Journal of University of Duhok., Vol. 22, No.1 (Pure and Eng. Sciences), Pp 90-105, 2019
https://doi.org/10.26682/sjuod.2019.22.1.11

BEST PRACTICES AND RECOMMENDATIONS FOR WRITING GOOD SOFTWARE

QUSAY **I**DREES **S**ARHAN
Dept. of Computer Science, College of Science, University of Duhok, Duhok, Kurdistan Region, Iraq

(Received: November 6, 2018; Accepted for Publication: March 27, 2019)

ABSTRACT

Writing good software is not an easy task, it requires a lot of coding experience and skills. Therefore, inexperienced software developers or newbies suffer from this critical task. In this paper, we provide guidelines to help in this important context. It presents the most important best practices and recommendations of writing good software from software engineering perspective regardless of the software domain (whether for desktop, mobile, web, or embedded), software size, and software complexity. The best practices provided in this paper are organized in taxonomy of many categories to ease the process of considering them while developing software. Furthermore, many useful, practical, and actionable recommendations are given mostly in each category to be considered by software developers.

KEYWORDS: Good Software, coding quality and standards, best practices, practical recommendations, software engineering.

1. INTRODUCTION

Writing software is the art and process of converting what is in mind into reality. This process aims to produce software-based solutions for real-world problems that we encounter in our everyday life. As a result, it makes the life easier and smarter in many directions. Software is used everywhere around us; for example in computers, mobiles, TVs, refrigerators, cars, aircrafts, and many others. Without software, we cannot imagine how our life would be [1]. Software is developed by coding using programming languages such as Java, C#, etc. Besides, different types of models, methods, and tools to manage this development are involved. Writing software is not an easy task at all, it requires a lot of thinking, imagination, and different types of skills. Software that serves our different types of needs is not written equally. Even those that are meant to serve the same purpose can differ greatly in their quality. Therefore, many software developers who are involved in this process suffer from the issue of how good software can be written or produced. Also, suffer from how given software can be considered as good [2]. And yet, people bet their jobs, their comfort, their safety, their entertainment, their decisions, and their lives on software. Thus, it is crucial to be written well. To address the aforementioned issues and to avoid the barriers or blockers of writing good software, we provide whatever helps in this context as best practices and recommendations. This paper is written to be used as a guide to support software developers to write good software. As well, a set of qualities, standards, rules, and practices which are closely related to the principles of good design and engineering but are not limited to any programming language, software domain, or the educational background of whom is writing the software has been gathered, organized, and presented. In literature, there are many efforts for writing good software but they are very specific to certain programming languages and focus only on few aspects of writing software while ignoring many others. We believe that this paper is an essential effort towards considering the most important factors of writing good software. However, we aim to contribute to the following:

- Provide state-of-the-art of major software development best practices and recommendations that help produce software with high quality.
- On top of that, the best practices and recommendations are organized in many categories in a proposed taxonomy to make them easy to be remembered and followed.
- As this comprehensive study provides deep understanding of many aspects regarding writing good software. We hope to provide a holistic

guideline as essential point to guide developers who want to follow up and be professional in this field.

The rest of this paper is organized as follows. Section 2 highlights the software nature and its most important characteristics. In Section 3, the proposed taxonomy of the best practices and recommendations of writing good software is given. Section 4 explains the taxonomy and the included categories. Finally, overall conclusions are provided in Section 5.

2. Software Nature

The software world is an amazing world. It combines creativity and imagination in all of its aspects. Software is the soul of mostly everything people use in their practical lives. Our world cannot be imagined without different types of software products covering various domains such as industry, healthcare, military, transportation, etc. These products have unique nature and characteristics that differentiate them distinctly from non-software products [3, 51]. Understanding software nature and its characteristics alongside many directions should ease the understanding of how software is used and developed. Therefore, in this section we provide the most important characteristics of software and its nature as follows:

- Software is everywhere around us and it changes the way people live. As the software engineer Mark Andreessen (the co-founder of Netscape, one of the first web browser companies) said, "*Software is eating the world*". More and more stuff are getting automated to make our life easier, faster, and smarter, and as much more automation is going to be just a function of software.
- Software plays a dual role. It is a product and a vehicle for delivering products at the same time. As a product, it is created to deliver its functionality to users via various forms including software applications, web applications, web services, embedded systems, etc. As a vehicle for delivering products, software constitutes the foundation for creating, managing, controlling, etc. of other software products such as software tools, frameworks, and platforms.
- Industrially, the majority of products existing around us today are physical and tangible things such as cars we drive, watches we wear, or TVs we watch. In contrast, a software product is the only product that can be used without touching it or seeing its internal structure.
- Once you design a physical product such as a lamp or teacup and make it exists, it cannot be evolved and changed over time. It fulfills its requirements and that is all; changing its requirements means a new product has to be produced from scratch. For software products, new requirements come in all the time so changes are applied on the same product without developing a new one from scratch.
- Software products are not like physical products; they do not break down after too much use.
- Not every software product is good even if it delivers its functionality in a proper way.
- A single mistake in writing software may collapse the work of many years, turns software useless, and causes catastrophic events. Take the Mars climate orbiter as an example; it crashed in 1999 because of a mistake of using different measurement units by different developers groups.
- The development and advancement in the field of software are growing very rapidly compared to other fields. Every single day there are new software innovations including developing new software applications, creating new software frameworks, and proposing new software technologies.
- Studying software is not similar to other study fields. As it is used in every device we use or see today, studying it requires studying other fields in depth. For example, planting a chip in a human body and program it requires the study of human body in detail alongside many directions.
- In most cases, developing software products may cost nothing in terms of manufacturing costs. And, may need no raw material, expensive devices and tools.
- Software is not a program. Software is produced by a team of people which is not the case in writing programs [4]. Besides, a number of differences between software and program exist regarding usage, size, users, development style, and many others [5]. Table 1 presents the main differences between them.

Table (1): Software vs. program

	Software	**Program**
Definition	A collection of several programs and related configuration files that manage their work alongside documentation.	A set of instructions (written in a computer understood language) manipulate a set of data to perform a specific task.
Problem scale	Developed to solve large-scale problems.	Developed to solve small-scale problems.
Developed by	A team of software specialists, each focusing on one part of the development process.	An individual.
Size	Extremely large.	Small.
Usage	Developed for a third party for the sake of money. Therefore, It is used by a different number of customers.	Developed mostly for the personal use. Therefore, the programmer himself is the sole user.
Functionality	Has full-scale functionality.	Has limited functionality.
User involvement	Most users are involved with the development.	A single developer is involved only.
User Interface	The user interface must be taken into consideration seriously as developers of the product and its users are totally different.	The user interface may not be very important as the programmer is the sole user.
Documentation	Must be well documented.	Very little documentation is expected.
Development process	Must be developed using software engineering principles and practices.	Can be developed according to the programmer's individual style of development.
Installation	Required.	Not required.

