IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received August 19, 2020, accepted September 1, 2020, date of publication September 7, 2020, date of current version September 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3022087

Efficacy of Inheritance Aspect in Software Fault Prediction—A Survey Paper

SYED RASHID AZIZ[1], **TAMIM AHMED KHAN**[1], **AND AAMER NADEEM**[2]
[1]Department of Software Engineering, Bahria University, Islamabad 44000, Pakistan
[2]Department of Software Engineering, Capital University of Science and Technology, Islamabad 45750, Pakistan

Corresponding author: Tamim Ahmed Khan (tamim@bahria.edu.pk)

ABSTRACT Software fault prediction (SFP) is a research area that helps development and testing process deliver software of good quality. Software metrics are of various types and are used in SFP for measurements. Inheritance is a prominent feature, which measures the depth, breadth, and complexity of object-oriented software. A few studies exclusively addressed the efficacy of inheritance in SFP. This provokes the need to identify the potential ingredients associated with inheritance, which can be helpful in SFP. In this paper, our aim is to collecting, organizing, categorizing, and investigating published fault prediction studies. Findings include identification of 54 inheritance metrics, 78 public datasets with various combinations of 10 inheritance metrics, 60% use of method level & use of private datasets, an increased number of studies using machine learning approaches. This study will facilitate scholars to studying previous literature on software fault prediction having software metrics, with their methods, public data sets, performance evaluation of machine learning algorithms, and findings of experimental results in a comfortable, and efficient way, emphasizing the inherited aspect specifically.

INDEX TERMS Object oriented paradigm, software inheritance metrics, software metrics, machine learning, software fault prediction.

I. INTRODUCTION

Measurement is needed to validate the effectiveness of software development process. The phrase software metrics describes measurements made on an artifact of software whereas a software artifact has two significant elements: the coded implementation, and the document of its design specification. The initially calculated McCabe, Halstead, and Albrecht metrics, presented during the 1970s, were typically constructed on the coded final software products. Examples of software science metrics include [1] function point analysis [2], and cyclomatic complexity metric [3], which predominated in the early 1980s to measure software product.

Worldwide software development expenditure, for year 2014, was 3.8 billion dollars which included 23% quality control and testing cost for business applications [4]. Early fault detection helps save costs, time, and reduce the complexity of the software because it is proportionate to the testing. It is a well known fact that extensive testing are impossible [5]. Testing cost sometimes amounts to over fifty percent of the entire software development cost. It is for these reasons that it is more feasible to detect and test classes with faults to produce software with better quality.

The faults are not uniformly dispersed within the software components. Some classes have a relatively high number of faults as compare to others and are clustered in a limited number of classes [6]. Source code quality is measured through internal metrics whereas the behavior or functionality of the software is measured by external metrics [4]. In general, these two types of metrics are utilized to assess the quality of the software to indicate the degree of reliability of the software. Presently in software engineering, numerous prediction approaches are being used in the research that includes prediction of reuse, prediction of testing effort, prediction of cost, prediction of security, prediction of faults, prediction of quality, and prediction of stress [7]. Out of these, software fault prediction is an emergent research domain where defective classes are identified during the initial phases of development project [8] by utilizing machine learning [9]. Many approaches make use of typical methods of machine learning, which consist of Support Vector Machines (SVM), Naive Bayes (NB) [10], Decision Trees [11], and Neural

The associate editor coordinating the review of this manuscript and approving it for publication was Claudio Agostino Ardagna.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Networks [12]. In SFP, these techniques are exercised by using metric measurements, and the fault information obtained by similar software projects [13] or previous versions to construct models to predict faults. Suppose, by using metrics to build a model of fault prediction [14] for the calculation of inheritance of software, cohesion, coupling, size, and complexity.

Typically, fault prediction process includes two stages. The first part is called, the training phase, while the part two called the prediction stage. In the first stage of training, the prediction model is constructed that utilizes method level or class level metrics of software with fault information associated with all the components of software programs. Later, the same model is used in the next version of the software for the prediction of fault proneness.

Methods of classification are utilized to put a label on classes as fault-free or faulty by employing metrics set with fault data. The software quality is improved by locating faulty classes in the software with the use of fault prediction models. The model performance influences the model technique [15], along with metrics [16]. Many scholars have built and endorsed machine learning, and statistical techniques on the models of fault prediction utilizing datasets, metrics, and reduction of feature techniques to make improvements in the performance of the models.

In the object-oriented paradigm, besides others, inheritance is an important feature. The metrics of inheritance are useful to recognize the class's complexity based on fault prediction [17]. Abreu, and Carapuca [18] mention, the larger the relation of inheritance, less a class is expected to inherit methods in larger numbers, therefore turn out to become further complex and consequently necessitating extra testing. Inheritance is helpful in the reuse, and many other aspects for example testability, complexity etc. [19]. It should be contained in limits to avoid technical hitches. In the literature, researchers suggested several metrics associated with inheritance to identify the fault tendency and quality software systems.The metric of inheritance defines the tree showing software inheritance, the order of classes with linkage within the master, and its child classes. Furthermore, specialization, and generalization offer code re-usability. It must utilize within a suitable range thus the software system does not turn out to be a complex one [19].

While exploring the SFP domain, it is observed that so much work has already been done on the properties supported by Object-oriented software like cohesion, coupling, etc. These are used either independently or in combination with other metrics. C&K metrics set is an example where these are used in combination and widely accepted by researchers [20]. It is observed that inheritance metrics are used in combine cases widely in C&K suite, however, exclusive usage and evaluation of inheritance metrics are missing. This spurs to conduct a study to focus specifically on inheritance to show the viability of its metrics in the context of SFP.

This paper aims to show up the available resource to draw the effectiveness of inheritance in SFP since it is not exclusively addressed in the research arena. This paper contributes to cataloging about 54 inheritance metrics defined so far in the literature. Object-oriented metrics, datasets, techniques, and performance measures used in software fault prediction. Also, identified 78 publicly available data sets that have inheritance metrics with various combinations. Lastly, discussion, and conclusion to show the effectiveness of inheritance in SFP.

Henceforth, the paper is planned into different Sections where:

- Section II explained the theoretical background of an object-oriented paradigm, software inheritance, SFP techniques, performance measures, and datasets.
- Section III depicted literature review where the survey of inheritance metrics, object-oriented metrics, their usage in SFP, datasets with inheritance metrics, algorithms, performance measures, and datasets used so far in the literature are enlisted.
- Section IV elaborates on discussions specifically focusing on the inheritance context.
- Section V discuss about Threat to Validity aspect.
- In the end, in Section VI, the conclusions, and future directions of the survey are provided.

II. THEORETICAL BACKGROUND

A. OBJECT ORIENTED PARADIGM

Ole-Johan Dahl and Kristen Nygaard from Norway are the innovators of the object-oriented language with their launch of Simula67 in 1967. The subsequent foremost development was Smalltalk, developed at XEROX by Palo Alto in the 1970s, followed by Eiffel who restored an earlier language to incorporate object-oriented capabilities, such as C++, Object Pascal, and Ada95.

Key feature in an object-oriented software is an object, which encloses both data named attributes, and functionality named methods. Besides, message passing, and inheritance functionalities. This is opposite to the traditional structured programming, where data is dealt with separately from the procedures that act on them. Objects are typically formed by an instantiation process, which utilizes a general template called a class. Classes may be master or root class, constructed in conjunction through the group of attributes, and methods. A child class gets features from a master class and may include or exclude functionality as preferred.

