International Journal of Engineering and Management Research
https://ijemr.vandanapublications.com
e-ISSN: 2250-0758 | p-ISSN: 2394-6962
Volume-13, Issue-3 (June 2023)
https://doi.org/10.31033/ijemr.13.3.33

Prediction Systems for Process Understandability and Software Metrics

Mohammad Saif Himayat[1] and Dr. Jameel Ahmad[2]
[1]PG Student, Department of Computer Science & Engineering, Integral University, INDIA
[2]Associate Professor, Department of Computer Science & Engineering, Integral University, INDIA

[1]Corresponding Author: mshimayat@student.iul.ac.in

Received: 01-06-2023 Revised: 16-06-2023 Accepted: 30-06-2023

ABSTRACT

The abstract of this research study outlines the objective of validating prediction systems for process understandability and software metrics. In this study, we focus on assessing the accuracy and reliability of prediction systems that aim to provide insights into complex processes and software-related metrics. The process of validation involves defining clear objectives, gathering relevant data, preprocessing the data, performing feature engineering, selecting appropriate prediction models, and training and validating these models using cross-validation techniques. Additionally, we emphasize the importance of interpretability and explainability in the prediction process, which enables us to gain meaningful insights into the underlying processes. Furthermore, a comparative analysis is conducted to compare the predictions generated by the system with ground truth or expert judgments, thereby ensuring the accuracy and reliability of the predictions. The study adopts an iterative refinement approach to enhance the performance, interpretability, and usability of the prediction system based on feedback and validation results. By following this comprehensive validation process, we aim to establish reliable prediction systems that provide meaningful understandability of processes and software metrics.

Software metrics play a crucial role in assessing the quality, maintainability, and performance of software systems. However, understanding these metrics and their implications can be challenging, especially for non-technical stakeholders. This research study focuses on the understandability of software metrics and proposes a validation framework to assess the effectiveness of prediction systems in providing understandable insights.

Keywords-- Software Metrics, Software Understandability, Prediction Systems

I. INTRODUCTION

Software architectural structures form the foundation of complex software systems, defining the organization, relationships, and interactions between various components[1,2,3]. The understandability of these architectural structures is crucial for effective communication, collaboration, and maintenance of software systems. However, assessing and quantifying the understandability of architectural structures can be challenging due to their inherent complexity and the subjective nature of human comprehensi[4,5]on.

Software metrics offer a systematic approach to evaluate software qualities and characteristics. They provide quantitative measures that aid in assessing different aspects of software systems, including understandability. By employing software metrics specifically designed for measuring the understandability of architectural structures, software practitioners and stakeholders can gain valuable insights into the complexity and comprehensibility of the software's architectural design[6,7].

The purpose of this research study is to identify and validate software metrics that effectively measure the understandability of architectural structures. These metrics will enable software practitioners to objectively evaluate and improve the understandability of their architectural designs, leading to more maintainable and robust software systems[8,9,10].

In this study, we will review existing literature and research related to software architecture, understandability, and software metrics. We will explore various architectural elements and their impact on understandability, such as modules, components, interfaces, and dependencies. By understanding the factors that influence architectural understandability, we can identify appropriate metrics that capture these aspects[11,12,13].

The selected metrics may include complexity measures, cohesion and coupling metrics, architectural layering or modularity indices, and architectural documentation coverage. Each metric will provide a numerical value representing a specific aspect of architectural understandability. These metrics will be carefully chosen based on their relevance, reliability, and applicability to architectural structures[14,15].

To validate the proposed metrics, an empirical study will be conducted using real-world software systems. These systems will represent a variety of architectural styles and complexities. The identified metrics will be calculated for each system, and subjective assessments of understandability will be collected from software practitioners. These assessments can be gathered through expert reviews, surveys, or cognitive walkthroughs[16,17,18].

Statistical analysis techniques, such as correlation coefficients and regression models, will be applied to analyze the relationship between the proposed metrics and subjective assessments of understandability. This analysis will help validate the metrics and determine their effectiveness in predicting architectural understandability.

The validated metrics will serve as practical tools for assessing the understandability of architectural structures in real-world software development projects. They will provide software practitioners and stakeholders with objective indicators to identify areas of improvement, detect potential design flaws, and guide architectural decisions towards more comprehensible and maintainable software systems[19.20,21].

The study begins by identifying a set of relevant software metrics that are commonly used in software development and maintenance. These metrics encompass various aspects, such as code complexity, code duplication, code churn, and software defect density[22,23].