3. The Proposed Taxonomy

This section presents the proposed taxonomy of the best practices and recommendations that help developers to produce good software. The taxonomy is divided into three main categories: (1) before writing software (2) while writing software, and (3) after writing software. It is worth mentioning that each main category has some sub-categories which are explained in the subsequent sections. Figure 1 shows the proposed taxonomy.

4. Best Practices and Recommendations

In this section, the proposed taxonomy that assists developers to create good software illustrated in categories. The categories cover different aspects of writing any software and they are based on software engineering, industry, and coding experiences.

4.1 Before writing software

The best practices and recommendations that have to be taken into consideration before starting the process of writing software are presented in the following subsections.

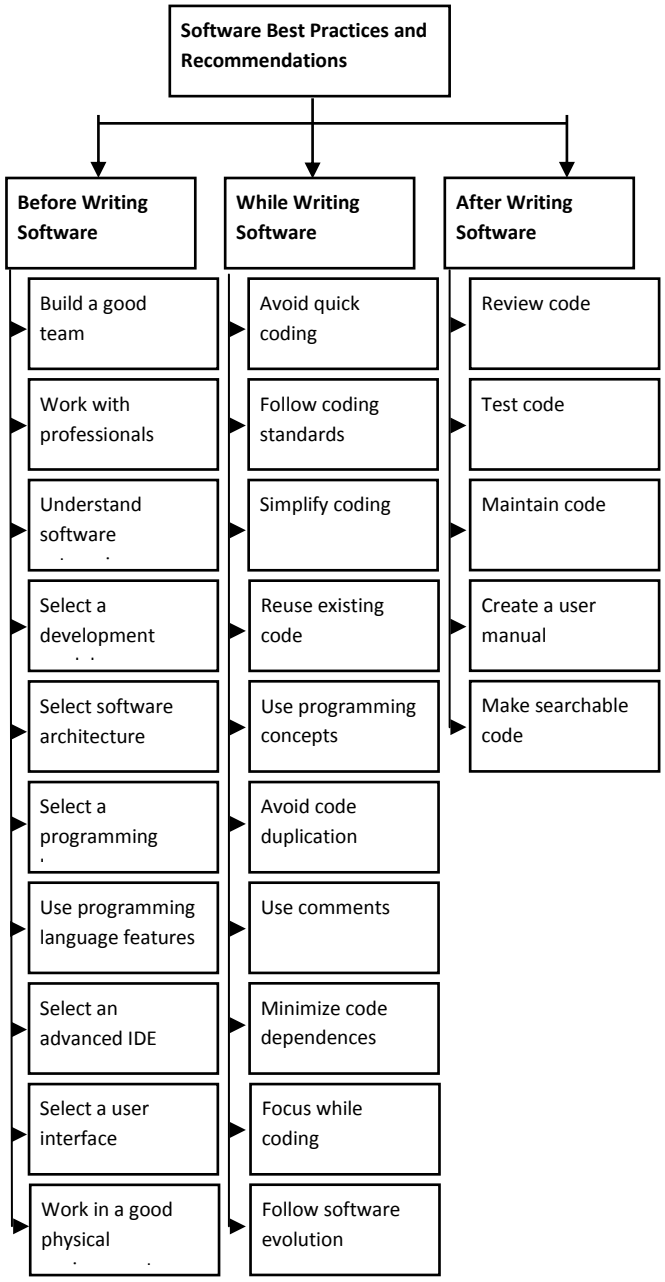

Fig. (1): Taxonomy of writing good software

4.1.1 Build a good team

Software development team is extremely necessary for planning, organizing, monitoring, and controlling the development process. Well-established team leads to timely and high-quality software delivery. Building a good team has an obvious positive impact on writing software and its quality [6]. Moreover, greater clarity in expressing ideas can be gained through group discussion, efficient use of resources (especially time) can be achieved, and workloads can be shared. To build a good software team to fulfill the vision mentioned above, a clear goal has to be set. Specifying the goal is the starting point of achieving everything that can be thought of. If writing good software is not the goal that you or your team seeks for, then never expect to achieve it. Good software is the result of a predefined goal and everything leads to achieve that goal. Therefore, a clear goal and vision have to be set before start looking for what helps to write good software. Software that makes our lives easier and smarter is good software. In contrast, software that makes things worse and does not serve people in their practical lives or does not deliver what customers want is not good software even if it is written perfectly. Some other important and most prevalent points should be taken into account while building a good software team [7], as follows:

- The team should be technically skilled.
- The team should closely represent the users of the software. Therefore, build a diverse team that involves users in the development process.
- The team should work collaboratively to achieve the required tasks. Therefore, team members should have good communication skills and understand each other.
- The team should have a common definition of success and believe that the whole is greater than the sum of parts.
- The team membership, size, skills, and resources should match the task at hand.
- The team should have a good leader who manages and plans tasks across the team. Encourages team members to show their best creativity, skills, and abilities.

In most cases, software is developed by a team. However, it can be developed by an individual when the software size is small. Therefore, this section can be considered mostly when the size is large.

4.1.2 Work with professionals

One of the enabling elements of writing good software is to work and be in touch with experts and professionals in the field of software development. Building a good team can provide the opportunity to achieve this in one way or another. Also, it can be achieved through collaborative projects, training courses, workshops, or conferences. Many well-known and senior software developers have websites,

Facebook pages, Blog pages, or YouTube channels where one can follow their software development activities and learn from them. Many others publish books and research articles on software development. One of the best examples to mention in this respect is using Livecoding.tv [8]. It is an interactive social coding platform where one can watch professionals coding software products in real time using a variety of programming languages. By doing so, software developers will improve their development skills to the next level and will learn innovative ways of coding never thought of before; which eventually leads to produce good software [9].

4.1.3 Understand software categories

Different categories of software as shown in Figure 2 are required to serve different types of our day-to-day activities. The software team should know and understand each category in detail with its own set of attributes. Understanding software attributes of a specific domain helps to create software with high quality.