B. SOFTWARE INHERITANCE

Software metrics are foundation to quantify complexity, quality of software, and project costs with effort estimation. Function points and cyclomatic complexity are traditional metrics. These are being utilized in the paradigm of procedure language. Nonetheless, these may not merely be utilized in the context of object-oriented [21]. The procedural languages are less complex while comparing with object-oriented programming language [22]. The majority of studies specified hurdles in moving from a procedural approach towards an object-oriented paradigm [23].In object-oriented, it is problematic to

comprehend how inheritance, abstraction, and encapsulation associated with each other.

It is vital to distinguish between functionally oriented approach and object-oriented design principles. Consecutively to explain numerous features of object orientation to allow superior administration, and quality management [24]. Pressman purposes five conditions, in which metrics of object-oriented may be established [25].

- **Location.** It is linked with information tendency as soon as it is centralized.
- **Inheritance.** Inheritance allows the choice to create a new class. Permitting the methods, and attributes of one or more classes fully or partially.
- **Encapsulation.** Encapsulation denotes the objects encompass their attributes and data.
- **Object abstraction.** The method object abstraction authorizes the designer to focus merely on rudimentary with essential aspects of particular program sections.
- **Information hiding.** Information hiding means to hide the object structures having attributes, and data.

In the object-oriented paradigm, inheritance makes it possible for newly created objects to utilize the elements of earlier objects. The source of inheritance is a base class or superclass whereas a derived class or subclass, which inherits through a base class. The term secondary class and main class may also be utilized interchangeably for super class, and sub class. The sub class may possess his elements and methods, but it may also inherit properties and visible methods from the main class. Inheritance acquisition offers [26]:

- **Reusability.** Reuse is a method established through inheritance where super class public methods utilize into sub class without code rewriting.
- **Overriding.** The secondary class may rewrite the primary class methods so the desired execution of the primary class method is constructed in the secondary class.
- **Data hiding.** Inheritance offers a data hiding feature. in which super class marks a method as private therefore this method may not be used or modify by child class.
- **Extensibility.** Expand logic of the main class as per the business logic of the secondary class.
- **Overriding.** The secondary class may rewrite the primary class methods so the desired execution of the method of a primary class is constructed in the secondary class.
- **Maintainability.** In the case where the program is splinted into parts. It is comfortable to walk through the code.

The foundation of inheritance is an IS-A relationship In the object-oriented paradigm, which explains ''R is a Z type of thing''. Red is a color; the computer is a machine. The inheritance is unidirectional, ''the house is a building'', but ''the building is not a house''. The inheritance has further additional prominent features [26]:

- **Specialization.** Increasing a class functionality is described as specialization [27].
- **Generalization.** Distribution of commonalities among multiple classes is described as a generalization.

There are many forms of inheritance in the literature. These are described in subsequent lines as under [28]:

- **Single inheritance.** A case where a subclass only inherits with one main class is denoted as single inheritance.
- **Multiple inheritance.** In the case of multiple inheritance, a child class is expending or inheriting from numerous main classes. The issue in this type of inheritance is that child class should handle the dependencies on multiple main classes.
- **Hierarchical Inheritance.** In the case of hierarchical inheritance a main class is expended by several child classes.
- **Multi-level inheritance.** Multilevel inheritance indicates a method where a subclass expanding through a derived class, turning a derived class to a parent of the newly created class.
- **Hybrid inheritance.** Hybrid inheritance is a combination of multiple inheritance and multi-layer inheritance. As in multiple inheritance child classes are expended with two main classes. But these main classes are not base classes rather derived classes.

The advantages of inheritance features in the object-oriented paradigm are mentioned as under:

- The Inheritance aspect supports reusability. After a class derives or inherits a new class, it may gain access to all the functionality of the inherited class.
- The reusability boosted reliability. Since the code of the supper class has previously tested and debugged.
- Since the present code is being reused so it shows the way to reduce efforts on software development, and subsequently on the cost of maintenance.
- Inheritance causes the subclasses to pursue a single general interface.
- Inheritance aids to decrease code duplication, and promotions extensibility of code.
- Inheritance makes the possible formation of class libraries.

Similarly, the drawbacks of inheritance features are explained as under:

- Inherited functions act normally sluggish since there is indirection.
- Misuse of inheritance might steer to bad results.
- Frequently, data members from the superclass remain not being used, which might result in wasted main memory.
- Inheritance enhances the coupling among the superclass with the child class. So a modification in superclass might affect every subclass.

C. SOFTWARE FAULT PREDICTION

Software fault prediction process typically involves two phases which are denoted as training phase and the second

is prediction phase where in the first phase, a model for prediction is constructed, which utilizes method or class level metrics of software with fault information associated with every single module of the software. Later, the same model is used to predict faulty classes in a new version of the software. SFP is effective to enhance the quality of software along with reducing the cost of testing. Also, it assists testing teams to limit testing on faulty classes only. Fault prediction in software possibly lays down yardstick for knowing, which areas may require attention. Many software fault prediction methods have been utilized [29], which contribute to three main elements [30]; Set of features, Label of Class, and finally the Model.

The set of features are consisting of single or multiple metrics derived through artifacts of software, it is believed that these are useful to predict labels of class. All of the metrics are grouped into the product, project, and process metrics. It is observed that metrics of the product are mostly utilized in the research arena [31]. The further level of product metrics includes method, class, and file levels where overall 60% method-level metrics are utilized followed by 24% class-level metrics [15]. Metrics of the product also consist of, volume, design, code, and complexity metrics. The performance of SFP heavily depends on these metrics. Researchers have assessed the utilization rate of metrics in [32] where highly use product metrics in software fault prediction are Halstead [1], McCabe [3], LoC in structural programming, and in object-oriented paradigm C&K metrics suite [33]. The subject metrics are become the approved standard metrics in software fault prediction. PROMISE and D'Ambros [34] are frequently used datasets repositories having these metrics. Both the repositories encompass datasets of about fifty-two percent of the research paper published after 2005 [9]. Since these datasets are publicly available therefore it is frequently used. The other reason is the non-availability of bug's data of industrial software.

The second extremely significant component in SFP is the class label, which contains the real value of the metrics. In SFP, the faulty / fault-free instances are marked as nominal-binary or continuous to mark the number of faults in any occurrence. Although, use of continuous labels exist in the literature [35], but dominating class labels in SFP or nominal class labels [7], [29].

The third important pillar of SFP is model building, which is a connection in-between class label, and feature set. This may be utilized with the assistance of Statistical methods, Machine learning algorithm (ML), or even expert opinion [15] where ML is an excessively used technique for model building [30]. It expressively expands the accuracy of classification [36]. In SFP many algorithms of ML are used. Malhotra et.al compared the performance of these ML algorithms who deduce Random forest, and Bayesian networks are outperformer as compare to other ML algorithms [9].

D. FAULT PREDICTION TECHNIQUES

Fault prediction is topic of numerous researches studies. A number of methods are recommended for the prediction of software faults, which include machine learning, and Statistical methods. Both methods are explained in detail in subsequent lines.

1) STATISTICAL METHODS

Several strategies of statistics are utilized to identify a simple straight mathematical numerical equation, which certainly recognizes in what way classification would be achieved. Kapila et. [37] utilized two approaches of statistic to do his study that includes univariate binary logistic regression (UBR), and logistic regression. Both the approaches are beneficial to investigate data contains binary variables. In the technique of Bayesian inference [37], a model strategy is to correlates metrics with software faults, and faults propensity. The regression analysis method is extensively applied for bad smell prediction, and method linear regression is applied in the case when the dependent variable available merely for dual classes.