Next, a prediction system is developed to generate predictions based on the selected metrics. The system employs machine learning or statistical models to analyze historical data and make predictions about future software outcomes or characteristics[24,25].

To validate the understandability of the prediction system, an evaluation framework is established. This framework includes both quantitative and qualitative measures. Quantitative measures assess the accuracy and reliability of the predictions, while qualitative measures focus on the comprehensibility and interpretability of the generated insights[26,27].

The validation process involves collecting a diverse dataset that includes software projects with varying characteristics and complexity levels. The prediction system is trained using this dataset, and its performance is evaluated using appropriate metrics such as accuracy, precision, recall, and F1-score[28].

To assess understandability, user studies, surveys, or interviews may be conducted to gather feedback from software practitioners and stakeholders. Their perception of the generated predictions and insights is analyzed to determine the effectiveness of the prediction system in conveying understandable information about software metrics. The results of the validation process are used to refine and improve the prediction system. This iterative approach ensures that the system becomes more accurate, reliable, and understandable over time[29].

By validating prediction systems for software metrics understandability, this research study aims to provide valuable insights into the development and maintenance processes, helping stakeholders make informed decisions and take appropriate actions to improve software quality and performance**.**

II. SUCCESSES AND FAILURES OF THE SOFTWARE

Successes and failures in software development are common occurrences that can have significant impacts on projects and organizations. Let's explore some examples of both successes and failures in software development:

Successes: Release of a Stable and Functional Product: A major success in software development is when a product is released that meets or exceeds user expectations. The software is stable, performs well, and fulfills its intended purpose effectively.

Timely Delivery: Completing a software project on schedule is considered a success. It demonstrates effective project management, resource allocation, and adherence to timelines. Timely delivery allows organizations to capitalize on market opportunities and gain a competitive advantage.

Positive User Feedback: When software receives positive feedback from users, it indicates that the product is meeting their needs and providing a satisfactory user experience. Positive user feedback boosts customer satisfaction and loyalty, which can lead to increased adoption and revenue generation.

Effective Bug Fixing and Maintenance: Successfully addressing bugs and maintaining software over time demonstrates a commitment to quality and ongoing improvement. Regular updates, bug fixes, and feature enhancements contribute to the longevity and relevance of the software[30].

Failures: Project Delays and Cost Overruns: One of the most common failures in software development is when a project exceeds its planned timeline and budget. This can occur due to poor project management, inadequate resource allocation, scope creep, or unexpected technical challenges.

Poor User Experience: When software fails to provide a user-friendly and intuitive experience, it can lead to frustration, low adoption rates, and negative reviews. Poor user experience often stems from inadequate user research, ineffective design, or usability issues.

Critical Security Breaches: Security vulnerabilities in software can lead to significant failures, compromising sensitive user data, damaging a company's reputation, and exposing organizations to legal and financial repercussions. Neglecting proper security measures and testing can result in severe consequences.

Software Defects and Unreliability: If a software product contains significant defects or experiences frequent crashes and errors, it can undermine user trust and damage the reputation of the development team or organization. High defect rates indicate a lack of thorough testing, quality assurance, or coding standards.

Lack of Scalability and Adaptability

When software fails to scale effectively or adapt to changing requirements, it can limit its usefulness and hinder organizational growth. Inflexible software architectures or inadequate planning for future needs can contribute to this failure[31].

III. LIMITATIONS OF METRICS FOR PREDICTING SOFTWARE QUALITY

While metrics play a significant role in assessing and predicting software quality, they also have certain limitations that need to be considered. Some of the limitations of using metrics for predicting software quality are:

1. Limited Coverage: Metrics may not capture all aspects of software quality. They often focus on specific measurable characteristics, such as code complexity, code coverage, or defect density, while neglecting other important qualitative factors like user experience, maintainability, and scalability. This limited coverage may result in an incomplete understanding of overall software quality[32].

2. Lack of Contextual Information: Metrics provide quantitative data but may not provide the necessary contextual information to interpret the results accurately. For example, a high defect density metric may indicate poor quality, but without understanding the complexity of the software or the severity of the defects, it can be challenging to determine the actual impact on quality.

3. Inherent Complexity: Software development is a complex process, and software quality is influenced by multiple interrelated factors. Metrics often simplify this complexity by quantifying specific attributes, but they may fail to capture the intricate dependencies and interactions among different software components. Consequently, relying solely on metrics may oversimplify the assessment of software quality.

