A New Technique for Understanding Large-Scale Software Systems

Thamer Al-Rousan and Hasan Abualese

Abstract — **Comprehending a huge execution trace is not a straightforward task due to the size of data to be processed. Detecting and removing utilities are useful to facilitate the understanding of software and decrease the complexity and size of the execution trace. The goal of this study is to develop a novel technique to minimize the complexity and the size of traces by detecting and removing utilities from the execution trace of object-oriented software. Two novel utility detection class metrics were suggested to decide the degree that a specific class can be counted as a utility class. Dynamic coupling analysis forms the basis for the proposed technique to address object-oriented features. The technique presented in this study has been tested by two case studies to evaluate the effectiveness of the proposed technique. The results from the case studies show the usefulness and effectiveness of our technique.**

Keywords — **Utility Classes, Software Comprehension, Dynamic Coupling Analysis, Object-Oriented Software.**

I. Introduction

MODERN software systems are characterized by being complex and large-sized. Maintenance of these systems is therefore a very difficult process. In particular, the understanding process is an essential requirement of software maintenance before making any modification [1]. Many studies have demonstrated that understanding activity consumes about two-thirds of the maintenance cost [2], [3], [4]. Hence, software understanding should be supported by several tools and techniques in order to achieve maintenance success and reduce its cost [3].

Most approaches to program comprehension can be classified into dynamic analysis and static analysis [5]. Dynamic analysis is based on analyzing the gathered information during the program execution. Alternatively, static analysis is based on analyzing the program source code without any execution. Both static and dynamic approaches have the power to make the comprehension process simpler, cheaper, and more efficient. However, static analysis techniques are not completely precise, particularly in the context of object-oriented systems [6]. Object-oriented features as late binding and polymorphism make it difficult to understand without using dynamic analysis [6], [7].

Paper received September 03, 2019; revised July 13, 2020; accepted July 17, 2020. Date of publication July 31, 2020. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miroslav Lutovac.

Thamer Al-Rousan is with the Faculty of Information Technology, Isra University, Jordan (e-mail: thamer.rousan@iu.edu.jo).

Hasan Abualese is with the Faculty of Information Technology, World Islamic Sciences and Education University, Jordan (hasan.abualese@wise.edu.jo).

Dynamic analysis, the focus of this study, revolves around the analysis of the traces generated by running the features of the program under study. In spite of the advantages of execution trace analysis however, it is frequently characterized by massive amounts of generated data that delay any feasible analysis. A large amount of trace data is not very important to the comprehension process, especially in regard to the utility components [5]. In order to extract the core functionality of execution traces without losing their main characteristics, many techniques have been proposed, the success of which depends on reducing the non-useful elements, such as utilities [8], [9]. This paper aims to propose two novel utility detection metrics to detect utility classes using dynamic coupling analysis.

This study is structured as follows: The related work is presented in Section 2. Section 3 introduces the necessary definitions for our technique. Two powerful utility class detection metrics are proposed in Section 4. A case study is presented in Section 5. Finally, we conclude this paper with a discussion of future research.

II. Related Work

Dynamic analysis means inspection of systems behavior by investigating their runtime data. Runtime data shows many aspects of the behavior of the software code, for instance, the coding flow and control flow [10].Runtime data is very useful in understanding systems functionalities by inspecting their behavior. Dynamic analysis has been selected in our study for several reasons [11]:

- Dynamic analysis supports a goal-oriented policy.
- Dynamic analysis can accurately deal with late binding and polymorphism.
- Object-oriented software has a dynamic structure that differs from a static structure.
- Dynamic analysis can go beyond the problems of infeasible paths and dead codes.

The goal-oriented policy means that just those interesting parts of the program code should be analyzed. This policy is effective in determining which parts precisely relate to the specific functionality of the program code. Polymorphism literally means a state of having many forms. Polymorphism is the feature of being able to assign a single name to present different meanings or usages in different contexts [11]. In programming contexts, different behavior can be expressed by a single name. Even though this procedure is effective in a programming context, it disturbs the understanding process because it postpones the accurate behavior of the program code in runtime. Hence, rather than taking into account all theoretical cases, a dynamic analysis should be able to determine the real cases that are executed. In object-oriented software, the dynamic structures usually form the "Spaghetti Architectures" phenomenon. The dynamic structures comprise very complex interwoven connections of relationships between the software entities. Therefore, the dynamic structures are quite different from static structures. Infeasible paths and dead codes happen to some extent in all applications. The applications contain methods and classes which are no longer in use, but that have not been eliminated from the source code. Static coupling tools will analyze all of the code under analysis, so that measurements may have become inaccurate [12].