2) MACHINE LEARNING

Typically machine learning focus on design and development of algorithms with techniques. That extricate rules, and patterns from massive databases. Neural Networks (NN) are previously being utilized within the software to create reliability growth models for the prediction of total modification or reusability metrics. The NN model is trained to do again a stipulated series of exact instances classification, as an alternative to creating formulas or rules. Mahajan et. al [38] mentioned that machine learning methods are beneficial to determine software faults as complete processing is performed by the computer.

Multilayer Perceptron (MLP) is used to manage classes with faults. Radial Base methods are utilized for the categorization of faults as per dissimilar faults classes [39]. Xing *et al.* outline the importance of the SVM model which can be utilized for an ample quantity of data. It brings better Accuracy while making comparisons with other techniques to predict software quality. The outcomes of SVM is at a depleted level on public data sets [38].

E. MACHINE LEARNING ALGORITHM

1) SUPPORT VECTOR MACHINE(SVM)

Vapnik accompanied by his fellow researchers suggested SVM. It has been extensively explored and used in several disciplines. The fundamental concept of this model is to find a similar gap among the two objects/classes by contemplating a distance metric among both of them. The gap among these entities/classes are by practice stated as a function of vectors of its feature [40]. Additionally SVM may be utilized to manage unbalanced classes [41].

2) DECISION TREE

The classifiers of decision trees utilize comparisons to segregate dissimilar occurrences of a group into suitable classes. Classification falls into supervised learning that primarily, a collection of acknowledged occurrences, so-called learning sets, are presented into a model. The model categorizes individual instances of the collection, relates every type with the attributes of each instance, and ascertains to, which group all instances be appropriate. Established upon whatever the trained model understood throughout the phase of learning, you may categorize the instances of a set [42], which were not ever realized earlier.

One of the applications of the use of decision trees refers to form algorithms of pattern recognition [42]. The second type of application is an analysis of images in machine learning that include detection of tumor cells, and brain tumors examples [43].

3) AdaBoost

The AdaBoost algorithm is superior among known algorithms for building a set of the classifier [44]. Each instance of the learning data set is measured. The early measurement is defined in weight (wi) = 1 / n, where wi represents training occurrence, and n is the sum of training occurrence. The furthermost applied algorithms with AdaBoost are single-level decision trees. Since decision trees are concise and encompass single classification decisions, so that named decision stumps. A dull classifier called the decision segment is formulated upon the learning of facts utilizing the weighted samples. The algorithm AdaBoost makes a powerful classifier by applying the weights via a recurring method [24].

4) RANDOM FOREST(RF)

Breiman [45] proposed a novel and promising classifier in 2001 named as RF. It offers numerous benefits [32] for instance its execution effectiveness on huge databases to manage thousands of input variables. Then presenting approximations demonstrating which is a significant variable in a classification phase.

5) K NEAREST NEIGHBORS (KNN)

While discussed in the research paper [46], amongst the several approaches for the recognition of supervised statistical patterns, the Nearest Neighbor rule attains a continuous high performance, deprived of a priori suppositions regarding the dispersion through, which training instances are take out. It comprises a collection of positive, and negative case training. A novel occurrence is categorized by gauging the gap with the closest training data. The classifier KNN stretches out this notion by captivating the neighboring spots and giving the majority mark. It is usual to opt for a quantity for the k which must be a lesser number with an odd figure to breakdown draws generally 1, 3, or 5. The greater values of k support in decreasing the impacts of the anomalous elements in the training dataset. The selection of k is usually made by cross-validation [46].

6) ARTIFICIAL NEURAL NETWORKS (ANNs)

ANNs is applied for classification purpose which is not similar to Naive Bayes, and tree-based algorithm. Since none of them need the discretization of the dataset. These may even manage a solo feature that appears in the dataset. ANNs is an utmost operative technique, utilized for a classification task that is done on metrics of object-oriented [47]. The state-of-the-art tendency in SFP is the explicit usage of ANNs [5], [48]. Moreover, the choice of method to model is appointed though considering the numerical details of the training set. It is a dataset that guides towards ANNs, and the response we deduce outcomes.

F. PERFORMANCE MEASUREMENT

Evaluating machine learning models require performance measures, which are explained as follows:

1) ACCURACY

Accuracy (equation 1) gauges the fraction of the files categorized correctly, to the total of files. Accuracy, however, excludes a thorough evaluation for example the number of correct labels of various classes, and in this sense investigators, to assess the model, utilize the F1 Score along with recall and precision that is explained in the following lines.

$$Accuracy = \frac{TP + TN}{(TP + TN) + (FP + FN)} \tag{1}$$

2) PRECISION

Precision (equation 2) gauges the fraction of files that were classified correctly as faulty over the totality of files categorized as either faulty or faulty-free. Simply, Confidence or Precision as it is called in data mining expresses the ratio of predicted occurrences these are indeed actually defective files. This measures how decent a prediction model is at detecting real defective files.

$$Precision = \frac{TP}{(TP + FP)} \tag{2}$$

3) RECALL

Recall explained in equation 3 computes the fraction of faulty instances, which are accurately labeled faulty by the sum of faulty instances existed. Sensitivity or Recall (as it is termed in Psychology) is the fraction of actual faulty instances which are properly predicted as faulty.

$$Recall = \frac{TP}{(TP + FN)} \tag{3}$$

4) F1 SCORE

The F1 Score is calculated by getting the (weighted) harmonic average of recall, and precision as demonstrated in the

equation (equation 4).

$$F1Score = 2 * \frac{Precision * Recall}{(Precision + Recall)} \quad (4)$$

5) SPECIFICITY (TRUE NEGATIVE RATE (TNR)

TNR is the quantity of accurately predicted classes likely to be fault-free among entirely real classes likely to be fault-free.

6) PF (FALSE POSITIVE RATE (FPR)

FPR is the fraction of all classes likely to be fault-free. These are erroneously predicted as faulty.

7) FNR (FALSE NEGATIVE RATE)

FNR is the proportion of faulty classes, which are categorized as non-fault prone.

8) ROC (RECEIVER OPERATING CHARACTERISTIC CURVE), AND AUC (AREA UNDER THE CURVE)

ROC, and AUC is mapped out with the values of recall on the y-axis, and the values of 1-specificity on the x-axis. The usefulness of the model is determined by quantifying the area underneath the ROC curve.

9) BALANCE

It denotes the best possible cutoff point of (sensitivity, pf) for example the normalized Euclidean distance as from (0,1) position in the ROC curve.

10) COMPLETENESS

It is the summation of all faults categorized as faulty classes above the sum of authentic faults within a system.

11) F-MEASURE

F-measure is the harmonic mean of precision and sensitivity.

12) CONFUSION MATRIX

Equation 5 represents a simple cross-tabulation of class tags mapped to those perceived in the field or reference data for a sample of cases in certain locations. The matrix provides a visual foundation for the evaluation of accuracy (Campbell, 1996; Canters, 1997), as well as for the description of classification accuracy, and characterization errors, that may help to improve the classification or derivative approximations through it [49].

$$ConfusionMatrix = \begin{Bmatrix} TP & FN \\ FP & TN \end{Bmatrix} \quad (5)$$

G. DATASETS IN SFP

Numerous data sets have been utilized in software fault prediction. These may be grouped into private, public, partial, and unknown dataset categories [15]. The public data sets usage has risen from 31% to 52% since 2005 onward [9]. Fault information usually not accessible for private projects but there are datasets openly available with fault information, these may be downloaded for free. There are several bug libraries, from this Tera-PROMISE, repositories, and D'Ambros repositories that are often used for predicting faults [34].