4. Lack of Standardization: There is often a lack of standardized and universally accepted metrics for measuring software quality. Different organizations or domains may adopt their own set of metrics, leading to inconsistencies and difficulties in comparing and benchmarking software quality across different projects or contexts. This lack of standardization hinders the reliability and generalizability of metrics-based predictions.

5. Subjectivity and Interpretation: The interpretation of metrics and their thresholds can be subjective and dependent on individual or organizational perspectives. Different stakeholders may have varying interpretations of what constitutes good or poor quality based on the same metrics. This subjectivity can introduce bias and lead to inconsistent predictions of software quality.

6. Evolving Nature of Software: Software systems are dynamic and constantly evolving. Metrics-based predictions are often based on historical data and assumptions that may not hold true in the future. As software changes, new features are added, or technologies evolve, the validity and relevance of metrics may diminish, reducing their predictive power.

7. Human Factors: Metrics tend to focus on technical aspects of software quality but may not adequately consider human factors, such as user needs, expectations, and satisfaction. Software quality is ultimately determined by how well it meets user requirements and provides a positive user experience, which metrics may not fully capture.

8. To address these limitations, it is essential to consider metrics as just one piece of the puzzle in assessing software quality. Combining metrics with other qualitative assessments, user feedback, and expert judgment can provide a more comprehensive understanding of software quality and improve the accuracy of predictions. Additionally, regular evaluation and refinement of metrics frameworks can help overcome some limitations by adapting to changing software development practices and incorporating new dimensions of quality[33].

IV. FACTORS AFFECTING SOFTWARE UNDERSTANDABILITY[1]

Software understandability is influenced by various factors that can impact how easily developers and other stakeholders comprehend and work with software systems.

We defined the membership function in fuzzy mathematics for different factors Software is maintained through the integrated use of source code and documents. Source code readability and quality of documentation should be taken into account while measuring the software maintainability. Comments Ratio (CR) is used to judge the Readability of Source Code (RSC). Quality of Documentation (QOD) is judged using Fog index[34].
understandability of software documentation we compiled is composed of source code (RSC) and documents (QOD) with membership function as follows:
(1) Membership functions for documents judged using Fog index:

$$\mu\text{poor}(x) = \begin{cases} 1 & x \leqslant 9 \\ 5x - 6 & 10 < x \leqslant 12 \\ 0 & x > 12 \end{cases}$$

$$\mu\text{average}\ (x) = \begin{cases} 0 & x \leqslant 10\ or\ x > 18 \\ 5x - 5 & 10 < x < 12 \\ 1 & 12 < x \leqslant 16 \\ 5x - 9 & 16 < x \leqslant 18 \end{cases}$$

$$\mu\text{good}[x] = \begin{cases} 0 & x \leqslant 16 \\ 5x - 8 & 16 < x \leqslant 18 \\ 1 & x > 18 \end{cases}$$

Membership functions for RSC judged using CR

$$\mu\text{poor}\ (x) = \begin{cases} 0 & x \leqslant 7 \\ x - 7 & 7 < x \leqslant 8 \\ 1 & x > 8 \end{cases}$$

$$\mu_{\text{avg}}\ (x) = \begin{cases} 0 & x < 5 \text{ or } 8 < x \\ x - 5 & 5 < x \leqslant 6 \\ 1 & 6 < x \leqslant 7 \\ -x + 8 & 7 < x \leqslant 8 \end{cases}$$

V. FRAMEWORK FOR EVALUATING MODELING TECHNIQUE UNDERSTANDING

Based on work by Mayer [5], Gemino and Wand proposed a framework for evaluating model understanding for arbitrary modeling techniques [3]. They differentiate between model creation (for representing parts of the real world) and model reading (creating a mental representation from a model) [3, p. 80]. In this paper, we deal with the second point. For this purpose, they suggest a model for knowledge construction and learning from models adapted from Mayer: Content, presentation method and the model viewer characteristics influence the knowledge construction and consequently the learning outcome. This cognitive process is not directly observable, but has to be observed indirectly through learning performance tasks. Here, Gemino and Wand list comprehension and problem-solving tasks. The former include questions regarding attributes of and relationships between model items—while the latter include questions going beyond the information given originally in the model. [3, pp. 82–83] For our problem (process understandability), comprehension tasks seem to be obvious.

VI. ASPECTS OF PROCESS UNDERSTANDABILITY

As we already discussed in Section 4, it is important to cover the different aspects of process understandability to fulfill the content validity requirement for metrics. In this paper, we concentrate on the aspects order, concurrency, exclusiveness and repetition. Doing so, we do not deny the possible existence of other aspects. Unlike in [7], we will give detailed definitions of the questions of the different aspects. We start with the definition of the term "activity period" which is later used in our questions[35,36].