Real-Time Software	System Software	PC Software
Embedded Software	Business Software	Web Software
Mobile Software	Engineering Software	Scientific Software
Software Categories		

Fig. (2): Different types of software

If the category or domain of the software required to be developed is Web for example; Web software attributes must be understood in detail by the software team. The most important Web software attributes are network intensive, content driven, and continuously updated. Thus, developing such software requires all these attributes to be implemented to get software with high quality.

4.1.4 Select a development model

Generally, software development models (also called software process models, software life cycle models, or software engineering paradigms) represent systematic manners that have to be followed to develop software. These models organize various tasks of software development into a number of phases. Then, specify how these phases (and tasks within each phase) are to be performed. To produce good software, developers should select a proper development model. The selection of a development model depends on many factors such as software team, software domain, software size, software complexity, development time, methods and tools to be used, deliverables that are required [10]. Currently, there are different types of software development models such as waterfall, prototype, incremental, spiral, and agile. Each model with its own usages, phases, advantages, and disadvantages [11]. For example, in the waterfall model (also called classical model or linear sequential model), the tasks that go into making software are performed sequentially in multiple phases. Figure 3 shows a version of the model with six phases.

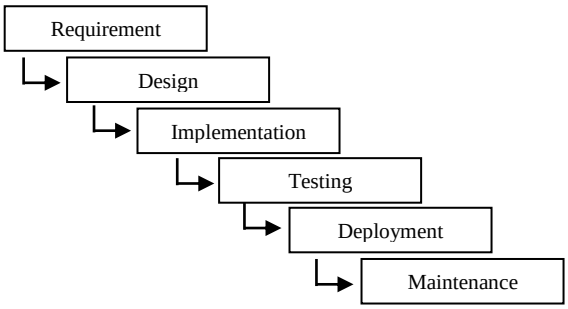

Fig. (3): Waterfall model

The phases of the waterfall model are briefly described below.

- **Requirement phase:** In this phase, all requirements regarding developing software are gathered then analyzed. Therefore, user participation is very important to get an approved set of requirements documented in a Software Requirements Specification (SRS) document. It is worth mentioning that the requirements gathered in this phase form the basis of all further phases in the model.
- **Design phase:** In this phase, the requirements in the SRS document are translated into a design. Besides, this phase acts as a bridge between what the user wants and the code that will be written to satisfy the user requirements.
- **Implementation phase:** In this phase, the code is written and the user manual may also be written.
- **Testing phase:** In this phase, the code is tested in form of unit testing for lowest level components, integration testing for groups of components, and testing of the software as a whole. The last and the most important task in this phase is usually the

acceptance testing to validate that the software meets user needs. Therefore, this task requires user involvement.

• **Deployment phase:** In this phase, the software is made operational. This phase includes software installation and user training to use the software properly. It is worth mentioning that when the software is operational, new requirements may be discovered especially if the requirements phase is not defined well and not analyzed with user involvement.

• **Maintenance phase:** In this phase, leftover defects found in the previous phase are corrected. In many cases, software is made more efficient, new features are added, or existing features are modified to meet the ever changing needs.

The main advantages of this model are:

• The model is simple to understand and use.

• The model is well-defined into phases each with its related tasks. Therefore, it provides a clear separation of tasks.

• Its phases are completed one at a time.

The main disadvantages of this model are:

• It assumes that user requirements are clearly defined at the beginning of the software development.

• User feedbacks are not taken into consideration as the software is being developed. Because it does not support changes in requirements that are identified in other phases.

• It is time-consuming in that a working version of the software will not be available until late in the development time-span.

It is worth mentioning that other software development models use the waterfall model as their basis and share some of its common phases. Many others have their own unique phases. Selecting a software development model has a big impact on building software. If the model is inappropriately chosen, the software product will distinctly suffer. Also, it may affect the development team. Thus, may need formulation of unique team roles but this should not affect the sequencing order in the proposed taxonomy.

All the tasks and activities within the selected software development model should be measured and evaluated regarding quality. The software development model is only as good as the requirements describe the current problem, the designs represent the solution, the codes lead to executable software, and the tests exercise the software to uncover errors.

4.1.5 Select software architecture

Software architecture generally means how the structural elements and functional components of a software product are organized and how they are communicating with each other to achieve a common goal. From software engineering perspective, the good software product is always based on well-selected software architecture. Therefore, the process of selecting which architecture out of many available options has to be used to develop a particular software product has a big impact on its quality [12]. The software quality will be affected as the selected architecture determines many aspects of the software including:

• How its elements and functional components will be organized.

• How they will communicate with each other.

• How its cost, size, and complexity will be derived.

• How its functionality will be delivered to users.

• In case of any maintenance, how it will be tested and maintained.

Currently, there are a various number of software architectures and each one has its own set of advantages and disadvantages [13]. For example, the traditional monolithic architecture is used today in a massive number of software products. Figure 4 shows monolith based software with all functional units in a single process.

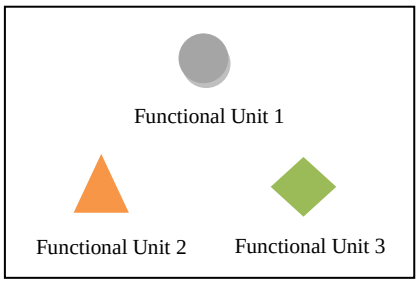

Fig. (4): Monolithic based software

Software products based on monolithic architecture are difficult to understand, maintain, and evolve which considered as the main drawbacks of this architecture type [14]. Another common architecture is the microservice architecture which recently gained more attention compared to the monolithic. The concept of microservice based software development is to divide any software into a number of small,

independent (not in the same process), and functionality-focused microservices [15]. It is worth mentioning that microservice is derived from the traditional Service Oriented Architecture (SOA). Figure 5 shows microservice based software where each service provides certain functionality.

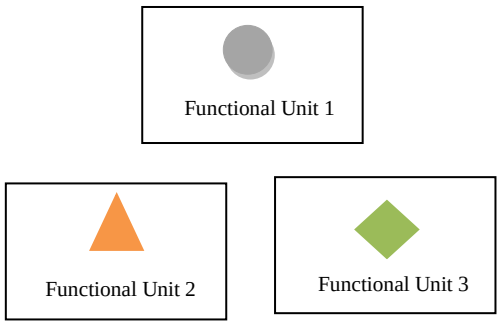

Fig. (5): Microservice based software

After both monolithic and microservice architectures described briefly, the main differences between them are presented in Table 2.

