Available online at www.worldscientificnews.com

World Scientific News

An International Scientific Journal

WSN 122 (2019) 56-70 EISSN 2392-2192

A Brief Overview of Software Reuse and Metrics in Software Engineering

Agbotiname Lucky Imoize[1,a], Damilola Idowu[2,b], Timilehin Bolaji[2,c]

[1]Department of Electrical and Electronics Engineering, University of Lagos, Akoka Lagos, Nigeria

[2]Department of Computer Engineering, University of Lagos, Akoka Lagos, Nigeria

[a-c]E-mail addres: aimoize@unilag.edu.ng , idamilola52@yahoo.com , bolajitimilehin@yahoo.com

ABSTRACT

This paper focuses on the importance of software reuse and metrics in software engineering. Software reuse is the process of reusing parts or all of an existing software system, architecture, or methodology to develop new software, potentially with different functionality. Software reuse metrics involve the different methods of ensuring that the reuse components are of the right quality, and reuse metrics serve to expose and predict the defects in the reusable process. In this paper, we introduced the concept of software reuse while exploring its merits and demerits, the types, and the impact of software reuse. From a survey outlook, we observed that high cost incurred whilst troubleshooting a reuse component and the difficulty integrating with development tools are some of the many potential problems associated with software reuse. In addition, we discussed software reuse metrics, the need for metrics, and their importance to software reuse (and its potential problems and solutions), and stakeholders in the software engineering process. Some of the issues discussed are case studies involving common software packages, object-oriented programming and its relationship to reuse, and the types of reuse metrics. Having presented an expository account of software reuse and metrics, this paper then provides recommendations for real world applications and future research.

***Keywords*:** Software Metrics, Software Reuse, Class, Object-Oriented, Software Engineering

(Received 03 February 2019; Accepted 20 February 2019; Date of Publication 21 February 2019)

1. INTRODUCTION

Software reuse and metrics are key components in the field of software engineering [1-3]. A metric is a quantitative indicator or benchmark of an attribute, which is a feature or property of an entity. Software reuse is simply the act of using existing software resources and knowledge, such as code, designs, and whole components to create a new software product. Implementation of software reuse influences the extent or degree to which a new software product is "original", where it tries to maximize the reuse of an existing software resource [4-6]. Software metrics thus refers to the extent or degree to which a component, process, module or system possesses a given attribute. They provide means for the quantitative measurement and validation of the different models used in software development life cycle. Software development life cycle (SDLC) defines the tasks performed at each stage of software development. The stages in SDLC are requirement gathering and analysis, design, development, testing, and maintenance. A process is synonymous to SDLC as it imposes a framework on the development of the software[1]. For example, in the waterfall process model, developers follow the SDLC steps in order but the spiral model is an improvement to this model because it is a risk driven approach and iterative at the same time [4].

Proper measurement of software metrics is essential to developing models for the software development process. It makes it possible to set expectations for the stakeholders, track and measure the success/failure of the project. The metrics and models developed are also invaluable tools used to estimate cost, measure productivity, and product quality. Therefore, this saves software development time and resources [5, 7]. This study aims to cover the basics of software reuse, the relevance and manifestations of software reuse in the real world and the importance of software reuse to the Object-Oriented Programming paradigm.

1. 1. Problem Statement

Software reuse makes possible to create software from already existing resources – by not "reinventing the wheel". There exists a need, however, to create material that can provide important, relevant, and relatable information, to aid the understanding of software reuse and metrics. The expansive nature of software reuse, as a discipline warrants the existence of a lot of material, which can prove cumbersome to navigate. This manifest as a bottleneck for engineers, students, and researchers, who may need to, in a timely manner, obtain knowledge about the topic. In addition, the relation of software reuse to metrics appears to be an under-researched area. Lack of proper documentation on this relationship and its application to the real world has also been a key challenge. Therefore, a dearth of information exists through which one can explore and learn more about the relationship.