Activity Period

An activity period of task t is the period between a point in time when t becomes executable and the next point in time when the actual execution of t terminates. Now, we c an define relations for the four aspects of process understandability[37].

Order

For the questions about task order,

the relations $O\exists, O\forall \subseteq TXT$

with the following meanings are used.

There is no process instance for

which an activity period of task t1 ends before an activity period of task t2 starts[38].

$(t_1, t_2) \in 0\exists \Leftrightarrow$,There is a process instance for which an activity period of task t1 ends before an activity period of task t2 starts.—But there also exists a process instance for which this does not hold.

$(t_1, t_2) \in o\forall \Leftrightarrow$, For each process instance, an activity period of task t1 ends before an activity period of task t2 starts.

Concurrency

For the questions about task concurrency, the relations

$\subset \nexists, c\exists \leq T \times T$

With the following meanings are used.$(t_1, t_2) \in c\exists \leftrightarrow$ There is no process instance for which the activity periods of tasks t1 and t2 overlap.(t1; t2) 2 c9 ,There is a process instance for which the activity periods of tasks t1 and t2 overlap at least once (Several executions of t1 and t2 per process instance are possible!).—But there also exists a process instance for which this does not hold[39].

$(t_1, t_2) \in c\exists \leftrightarrow$,For each process instance, the activity periods of tasks t1 and t2 overlap at least once.

Object-Oriented Metrics

When it comes to object-oriented programming, several metrics can be used to assess the quality and maintainability of code. These metrics specifically focus on aspects related to object-oriented design principles and concepts. Here are some commonly used metrics for object-oriented code:

1. Class Coupling: Class Coupling measures the degree of interdependence between classes in a codebase. It counts the number of unique classes that a particular class relies

e-ISSN: 2250-0758 | p-ISSN: 2394-6962
Volume-13, Issue-3 (June 2023)
https://doi.org/10.31033/ijemr.13.3.33

on or interacts with. High coupling indicates tight dependencies, which can make code more difficult to understand and maintain[40].

2. Cohesion: Cohesion measures how closely the methods and attributes within a class are related to each other. High cohesion suggests that the methods and attributes in a class are closely related to its purpose and responsibilities, leading to better code understandability and maintainability.

3. Depth of Inheritance Tree (DIT): DIT measures the number of inheritance levels in a class hierarchy. It indicates the depth of the inheritance tree and the potential complexity involved in understanding the relationships between classes. High DIT values can imply increase complexity and reduced understandability.

4. Number of Children (NOC): NOC measures the number of immediate subclasses that inherit from a particular class. Higher NOC values suggest a larger number of derived classes, which can indicate a complex class hierarchy and potentially affect code understandability and maintainability.

5. Lack of Cohesion in Methods (LCOM): LCOM measures the lack of cohesion within a class. It quantifies the number of pairs of methods in a class that do not share common attributes or methods. Higher LCOM values suggest lower cohesion and can indicate potential design issues that affect understandability and maintainability.

6. Weighted Methods per Class (WMC): WMC measures the number of methods within a class, giving each method a weight based on its complexity (e.g., cyclomatic complexity). It provides an indication of the complexity and potential understandability challenges of a class.

7. Response for a Class (RFC): RFC measures the number of methods that can be invoked in response to a message or request to a class, including its own methods and those inherited from super classes. Higher RFC values can indicate increased complexity and potentially impact code understandability.

8. Lack of Cohesion in Hierarchies (LCH): LCH measures the lack of cohesion within an inheritance hierarchy. It quantifies the number of pairs of methods in different classes within the hierarchy that do not share common attributes or methods. Higher LCH values indicate lower cohesion and can suggest potential issues with the hierarchy's design and understandability.

These metrics provide insights into the design and complexity of object-oriented code, highlighting areas that may need attention to improve understandability and maintainability. However, it's important to remember that these metrics should be used in conjunction with other qualitative assessments, code reviews, and the specific context of the project to make informed decisions about code quality and design.

VII. DATA ANALYSIS

To perform a data analysis of software understandability using software metrics, you can follow these general steps:

1. Define the Software Metrics: Identify the software metrics that are relevant to understanding the code's complexity, maintainability, and readability. This can include metrics such as Cyclomatic Complexity, Lines of Code (LOC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), or any other metrics that you find suitable.