In dynamic analysis, the event traces are collected into a file labeled as "an execution trace file" during the running of the program [13]. The size of the execution trace is a key challenge that faces the dynamic analysis technique [14]. Therefore, in order to make the dynamic analysis more efficient, it is important to decrease the size of the trace data without losing its main features. In dynamic analysis, many techniques have been proposed to face this challenge such as the sampling technique, clustering technique, visualization techniques, and the utility detection and removal technique.

The idea of the sampling technique is based on executing a number of samples from event traces instead of executing the whole trace file. The sample events are selected randomly or by employing a customized procedure. The problem with this technique is that some of the main events might have been missed using this technique [15].

In the clustering technique, the events are grouped based on specific criteria [16]. Depending on the objects under study, the events are grouped by using a number of rules in order to make the understanding of the trace easier. The outcome of this technique is a different set of clusters that may make the understanding of the trace easier.

Detection and removal of utility modules that have a bad impact on the relationships between other modules can be considered as an essential trace data reduction technique [9]. Detection and removal of utility modules will decrease the size of the trace event and make the data more favorable in terms of understanding the program. However, detecting and removing utilities is not a simple task, especially, when the maintenance phase is accomplished under many maintenance rounds.

Few studies have been designed to detect the utility modules. Nevertheless, the majority of them employ static fan-in approaches based on static graphs. The study of Meszaros [17] denotes that the utility modules are ubiquitous and these modules have a bad impact on the understanding of the relationships between the system modules. Consequently, Meszaros recommends they may be removed in order to obtain a good understanding of the software structure. Despite this, the author didn't present any metrics to express utility modules.

The QNX software engineers have developed an intuitive concept such that utilities can be packaged or grouped together in a class, a library, or in another form [8]. In fact, some utilities are usually not grouped in certain modules. For example, in most classes, the accessing methods can be counted as utilities while the classes that include them are not necessarily utilities.

The [8] study used the content selection to extract a summary of important content from a trace by eliminating unnecessary details. The important content is usually selected from a trace document by classifying the document phrases based on their importance. The importance is measured by using different techniques, for instance: word distribution, cue phrases, and the place of phrases in the trace document [18]. The study used brainstorming sessions to answer the main question which is: what constitutes the most important content of trace. The content generalization consists of allocating a high-level description to specific content; specifically, exchanging it with more abstract info. However, in this study, the author did not present any metric to express the utility modules.

Trace summarization has been defined by Hamou-Lhadj et al. [9] as an abstract description of the trace results by removing unwarranted details such as utility modules. The main goal of trace summarization is to extract a view of a trace that testers can easily work with when attempting to understand the most important information traced. The Hamou-Lhadj study combined the fan-in and fan-out techniques to define the utilities. The summarization of the trace is the initial attempt to declare a detection utility metric. Nevertheless, their detection metrics still use static fan-out and fan-in techniques which have less precision in object-oriented software.

Understanding a huge trace is not a straightforward task due to the size and complexity of the data to be processed. Not all implementation specifics will be counted as utilities. Using existing detection techniques that are dependent on static techniques will still result in less accurate detection of utilities in object-oriented systems. In the following sections, this study proposes a new technique to detect the utilities in object-oriented systems by using dynamic analysis as a guide to decide the degree that a specific class can be counted as a "utility class."

III. Execution Traces Analysis Techniques

The generated information from the execution of the program code is normally saved in an execution trace file. In recent object-oriented systems, the execution files are extremely huge. In particular, complexity and size are key factors that have negative impacts on software maintenance and software comprehension [19]. So as to simplify the execution traces and reduce the time and effort required for the software maintenance process, there is a need to use trace analysis techniques.

This study proposes a new technique for trace analysis that makes the process of comprehension easier. The new trace analysis technique is based on filtering out of the utility modules which can disturb the relations between other modules of the execution trace. Dynamic coupling measurements were used in our proposed technique as an indicator to determine which modules could be counted as utilities. The next subsections present the required definitions for our techniques.