A publicly accessible repository called Tera-PROMISE offers a large data set for multiple projects. An earlier version of this repository has named as NASA repository [50]. The datasets of NASA are an important source of the Tera-PROMISE repository since its datasets are an extensively utilized library of software fault prediction. About 60% of the papers published from 1991 to2013 make use of this library [51]. The repository of PROMISE offers metrics associated with the process, and product with digital nominal class labels for building classification, and regression models.

The repository of D'Ambros holds data sets of five open-source software applications, these are: Equinox, Eclipse JDT Core, Lucene, Eclipse PDE UI, Mylyn, and Framework.

III. LITERATURE REVIEW

Fault prediction of software is a vital field of study and the topic of several earlier types of research. These researches generally give off fault prediction models that permit software engineers to concentrate on development activities on source code failures, increasing software quality, and boosting resource utilization. In this section, the focus is on the object-oriented paradigm, specifically on the inheritance aspect, which helps predict faults. There is no systematic review of the literature in this study, but to investigate in what way distinct inheritance features are beneficial in software fault prediction.

A. INHERITANCE METRICS

Inheritance is one of the utmost strong facets of an object-oriented paradigm. Programming deprived of inheritance is not object-oriented programming rather termed as programming with an abstract data type. During the last decade or so, numerous metrics of object-oriented have been suggested by researchers, and in fact, only a few of them are being utilized by many establishments as a measurement to accomplish quality [52]. The metrics of inheritance are constructed on:

- Deepness of the inheritance tree.
- Over-all quantities of classes inherited in a program.
- Classes quantities indirectly or directly inherited by a class.
- Classes quantities indirectly or directly inheriting from a class.
- Methods quantities inherited by a class.
- Methods quantities overridden by a class etc.

The complexity of inheritance depends not solely upon the number of inherited classes, but additionally upon the number of inherited methods, together with their complexity, as endorsed by Abreu and Carapuca, [18]. The higher the inheritance relationship, the more complex the class, and the more it requires testing because the number of methods the class may inherit increases. Methods with complex decision-making structures are difficult to test, maintain, and

are prone to errors [18]. They allow complexity metrics based upon the previously mentioned criteria to identify possibly large methods or classes, assist with the preparation of review, and testing jobs, which is a key part of the suggested software quality process that mentions [53]. There is no complete object-oriented inheritance metric that addresses all of these issues. The main motivation for this paper is to collect and organize the inheritance metrics defined so far by researchers in the field of SFP.

1) DEPTH OF INHERITANCE TREE (DIT) [33]

The depth of inheritance metrics is to determine how much ancestor classes might influence the metrics of this class. In the situation of multiple inheritances, DIT will be the greatest length from the root tree to the node.

$$DIT = Max\ inheritance\ path\ from\ the\ class\ to\ the\ root \quad (6)$$

- The deepest class in the hierarchy will inherit a large number of methods, so predicting their behavior will be difficult.
- Into the design of software, at the bottom of the trees, will create more complexity since several classes, and methods have been used.
- At the bottom of a specific class within the hierarchy, it is possible the larger the reuse of methods which are inherited.

Visual Studio .NET suggested that DIT ≤ 5, and some sources recommend up to 8. Extremely deep class hierarchies are very difficult to build. It is noted that a rise in DIT will increase the degree of errors, and eventually decrease the software quality.

2) NUMBER OF CHILDREN (NOC) [33]

- Growing the NOC will rise in the reuse of methods since inherited methods are a mode of reusability.
- The larger the subclass, the larger the probability of inadequate abstraction of the parent class. In the case where class contains a large number of subclasses, that might be a situation of misappropriation of the child class.
- The quantity of subclasses provides the objectives of the possible effect of the class has on design. In the case where a class has a huge number of subclasses, it may require more method testing in the class.

$$NOC = number\ of\ immediate\ sub-classes\ of\ a\ class \quad (7)$$

High NOCs show fewer faults since this might be due to excessive reuse that is suitable.

The breadth of a class hierarchy is measured by NOC since the depth is measured by the highest DIT since the depth is usually better as compared to the breadth.

3) ATTRIBUTE INHERITANCE FACTOR (AIF) [54]

Attribute Inheritance Factor is the proportion of the total of inherited attributes of all classes in the system to the sum of accessible attributes of all classes. It is a system-level metric that calculates the scope of attribute inheritance in the system. Mathematically AIF is calculated as follows:

$$AIF = \frac{\sum Ai(Ci)}{\sum Aa(Ci)} \quad (8)$$

4) METHOD INHERITANCE FACTOR (MIF) [54]

Method Inheritance Factor is the fraction of the total inheritance methods of all classes of the system to the sum of available methods of all classes. MIF is a system-level metric. It is recommended to keep the MIF between 0.25, and 0.37. Mathematically MIF is calculated as follows:

$$MIF = \frac{\sum Mi(Ci)}{\sum Ma(Ci)} \quad (9)$$

Subclasses that inherit more than one method or attribute from their parent class contribute to a higher MIF or AIF. Redefining methods or properties of its parent class, and adding subclasses of new classes may help reduce MIF (AIF). An independent class with no inheritance and no children helps to reduce the MIF (AIF).

The acceptable range of MIF is from twenty percent to eighty percent, and the suitable limit of AIF is from zero percent to forty-eight percent [55]. An alternative opinion about the AIF is that it must idyllically be a zero since every variable would be defined as private.

5) NUMBER OF METHODS INHERITED (NMI)

The metric NMI calculates the quantity of inherited methods by a child class.

6) NUMBER OF METHODS OVERRIDDEN (NMO) [56], [57]

A larger quantity of methods overridden points out a issue in design, signaling these methods overridden as a last-minute design. Since it is recommended, a child class must be a specialization of all its superclasses, which resultantly create a new exclusive name for the methods.

7) NUMBER OF NEW METHODS (NNA) [56], [57]

The typical expectancy of a child class is how to specialize (or add) superclass objects. If there is no method of the same name in a superclass, the said method is described as an appended method in the child class.

8) INHERITANCE COUPLING (IC) [58]

The IC metric is a connection between classes which allows the usage of formerly defined objects, comprising variables, and methods. Inheritance reduces the complexity of a class by decreasing the number of elements of a single class, however, due to this maintenance, and design became tough. Inheritance improves effectiveness, and reuse while using present objects. Concurrently, inheritance has led to complexities in software testing and understanding. This implies that inheritance coupling affects many quality attributes such as complexity, efficiency, reusability, testability, understandability, and maintainability.

9) NUMBER OF INHERITED METHODS (NIM)

The metric NIM is a fair metric that describes the quantity of behavior a class is supposed to reuse. It calculates the sum of methods a class may retrieve from his parent class. The higher the quantity of inherited methods, the superior the class reusability by the subclasses. Evaluating this metric with the number of parent classes sent, and the way it is sent to undefined methods in the class may be interesting because it indicates the number of internal reuses between the calling class, and its parent class. It may be an incoming call to an incoming method, although it is difficult to statistically evaluate it. Besides, inheriting large superclasses may be problematic because merely a subset of the methods may be required/desired within the subclasses. So it is a threshold of single inheritance basing on the object-oriented programming language.