Table (2): Monolithic vs. microservice

	Monolithic	Microservice
Structure	The functional components or modules are gathered into a single file. Thus, user interface, business logic, and data layer are not clearly separated	A number of small-scale functional units deployed as services. Besides, these services can collaborate with each other by integration and composition.
Reusing	Other developers cannot reuse monolithic based software as it is not intended for reusing.	Other developers have the ability to reuse complete functional units through composition.
Complexity	Lower complexity as there is no interaction required between its various components.	Higher complexity as there are different issues to deal with including interactions, locking, integrity etc.
Development	Usually, it is written in a single programming language.	Usually, it is written in multiple programming languages.
Accessibility	It cannot be used by multiple users at the same time.	It can be used by multiple users at the same time.
Maintenance	As everything is placed in one place and there is no clear separation between the functional components, any attempt to maintain code may collapse everything or may cost so much.	Microservices are small in size. Thus, the cost to replace them with a better implementation, or even delete them altogether, is cheap in terms of cost, time, and effort.
Deployment	It is executed as one complete unit.	Each microservice can be deployed independently. A system may have a number of deployed microservices with the ability to manage the work of each one separately.
Resilience	A small fail may break down the whole program.	A single-component failure does not break down the whole software.

4.1.6 Select a programming language

There are a tremendous number of programming languages that can be used to create different types of software. Many of them share common features, capabilities, and target common software domains such as desktop, web, mobile, and embedded. Many others have their unique features, capabilities, and target only specific domains. Therefore, choosing which language (sometime multiple languages) to be used for developing software has a direct impact on its quality and functionality [16]. For example, microservice based software development is a hot topic nowadays and currently there are many languages including Java and Go that support developing this kind of software. But Jolie is the first and the only programming language that has been designed completely to develop microservice based software and supports its full concepts [17]. Therefore, using a language other than Jolie in this respect will produce software that does not implement all concepts of microservice which is not the right thing to go for. Based on what mentioned above, software developers should take this important point into account to produce good software with the right functionality and fully implemented concepts.

4.1.7 Use a programming language's features

Many developers do not realize that the way they select, learn, understand, and use a programming language affects the quality of their

software in one way or another. Inexperienced developers learn many languages over time but do not dive deeply into language's architecture, concepts, and full features [18]. In many cases, those developers write many pieces of code to achieve a specific task (for example, displaying the contents of a given folder), whereas the same task can be achieved using a single built-in function provided by the used language itself. In many other cases, developers use external libraries to perform tasks, whereas the used language provides a built-in library that performs the same tasks even more efficiently. Going deeply through all language-related concepts, features, and studying them across many aspects helps to use that language in an effective and efficient way never expected before.

4.1.8 Select an advanced IDE

An Integrated Development Environment (IDE) is a programming environment to help developers write software and manage related activities. Generally, any IDE consists of code editor, compiler/interpreter, debugger, and built-in utilities that simplify the software development and hide its intricacies (For example, developers will be shielded from going through the internal details of how their codes are compiled and converted into machine codes). Using an advanced IDE simplifies the learning of the language being used, helps developers to measure the performance and resource allocation of their software in many directions, and increases their productivity [19]. Also, it helps making less coding errors. For instance, if a variable or a method has to be renamed, IDEs provide the "refactor" option to help in this respect. This option changes all the calls to that variable/method name. Thus, there is no need to change them manually one by one which will make the code prone to various errors. Currently, there are a number of different IDEs each having its own set of features including supporting a number of different programming languages. For example, one of the widely used IDEs for Java developers is NetBeans IDE [20]. In this IDE, developers can use Profiler tool to measure the CPU time, RAM allocation, execution time of each method, and many other available important features which are not available in traditional Java IDEs. Doing so, help them to decide if their software products meet specific performance and resources requirements or not.

4.1.9 Select a user interface

Any software code is good when it provides a good user interface to its users to interact with its functionality. Therefore, software developers should take this point into account while coding. User interfaces represent one of the most important parts of any software code because it determines how easily users can use it and as it is the only visible part for users. It is worth mentioning that any powerful software code with a poorly designed user interface has no or little value. Generally, user interfaces are divided into five types [50], as follows:

- **Command Line Interface (CLI):** It is the simplest one among other types of user interfaces. In the CLI, the user interacts with the software by typing commands in a screen using the keyboard and the software provides back the output by printing it on the same screen. Command prompt application in Windows operating system is one of the best examples that utilize CLI to deliver its functionality to users.
- **Graphical User Interface (GUI):** The GUI presents a user-friendly mechanism for interacting with any software via a set of graphical components including menus, buttons, labels, and many others. Besides, it provides a distinctive look and feel to the underlying software. Paint application in Windows operating system is one of the best examples that utilize GUI to deliver its functionality to users.
- **Zoomable User Interface (ZUI):** The ZUI is a special type of GUI. In the ZUI, users are able to see more detail or less by changing the scale of the viewed area. In other words, the user can browse almost everything simply by zooming in and out. Eagle Mode application [52] is one of the best examples that utilize ZUI to deliver its functionality to users.
- **Voice User Interface (VUI):** In the VUI, users interact with the software through voice/speech based commands to utilize its functionality. The most important feature of VUI, it provides hands-free and eyes-free interaction. Speech Recognition and Cortana applications in Windows operating system, Amazon Alexa, and Apple Siri [53-55] are the best examples that utilize VUI to deliver its functionality to users.
- **Activity User Interface (AUI):** It is also called gesture user interface. In the AUI, gestures can originate from the human face or hands and then recognized as commands to the underlying software. Touchscreen in smart phones is one of

the best examples that utilize AUI to deliver its functionality to users.

However, selecting the appropriate user interface depends mainly on the software aim and domain. Besides, any good software should have a user interface with important characteristics, as follows [21]:

- **Clear:** Everything in the user interface should be clear including its objectives, how to use it, and many others.
- **Concise:** The words used to describe the aspects of the user interface should be concise and give useful meanings. In GUI for example, the title of a button should be simple and reflect the function of that button.
- **Attractive:** The user interface should be attractive in terms of colors, fonts, and shapes. This creates a nice user experience while using it.
- **Efficient:** The user interface should perform its tasks properly and efficiently. For example, if the user clicked on an EXIT button to exit from the current application, the application should be closed directly without any delay.
- **Forgiving:** The user interface should not allow users to make mistakes while using it. In case of any, the interface should be able to recover or undo the process. For example, if the user deleted by mistake an important file, the interface may offer options to recover it.
- **Feedback:** Users should be notified about the tasks requested to be performed. For example, the user interface should notify the user after the task is performed successfully or in case of any unexpected error due to certain reasons.
- **Speed of learning:** The interacting with the user interface should be easy and does not require special trainings to learn how to use it.