A. *Operational Definition of Utilities*

There is no common definition of the term “utility.” The operational definition defined by Hamou-Lhadj et al. [8] was applied in this study which is: “Any modules of a software system designed for the suitability of the implementer and designer and it can be accessed from different places within certain program scope”.

Utilities are used to provide support to the functions that implement the core functionality, and they do not have a critical role in the flow control of the program. A lot of utilities are designed to be reused or invoked by several other modules. Based on the Hamou-Lhadj et al. [8] definition, a utility could be a class, package, method, or another element. It may also be accessed from more than one place. In addition, this definition permits the accessing methods to be counted as utilities, and does not need to be grouped in any way, even though it does not prevent that.

B. *Utility Detection using Dynamic Coupling*

This study aims at employing the “dynamic coupling measures” to improve the program comprehension process. Dynamic coupling measures can capture the actual coupling behaviors in an object-oriented system as they are analyzed from data generated during runtime. Dynamic coupling measures have varied classifications, depending on the context of the application in which such measures are to be applied [20]. Thus, it is significant to determine which dynamic measures can accomplish this goal. To achieve this, this study employs a common framework proposed by Arisholm [21] which is widely used in dynamic coupling measures. The Arisholm framework describes coupling according to the three criteria: Scope, Granularity, and Entity of the measurement. For the purposes of this study, we refined these criteria as:

- The entity of measurement: the study concentrates on how testers attempt to understand the software program. In this trend, the testers always select the class level rather than object-level because the entities at the class level are more recognizable in the software system, so the study will employ the classes as a measurement entity.
- Granularity: since the study selects the classes as a measurement entity, so the granularity level must be a class or its objects.
- Scope: the study considered the execution trace file as the scope of measurement. While the execution trace file is generated based on scenarios that cover one feature or more. The testers have the freedom to define their sub-scopes, so the testers can concentrate on only the interesting parts of the execution trace instead of the whole execution trace.

Dynamic coupling is intended to be measured in two forms: IC (import coupling) or EC (export coupling) depending on the direction of the method calls. Import coupling counts the messages which are sent from an object or class. Export coupling counts the messages which are received by an object or class [22]. As this study is interested in both directions (import and export coupling), to determine the utilities and level of granularity, the study adopted two metrics of the Arisholm framework.

a) Export Coupling-Distinct Class (*EC_CC*): this metric calculates the number of distinct classes to which a class **A** has received messages in a given scenario. *EC_CC* stands for Export Coupling-Class level-Distinct Classes.
b) Import Coupling-Distinct Classes (*IC_CC*): this metric calculates the number of distinct classes that methods of a given class **A** have sent messages in a given scenario.

IV. Metrics for Detecting Utility Classes

The majority of dynamic coupling metrics, such as metrics in the Arisholm framework, are used to evaluate the overall service of quality according to its coupling, only very few calculate the significance of the modules in the comprehension practice [23]. In this regard, our study proposes two detection metrics to improve the comprehension process. The first detection metric took into consideration only one direction of coupling (namely export coupling), while, the second detection metric takes into consideration the coupling in both directions (import and export coupling). In the following subsections, the study will discuss in detail these detection metrics.

A. *Detection Metric for Export Utility Classes*

The next metric was defined by this study to calculate to what extent a certain class may be counted as a utility. The metric for export utility for a given class **C** is:

$$EUC(c) = \frac{EC\_CC(c)}{Max\_EC\_CC(c)} \tag{1}$$

where: *EUC*(*c*) - Export Utility Classes metric for a given class **C**; *EC_CC*(*c*) - the number of calls between objects of class **C** and the rest of classes in the execution trace; *Max_EC_CC*(*c*) - the highest possible value for *EC_CC*(*c*) in the execution trace.

In our export metric, the measure of export coupling (*EC_CC*()) was defined for a certain class with a range between **0** and the total amount of methods in the rest of classes in the execution trace. Particularly, *EC_CC*() calculates the amount of "(m2, c2, c1) triples". The technique indicates to the highest value of the *EC_CC*() metric by *Max_ EC_CC*(). Consequently, to restrict the "peak value" to **1** the technique divides *EC_CC*() by *Max_EC_CC*(). Consequently, we can easily distinguish between utilities and non-utilities. It is important to mention that the value will be different from one class to another because the *Max_EC_CC*() does not count the local operations.