10) FANIN AND FANOUT METRIC [59]

The Fanin/Fanout metrics first defined by Henry and Kafura [59], that is "module-level" metrics. Subsequently, these are expanded for the object-oriented paradigm. For each class A we record its fan-in as the count of classes, which use features of class A. Similarly, the fan-out for class A is the count of classes used by A.

Sheetz, Tegarden, and Monarchi derive a set of primitive counts. The complexity between modules called Structural complexity has been recognized as a significant part of structured system complexity. Several researchers have used module-defined fan-in and fan-out [60]. Expanding these thoughts to variables in object-oriented systems looks suitable, and up-front. The number of methods that use variables (fan input variable) is quite identical to the number of modules that call the module (input input), and the number of objects accessed by the variable (variable output input), and the digital module called by the module (fan).

11) FAN-DOWN

The metric Fan-Down is the quantity of classes under the hierarchy of classes (subclasses).

12) FAN-UP

The metric Fan-Up is the quantity of classes over in the hierarchy (superclass).

13) OBJECT-To-ROOT DEPTH

The metric Object to Root Depth is the largest number of stages within the hierarchy which is directly over the class object.

14) OBJECT-TO-LEAF DEPTH

The metric Object to Leaf Depth is the highest number of steps in the hierarchy of object which is under the class object.

15) MEASURE OF FUNCTIONAL ABSTRACTION (MFA)

The MFA metrics are the share between the number of methods inherited by a class to the sum of methods of the class. Its limits are between zero to one.

16) IFANIN

The metric IFANIN calculate the sum of immediate base classes of the hierarchy.

17) NUMBER OF ANCESTOR CLASSES (NAC) [61]

Number of Ancestor Classes (NAC) metric quantifies the over-all quantity of child classes, which a child class inherits in the class inheritance hierarchy. NAC is of the same type as `{dit}`, since it calculates the number of ancestors of a class. Li [62] validated the piece of evidence that the unit of the NAC metric is "class" as the trait obtained by the NAC metric is the number of other classes.

18) NUMBER OF DESCENDENT CLASSES (NDC) [61]

Number Descending Classes (NDC) metric specifies the overall quantity of child classes of the class. It is a substitute for NOC. The metric NOC calculates the extent of a class's influence on its inheritance subclass. Li [62] argued that NDC metrics obtain class attributes superior than `{noc}`.

19) CLASS-TO-LEAF DEPTH(CLD)

Tegarden *et al.* proposed the metric CLD, which is the highest quantity of stages in the hierarchy that are underneath the class [63].

20) NUMBER OF ANCESTOR(NOA)

Tegarden *et al.* proposed the NOA metric, which is the number of classes that a specified class directly or indirectly inherits from [63].

21) NUMBER OF PARENTS(NOP)

Lake and Cook suggested the metric, which is the number of classes that a particular class inherits directly [64].

22) NUMBER OF DESCENDANTS(NOD)

Lake and Cook [64] suggested the metric NOD, which is the number of classes that directly or indirectly inherit from a class.

23) DEPTH OF INHERITANCE TREE OF A Class (DITC)

Rajnish *et al.* [65], [66] introduced the metric. The DITC class inheritance hierarchy is calculated by totaling the protected, private, public inherited attributes, and the protected, private, public, and legacy methods at every stage [66]. The DITC metric of a class at the respective stage is measured as follows:

$$DITC(C_J) = \sum_{i=1}^{L} LEV_i * i \tag{10}$$

where, LEVi = Method (Ca) +Attribute (Ca)

Ca = A class in the ith level of the class inheritance hierarchy. Attribute (Ca) = Calculate the sum of private, protected, public, and inherited attributes inside a class in the class inheritance hierarchy at each stage.

Method (Ca) = Calculate the sum of private, protected, public, and inherited methods contained by a class in the class inheritance hierarchy at each stage.

L = Total tallness in the class inheritance hierarchy. The most extreme distance from the last node for example top level in the class inheritance hierarchy, ignoring any shorter ways if there should be an occurrence of multiple inheritance is being utilized.

24) CLASS INHERITANCE TREE (CIT) METRIC

Rajnish *et al.* [65], [66] proposed the (CIT) metric, which is used to calculate the class inheritance tree [66]. The key objective of (CIT) is to calculate how the class is inherited by various classes, and in what way class inherits numerous classes at any level in the inheritance tree. CIT is expressed as under:

$$CIT(C_i) = \sum_{i=1}^{L} CIN(C_i) + COUT(C_i) \tag{11}$$

where Ci represent a class at the ith stage in the inheritance tree. The value of CIN (Ci) = 1 in case Ci is inherits various classes in the inheritance tree, 0 in other case. The value of COUT (Ci) = 1 in case Ci is inherited by various classes in the inheritance tree, 0 in other case.

25) INHERITANCE COMPLEXITY OF CLASS(ICC) [67]

Sandip *et al.* [56], [68] presented metric for evaluation. The ICC metric is explained in [68], and is explained in equation 12 in the following lines:

$$\begin{aligned} ICC(C_i) &= M(C_i) + A(C_i) + IF(C_i) \\ IF(C_i) &= \frac{NoOfClasses\ InheritedDirectlyByC_i}{1 + NoOfClassesInheritedDirectlyByC_i} \end{aligned} \tag{12}$$

where ICC represents the quantity of a class of an inheritance tree.

Ci = Represents the classes at the ith stage in an inheritance tree. A (Ci), calculate the quantity of private, protected, and public inherited attributes at every stage in an inheritance tree.

M (Ci) = calculate the quantity of private, protected, and public inherited methods at every stage in an inheritance tree.

26) INHERITANCE COMPLEXITY OF TREE (ICT)

The ICT metric is described in [67] study, and is measured as per equation 13:

$$\begin{aligned} ICT(C_i) &= \frac{M(C_i) + A(C_i) + IF(C_i)}{N} \\ IF(C_i) &= \frac{NoOfClasses\ InheritedDirectlyByC_i}{1 + NoOfClassesInheritedDirectlyByC_i} \end{aligned} \tag{13}$$

N = Sum of classes in an inheritance tree.

27) NUMBER OF METHODS (NOM)

Quantity of Methods declared in a class.

28) NUMBER OF ATTRIBUTES (NOA)

Quantity of Attributes declare in a class.

29) REUSE RATION (RR)

$$RR = \frac{No\ of\ Subclasses}{Total\ No\ of\ Classes} \tag{14}$$

30) SPECIALIZATION RATION (SR)

$$SR = \frac{No\ of\ Subclasses}{No\ of\ Superclasses} \tag{15}$$

31) NUMBER OF INHERITED ATTRIBUTE(NIA)

Quantity of inherited attribute in a class.

32) NUMBER OF INHERITED METHOD (NIM)

Quantity of inherited Methods in a class

33) NUMBER OF OVERRIDDEN METHODS(NoVM)

Quantity of overridden methods in a class.