2. Gather Data: Collect the required data for the selected software metrics. This data can be obtained from the source code repository, version control system, or static code analysis tools. Ensure that you have the necessary information, such as the number of classes, methods, lines of code, and any other metrics you plan to analyze.

3. Calculate Metrics: Use appropriate formulas or existing tools to calculate the software metrics for each code component (class, method, etc.) in your dataset. Apply the formulas to the relevant data points and calculate the corresponding metric values.

4. Establish a Baseline: Determine a baseline or reference point for software understandability. This can be done by analyzing a representative sample of the codebase or by using expert judgment. This baseline will serve as a benchmark for comparison and evaluation.

5. Analyze the Data: Perform data analysis techniques to gain insights into software understandability. This can include statistical analysis, data visualization, correlation analysis, or any other suitable techniques based on the nature of the metrics and your research questions. Look for patterns, trends, and relationships among the metrics to identify factors that may impact understandability.

6. Interpret the Results: Interpret the results of the data analysis to draw conclusions about software understandability. Identify code components that deviate significantly from the baseline or exhibit patterns indicating low understandability. Investigate the potential causes of these deviations, such as high complexity, excessive coupling, or poor code structure.

7. Take Action: Based on the findings, determine appropriate actions to improve software understandability. This can involve refactoring code, improving documentation, addressing high complexity areas, or introducing coding guidelines and best practices.

8. Monitor and Iterate: Continuously monitor the software metrics and repeat the analysis periodically to track the progress of understandability improvements over time. This iterative process helps in identifying areas that still require attention and evaluating the effectiveness of the actions taken.

It's important to note that software understandability is a multidimensional aspect influenced by various factors. Therefore, a comprehensive analysis may involve considering multiple software metrics simultaneously and integrating qualitative assessments from developers or code reviewers to get a holistic view of understandability.

Software Understandability Based on Fuzzy Matrix

Software understandability is a subjective measure that depends on multiple factors and cannot be precisely quantified. However, fuzzy logic can be applied to create a fuzzy matrix that assesses the understandability of software based on linguistic variables. The fuzzy matrix allows for representing and reasoning about imprecise and uncertain information. Here's a general approach to constructing a fuzzy matrix for software understandability:

1. Identify linguistic variables: Determine the linguistic variables that contribute to software understandability. These variables can include code readability, complexity, modularity, naming conventions, documentation, and so on. Each linguistic variable should have a set of linguistic terms or labels that represent different levels or degrees.

2. Define membership functions: For each linguistic term, define membership functions that describe the degree of membership or relevance to that term. Membership functions can be triangular, trapezoidal, Gaussian, or any other suitable shape that represents the fuzzy membership.

3. Determine fuzzy rules: Establish a set of fuzzy rules that map the input linguistic variables to the output linguistic variable, which represents the understandability level. These rules capture expert knowledge or heuristics about how different linguistic variables influence software understandability. For example, a rule might state that if the code readability is "high" and the modularity is "moderate," then the understandability is "good."

4. Evaluate linguistic variables: Assess the linguistic variables based on their corresponding membership functions. Evaluate the linguistic terms for each variable to determine the degree to which they apply to the software being analyzed. This evaluation can involve linguistic assessments by experts or automated analysis techniques.

5. Apply fuzzy inference: Utilize fuzzy inference methods, such as Mamdani or Sugano, to compute the output fuzzy set for understandability based on the fuzzy rules and the evaluated linguistic variables. Fuzzy inference combines the fuzzy rules and their degrees of applicability to derive a fuzzy output.

6. Defuzzification: Convert the fuzzy output into a crisp value using defuzzification techniques. Common methods include centroid, mean of maxima, and weighted average.

7. Interpretation: Interpret the crisp value obtained from defuzzification as a numerical representation of the software's understandability level. This value can be mapped to linguistic terms, such as "low," "medium," or "high," to provide a more human-readable understanding of the software's understandability.

It's important to note that constructing a fuzzy matrix for software understandability requires domain expertise and knowledge. The fuzzy matrix should be continuously refined and validated based on real-world data and expert feedback to ensure its accuracy and effectiveness in capturing software understandability.