The *EUC* metric has a range from **0** to **1**. In the execution trace, the *EUC* value is equal to **0** when a certain class doesn’t deliver any services to other classes, which implies that class isn't a “utility class” as specified by the definition of utility. In contrast, when the ***EUC*** metric converges to **1**, this means that the class provides a lot of services to different classes existing in the execution trace. Consequently, based on the *EUC* metric, the class with the highest measured value most likely will be the utility class. Actually, the *EUC* metric can determine the most coupled classes, which can be considered to be a new contribution to our metric. This can help the testers to understand the significance of the classes.

It is important to mention that during the comprehension process, extracting the candidate utility class is not easy and is in fact risky. This is because different testers have a different understanding of the software functions and different concerns as of the software details [24]. Thus, what can be considered as a utility to one tester may not be considered as such to another tester. Consequently, the *EUC* metric can be used as a technique to make decisions regarding what can be considered as utilities.

B. Detection Metric for "Export-Import" Utility Classes "EIUC"

EIUC takes into consideration the coupling in both directions (export and import coupling). The reason behind adopting this metric is: there is a positive relationship between outgoing calls of a certain class and their importance to control the flow of the object-oriented system. This means that: if the amount of outgoing calls from a certain class is high, then its role in the program control flow is very important and thus there is a greater need to keep it. On the other hand, the relationship between import coupling and the utility concept is negative, this means that more outgoing calls indicate a lower probability of it being a utility class.

It is remarkable to mention that the import coupling is less important to the utility concept compared with export coupling. Thus, in our utility detection metrics, the study gives less weight to the import coupling.

In our study, in order to calculate the import coupling, we use the log function which is inspired by the famous technique called "**tf-idf weight"** [25]. **Tf-idf** is short for the term "frequency-inverse document frequency," and the **tf-idf weight** is often used in searches for information retrieval, text mining, and user modeling techniques. Therefore, The Import-Part (IP) for a certain class **C** can be expressed as follows:

$$IP(c) = \text{Log}\left(\frac{Max\_IC\_CC(c)}{IC\_CC(c)}\right) \qquad (2)$$

where: *IC_CC*(*c*) - the number of distinct classes that be utilized by methods of the object of a class **C** in the execution trace. These values must be placed in the denominator because the relationship between import coupling and the utility concept is negative; *Max_IC_CC*(*c*) - the highest possible value for *IC_CC*(*c*) in the execution trace.

If the *IC_CC* value closes to the *Max_IC_CC* value then class **C** has strong import coupling, so the $IP(c) = \text{Log}(1) = 0$. While, if class **C** has weak import coupling (that is "close to **1**"), the value of *IP*(*c*) will be inclined to equivalent to Log(*Max_IC_CC*(*c*)). To eliminate division by zero, we eliminate *IC_CC*=0.

To guarantee that the definitive consequence of *IP*(*c*) ranges amongst **0** and **1**, we divide the *IP* weight by its highest value. Thus, the **Final-Import-Part (***FIP***)** can be expressed as:

$$FIP(c) = \frac{\text{Log}\left(\frac{Max\_IC\_CC(c)}{IC\_CC(c)}\right)}{\text{Log}\left(Max\_IC\_CC(c)\right)} \qquad (3)$$

The **Export-Import utility Class** (*EIUC*) will exploit the **Export Utility Metric** (*EUC*) via multiplying it by the *FIP* value that takes into account the import coupling if it exists; else, *EIUC* will be equivalent to the *EUC* metric. Consequently, we refine *EIUC* for a certain class **C** in this way:

$$EIUC(c) = \begin{cases} EUC(c) \times FIP, & \text{if } IC - CC \neq 0 \\ EUC(c) & \text{if } IC - CC = 0 \end{cases} \qquad (4)$$

The export coupling will be dismissed if the *FIP*() value is near to **0**, which means that the import coupling in class is strong. In this circumstance, the **C** class mustn't be counted as a "utility" because the **C** class has many "control dependencies." On the other hand, when import coupling in a certain class is very weak (*FIP*() value close to **1**), then the value of its export coupling will be the only scale to assess the degree to which that **C** class will be counted as "utility class."