2. LITERATURE REVIEW

A handful of works on software reuse and metrics exists. This section presents a careful review of these related works. It is noteworthy that although different concepts and methodologies were proposed, the level of implementation differs greatly. Frakes and Terry

[1] SDLC defines the phases a software project goes through while a software process defines the specific steps for software production used by a person or an organization.

[4], in their work on software reuse metrics and models, surveyed metrics and models of software reuse and reusability. They provided a classification structure to help users select metrics, noting the need for organizations to identify the best reuse strategies and obtain data on the progress made. Like [4], Somerville [6] also explores the topic of software reuse, giving an introduction to reuse, and expatiating on the benefits, problems of software reuse, the ways through which software reuse is implemented, and introduced the reader to ways in which quick system development can be achieved.

Juan *et al.*, [1] reported the importance of measuring the reuse capability of a test case and emphasized how important it is to modern researches in the area of software test reuse. The authors applied the Bayesian network approach to measure the reuse of test case, and found that the reuse of test case could be divided into three aspects depending on the reuse preference that improve the accuracy of metric. Lee [8] identified the various types of knowledge that are often required at every stage of the software development life cycle. He argued that some knowledge could get lost especially when it is not well documented in the software artifact. However, it was noted that the knowledge encoded in a software artifact as software concepts could find useful applications in software reuse. The author proposed a model, which has similar features like CASE tools to depict the various dimensions of software reuse and demonstrated the degree of knowledge reuse.

Mills [7] however introduced the most commonly used software metrics and reviewed their use in constructing models of the software development process. He noted that some metrics (such as the number of defects, lines of code etc.) are observable and that effective use of software metrics can avert software crisis – difficulty writing proficient and useful software in a given time. In another study on software reuse metrics, Suri and Garg [9] focused on enumerating the various metrics of software for evaluating module reusability. They then proceeded to introduce a new metric for the evaluation of software component independence, which indicates the degree of reusability.

Ravichandran [10] proposed a way to assess module reusability in software. The author presented the relevance and merits of software reuse, including the concept of reusability, giving examples of favorable situations for reuse, and software development phases where reuse comes into play. Antovski and Imeri [11], in a review of software reuse processes, presented the basic principles of software reuse, the driving factors that facilitate reuse, and the potential benefits. Furthermore, the authors focused on the key issues one needs to take into consideration, for the successful adaptation of software reuse. In a related study, Mahmood [12], using Frakes [4] as a basis, reviewed the metrics and models of software reusability and provided a classification structure to help users select them. Mahmood concluded that accurate project scoping is an essential element of metrics and modeling, and techniques of reuse and reusability metrics and models must change as the field changes.

3. METHODOLOGY

Software reuse is a highly documented and well-researched topic. Since the early days of software engineering, software reuse has carved a niche for itself as a professional practice. It, therefore, follows that in carrying out this study, we review multiple works on software reuse and metrics, in order to build upon the wealth of knowledge made available by previous authors. The works reviewed came from many sources, such as textbooks and peer-reviewed

publications. Due to the evident expertise of the authors of the source materials and their understanding of the problem area, the said sources thus served as a reliable trove of information on the topic of software reuse and metrics. In order to verify the integrity of the data sources, it was necessary to review the material, investigate the quality of the material and research that they were based on. Additional research on the reliability of the authors in question gave greater assurance on the quality of work on offer.

4. SOFTWARE REUSE FUNDAMENTALS

It has been noted that software reuse, in simple terms, is an act involving the recycling of components and entities in the process of software development. Different entities, at different levels of development, are reusable, ranging from design patterns to functions/procedures in the source code of a software product. Reuse can be found everywhere in modern software development, especially with the advent of Object-Oriented Programming.

4. 1. Benefits of Software Reuse

Software reuse aids in reducing the cost of projects, by saving the trouble of developing new components or concepts for use in software. There is also less risk in software development processes due to software reuse. This is because the behaviour of reused software components are already known and understood, leading to the development of more reliable software. In the case of a fault, it is marginally easier to trace the issue, as reused components can be isolated, due to their already tested nature[2].

