. While efforts are underway to automate the understanding process, such significant amounts of knowledge and analytical power are required that today software comprehension is largely a manual task. The developer/programmer needs a well-defined and an efficient strategy for comprehending the program in order to identify the desired code. This paper presents a generalized approach for programmers, particularly "new buddies", to develop an understanding of how applications work so that can modify or add code at the right place. It also shows how to translate this comprehension into effective coding. The programmers should come out with an efficient understanding of correctly framing the software problem and gather resources needed to obtain an output. Thus writing code has become very easy as compared to making changes in the existing code written by somebody else, which is indeed a challenging job.

References

[1] G. M. Weinberg, Editor, "*The Psychology of Computer Programming*", vol. 932633420, 1971. New York: Van Nostrand Reinhold.

[2] M. A. Storey, "*Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future*", Software Quality Journal, vol.14 (3), pp. 187-208, 2006.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster,"*Program Understanding and the Concept Assignment Problem*", Communications of the ACM, vol. 37(5), pp. 72-82, 1994.

[4] R. P. Gabriel, Editor, "*Patterns of Software*", vol. 62, 1996, Newyork, Oxford University Press.

[5] S. Rugaber, "*Program Comprehension for Reverse Engineering*", In AAAI Workshop on AI and Automated Program Understanding, San Jose, California, pp. 106-110, 1992.

[6] D. L. Parnas, "*On the Criteria to be used in Decomposing Systems into Modules*", Coomunications of the ACM, vol. 15(12), pp. 1053-1058, 1972.

[7] V. Rajlich and N. Wilde, "*The Role of Concepts in Program Comprehension*", In Program Comprehension, Proceedings, 10th International Workshop on IEEE, pp. 271-278, 2002.

[8] J. Siegmund, C. Kastner, S. Apel, A. Brechmann and G. Saake," *Experience from Measuring Program Comprehension-Toward a General Framework*, 2013.

[9] B. Di Martino, C. W. Kebler, "*Two Program Comprehension Tools for Automatic Parallelization*," IEEE Concurrency, vol. 8(1), pp. 37-47, 2000.

[10] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan ans S. Li," *Measuring Program Comprehension: A Large-Scale Field Study with Professionals*", IEEE Transactions on Software Engineering, 2017.

[11] R. Schauer and R. K. Keller, " *Integrative Levels of Program Comprehension*", In Reverse Engineering, WCRE'08, 15th Working Conference on IEEE, pp. 145-154, 2008.

[12] N. Saroni, S. A. Aljunid, S. M. Shuhidan, and A. Shargabi, "*An Empirical Study on Program Comprehension Task Classification of Novices", In e-Learning, e-Management and e-Services (IC3e),* 2015 IEEE Conference on IEEE, pp. 15-20, 2015.

[13] R. Wettel and M. Lanza, "*Program Comprehension through Software Habitability*", in Program Comprehension, ICPC'07,15th IEEE International Conference on IEEE, pp. 231-240, 2007.

[14] N. Sasirekha, a. E. Robert and D. M. Hemlata, "*Program Slicing Techniques and its Applications*", arXiv preprint arXiv: pp. 1108-1352, 2011.

[15] N. Carvalho, C. da Silva Sousa, J. S. Pinto and A. Tomb, "*Formal Verification of kLIBC with the WP Frama-C Plug-in*", in NASA Formal Methods, pp. 343-358, 2014.

[16] M. A. Storey, K. Wong and H. A. Muller," *How do Program Understanding Tools Affect How Programmers Understand Programs?.*", In Reverse Engineering, Proceedings of the 4th Working Conference on IEEE, pp. 12-21, 1997.

[17] Y. Liu, X. Sun, X. Liu and Y. Li, "*Supporting Program Comprehension with Program Summarization*", In Computer and Information Science (ICIS), IEEE/ACIS 13th International Conference on IEEE, pp. 363-368, 2014.

[18] A. Von Mayrhauser and A. M. Vans, "*Program Comprehension During Software Maintenance and Evolution*," Computer, vol. 28(8), pp. 44-55, 1995.

[19] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen and R. Koschke, " *Systematic Survey of Program Comprehension through Dynamic Analysis*," IEEE Transactions on Software Engineering, vol. 35(5), pp. 684-702, 2009.

[20] E. Soloway, K. Ehrlich, "*Empirical Studies of Programming Knowledge*,"IEEE Transactions on Software Engineering, vol. 5, pp. 595-609, 1984.

[21] F. Détienne,"*Expert Programming Knowledge: A Schema-Based Approach*", Psychology of Programming, pp. 205-222, 1990.

Author's Profile

Jyoti Yadav is currently working as an Assistant Professor in Department of Computer Science, Savitribai Phule Pune University. She has 22 years of teaching experience. She has been awarded Ph.D. (CS) and M.Phil. (CS) from same University. She has published research papers in many National and International journals. She has been actively working on many major and minor research projects in the University. The broad perspective of her research areas include Soft Computing, Fuzzy Logic, Fuzzy Image Processing, Big Data Analytics, Genetic Algorithms, and Data Science and Data Mining, Cloud Computing, Blockchain Technology, Software Defined Networks and Quantum Computing etc.