4.1.10 Work in a good physical environment

Many developers believe that they can code effectively in any physical environment. And, consider that different physical environments including offices and laboratories do not affect the code quality and productivity. Many studies proved the opposite, physical environments have obvious impacts on quality and productivity. Often, software bugs are produced when developers lose their attentions while coding. To overcome this issue, concentration is required. One of the main factors that help writing good code is working and coding in inspiring environments. That is, the environment where everything in it supports your work and creativity, a place where you can concentrate and be motivated while coding. Generally, physical environments have different elements that affect the code quality if they were set or used in improper ways [22], as follows:

- **Lighting:** As software developers spend most of their time doing coding in offices and laboratories, the lighting of these environments have a direct effect on them. When there is no light or the light is dim, the human body produces a hormone called melatonin or darkness hormone. This hormone makes the body in an inactive or sleeping state. The natural defender against this hormone is the sunlight. It stops the production of the melatonin and keeps the body active and awake. Based on the aforementioned information, offices and laboratories where developers work at should consider the effect of lighting on developers and provide an appropriate lighting system. With no doubt, utilizing the sunlight is more preferred compared to others. This can be achieved by using a lot of windows in such environments. If providing the nature sunlight is difficult for any reason, lighting that simulates the sunlight can be used instead. By this, the developers will be awake and active while coding. As a result, they will be more productive.
- **Coloring:** Different colors have different effects on each one of us and especially software developers. Each developer likes colors that motivate him/her in working. Therefore, offices and laboratories where developers work at should consider the impacts of colors on the developers and their work. Colors such as green, red, and blue generally are the creativity colors. Offices and laboratories can use these colors (rather than using only white or light yellow) in balance to paint their all or some wall surfaces to motivate developers and increase their working enthusiasm.
- **Ventilation:** When there are many developers work in a small office environment with no good ventilation, do not expect from them to do good while they code. Bad ventilation usually caused by either no availability of enough windows or no good ventilation system is supported in the office. However, it decreases the oxygen which affects directly on the developers such that they cannot breathe properly and then feel tired. On top of that, this will have a direct bad impact on their performance while coding.
- **Air-conditioning:** The productivity of software developers gets affected by the air-conditioning.

Therefore, office environments should consider this issue too. They should not be warm in summer nor cold in winter. It is recommended that the temperature be around 25 °C over all seasons to provide a productive working environment.

- **Adjustability:** As software developers sit a lot on chairs in front of computers and use keyboard and mouse devices to do coding, they should feel comfortable in their sittings and in using devices. For example, some developers like sitting on circled chairs, using wireless mouse/keyboard devices, and specific type of monitors. The aforementioned preferences are existent due to the physical anatomy and issues of the human body. As an example, sitting on a chair for a long time makes body pain for developers. Therefore, it is preferred to equip office environments with chairs, monitors, and many other equipment that are highly adjustable. As a result, every developer will be able to make his/her own configurations easily. Providing adjustable and configurable equipment in office environments decreases body pain and eventually makes developers comfortable and more productive.

4.2 While writing software

The best practices and recommendations that have to be taken into consideration while writing software are presented in the following subsections.

4.2.1 Avoid quick coding

Quick coding has direct and bad impact on code quality. In many cases, software developers become under pressure to follow a schedule, meet a deadline, or add a new feature within a limited time. Such cases lead them to code more in a short period. As a result, they produce various defects [23]. In the short-term, many defects have no instant effects on code functionality which make them ignored totally by developers or they will be added to a list of problems that have to be fixed later. With continuing coding in the same wrong manner, new defects will be added to the list. By the time, developers will face a full list of defects based on the ignored ones in the earlier phases of coding making code much harder to be corrected. However, defects have to be fixed directly once they are found to overcome issues of fixing them later in terms of time, cost, and risk. Or, to not forget them; they can be tracked and logged using different tools and frameworks. For example, the open-source Apache Log4j is a well-known Java-based logging framework that could help very well in this context [24].

4.2.2 Follow coding standards

Coding standards are found mainly to format and style software code in a uniform way. As a result, these standards prevent developers (especially when there are many developers working on the same software code) to format and style a piece of code in their own personal and non-uniform way. Alongside that, they help to produce code without bugs [25] and to improve the readability of the software, allowing developers to understand a given code more quickly and thoroughly. Coding standards are of many forms, as follows:

- Using spaces to indent code.
- Using easy-to-type, easy-to-remember, and self-explained variable names.
- Naming conventions for classes, interfaces, and methods.
- Using an empty line to separate two code sections.

Many software organizations provide coding standards and conventions for different programming languages. For example, Java developers can use Google Java Style [26] or Sun Code Conventions [27] in their coding. Currently, there are a tremendous number of tools and frameworks to help developers analyzing a given code (written in any language) to ensure applying coding standards or to avoid unwanted coding standards. For example, Java Coding Standard Checker (JCSC) [28] and Checkstyle [29] are free and open-source tools that can be used by Java developers in this context. In many cases, the size of software code is extremely large which means developers will not be able to apply the standards to the whole code. Instead, only the most important pieces of code can be standardized.

4.2.3 Simplify coding

Simplicity is the foundation of readability and maintainability for any software code. It is independent of the size and functional complexity of code. Usually, it is associated with many properties such as creating simple classes with very well-defined functionalities containing simple methods each with a single functional responsibility. On top of that, the relationship between classes should be simple and understandable [30]. Sometimes, the used programming language has an obvious impact on code simplicity. For example, printing a simple

string on the output screen using Java requires multiple lines of code. On the other hand, printing the same string requires only a single line of code in Python as presented in Table 3.

Table (3): Printing a string - Java vs. Python

Language	*Code Example*

Java	public class Hello
	{
	public class void main (String argv [ ])
	{
	System.out.println("Hello World");
	}
	}

Python	print "Hello World"

Regardless of the aim of code, simplicity and cleanness make the code easy to read and maintain. Furthermore, they are considered as the must-have properties of any good code.