Software reuse allows the expertise of the software engineer to be used more efficiently; instead of doing similar tasks repeatedly, expert engineers are able to focus on developing singular, higher quality, reusable components [13]. This obviously increases overall software quality and even, employee satisfaction.

Software reuse also increases the speed of completion of a software project. This ensures fast completion of pending projects, sometimes ahead of schedule and then released to the market. Software reuse also makes it easier for the software engineer to develop software modules and components in accordance with defined rules and standards. This is possible, as it is likely that reused components and concepts are already in conformity with defined industry guidelines. In addition, reused components tend to have defined interfaces for communication and integration. These defined interfaces make it possible, that when the components are integrated, the engineer is compelled to abide by guidelines needed to integrate the component properly such that everything works together.

4. 2. Problems created by Software Reuse

With all the merits and benefits that software reuse brings to the development process, it is pertinent to note that it comes with its tradeoffs. Reuse in development has its pitfalls, which could make the decision to reuse software, less straightforward. The following paragraphs discuss some of the potential problems associated with software reuse and show why software metrics is required. Although reuse may help to reduce the original development costs, there is a cost trade off when it comes to maintenance. This is especially the case when one does not

[2] It is obvious from this that the reduction of cost also translates into a reduction of development risk.

have all the information (documentation, data) about a reused component. That is, it is much harder and thus costlier to troubleshoot a reused component, or even improve the functionality of the software. Sometimes it is impossible, especially in the case of non-open-source libraries. Additionally, there are tools used to carry out software development. However, there exists the issue of support of reused components with the development tools. Usually, these tools integrate well with the software under development, but reuse can throw a spanner in the works, by making it more complicated to integrate properly with the tools [14-16]. This hampers productivity and therefore, can constitute a nuisance.

There is a great deal of human input in software engineering. This obviously means that certain human-specific factors come into play. There exists the issue of trust and the hunger for a challenge. That is, engineers may have issues working with reused components, as they may not trust the functionality, or may want the challenge of creating their own solution [17].

Reuse, in many cases, can be tasking and expensive to set up. In large organizations, for example, to reuse components, there is a need to create libraries and populate them with adequate elements for engineers to take advantage of them. Creating these libraries can be expensive, however, especially considering that such libraries be tailored to make it usable.

Finally, to be able to reuse software components to great effect, one needs to understand properly the intricacies of their functionality and behavior. However, this tends to be a difficult task reason being that not all reusable components have proper documentation attached to them. That is, there is sometimes a lack of information about components, so engineers need to make a concerted effort to examine the component, to be able to make proper use of the component. In some cases, software reuse is impossible, and the project could be adversely affected due to a lack of proper documentation about a component.

5. REUSE IN THE REAL WORLD

Software reuse manifests in today's world in many ways. Reuse can be seen in the adoption of code libraries which perform specific predefined operations in software, in operating systems, code modules, "include files", software updates, etc.

In the C++ programming language, to perform an Input or Output (I/O) operation, a programmer needs to "include" the *"iostream"* header file. This header file is a library containing function declarations and definitions, which are necessary to be able to perform I/O operations. Effectively, "including" this header file in the program being written, adds the code from the file to the current program. That is, the code in the header file are “reused” in the new program. The same holds for other operations, and there is a huge number of standard header files that are available for use in the C++ programming language. It is also possible for a programmer to define custom header files for use, and thus, reuse the code he has written in the header files. These programmer-defined header files make possible class declarations, under the OOP paradigm, as addressed in the next section [18].

In the Microsoft Windows operating system, for example, a user can observe that certain components barely change across operating system versions. An example is the Windows "Control Panel" which when examined closely, can be seen to have undergone few changes, between the Windows 7 and Windows 10 releases. This is a very common occurrence across versions of the Windows operating system. In this case, it holds true that components of a particular version of an operating system are “reused” in the next version.