Measured Software Artefacts and Attributes (RQ1)

Different authors claim the importance of both, metrics that measure individual artefacts (components) in the system and metrics that measure the whole architectural

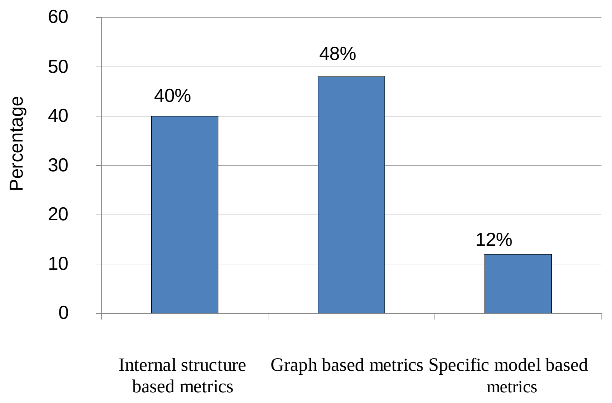

Figure 1: The distribution of the studies related to the approach type

etc.). For example, the metrics like the total number of components or links in the system's component view represent the metrics related to the architecture artefact. The metrics related to the component artefact are defined from different authors using the term component as a high-level artefact in different contexts. Kanjilal et al. [32] considers component as a system element that can be composed with other components, offers a predefined service and is able to communicate with other components. Misic [47] considers component as a set of objects at different abstraction levels (libraries, project objects). Sartipi [58] considers component as a group of system entities in form of a file (to evaluate a design), or module and sub-system (to evaluate the architecture). Shereshevsky et al. [60] consider primitive components (at the lowest level) that exchange the information between each other and do not contain any other components and upper-level components that contain those at the level below them without overlapping. Wei et al. [15] consider components as autonomous pieces of software code with well-defined functionality and interfaces, similarly, to the work by Kanjilal et al. At the end Yu et al. [12] consider primitive components, that represent the smallest units, and can be composed into compound components that are further composed into higher level compound components so that the layered component structure is formed (similarly to the work by Shereshevsky etal.). In that context different artefacts can be considered as components such as packages, classes, programs, etc. Considering the previous considerations, we can say that the term component is used as a higher-level artefact that can be composed of other components or lower-level artefacts and that has well-defined functionality. Component level metrics consider incoming/outgoing interactions of a component, relations between the entities within a component, etc.

Component-to-component metrics consider pairs of components. Some examples of those metrics are the total number of interfaces between any pair of components or the number of connectors on the shortest path between a pair of components.

Package artefacts are considered also from different authors but they all consider packages as artefacts that

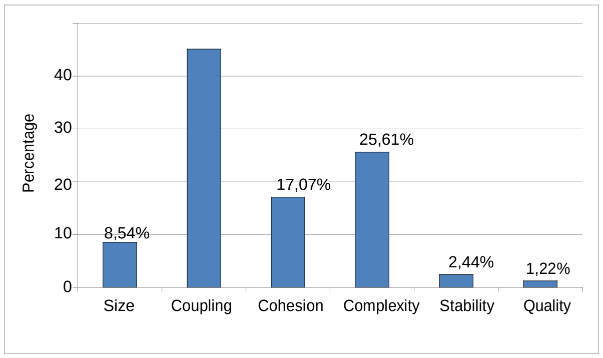

Figure 2: The distribution of the metrics related to the measured software attributes

sider module as a group of components (component represents any abstract high-level artefact). Based on dependency analysis components are grouped into modules (architectural slicing). In the work by Hwa et al. [6] module contain classes as well as other modules which leads to a hierarchical structure very similar to the package hierarchical structure explained above. Lundvall et al. [7] and Sarkar al. [8] consider modules as sets of classes like the work by Hwa et al. Some examples of module level metrics are the number of classes outside a module that are commonly shared by the classes inside a module, the number of classes inside a module that are used by other classes in other modules, etc. Finally, graph node metrics consider nodes in the graph that is used to represent a software system in a very abstract way. Graph node metrics are for example the degree of a node in a graph, the importance of a node in a graph, etc.

The distribution of the metrics related to the measured artefacts is shown in Figure 4. Regarding the software attributes that are measured the following categories emerged during the data analysis:

• Size Metrics are related to the number of constituents elements of the corresponding design units (artefacts) in the system or to an information theory-based size. For example, the number of components and modules are related to the overall structure of the system. The number of classes is related to single entities, but also it can be related to the overall structure of the system. Information theory based size metrics calculate the amount of information in the system graph using Shannon entropy.

• Coupling Metrics are concerned with the relations between the design units. Those relations are reflected through the number of interfaces, the links or paths between the design units, the extent to which some design units use other design units, the Shannon entropy of the information transmission between design units (information theory-based coupling metrics, see for example [5]), etc. Coupling mechanisms are also distinguished in terms of the direction of coupling (import or export coupling), and through direct and/or indirect connections between the design units.