4.2.4 Reuse existing code

Writing a piece of code from scratch is not a recommended practice in the software development process, especially when the scale of the code being developed is large. Often, many pieces of code that provide functionality needed to develop the current software are available somewhere on the Internet for free as part of third-party libraries, frameworks, or other software projects. However, if a function that only converts between units of temperature is required to be reused, then there is no need to import or reuse the complete library that contains that function and a hundred of other functions that are not required to be reused. Some of them are even written in your preferred programming language by well-known experts in the field. Or, they are reviewed and tested by experts. However, other pieces of code are not [31]. It is worth mentioning that before integrating the reused code into your own code, it is recommended to test its functionality separately to ensure it has the required functionality and works properly. This testing process is important and critical to reveal the strengths and weaknesses of the reused code. As a result, the weaknesses can be eliminated or maintained then can be integrated into the code at hand. Generally, free software components and frameworks that are used widely have large support community or licensed under well-known software licenses such as Apache, GNU's Not Unix General Public License (GNU GPL), or Berkeley Software Distribution (BSD). Thus, they have fewer bugs compared to others that are developed in-house and not used or supported widely. Code search tools and engines that help in this context gained more attention from researchers and developers due to their good impact on the software development process. Currently, many code search engines are available to help developers find and reuse relevant existing code. For example, SearchCode [32] is one of the leading code search engines that help developers to search for existing code written in various programming languages. Reusing existing code in proper ways saves a lot of time, effort, and increases productivity over time.

4.2.5 Use programming paradigm-related concepts

Almost all software codes are developed based on a software design or programming paradigm. In the software world, there are a number of programming paradigms each with its own set of concepts. Currently, the superior ones are object-oriented and service-oriented paradigms [33]. Developers are using such paradigms in their coding, but most of them do not really utilize all the concepts and features offered by a specific paradigm. For example, developers and especially the newbies use object-oriented paradigm only in terms of creating classes and initiating objects. As a result, their code is full of classes and objects only while ignoring other important concepts and features of object-oriented including inheritance, polymorphism, and many others [34]. Therefore, developers should not limit their using and understanding of object-oriented to classes and objects creation only. These paradigms are found to help writing good software code alongside many directions, so their concepts and features have to be utilized rigorously.

4.2.6 Avoid code duplication

One of the fundamental factors that make any code clean and good is making no code duplication. Code duplication can be found in three main forms, as follows [35]:

- A piece of code such as a class or method is duplicated exactly as it is but with different names.
- A piece of code is duplicated with a very little change.

- A piece of code doing a specific purpose is repeated many times but in different ways and styles.

Coding with duplication affects badly on code in many ways including increasing its size, complexity, and making it difficult to be understood and maintained.

4.2.7 Add comments to describe code

Comments are very useful to explain the purpose and functionality of code. They are considered one of the main properties of good software code. The good code should explain itself to other developers or even to its developer when it is required to make maintenance after sometime of its creation [36]. Generally, code comments are classified into two types, as follows:

- **Header comments:** Are used at the beginning of code files to give general information about the code including the purpose, functionality, names of developers, last date/time modified, and many others.
- **Inline comments:** Are used anywhere between code lines when it is required to explain the reason of doing something, to clarify the functionality of a method, to specify the relationships between classes, methods, and many others.

It is worth mentioning that using comments require attention and skills. For example, developers should know where to use comments, comments should be simple and informative, and not to over use or use inaccurate comments. Currently, many programming languages have built-in tools to parse header and inline comments from source code to produce complete and formatted code documentation. For example, Java language provides a code documentation generator tool called Javadoc [37] that can be used in this respect to parse code comments into Hyper Text Markup Language (HTML) document format that can be viewed from a web browser such as Mozilla Firefox or Google Chrome.

4.2.8 Minimize code dependencies

Making any software code depends heavily on its software or hardware environment might have a bad impact on its performance and quality in case of changing that environment [38]. For example, writing a code with dependencies to work only on a Windows operating system will have a certain level of performance and quality differs from the same code working on a Linux operating system. In this context, dependency can be of many forms, as follows:

- The software code depends on a specific operating system to work properly. Or, it depends on the existence of a system file or folder which can be deleted or modified by the system itself anytime.
- The software code depends on a specific hardware device or hardware specifications to deliver its functionality properly.

Therefore, developers should minimize the number and type of dependencies of their code as possible. Code with a lot of dependencies is not considered a good code.

4.2.9 Focus while coding

Many developers believe that spending a lot of time on coding will increase the productivity and improve quality. But the truth is that coding less with focusing and sustainable pace achieve much more in terms of productivity and quality. Coding so much in a short period with no focus opens the doors to produce code with different types of defects that lead to code with no quality. Therefore, developers should give themselves enough time to observe and analyze the impacts of what they are coding and react accordingly.

4.2.10 Follow software evolution

Software libraries, frameworks, and tools are updated and improved daily. Furthermore, many new ones with new features are developed from scratch to compete with the existing ones in one way or another. As a result, using a framework for developing a part of the software at hand may not be good choice after sometime [39]. Therefore, software developers should always follow and utilize new software innovations to make their software development up-to-date. However, developers should not expect that all new libraries and frameworks are good in quality and have big improvements and features compared to the existing ones. They are recommended to perform a deep analysis before deciding to use a new library or framework. Generally, utilizing new innovations in the field of software will result in significant improvements in the functionality, maintainability, and productivity of software code.

4.3 After writing software

The best practices and recommendations that have to be taken into consideration after writing software are presented in the following subsections.

4.3.1 Review code

Code reviewing is an important process to decrease the number of errors/bugs, detect undesired coding styles, and increase code quality [40]. Often, this process is conducted by a senior software developer who has a wide experience in reviewing code. One of the main advantages of involving a senior developer in the review process is his/her ability to identify errors, defects, or mistakes faster with more accuracy which saves a lot of valuable time. However, code reviews can also be conducted by a team of reviewers each having a unique reviewing task. Involving a team in the reviewing process has its own advantages [41]. For example, inexperienced reviewers or newbie reviewers can join the team not only to gain knowledge and understand how the reviewing process is conducted but also to learn how to code with a high quality by analyzing other's code. In software engineering perspective, code reviewing is one of the must-to-do processes to produce a code with quality. Therefore, it is recommended to apply reviewing on any code after its completion or after each functional development phase.

4.3.2 Test code

Testing is one of the most important processes that ensure a piece of code is without defects or undesired functionality. It is important to ensure that any code should pass a set of tests after each development, maintenance, integration, or reuse processes. Also, code should be tested on different machines using different configurations. On top of that, extreme and thorough test scenarios have to be used for testing code. Generally, testing can be divided into three types depending on how the test is performed [42], as follows:

- **White-box testing:** It is used to test the internal structure of a given code. It tests how code statements have been written, how classes and objects have been declared and used, tests the relationships between functional units, and many others.
- **Black-box testing:** It is used to test only the functionality of a given code without going through its internal structure and how it has been written and developed. In other words, it tests only the input and output of code without considering the technical details.
- **Gray-box testing:** It combines the two aforementioned testing types.