A final case study is evident in the case of the Open Graphics Library (OpenGL). OpenGL is a library that renders two- and three-dimensional graphics. It finds use by applications for various purposes, in different fields, to achieve functions associated with graphic rendering. It serves as an example of software reuse, as its inbuilt functions are set up to achieve hardware-accelerated rendering with little additional effort. OpenGL has an extensive library of functions that greatly simplify graphics rendering and processing, and without this manifestation of software reuse, software that required graphics rendering would have to implement the hardware rendering individually. This would not only yield suboptimal results but also come at a greater cost to the development company [19, 20].

From the above case studies, it is clear that reuse is especially important to the production and continuous improvement of software. This is especially important to software which remains relevant for a long period, and which requires continual improvements to keep up with changing times.

6. OBJECT-ORIENTED PROGRAMMING AND REUSE

Object-Oriented Programming (OOP) is a software-programming paradigm, centered on the use of "objects", wherein objects are constructs containing data (fields or attributes) and procedures (methods). Objects denote logical units sometimes used to model real-world constructs. It is possible to combine them in multiple ways to model portions of processes, or even complete processes. Objects, when seen as blocks, combine in various ways to form larger, more complicated functional constructs. Object-Oriented Programming is one of the most common forms of formalized software reuse [21-23].

Usually, in OOP, a template is the basis for the nature and characteristics of an object, so to speak, which in turn defines the attributes and methods associated with any object created based on that "template". This template, in most OOP languages, is termed a class. Therefore, a class is the blueprint for creating unique objects or instances of said class. The class paradigm is an especially elegant form of software reuse, wherein unique constructs though created based on a predefined template, require a minimal amount of additional code written. In some cases, the code, wherein the object instance’s uniqueness resides, is termed a constructor.

One of the fundamental properties of OOP is Inheritance, which is yet another manifestation of software reuse in OOP. With inheritance, previously defined (base) classes serve as the basis to create new (sub) classes. The subclasses **inherit** (reuse) the attributes and methods from their parent classes, and can then have their own unique attributes and methods specifically defined to distinguish them from their parent classes.

Figure 1 illustrates the concept of software reuse in OOP, by using the C++ programming language to model mobile phones. This is a header file for the definition of a base class (Mobile) to be “reused” by other sub-classes (Nokia and Samsung). Figure 2 shows the main source file as well as the console output. Figures 3 and 4 show the Python language equivalent.

All Samsung models are mobile phones, but of course, not all phones are Samsung. That is, there are some attributes common to all mobile phones, and some unique to Samsung phones. Thus, the “Mobile” class contains definitions for some properties and functions that are common to all mobile phones e.g. the IMEI number, and the dialing functions. To specify the unique characteristics for various phone brands, it becomes pertinent to create another class.

It is possible to define a completely unaffiliated class for Samsung phones, and another for Nokia phones, but due to the manifestation of software reuse, the code from the general "Mobile" class is "reused", through the inheritance feature available in the C++ language, with extra attributes being defined in the Samsung/Nokia subclass. That is, the Nokia and Samsung subclasses inherit (reuse) the functions and data in the Mobile class.

```
#ifndef __MOBILE_PHONES_
#define __MOBILE_PHONES_
#include <iostream>
using namespace std;

//base class
class Mobile
{
protected:
    string IMEI;
public:
    Mobile();
    string SIMCard;
    string Processor;
    int internalMemory;
    bool isSingleSIM;
    void PrintIMEICode();
    void Dial(int);
    void Recieve();
    virtual void sendMessage(string);
    ~Mobile();
};

//derived classes
class Nokia : public Mobile
{//contains members inherited from Mobile class
public:
    //properties
    int RamSize;
    string Processor;
    //constructor
    Nokia();
}

class Samsung : public Mobile
{//also contains members inherited from Mobile class
public:
    //additional properties
    bool hasStylus;
    float WifiBand;
    //additional method
    void FingerprintUnlock();
};
#endif
```