34) INHERITANCE TREE DEPTH (ITD)

Classes are designed for inheritance purposes, hierarchically in a tree structure. The depth of a class within the inheritance hierarchy is the highest distance from the class node to the root of the tree, and is computed through the quantity of ancestor classes.

$$ITD = max\{Inheritance\ Tree\ Path\ Length\} \tag{16}$$

The deep a class is in the inheritance hierarchy, the additional protected, and public methods it is probably attaining, making it extra complicated. Deeper trees specify more complexity of the design. Thus, deeper class inheritance is neither needed nor required. The tests are more problematic, and the world doesn't normally hold considerably specializations. Deeper hierarchies are also a theoretical integrity apprehension since it turns out to be hard to ascertain specialization from, which class. This metric principally appraises reusability. Besides also associate with understandability, and testing ability [69].

35) AVERAGE DEGREE OF UNDERSTANDABILITY (AU)

Metric of F. T. Sheldon et al defines understandability, the ease of understanding a structure of the program, or the structure of class inheritance [70]. To compute AU, firstly expressing the level of class understandability, which is referred to by U, and explained in equation 17.

$$U\ of\ class\ Ci = PRED(Ci) + 1 \tag{17}$$

In the above equation C_i is i^{th} class.

$PRED(C_i)$ is the overall number of ancestors of ith class.

Next, the Total Degree of Understandability (TU) of a class inheritance tree is expressed as under:

$$TU\ of\ a\ class\ Inheritnce = \sum_{i=1}^{t}(PRED(C_i)+1) \quad (18)$$

In the above equation t is the sum of the classes in the class inheritance tree. Finally, the Average Degree of Understandability (AU) of a class inheritance tree is described as under:

$$AU\ of\ a\ class\ Inheritnce = \frac{\sum_{i=1}^{t}(PRED(C_i)+1)}{t} \quad (19)$$

36) AVERAGE DEGREE OF MODIFIABILITY (AM)

F.T. Sheldon et al defines Average Degree of Modifiability(AM) metrics. The definition of modifiability in the comfort by, which modification or amendments may be incorporated into program structure or in a class inheritance hierarchy [70]. To compute AM, firstly declare the level of class modifiability refer to by M. The AM is calculated as mentioned in equation 20.

$$M\ of\ a\ class\ Ci = \frac{U(Ci)+SUCC(Ci)}{2} \quad (20)$$

In the above equation C_i is i^{th} class.

$SUCC(C_i)$: the overall quantity of descendants of class i.

Next, the total degree of modifiability (TM) of a class inheritance tree is defined as follows:

$$TM\ of\ a\ class\ Inheritnce = TU + \sum_{i=1}^{t}(SUCC(C_i)/2) \quad (21)$$

where t is sum of classes in the class inheritance hierarchy tree. Finally, the Average Degree of Modifiability (AM) of a class inheritance tree is calculated as per equation mentioned as under:

$$AM = AU + \sum_{i=1}^{t}(SUCC(C_i)/2)/t \quad (22)$$

37) AVERAGE INHERITANCE DEPTH (AID)

Henderson-Seller stated that the AID metric of a class is computed by a following equation:

$$AID = \frac{(\sum Depth\ Of\ each\ Class)}{Number\ Of\ Classes} \quad (23)$$

38) DERIVE BASE RATIO METRIC (DBRM)

DBRM is the share of the sum of derived classes to the sum of root classes in the class inheritance tree. DBRM is described as underneath:

$$DBRM = \frac{\sum_{i=1}^{N} TD(C_i)}{\sum_{i=1}^{N} TB(C_i)} \quad (24)$$

In the above equation $\sum_{i=1}^{N} TD(C_i)$: Sum of derived classes in the class inheritance tree.

$\sum_{i=1}^{N} TB(C_i)$: Sum of root classes in the class inheritance tree.

N: Sum of classes in the class inheritance tree.

39) AVERAGE NUMBER OF DIRECT CHILD (ANDC) METRIC

The ANDC metric is the percentage of the overall quantity of instant child with the overall quantity of classes in the inheritance tree. The calculation of the ANDC metric is explained as under:

$$ANDC = \frac{\sum_{i=1}^{N} TDC(C_i}{N} \quad (25)$$

where $\sum_{i=1}^{N} TDC(C_i)$: Sum of instant child in the class inheritance tree.

N: Sum of classes in the class inheritance tree.

40) AVERAGE NUMBER OF INDIRECT CHILD (ANIC) METRIC

The ANIC metric is the ratio of sum of indirect child to the sum of classes in the inheritance tree. ANIC metric is described as follows:

$$ANIC = \frac{\sum_{i=1}^{N} TIC(C_i}{N} \quad (26)$$

$\sum_{i=1}^{N} TIC(C_i)$: Sum of indirect child in the class inheritance tree.

N: Sum of classes in the class inheritance tree.

41) CLASS COMPLEXITY DUE TO INHERITANCE (CCI)

The idea of Class Complexity due to Inheritance (CCI) metrics is to forecast the superiority of a class in terms of reusability, understandability, maintainability, and testability. Class Complexity due to Inheritance (CCI) may be measured as mentioned in equation 27.

$$CCI_i = \sum_{inherit=1}^{K} CCI_{inherit} + \sum_{l}^{j=1} MC_j \quad (27)$$

In the above equation CCI_i is the complexity of an ith class due to inheritance. k is the number of classes, and ith class is directly inheriting class. CCI inherit the complexity of an inherited class. l is the number of methods excluding constructors, destructors, pure virtual function, the ith class has. MCj is the complexity of the jth method in the ith class, and it may be computed by utilizing a recently suggested method complexity metric (MC). The classes with greater CCI quantities are complex. Thus considered further prone to errors.

42) METHOD COMPLEXITY (MC)

Method Complexity is constructed on McCabe's cyclomatic complexity [3]. Besides it also takes into consideration, control structure's depth. MC metrics are described as under:

$$MC_j = P_j + D_j + 1 \quad (28)$$

In the aforementioned equation MC_j is the complexity of jth method, P_j is the number of predicates j^{th} method has D_j is the highest depth of control structures in j^{th} method; the value of $D_j = 0$ in the case where not nested control structures exist; if there is one inside another then $D_j = 1$, and so on ...

43) AVERAGE COMPLEXITY OF A PROGRAM DUE TO INHERITANCE (ACI)

The use of the average complexity of a program due to inheritance metrics (ACI) is to forecast the quality of design of the program having multi-classes. ACI metrics described as under:

$$ACI = \frac{\sum_{i=1}^{n} CCI_i}{n} \tag{29}$$

where: CCI_i is the complexity of an ith class due to inheritance, n is the sum of the classes in the program.

44) NUMBER OF METHODS EXTENDED (NME)

Abreu & Carapuca defined NME metrics. The number of methods extended (NME) i.e. redefined in child class by invoking a method with a similar name on a parent class.

45) TOTAL CHILDREN COUNT (TCC)

Abreu & Carapuca defined quantity of classes that inherit directly from a class is the TCC of that class.

46) TOTAL PROGENY COUNT (TPC)

Abreu & Carapuca defined quantity of classes that inherit directly or indirectly from a class is the TPC of that class.

47) TOTAL PARENT COUNT (TPAC)

Abreu & Carapuca defined The number of super classes from, which a class inherits directly is the TPAC of that class.

48) TOTAL ASCENDANCY COUNT (TAC)

Abreu & Carapuca defined the number of super classes from, which a class inherits directly or indirectly is the TAC of that class.

49) TOTAL LENGTH OF INHERITANCE CHAIN (TLI)

Abreu & Carapuca defined the sum of edges in the inheritance hierarchy graph.