• Cohesion Metrics are very similar to the coupling metrics except that they are bound to the relations between the constituting parts of the same design unit (artefact). Functional cohesion introduces external and internal cohesion, where external cohesion considers the relations between the elements inside a given design unit and the elements outside that design unit, while internal cohesion considers relations between the elements inside a given design unit. Cohesion is also measured as the extent to which the elements within one design unit are commonly used from other design units or as information-based cohesion that measures the information flow within design units using the aforementioned Shannon entropy.

• Complexity Metrics measure the degree of connectivity between elements by considering the relationships within design units and between them together. They are concerned with the metrics related to network parameters (graph-based metrics), information theory-based complexity, etc. They also measure the hierarchical structure (degree of composition) in the system.

• Stability Metrics measure how easy it is to make changes to the elements in a design unit without affecting elements in other design units in the system.

• Quality Metric is based on the Multi-Attribute Utility Technique (MUAT) which argues that the quality of a component is decided by its N attributes such as complexity and maintainability [30]. This metric considers composite based software architecture which provides a way to separately describe control flow and computation.

e-ISSN: 2250-0758 | p-ISSN: 2394-6962
Volume-13, Issue-3 (June 2023)
https://doi.org/10.31033/ijemr.13.3.33

Figure 5 shows the distribution of the metrics related to the measured software attribut

VIII. CONCLUSION

Software understandability affects quality of overall software engineering. If software under- standability is favorable, software development process can be mastered. In this work, we considered so many different types of metrics. But we want to focus few more metrics on our further research. Here in chapter 4, we used a rough set approach to detect the project which is having abnormal behavior. This type of behavior tells us that the project is either easily understandable or very much difficult to understand. The algorithm which is used by us is having less time complexity than fuzzy based approach. In our further work we want to include threshold values which have been calculated based on the standard values of different attributes, based on that threshold value we will give outlier ranking.

REFERENCES

[1] Lin, J. C. & Wu, K. C. (2006, September). A model for measuring software understandability. In: *Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, pp. 192-192. IEEE.

[2] Melcher, J. & Seese, D. (2008, September). Towards validating prediction systems for process understandability: measuring process understandability. In: *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 564-571. IEEE.

[3] Srivastava, M. Haroon, & A. Bajaj. (2013). Web document information extraction using class attribute approach. *4th International Conference on Computer and Communication Technology (ICCCT), Allahabad, India*, pp. 17-22. DOI: 10.1109/ICCCT.2013.6749596.

[4] Haroon, M., Tripathi, M. M. & Ahmad, F. (2020). Application of machine learning in forensic science. In: *Critical Concepts, Standards, and Techniques in Cyber Forensics*, pp. 228-239. IGI Global.

[5] R. Khan, M. Haroon & M. S. Husain. (2015). Different technique of load balancing in distributed system: A review paper. *Global Conference on Communication Technologies (GCCT), Thuckalay, India*, pp. 371-375. DOI: 10.1109/GCCT.2015.7342686.

[6] M. Haroon & M. Husain. (2015). Interest attentive dynamic load balancing in distributed systems. *2nd International Conference on Computing for Sustainable Global Development (INDIA Com), New Delhi, India*, pp. 1116-1120.

[7] Fenton, N. E. & Neil, M. (2000, May). Software metrics: roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*, pp. 357-370.

[8] Harrison, R., Counsell, S. & Nithi, R. (1998, November). Coupling metrics for object-oriented design. In: *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, pp. 150-157. IEEE.

[9] Fenton, N. E. & Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, *47*(2-3), 149-157.

[10] Haroon, M. & Husain, M. (2013). Analysis of a dynamic load balancing in multiprocessor system. *International Journal of Computer Science Engineering and Information Technology Research*, *3*(1).

[11] Wasim Khan & Mohammad Haroon. (2022). An unsupervised deep learning ensemble model for anomaly detection in static attributed social networks. *International Journal of Cognitive Computing in Engineering, 3*, 153-160. https://doi.org/10.1016/j.ijcce.2022.08.002.

[12] Lin, J. C. & Wu, K. C. (2006, September). A model for measuring software understandability. In: *Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, pp. 192-192. IEEE.

[13] Khan, W. (2021). An exhaustive review on state-of-the-art techniques for anomaly detection on attributed networks. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, *12*(10), 6707-6722.

[14] Stevanetic, S. & Zdun, U. (2015, April). Software metrics for measuring the understandability of architectural structures: a systematic mapping study. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1-14.