Figure 1. A demonstration of Software Reuse in OOP: C++ header file

```
#include "MobilePhones.h"

int main()
{
    Nokia _3310;
    Nokia N95;
    Samsung GalaxyJ1;

    _3310.Dial(912344567);
    N95.PrintIMEICode();
    GalaxyJ1.hasStylus = false;
}
```

```
C:\Windows\system32\cmd.exe
Dialing... 912344567
IMEI - IEDF3454656768
Press any key to continue . . .
```

Figure 2. A demonstration of Reuse in OOP: C++ source file and console output

```
#base class
class Mobile:
    IMEI = ""                               #properties
    SimCard = ""
    Processor = ""
    internalMemory = 0
    isSingleSIM = False
    def __init__(self):                     #methods
        #constructor
        self.IMEI = "AAA00000000000"        #default
    def isSingleSim(self):
        return isSingleSIM
    def PrintIMEICode(self):
        print(self.IMEI)
    def Dial(self, num):
        print("Dialling... ", num)
    def Recieve(self):
        print("Recieve a call")
    def sendMessage(self, msg, num):
        #definition
        print("Message sent!")
#derived classes
#some members are inherited from the base (Mobile) class.
#Ditto for Samsung
class Nokia(Mobile):
    RamSize = 4
    Processor = "ARM"
    #constructor overrides that of base class
    def __init__(self):
        self.IMEI = "F3GDA34354665"
        print ("Nokia object created")
class Samsung(Mobile):
    hasStylus = False
    WifiBand = 5.0
    def FingerprintUnlock(self):
        print("Processing...")
def main():
    n1 = Nokia()                        #function within class
    s1 = Samsung()
    n1.Dial(912454546)                  #function in base class
    s1.FingerprintUnlock()              #function within class
    s1.sendMessage("Hello Word", 912345678) # "
    n1.PrintIMEICode()                  #prints unique IMEI code
main()
```

Figure 3. A demonstration of Software Reuse in OO: Python script

```
>>> ================================ RESTART ================================
>>>
Nokia object created
Dialling...  912454546
Processing...
Message sent!
F3GDA34354665
```

Figure 4. A demonstration of Software Reuse in OOP: Python shell output

7. SOFTWARE REUSE METRICS

A metric is a system or standard of measurement. More specifically, a metric is simply a qualitative indicator of the attributes of an element. Software reuse metrics thus simply involve the continuous application of measurement-based techniques to improve software reuse processes [24, 25].

1) Why do we need metrics in software reuse?

i. To provide information on the quality of the software component being "reused" by predicting defects in the software reuse process.
ii. To ensure proper scheduling of an ongoing project. Metrics can ensure that developers carry out proper documentation of reused modules, functions etc. This is the case because improper documentation of software components will hamper the measurement processes.
iii. It shows maintenance, research, and costs associated with a reuse process. This is especially relevant
iv. It helps in making decisions about the improvement needed for a particular reuse process.

2) Goals of Reuse Metrics

i. *To provide realistic measures of reuse:* In large organizations (that adopts a systematic software reuse), information about software reuse degrees and nature in the organization is usually required, for strategic planning. This information is obtained through measures supplied by reuse metrics
ii. *To estimate the benefits of reuse:* In many fields, the norm is that there must be justification for every action or activity taken. That is, one must know the positive impact of software reuse, in order to justify its continued application. Software reuse metrics provide engineers and management with a way to ascertain how beneficial software reuse as an approach is and the knowledge of this serves to justify the continuous adoption of reuse.
iii. *To provide feedback to stakeholders:* Stakeholders, with varying levels of interest in the development venture, have one thing in common, they need information on progress. The information that stakeholders need varies, however, depending on their role. For example, financial stakeholders would require information on the financial impact of software reuse, while technical staff, could require information about the impact of software reuse on development efficiency.

iv. *To give simple easy to understand data:* Having established that metrics provide information through measurements; the information supplied by metrics also needs interpretation and comprehension. Therefore, it is necessary that metrics are developed and tailored such that they produce easily interpretable and understandable values, for the use of stakeholders.
v. *To encourage software reuse:* Noting that software reuse metrics help to provide information about the benefits of reuse, and hence justify reuse, it is clear that properly developed software reuse metrics will encourage software reuse.