50) CLASS COMPLEXITIES

There are two types of class complexities: Member Complexity (MC) and Relational Complexity (RC). A class consists of data members, methods, and relationships with other classes. Complexity due to data members, and methods are called member complexity, and complexity of class due to relationships is called relational complexity. Therefore, complexity (C) of a class may be denoted as

$$C = MC + RC \tag{30}$$

The complexity (MC) due to the member function of the class depends upon the logic of the method whereas relational complexity (RC) of the class depends upon coupling and inheritance. Higher complexity causes higher fault proneness.

51) DEGREE OF INHERITANCE OF A CLASS (DIC)

DIC is constituted for two foremost characteristics of inheritance: with the growth in the level of inheritance, reuse rises to three levels of depth, but further, than three-level, the capability for reuse has a tendency to drop as it represents a deprived subclassification. Therefore, for each level of depth, if it increases further than three, there is a penalization factor that rises (DIC increases) and affects a reduction in reuse. The quantity of inherited methods or attributes plays a significant role in class inheritance and offers a clearer image of the inheritance factor. Inheritance of a class (DIC) is quantified in terms of methods and attributes such as:

$$DIC = \left\{ \begin{array}{l} \text{N_O_Inher_Attb * (4-Lvl) if lvl} \leq 3 \\ \text{N_O_Inher_Attb * (Lvl-3) if lvl} \geq 4 \\ \text{N_O_Inher_Mth * (4-Lvl) if lvl} \leq 3 \\ \text{N_O_Inher_Mth * (Lvl-3) if lvl} \geq 4 \end{array} \right\} \tag{31}$$

In the above equation, the level is the highest distance from the class to the root in an inheritance hierarchy, disregarding any concise route in the situation of multiple inheritance. Quantity of inherited attributes is a summation of attributes public, and protected methods within a class that are inherited from their instant superclass. The degree of inheritance of a class (DIC) is one of the main constructions to improve reuse and, thus, the software design quality.

52) EXTENDED DERIVED BASE RATIO METRICS (EDBRM)

This EDBRM make available the average quantity of the percentage of derived, and root shared attributes from each class in the class inheritance tree as described in the following equation:

$$EDBRM = \sum_{i=0}^{N} DF(C_i)/BF(C_i)/N \tag{32}$$

53) EXTENDED AVERAGE NUMBER OF DIRECT CHILD METRICS

The Extended Average Number of Direct Child Metrics acronym as EANDC, calculates the averages of shared methods and attributes to its direct sub classes. When the number of shared properties, and directly connect sub classes are higher, the more shared each class, and it needs additional testing to be performed to guarantee that it does not disrupt the sub classes after modification are incorporated. The equation mentioned below is used to calculate EANDC:

$$EANDC = \sum_{i=0}^{N} (TSP(C_i) * TDC(C_i))/N \tag{33}$$

54) EXTENDED AVERAGE NUMBER OF INDIRECT CHILD

The metrics EANIC calculates the averages of the number of shared methods and attributes to its indirect sub classes. The Class, which is at the deepest in inheritance tree has a greater EANIC number as compare to the one in inheritance tree with a lower depth. The depth quantity of a class inheritance tree must be more than three to calculate these metrics. The classes, which have depth quantity fewer than three may not have any indirect classes, hence these have zero value of

EANIC. The EANDC may be calculated through the equation mentioned below.

$$EANDC = \sum_{i=0}^{N} (TSP(C_i) * TIC(C_i))/N \quad (34)$$

B. INHERITANCE METRICS AND THEIR USAGE

Inheritance aspect is a vital attribute in the object-oriented paradigm. It facilitates design at class level, and forms 'IS-A' relations among classes. The design of the class is a basic component of system development [71]. The utilization of the inheritance aspect, shrinkages the cost of testing efforts along with maintenance of system [33]. The reuse with inheritance will consequently provide software, which is more maintainable, understandable, and reliable [72]. An experimental study of Harrison *et al.* depicts inheritance absence is simpler to grasp or control than software systems having inheritance aspect [72]. But, experiments of Daley shows software systems with tertiary inheritance may be easily updated compared to systems without inheritance [73].

The metrics of Inheritance compute several facets of inheritance, for example, the breadth, and depth of the hierarchy, in addition to the complexity of overriding [74]. Likewise, Bhattacherjee and Rajnish performed an investigation inheritance metrics addressing classes [62]. Yet, it is established that the greater the hierarchy of inheritance, the reusability of a class will be better but maintainability of the system under test (SUT) will be difficult. To simplify the understanding, designers trying to maintain inheritance hierarchy shallow, and discard reusability by the use of inheritance [33]. Therefore, it's essential to evaluate the complexity of the inheritance hierarchy to solve the difference between shallowness, and depth.

Many metrics addressing inheritance are defined by the researchers. In this regard, Table 1 showing these metrics with references. In this paper specifically, we get these inheritance metrics from the perspective of SFP.

TABLE 1. Catalog of Inheritance Metrics.

Author Name	Metrics Name
Chidamber and Kemerer [33]	Depth of Inheritance Tree (DIT)
	Number of Children (NOC)
Abreu Mood metrics suit [54]	Attribute Inheritance Factor (AIF)
	Method Inheritance Factor (MIF)
Bansiya J. et al QMOOD[75]	Number of Hierarchies (NOH)
	Average Number of Ancestors (ANA)
	Measure of Functional Abstraction (MFA)
Henry's & Kafura[59]	Fan in
	Fan out
Tang, Kao and Chen, [58]	Inheritance Coupling (IC)
Lorenz and Kidd	Number of Method Inherited (NMI)
	Number of Methods Overridden (NMO)
	Number of New Methods(NNA)
	Number of Variable Inherited (NVI)
Henderson-Sellers	Average Inheritance Depth (AID)
Li [61]	NAC (number of ancestor classes)
	NDC (number of descendent classes)
Tegarden et al. [63]	CLD (class-to-leaf depth)
	NOA (number of ancestor)
Lake and Cook [64]	NOP (number of parents)
	NOD (number of descendants)
Rajnish et al. [66][62]	DITC (Depth of Inheritance Tree of a Class)
	CIT (Class Inheritance Tree)
Sandip et al. [7][67]	ICC (Inheritance Complexity of Class)
	ICT (Inheritance Complexity of Tree)
Gulia et al. [76]	CCDIT (Class Complexity Due To Depth of Inheritance Tree)
	CCNOC (Class Complexity Due To Number of Children)
F. T. Sheldon et al [70]	Average Degree of Understandability (AU)
	Average Degree of Modifiability (AM)
Rajnish and Choudhary [71]	Derive Base Ratio Metric (DBRM)
	Average Number of Direct Child (ANDC)
	Average Number of Indirect Child (ANIC)
Mishra et al. [77]	CCI (Class Complexity due to Inheritance)
	ACI (Average Complexity of a program due to Inheritance)
	MC (Method Complexity)
Abreu and Carapuc [77][18]	Total Children Count (TCC)
	Total Progeny Count (TPC)
	Total Parent Count (TPAC)
	Total Ascendancy Count(TAC)
	Total Length of Inheritance chain(TLI)
	Method Inheritance Factor(MIF)
K. Rajnish et al. [71]	Extended Derived Base Ratio Metrics (EDBRM)
	Extended Average Number of Direct Child (EANDC)
	Extended Average Number of Indirect Child (EANIC)
Rajnish and Bhattacherjee[18]	Inheritance Metric Tree (IMT)
Chen, J. Y., and J. F. Lu[78]	Class Hierarchy of Method (CHM)
Lee et al[79]	Information-flow-based inheritance coupling (IH-ICP)

C. INHERITANCE IN SFP

The metrics of object-oriented are utilized for prediction of software quality. The characteristics that ascertain the software quality are understandability, fault tolerance, maintainability, defect density, reusability, normalized rework rate, and many others. Numerous studies have been performed including object-oriented metrics verification through empirical research paper on open-source software in the context of fault prediction utilizing {loc}, {lcom}, {cbo}, {dit}, and {noc} metrics [80]. Reusability investigation on systems based on object-oriented utilizing metrics of coupling, inheritance, and cohesion [81], heuristic-based C&K metrics evaluation [82], reusability metrics for the design of object-oriented [83], empirical scrutiny of C&K metrics for the complexity of object-oriented design [84].