V. Case Studies

In this section, two case studies are presented in order to evaluate the effectiveness of the proposed work. For the former one, we used an open-source tool called **Checkstyle** [26]. **Checkstyle** is a Java-based tool that validates Java code and confirms if a Java source code employs coding standards. It contains 58,000 lines of code to represent 21 packages and 311 classes. Thus, it is counted to be a good representation of actual software systems. Furthermore, the results of this experiment can be repeated, as **Checkstyle** is an "open-source" system and its execution traces are provided in a common dataset benchmark. For the latter case study, we used **JHotDraw** [27]. **JHotDraw** is a customizable Java framework that distinguished for graphics editing. It comprises 73,000 lines of code to represent 21 packages and 344.

A. Usage Scenario

Because of that individual datasets result in several drawbacks such as the limited generalization of the evaluation results and the uncertainty of the sound of the individual datasets [28] [29]. We opted to use execution traces of **Checkstyle** and **JHotDraw** from a common dataset benchmark. The execution traces are generated from executing **Checkstyle** and **JHotDraw** to specific scenarios. For example, **Checkstyle** was executed from the command line where 64 types of checks are specified. Whilst, a new drawing was created in **JHotDraw**, then 5 distinct figures were added. Subsequently, the size of the generated **Checkstyle** execution trace has 107 classes, 1243 methods, and 31,260 calls. We opted to remove self-callings and library classes, therefore, the new execution trace has 100 classes, 798 methods, and 11,632 calls. In addition, the size of the generated **JHotDraw** execution trace has 161,087 calls after removing mouse movement events. Also, we opted to remove self-callings and library classes, therefore the new execution trace has 53,638 calls.

B. Quantitative results

Table 1 and Table 2 present the results of applying the "proposed utility detection metrics" to the **Checkstyle** and **JHotDraw** execution traces respectively. The info is organized in descending consistently with the EIUC value.

Therefore, classes that appear in front could be considered as utilities, while classes that appear at the end are non-utility classes. For example, Table 1 shows that there is no chance for TreeWalker and ConfigurationLoader classes to be utilities because they sit at the bottom of the table. Whilst, DefaultContext, DefaultConfiguration, DetailAST classes are candidate utility classes because they sit at the top of the table. The same observations could be applied to Table 2.

The results in the two case studies conform to the results in Cornelissen's study [30]. Hence, the proposed technique has the capability to classify classes of object-oriented systems consistent with their probability of being candidate utilities and put them at the highest point of the ranking table.

Nevertheless, we have to set some parameters to guide the process. For example, a parameter is optionally required to enable us to concentrate on particular portions of the execution trace. This is important when analyzing very large execution traces that comprise several features. Hence, we can analyze only a specific feature, exclude a specific feature, or analyze the features separately.

Another very important parameter is the size ratio parameter (SR) to decide the number of classes that we wish to appear in the final execution trace (CA). For example, if the size of the original execution traces is represented by the total number of its classes (CT), then CA = CT * SR.

We randomly chose SR=80% for **Checkstyle** and **JHotDraw** execution traces which means that only the events of 80classes (out of 100 classes) will appear in the final **Checkstyle** execution trace and only the events of 90 classes (out of 112 classes) will appear in the final **JHotDraw** execution trace. Consequently, the final execution traces contain only 4,372 calls (37% of the original size) and 170 calls (0.32% of the original size) respectively.

The aforementioned results show that 20% of the **Checkstyle** execution trace classes are responsible for nearly 63% of the interactions within the execution trace and 20% of the **JHotDraw** execution trace classes are responsible for nearly 99.68% of the interactions within the execution trace. The results show that the value of the SR parameter should not be fixed for all case studies. For example, what is adequate for one case study may not be for another. In addition, it should not be fixed for the same case study. For example, when the user gains some knowledge about the case study he may request to explore more details.

However, the user may adjust the size ratio parameter if the simplified execution trace is inadequate. Therefore, the setting of this parameter is totally up to the user until it is suitable for the purpose at hand. In other words, the final execution trace should be customized on request (i.e. expanded or collapsed). In particular, we can easily adjust the size ratio parameter and apply the proposed technique again. For example, we can adjust the parameter value to 95% for the **JHotDraw** execution trace if the final trace is too abstract or if we need to show some classes that are excluded for the first time. According to the adjusted parameter value, the final execution trace contains 31055 calls (58% of the original size). Furthermore, if the final trace is too detailed or if we need to exclude more classes from the trace, the parameter can be adjusted again with less value. For example, if we choose SR=90%, then the final execution trace will contain 5,241 calls (less than 10% of the original trace).