3) Types of Software Reuse Metrics

There are different metrics for improving the quality and productivity of software reuse projects. These metrics arose due to the level of complexities in the software reuse process. This paper will divide the numerous metrics into six distinct types and they are highlighted [4]:

i. Maturity Assessment
ii. Reuse Cost Benefits
iii. Amount of Reuse
iv. Reuse Library Metric
v. Failure Mode
vi. Reusability Assessment

a) Reuse Cost Benefit Analysis

This involves the process of weighing up the benefits of using a particular reuse process and comparing it with the cost associated with the usage of that process. The cost of a reuse process is dependent on how large or small the application domain is. In a large application domain that lacks clearly defined reuse requirements, a small application domain with fewer reuse components will be more cost effective. This is so because a mistaken requirement for reuse by an individual that would necessitate a team of software developers would greatly increase the cost of the reuse process. An advantage of this reuse metric is that it assigns a monetary value to both cost and benefit.

b) Maturity Assessment

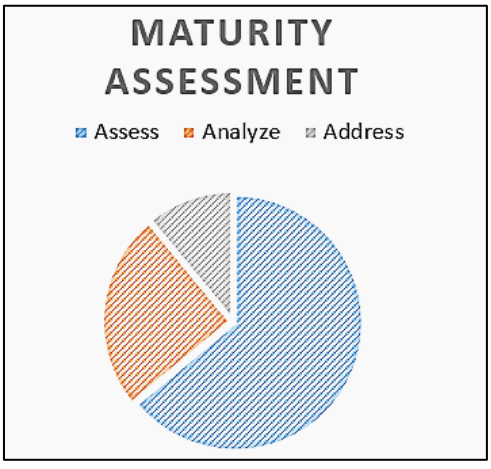

Figure 5. The Maturity Assessment metric

This is a self-evaluation tool for checking where a project stands and where it needs to be so that one can attain set goals for the project. Maturity Assessment involves three key areas as depicted in Figure 5.

c) Amount of Reuse

The amount of reuse metric is a fundamental tool for assessing and monitoring a reuse improvement effort by tracking percentages of reuse of software components over time [12].

It shows the amount or percentage of data that require reuse in order to make software more efficient. The number of lines of code is an example of quantities used to represent the amount of data that is "reused" in this metric. Equations (1) and (2) show some of the important sub categories.

- **Reuse Level**

$$R = \frac{ALOR}{T} \tag{1}$$

where:

ALOR is the amount of lifecycle objects reused

T is the total size of lifecycle object

- **Reuse Percentage**:

$$\%R = \frac{LOC}{T} \tag{2}$$

where:

LOC is the reused Lines of Code

T is the total number of Lines of Code

d) Failure Mode

This helps to evaluate the quality of a reuse program, to determine impediments to reuse in an organization, and to improve systematically the reuse strategy or model, considering technical and non-technical factors. According to Mahmood [12], the reuse failure modes are:

i. No attempt to reuse
ii. Part does not exist
iii. Part is not available
iv. Part is not found
v. Part is not understood
vi. Part is not valid
vii. Part cannot be integrated

e) Reusability Assessment

This metric module determines whether the component or artifact is reusable or not. Selby [26] identified some module attributes for assessing reusability of reuse modules, compared to others. The author suggested that modules possessing those attributes would be more reusable.