These types of testing are very useful to test many aspects of code including its functionality. Moreover, they ensure that a given code has an acceptable level of performance and good quality. Nowadays, many tools and frameworks are available in many programming languages to support developers in this context. For example, developers can use Apache JMeter application [43] to measure the performance and functionality of their software. Also, it supports testing a variety number of software including web, database, and network applications/services. Another well-known testing framework is JUnit [44]. It is used widely to test the functionality of software code. JUnit is written in Java programming language but there are many JUnit-inspired testing frameworks to support other programming languages such as CPPUnit for C++, NUnit for C#, and PHPUnit for Personal Home Page (PHP).

4.3.3 Maintain code

Code maintenance usually performed after the testing phase or after the real installation and deployment of software [45]. Different types of maintenance are required for any software in the following cases:

- **Corrective maintenance:** To correct software defects, errors, bugs, etc. found in the testing phase or in the operating phase.
- **Perfective maintenance:** To enhance the existing software code via adding new features or functions. Thus, to extend its original functional requirements.
- **Adaptive maintenance:** To make changes to the existing software to adapt to changes in its external environment such as CPU, RAM, or operating system.
- **Preventive maintenance:** To make changes to the existing software so it can be corrected, enhanced, or adapted more easily.

One of the main factors that have an obvious effect on code maintenance is code simplicity. If a software code is written in a simple way, developers will rapidly find the place to make the required modification [46]. However, it is not recommended to perform one massive maintenance process in one time on the code. As often it affects badly and in many directions on code quality and functionality. Also, massive maintenance failures are not manageable and even can lead to bad decisions including cancelling the maintenance. On the other hand, multiple simple maintenance processes in different times help

developers to measure the effects of each maintenance process on the code and to handle failures (if any) of each process in an easy and manageable way.

4.3.4 Create a user manual

The good software is always delivered with a user manual (also called a user guide) to support its users while using it. Writing user manuals is not like commenting on the code. Code comments are only used by developers to understand a given code whereas manuals are mainly for non-technical persons including users [47]. The main advantages of software user manuals are:

- They are used to describe the software functionality.
- They are used to explain how to use the software properly.
- They are used to compare the functionality of two or more software products without knowing their internal technical intricacies or even without executing them.

Generally, any manual includes a number of sections; the most common ones are listed below:

- **Title section:** Includes software name, software version, developers' names, copyright, and manual created date.
- **Contents section:** Contains information on how to navigate the manual.
- **Introduction section:** Explains the functionality of software and gives an overview of the manual.
- **Requirements section:** Provides a list of software and hardware requirements that enable the software to deliver its functionality properly.
- **Frequently Asked Questions (FAQ) section:** Includes a list of questions that are commonly asked in the context of software and their answers.
- **Installation section:** Provides a set of instructions to help users properly installing the software.
- **Uninstallation section:** Provides a set of instructions to help users properly uninstalling the software.
- **Using section:** Includes a guide on how to interact and use the software.
- **Troubleshooting section:** Presents a list of possible errors or problems that may occur while using the software, along with how to fix them.
- **Contacts section:** Gives details on how or where to find the technical help and contact details.
- **Glossary section:** Provides a list of terms that are either newly introduced, uncommon, or specialized with their definitions.

It is worth mentioning that user manuals should be created in a clear, simple, and understandable way without diving into the software technical aspects as they are intended to be used mainly by non-technical users. There are a number of tools that help creating manuals automatically. For example, Dr.Explain [48] can be used in this respect to create manuals in different file formats including HTML, Rich Text Format (RTF), or Portable Document Format (PDF). It works automatically by analyzing software's user interface, taking screenshots of all its graphical controls and elements, and then allowing developers to smoothly add desired descriptions to the created manuals.

4.3.5 Make searchable code

As we are living in the world of open-source software, any software code that is discovered and accessed easily in the digital world is considered good code. Only the search for software code makes them usable and more valuable in our practical life. Therefore, developers should make their code (in case of open-source development) searchable and accessible using suitable metadata to describe their code [49]. Often, such data are related to the code file such as name, size, extension, and attributes. Besides, developers can provide other data about code files such as the number of used classes, objects, methods, variables, and comments. All these information can help other developers to search published code files and reuse them for example. Currently, many code search engines and tools are available to help in this respect as mentioned in subsection 4.2.4.

5. CONCLUSION

In this paper, we provided the taxonomy of the most important best practices, recommendations, and factors that software developers should follow and take into account to produce good software. Some of the influential factors target the technical aspects of any software development process. Many others, target the non-technical aspects in terms of the effects of working environments on the productivity and quality. Generally, everything given in this paper helps developers in one way or another to write good software regardless of the programming language used in the development and regardless of the software domain whether for

desktop, mobile, etc. Besides, this paper helps developers to measure the goodness of given software alongside many directions.

REFERENCES

[1] R. Selby, "Being a Software Engineer in the Software Century", Wiley-IEEE Press, pp. 797-806, 2007.

[2] L. Baresi, E. Di Nitto, and C. Ghezi, "Toward open-world software: Issues and challenges", Computer, vol. 39, no. 10, pp. 36-43, 2006.

[3] R. Kitchin and M. Dodge, The Nature of Software, 1st edition of Code/Space: Software and Everyday Life, MIT Press, 2011.

[4] T. F. Kusumasari, I. Supriana, K. Surendro, and H. Sastramihardja, "Collaboration Model of Software Development", International Conference on Electrical Engineering and Informatics (ICEEI), pp. 1-6, 2011.

[5] S. Bansal and A. Goyal, "An Innovative Research on Software Development Life Cycle Model", International Journal of Innovation Research in Computer and Communication Engineering (IJIRCCE), vol. 3, no. 11, pp. 10445-10451, 2015.

[6] M. Grottke, L. M. Karg, and A. Beckhaus, "Team Factors and Failure Processing Efficiency: An Exploratory Study of Closed and Open Source Software Development", IEEE 34th Annual Computer Software and Applications Conference, pp. 188-197, 2010.

[7] S. Basri and R. V. O'Connor, "Software Development Team Dynamics in SPI: A VSE Context", The 19th Asia-Pacific Software Engineering Conference, vol. 2, pp. 1-8, 2012.

[8] Live Coding Television, https://www.livecoding.tv/, accessed 01/08/2018.

[9] M. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The Impact of Social Media on Software Engineering Practices and Tools", The FSE/SDP workshop on Future of Software Engineering Research (FoSER), pp. 359-364, 2000.