TABLE 1 CHECKSTYLE'S CLASSES SORTED DESCENDING ACCORDING TO EIUC ( ) VALUE.

EUC( )	*EIUC( )*	*CLASS NAME*
0.3316	0.3316	DefaultContext
0.1968	0.1968	DefaultConfiguration
0.131	0.131	DetailAST
0.0816	0.0816	FullIdent
		.
		.
		.
0.0102	0	TreeWalker
0	0	grammars.GeneratedJavaLexer
0	0	ConfigurationLoader
0	0	PackageNamesLoader
0	0	TreeWalker$SilentJavaRecognizer
Sum of Methods= 798		
Number of total classes = 100		
Number of well-known library classes = 1		
Number of undesired classes = 0		
Number of classes involved in the analysis = 99		
Number of classes involved in the final trace file =80		

TABLE 2 JHOTDRAW'S CLASSES SORTED DESCENDING ACCORDING TO EIUC ( ) VALUE.

EUC( )	*EIUC( )*	*CLASS NAME*
0.2362	0.2362	NullDrawingView
0.2358	0.2358	AbstractCommand$EventDispatcher
0.3183	0.19	ZoomDrawingView
0.1584	0.1584	MainClass
0.1826	0.1384	AbstractTool$EventDispatcher
		.
		.
		.
0.0023	0.0016	StandardDrawing
0.0023	0.0014	TextFigure
0.0023	0.0012	DesktopEventService$1
0.0023	0.0012	MDIDesktopPane$1
0.0023	0.0009	StandardDisposableResourceManager
Sum of Methods= 445		
Number of total classes = 112		
Number of well-known library classes = 0		
Number of undesired classes = 0		
Number of classes involved in the analysis = 112		
Number of classes involved in the final trace file =?		

VI. CONCLUSION

Software Comprehension is a significant phase of software maintenance as problem comprehension is the main part of problem-solving. The contribution of this paper is to propose two novel utility detection metrics to detect utility classes for a certain scope of the specified execution trace. The goal of utility detection metrics is to improve understanding of the process and reduce the time and effort required for the software maintenance process. These metrics are based primarily on the dynamic coupling to capture object-oriented system properties in runtime that static coupling cannot address them. The first detection metric takes into consideration only one direction of coupling (namely export coupling), while, the second detection metric takes into consideration the coupling in both directions (export and import coupling).

We demonstrated our technique using two case studies, namely **Checkstyle** and **JHotDraw** execution traces, which were specifically chosen because of the availability of their execution traces in a common dataset benchmark. The quantitative results of our case studies have shown that our technique is able to automatically detect and remove utility classes.

In particular, when we choose to filter out 20% of the total classes in **Checkstyle** execution trace, (i.e. the size ratio parameter = 80%), the original execution trace is reduced by 63% when interactions to the removed classes are excluded. In addition, the same result is reported when we filter out only 5% of the total classes in **JHotDraw** execution trace. The results prove that a small percentage of execution trace classes causes the execution trace to inflate to a very large size.

Integrating this technique into a complete trace analysis framework will be our forthcoming contribution. The framework suggests the decoupling of the utility classes rather than simply removing them. The basic idea behind using decoupling is to prevent the creation of gaps in the structure of execution traces and to prevent the removal of some critical dependencies to utility classes.

References

[1] E. Bousse, et al., "Advanced and Efficient Execution Trace Management for Executable Domain-Specific Modeling Languages," Software & Systems Modeling, vol. 17, no. 1, pp. 1-37, 2017.

[2] C. Chapman, et al., "Exploring Regular Expressionsion Comprehension," in *32nd International Conference on Automated Software Engineering(ASE 17)*, 2017,pp.348-356.

[3] D. Kauchak and G. Leroy, "Moving Beyond Readability Metrics for Health-Related Text Simplification," *IEEE Journals & Magazines,* vol. 17, no. 1, pp. 41-51, 2016.

[4] J. Chhabra and V. Gupta, "Survey of Dynamic Software Metrics," *Computer Science and Technology,* vol 25, no. 4, pp. 1016-1029., 2010.

[5] H. Virdi and B. Singh, "Analysis of the Software Code based upon Coupling in the Software," in *Third International Conference on Computing Communication & Networking Technologies (ICCCNT'12)*, 2012, pp.651-659.