The attributes are as follows:

i. Calls per source line
ii. Fewer I/O Parameters per source line
iii. Fewer read/write statements per line
iv. Higher comment to code ratios
v. More utility function calls per source line
vi. Fewer source lines

f) Reuse Library Metric

The Reuse Library is where reusable assets are stored and properly managed. To reuse a software component, a software engineer must not only find it but also understand it [12]. Figure 6 gives an illustration of some reuse metrics. The suitability for use of a reuse library depends on the quality of its assets. That is, the quality of assets is a very important attribute of a reuse library. Frakes and Nejmeh [27] proposed some metrics as indicators of the quality of assets in a reuse library:

i. Time in use
ii. Reuse Statistics
iii. Reuse Review
iv. Complexity
v. Inspection
vi. Testing

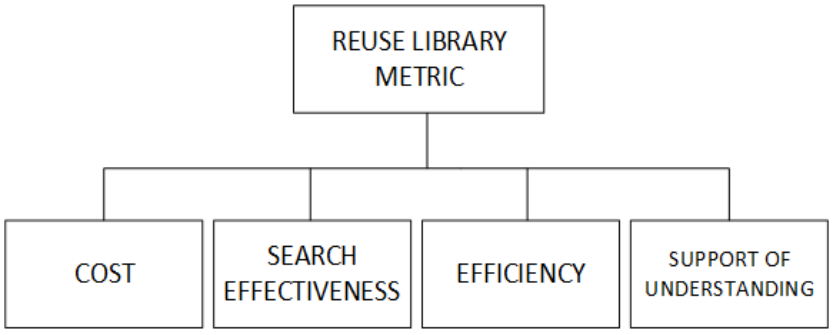

Figure 6. Reuse Library metrics

8. DISCUSSION

The overall purpose of this paper is to give an exposition about software reuse and metrics pertaining to reuse. It is seen that software reuse - as a topic - is an exceptionally wide and well-studied area of software engineering. Software reuse is especially beneficial to software engineering processes. Software reuse has a positive impact on cost, development risk, employee efficiency and satisfaction, project completion speed, fault detection and correction, and ease of compliance with standards. However, like many other concepts and practices, software reuse comes with its demerits as already outlined. In an organization that employs

software reuse, there are different stakeholders (people who have an interest in the actualization of the software engineering goals) in the development process, such as:

i. Technical stakeholders
ii. Financial stakeholders
iii. Management stakeholders

The purpose of software reuse metrics is mainly to provide these stakeholders with information about software reuse processes in an organization. Software reuse metrics aid in decision-making, and help to show factors associated with projects, such as costs. Reuse metrics enable realistic reuse measurement and reuse benefit estimation; in designing metrics, it is necessary to aim to produce easily interpretable values. There are different types of metrics, having varying levels of relevance/importance to different classes of stakeholders. That is, different stakeholders will use different metric types to ascertain the information they need, and make decisions accordingly.

9. CONCLUSIONS

This study provided an exposition of the nature and importance of software reuse and reuse metrics to software engineering processes and stakeholders. Software reuse as a practice is a major part of efficient software engineering and as such requires a careful investigation. This review found that software reuse plays an exceptionally key role in reducing cost, and boosting development efficiency and productivity in software engineering processes. Thus, software development stakeholders need harness the enormous benefits of software reuse and metrics to mitigate the potential problems inherent in software engineering by engaging in proper process documentation. Future studies could focus on developing singular indices for various stakeholders, based on the types of software metrics available. An ideal way to develop these would take into account a stakeholder's need to compute the singular relevant "reuse index" value and be able to make decisions based on the magnitude of the value. This would enable easy analysis and effective decision-making on appropriate software reuse and metrics in software engineering.

References

[1] Z. Juan, L. Cai, W. Tong, Z. Liu and Y. Li, A Dynamic Metrics Method for Test Case Reuse Based on Bayesian Network, in *IEEE International Conference on Computational Intelligence and Software Engineering*, Wuhan, China, 11-13 December 2009.

[2] M. Abdellatief, A. Sultan, A. Ghani and M. Jabar, A mapping study to investigate component-based software system metrics. *Journal of Systems and Software*, vol. 86, no. 3, pp. 587-603, 2013.

[3] D. Bombonatti, M. Goulão and A. Moreira, Synergies and tradeoffs in software reuse – a systematic mapping study. *Software: practice and experience* vol. 47, no. 7, pp. 943-957, 2017.