[15] Husain, Mohammad Salman & Haroon, Dr. Mohammad. (2020). An enriched information security framework from various attacks in the IoT. *International Journal of Innovative Research in Computer Science & Technology, 8*(3). Available at: SSRN: https://ssrn.com/abstract=3672418

[16] Husain, Mohammad Salman. (2020). A review of information security from consumer's perspective especially in online transactions. *International Journal of Engineering and Management*

Research, *10*(4). Available at: SSRN: https://ssrn.com/abstract=3669577

[17] A.M. Khan, S. Ahmad & M. Haroon. (2015). A comparative study of trends in security in cloud computing. *Fifth International Conference on Communication Systems and Network Technologies, Gwalior, India*, pp. 586-590. DOI: 10.1109/CSNT.2015.31.

[18] Khan, W. & Haroon, M. (2022). An efficient framework for anomaly detection in attributed social networks. *International Journal of Information Technology*, *14*(6), 3069-3076.

[19] Khan, W. & Haroon, M. (2022). An unsupervised deep learning ensemble model for anomaly detection in static attributed social networks. *International Journal of Cognitive Computing in Engineering*, *3*, 153-160.

[20] Husain, M. S. & Haroon, D. M. (2020). An enriched information security framework from various attacks in the IoT. *International Journal of Innovative Research in Computer Science & Technology (IJIRCST)*.

[21] Husain, M. S. (2020). A review of information security from consumer's perspective especially in online transactions. *International Journal of Engineering and Management Research*, *10*.

[22] Siddiqui, Z. A. & Haroon, M. (2022). Application of artificial intelligence and machine learning in blockchain technology. In: *Artificial Intelligence and Machine Learning for EDGE Computing*, pp. 169-185. Academic Press.

[23] Shakeel, N., Haroon, M. & Ahmad, F. (2021). A study of wsn and analysis of packet drop during transmission. *International Journal of Innovative Research in Computer Science & Technology*.

[24] Khan, W. & Haroon, M. (2022). A pilot study and survey on methods for anomaly detection in online social networks. In: *Human-Centric Smart Computing: Proceedings of ICHCSC*, pp. 119-128. Singapore: Springer Nature Singapore.

[25] Haroon, M., Tripathi, M. M., Ahmad, T. & Afsaruddin. (2022). Improving the healthcare and public health critical infrastructure by soft computing: An overview. *Pervasive Healthcare: A Compendium of Critical Factors for Success*, pp. 59-71.

[26] Haroon, M., Tripathi, M. M., Ahmad, T. & Afsaruddin. (2022). Improving the healthcare and public health critical infrastructure by soft computing: An overview. *Pervasive Healthcare: A Compendium of Critical Factors for Success*, pp. 59-71.

[27] Khan, N. & Haroon, M. (2023). A personalized tour recommender in python using decision tree. *International Journal of Engineering and Management Research*, *13*(3), 168-174.

[28] Khan, A. M., Ahmad, S. & Haroon, M. (2015, April). A comparative study of trends in security in cloud computing. In: *Fifth International Conference on Communication Systems and Network Technologies*, pp. 586-590. IEEE.

[29] Haroon, M. & Husain, M. (2013). Analysis of a dynamic load balancing in multiprocessor system. *International Journal of Computer Science engineering and Information Technology Research*, *3*(1).

[30] Haroon, M. & Husain, M. (2013). Different types of systems model for dynamic load balancing. *IJERT*, *2*(3).

[31] Scholar, P. G. (2021). *Satiating a user-delineated time constraints while scheduling workflow in cloud environments*.

[32] Fenton, N. & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

[33] Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, *20*(3), 199-206.

[34] Lessmann, S., Baesens, B., Mues, C. & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, *34*(4), 485-496.

[35] Dewangan, S., Rao, R. S., Mishra, A. & Gupta, M. (2021). A novel approach for code smell detection: an empirical study. *IEEE Access*, *9*, 162869-162883.

[36] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. & Turski, W. M. (1997, November). Metrics and laws of software evolution-the nineties view. In: *Proceedings Fourth International Software Metrics Symposium*, pp. 20-32. IEEE.

[37] Zuse, H. (2013). *A framework of software measurement*. Walter de Gruyter.

[38] Lanza, M. & Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[39] Mendling, J., Reijers, H. A. & van der Aalst, W. M. (2010). Seven process modeling guidelines (7PMG). *Information and software technology*, *52*(2), 127-136.