[6] I.Sommerville, Software Engineering. 10 edition., Pearson Education, 2016.

[7] N. Al-Saiyd, "Source Code Comprehension Analysis in Software Maintenance," in *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*,2017, pp. 1-5.

[8] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behavior of a Software System," in *14th IEEE International Conference on Program Comprehension*, 2006.

[9] A. Hamou-Lhadj and T. Lethbridge, "Understanding the complexity embedded in large routine call traces with a focus on program comprehension tasks.," *IET Software,* vol. 9, no. 2, p. 61–177, 2010.

[10] X. Xia, L. Bao, Z. Xing, E. Hassan, and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals, "*IEEE Transactions on Software Engineering,* vol. 44, no.10, 2017.

[11] L. Soares, et al., "Lightweight Process for Dynamic Inspection of Feature Interactions," in *12nd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS2018)*, Madrid, Spain, 2018.

[12] F. Palomba and A. Zaidman, "Does Refactoring of Test Smells Induce Fixing Flaky Tests?,"," in *33rd International Conference. Software Maintenance and Evolution (ICSME 17)*, 2017,pp.167-175.

[13] N. Ganesh and S. Babu, "Analysis of Static Coupling Versus Dynamic Coupling in a Distributed Object-Oriented System Based on Trace Events," *World Engineering & Applied Sciences Journal,* vol. 6, no. 1, pp. 91-95, 2015.

[14] S. Jayaraman, et al., "Compact Visualization of Java Program Execution," *Software: Practice and Experience,* vol. 47, no. 2, pp. 163-191, 2016.

[15] L. Mengleng, et al., "Fault Localization Guided Execution Comparison for Failure Comprehension," in *IEEE International Conference on Software Quality, Reliability and Security Companion*, 2016.

[16] Y. Jia and M. Harman, "An Analysis and Survey Of The Development of Mutation Testing," *Transactions on Software Engineering,* vol. 43, no. 5, p. 649–678, 2016.

[17] G. Meszaros, xUnit Test Patterns: Refactoring Test Code, Pearson Education, 2007.

[18] R. Chauhan and I. Singh, "Latest Research and Development on Software Testing Techniques and Tools," *INPRESSCO International Journal of Current Engineering and Technology,* 2014,pp.423-429.

[19] T. Al-Rousan and H. Abualese, "Simplifying the Structural Complexity of Software Systems," Cybernetics and Information Technologies, Vol. 19, No. 3. 2019.

[20] T. Al-Rousan and H. Abualese, "A New Technique for Utility-Class Detection in Object-Oriented Software," *Tem Journal - Technology, Education, Management, Informatics*, Vol.8, No.2. 2019.

[21] E. Arisholm, et al., "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Transactions on Software Engineering,* vol 30, no. 18, p. 491–506., 2004.

[22] N. Bhateja, "A Study on Various Software Automation Testing Tools," *International Journal of Advanced Research in Computer Science and Software Engineering,* vo.l 5, no. 6, 2015.

[23] H. Abualese, et al., "Utility Classes Detection Metrics For Execution Trace Analysis," in *8th International Conference on Information Technology (ICIT)*, Amman, 2017, 156-164.

[24] H. Abualese, et al., "A Trace Simplification Framework," in *8th International Conference on Information Technology (ICIT)*, Amman, 2017, pp. 372-278.

[25] S. Anand, "An Orchestrated Survey on Automated Software Test Case Generation," *Journal of System and Software*, vol. 78, no. 1, pp. 1978-2001, 2013.

[26] Checkstyle Tool, Available: http://checkstyle.sourceforge.net/. [Accessed 9- 4 -2018].

[27] JHotDraw Tool, Available: https://sourceforge.net/projects/jhotdraw//. [Accessed 25- 3 -2020].

[28] JK. Chhabra, and V. Gupta, "A Survey of Dynamic Software Metrics," *Journal of Computer Science and Technology*, 25(4), pp. 1016-1029. 2010.

[29] S. Pfleeger, "Software Engineering Theory and Practice", Pearson India, 2013.

[30] B. Cornelissen, A. Zaidman and A. van Deursen, "A Controlled Experiment for Program Comprehension through Trace Visualization," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 341-355, May-June 2011.