[4] W. Frakes and C. Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys* vol. 28, no. 2, pp. 415-435, 1996.

[5] N. Fenton and J. Bieman, Software Metrics: A Rigorous and Practical Approach, 3rd ed., Boca Raton, Florida: CRC Press, 2014.

[6] Sommerville, Software Engineering, 8th ed., Boston, Massachusetts: Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] E. E. Mills, Software Metrics: SEI Curriculum Module SEI-CM-12-1.1. Carnegie-Mellon University, Pittsburg, 1988.

[8] H. Lee, Software engineering knowledge for software reuse. *Proceedings of IEEE 6th International Workshop on Computer-Aided Software Engineering*, pp. 263-269, 19-23 July 1993.

[9] P. Suri and N. Garg. Software Reuse Metrics: Measuring components independence and its applicability in Software Reuse. *International Journal of Computer Science and Network Security*, vol. 9, no. 5, pp. 237-248, 2009.

[10] S. Ravichandran. A detailed systematic literature review on software reusability metrics for object oriented software programs. *International Journal of Information Technology & Management Information Systems* vol. 3, no. 1, pp. 33-44, 2012.

[11] L. Antovski and F. Imeri. Review of Software Reuse Processes. *International Journal of Computer Science Issues*, vol. 10, no. 6, pp. 83-88, November 2013.

[12] S. Mahmood. Conception and Experience of Software Reuse and Reusability Metrics and Models - An Academic Review. *International Journal of Engineering Technology*, vol. 3, no. 2, pp. 35-40, September 2006.

[13] P. Mohagheghi and R. Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, vol. 12, no. 5, pp. 471-516, 2007.

[14] F. O. Bjørnson and T. Dingsøyr. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Information and Software Technology*, vol. 50, no. 11, pp. 1055-1068, 2008.

[15] T. Vale, I. Crnkovic, E. De Almeida, P. Neto, Y. Cavalcanti and S. de Lemos Meira. Twenty-eight years of component-based software engineering. *Journal of Systems and Software*, vol. 111, pp. 128-148, 2016.

[16] R. Selvarani and P. Mangayarkarasi. Modeling of Reusability Estimation in Software Design with External Constraints. *In Software Project Management for Distributed Computing, Springer, Cham*, pp. 3-23, 2017.

[17] V. Alves, N. Niu, C. Alves and G. Valença. Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, vol. 52, no. 8, pp. 806-820, 2010.

[18] B. Stroustrup, The C++ Programming Language, Pearson Education India, 1995.

[19] Mahmood and A. Oxley. Reusability assessment of open source components for software product lines. *International Journal of New Computer Architectures and their Applications* vol. 1, no. 3, pp. 519-533, 2011.

[20] D. Shreiner, OpenGL reference manual: The official reference document to OpenGL, version 1.2, 3rd ed., Boston, Massachusetts: Addison-Wesley Longman Publishing Co., Inc., 1999.

[21] Ahmaro, A. Abualkishik and M. Yusoif. Software reusability: A review of the research literature. *Journal of Applied Sciences,* vol. 14, no. 20, pp. 2396-2421, 2014

[22] U. Afzal, T. Mahmood and Z. Shaikh. Intelligent software product line configurations: A literature review. *Computer Standards Interfaces,* vol. 48, pp. 30-48, 2016.

[23] Y. Ye. Supporting component-based software development with active component repository systems. University of Colorado, Colorado, 2001.

[24] N. H. Bakar, Z. M. Kasiru and N. Salleh. Faetures extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. *Journal of Systems and Software,* vol. 106, pp. 132-149, 2015.

[25] V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics,* vol. 9, no. 3, pp. 1234-1249, 2013.

[26] R. W. Selby, Quantitative studies of software reuse. *Software reusability. ACM,* vol. 2, pp. 213-233, June 1989.

[27] W. Frakes and B. A. Nejmeh. Software reuse through information retrieval. *ACM SIGIR Forum,* vol. 21, no. 1-2, pp. 30-36, 1986.