[10] S. Kishore and R. Naik, Software Requirements and Estimation, Eighth reprint 2007, Tata Mc Graw Hill, 2001.

[11] H. Sarker, F. Faruque, U. Hossen, and A. Rahman, "A Survey of Software Development Process Models in Software Engineering", International Journal of Software Engineering and Its Applications, vol. 9, no. 11, pp. 55-70, 2015.

[12] R. Land, "A Brief Survey of Software Architecture", Mälardalen Real-Time Research Center (MRTC) Report, Mälardalen University, Sweden, 2002.

[13] C. STRIMBEI, O. DOSPINESCU, R.-M. STRAINU, and A. NISTOR, "Software Architectures - Present and Visions", Informatica Economica, vol. 19, no. 4, pp. 13-27, 2015.

[14] R. Terra, M. T. Valente, and R. S. Bigonha, "An approach for extracting modules from monolithic software architectures", IX Workshop de Manutenção de Software Moderna (WMSWM), pp. 1-18, 2012.

[15] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture", International Journal of Open Information Technologies, vol. 2, no. 9, pp. 24-27, 2014.

[16] C. Wohlin, "Is prior knowledge of a programming language important for software quality?", Proceedings of International Symposium on Empirical Software Engineering, pp. 27-34, 2002.

[17] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with Jolie", Web Services Foundations, pp. 81-107, 2014.

[18] M. Ben-Ari, Understanding Programming Languages, John Wiley and Sons, 1st edition, 2007.

[19] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini, "IDE 2.0: Collective Intelligence in Software Development", The FSE/SDP workshop on Future of Software Engineering Research (FoSER), 2010.

[20] NetBeans IDE, https://netbeans.org/, accessed 01/08/2018.

[21] W. O. Galitz, The Essential Guide to An User Interface Design, John Wiley and Sons, 2nd edition, 2002.

[22] Ş. Çetiner, "Effects of Office Environment on Software Developer's Productivity", 2012, http://scstrider.blogspot.com/

[23] K. Henney, 97 Things Every Programmer Should Know, 1st edition, O'Reilly, 2010.

[24] Apache Software Foundation, Apache Log4j v. 2.5 User's Guide, 2015.

[25] P. Goodliffe, Code Craft: The Practice of Writing Excellent Code, 1st edition, No Starch Press, 2006.

[26] Google Java Style Documentation, http://checkstyle.sourceforge.net/reports/google-java-style.html, accessed 01/08/2018.

[27] Code Conventions Documentation, http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html, accessed 01/08/2018.

[28] Java Coding Standard Checker Tool, http://jcsc.sourceforge.net/, accessed 01/08/2018.

[29] Checkstyle Tool, http://checkstyle.sourceforge.net/, accessed 01/08/2018.

[30] Y. Li and H. Yang, "Simplicity: A key engineering concept for program understanding", IEEE Workshop on Program Comprehension, pp. 98-107, 2001.

[31] R. Keswani, S. Joshi, and A. Jatain, "Software Reuse in Practice", The 4th International Conference on Advanced Computing and Communication Technologies, pp. 159-162, 2014.

[32] Source Code Search Engine, https://searchcode.com/, accessed 01/08/2018.

[33] H. Zhu, "From OOP to SOP: What Improved?", IEEE International Conference on Service Operations and Logistics, and Informatics, pp. 956-961, 2005.

[34] N. M. A. Munassar and A. Govardhan, "Comparison study between traditional and object-oriented approaches to develop all projects in software engineering", The 5th Malaysian Conference in Software Engineering (MySEC), pp. 48-54, 2011.

[35] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication", The 11th Working Conference on Reverse Engineering, pp. 100-109, 2004.

[36] J. L. Freitas, D. da Cruz, and P. R. Henriques, "A Comment Analysis Approach for Program Comprehension", The 35th Annual IEEE Software Engineering Workshop, pp. 11-20, 2012.

[37] Javadoc Tool, http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html, accessed 01/08/2018.

[38] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures", IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 864-878, 2009.

[39] M. W. Godfrey and D. M. German, "The Past, Present, and Future of Software Evolution", Frontiers of Software Maintenance, pp. 129-138, 2008.

[40] M. Bernhart and T. Grechenig, "On the understanding of programs with continuous code reviews", IEEE International Conference on Program Comprehension, pp. 192-198, 2013.

[41] Bac chelli and C. Bird, "Expectations, outcomes, and challenges of modern code review", International Conference on Software Engineering, pp. 712-721, 2013.

[42] S. H. Trivedi, "Software Testing Techniques", International Journal of Advanced Research in Computer Science and Software Engineering, vol. 2, no. 10, pp. 433-439, 2012.

[43] S. Shenoy, N. A. A. Bakar, and R. Swamy, "An adaptive framework for web services testing automation using JMeter", IEEE 7th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 314-318, 2014.

[44] M. Wahid and A. Almalaise, "JUnit framework: An Interactive Approach for Basic Unit Testing Learning in Software Engineering", The 3rd International Congress on Engineering Education (ICEED), pp. 159-164, 2011.

[45] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap", The International Conference on The future of Software engineering (ICSE), pp. 73-87, 2000.

[46] C. F. Kemerer, "Software Complexity and Software Maintenance: A Survey of Empirical Research", Annals of Software Engineering, vol. 1, no. 1, pp. 1-22, 1995.

[47] Y. Murakami and Y. Hori, "Automatic Generation of Usage Manuals for Open-Source Software", The 6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), pp. 671-676, 2012.

[48] Dr.Explain Tool, http://www.drexplain.com/, accessed 01/08/2018.

[49] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere, "Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts", The 18th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 242-245, 2013.

[50] Q. I. Sarhan, "Web Applications and Web Services: A Comparative Study", Science Journal of University of Zakho (JUOZ), vol. 6, no. 1, pp. 35-41, 2018.

[51] Ron Jeffries, The Nature of Software Development: Keep It Simple, Make It Valuable, Build It Piece by Piece, 1st Edition, Pragmatic Bookshelf, 2015.

[52] Eagle Mode Zoomable User Interface, http://eaglemode.sourceforge.net/, accessed 01/08/2018.

[53] Cortana Voice User Interface, https://www.microsoft.com/en-in/windows/cortana, accessed 17/02/2019.

[54] Amazon Alexa Voice User Interface, https://www.amazon.com/b?node=17934671011, accessed 17/02/2019.

[55] Apple Siri Voice User Interface, https://www.apple.com/siri/, accessed 17/02/2019.