The metrics suite of C&K developed, and implemented by Chidamber *et al.* is the utmost often use metrics collection for object-oriented software [33]. Basili *et al.* [85] studied the set of design metrics of object-oriented proposed by Chidamber *et al.* [86]. R. Subramanyam endorsed that {dit}, {cbo}, and {wmc} metrics are class level fault predictor [84].

Empirical assessments of the classification algorithm have been constructed to predict faults through studies [87]. Basili *et al.* numerous C&K metrics are found to be related to failure propensity [86]. Tang *et al.* evaluated metrics suite of C&K, and discovered that not any of these metrics except {rfc}, and {wmc} were considered important [88]. Briand *et al.* takeout forty-nine metrics to determine, which model to use for fault prediction. Findings reveals except {noc} all metrics are helpful to predict failure propensity [85]. Wust and Briand discovered that {dit} metrics are inversely correlated to fault proneness, and {noc} measure is a trivial predictor of failure propensity [89]. Yu *et al.*

selected 8 metrics, and they studied the association among them with the propensity to failure. Initial, they investigated the correlation amongst the metrics where they discovered four extremely related groups. Then they use univariate analysis to notice, which indicators may identify failures [90]. Malhotra and Jain used logistic regression methods to study the association among the object-oriented metrics with faults tendency. The receiver operating characteristics (ROC) investigation was utilized, and the accomplishment of the predictive model was assessed using ROC [90]. Yeresime *et al.* explored the methods including linear regression, logistic regression, and ANN to predict software failure using C&K metrics. They concluded the importance of {wmc} metric for classification of fault [91].

The review of the literature reveals that the {noc}, and {dit} metrics, correlated to inheritance features are used in fault prediction together within C&K metrics only. Consequently, it is deemed vital to authenticate the worth of inheritance metrics from the perspective of SFP.

D. METRICS UTILIZATION IN SFP

A sizeable quantity of object-oriented metrics has been formed previously, which includes Li and Henry (1993), Abreu and Carapuca (1994), Henderson-Sellers (1996), Lorenz and Kidd (1994), Bansiya and Davis (QMOOD metrics suite) (1997), Etzkorn *et al.*, Tegarden *et al.* (1999), Briand *et al.* (1997), Benlarbi and Melo (1999), Tang *et al.* (1999) and Cartwright and Sheppered (2000) including Bieman and Kang, Halstead, McCabe, Briand *et al.* Hitz and Montazeri, Lee *et al.*, Li and C&K metrics suite, to evaluate the internal structure of software systems [35].

Metrics have different levels including class, method, file, process, component, and quantitative levels. The metrics at method-level are extensively utilized for the problem of fault prediction. McCabe (1976) and Halstead (1977) metrics recommended in the 1970s however these are even now the furthermost prevalent method-level metrics.

The metrics at class-level are only used in object-oriented programs where the metrics suite of C&K (Chidamber & Kemerer, 1994) is even now the utmost prevalent class-level metrics suite being utilized for prediction of faults.

Few scholars including Khoshgoftaar, Gao, & Szabo, 2001; Ostrand, Weyuker, & Bell, 2005 utilized metrics gathered per source file, which is termed file-level metrics [27]. Table 2 summarized the frequently used metrics set at method, class, and file-level for SFP. The overall distribution of metrics for SFP is shown in Figure 1.

TABLE 2. Frequently used Metrics in Software Fault Prediction.

Method Level Metrics	Class Level Metrics	File Level Metrics
1. **loc** McCabe's line count of code 2. **v(g)** McCabe "cyclomatic complexity" 3. **ev(g)** McCabe "essential complexity" 4. **iv(g)** McCabe "design complexity" 5. **n** Halstead total operators + operands 6. **v** Halstead "volume" 7. **l** Halstead "program length" 8. **d** Halstead "difficulty" 9. **i** Halstead "intelligence" 10. **e** Halstead "effort" 11. **b** Halstead "bug" 12. **t** Halstead's time estimator 13. **lOCode** Halstead's line count 14. **lOComment** Halstead's count of lines of comments 15. **lOBlank** Halstead's count of blank lines 16. **lOCodeAndComment** Lines of comment and code 17. **uniq_Op** Halstead Unique operators 18. **uniq_Opnd** Halstead Unique operands 19. **total_Op** Halstead Total operators 20. **total_Opnd** Halstead Total operands 21. **branchCount** Branch count of the flow graph Converted method-level metrics into class-level ones using minimum, maximum, average and sum operations (21 *4 = 84)	**1.** Coupling Between Objects **CBO)** 2. Depth Of Inheritance Tree **(DIT)** **3.** Lack Of Cohesion Of Methods **(LCOM)** **4.** Num Of Children **(NOC)** 5. Response For Class **(RFC)** 6. Weighted Method Per Class **(WMC)** 7. Percent_Pub_Data 8. Access_To_Pub_Data 9. Dep_On_Child 10. Fan_In is the # of calls by higher modules. No_Of_Method No_Of_Attribute NO_Of_Attribute_Inher No_Of_Method_Inher Fan-in Fan-out No_Of_Pvt_Method No_Of_Pvt_Attribute NO_Pub_Method NO_Pub_Attribute NLOC	1. # of times the source file was inspected prior to system test release. 2. # of LOC for the source file prior to coding phase (auto-generated code) 3. # of LOC for the source file prior to system test release, 4. # of lines of commented code for the source prior to code (auto-generated), 5. # of lines of commented code for the source file prior to system test release

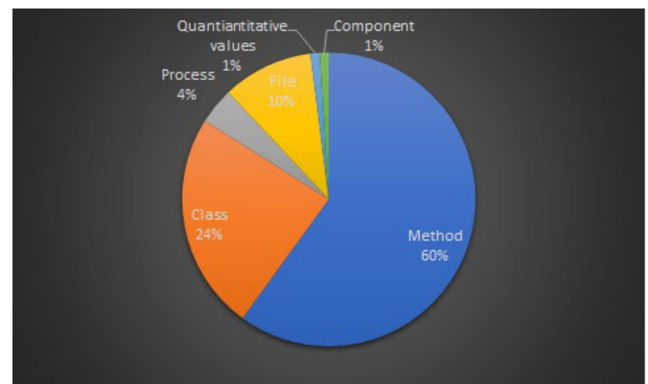

FIGURE 1. Distribution of Metrics.

E. DATASET

The information regarding the faults of software projects is honestly rarely reachable. The issue is that information on faults for the enterprise's business projects are accumulated digitally as these are propriety to the organization. On the other hand, in the smaller projects fault information is not adequate. Consequently, data with labels seldom exist. The accessibility of public datasets will grant the assessment of the inheritance metrics in software fault prediction. Many datasets having inheritance metrics are found [92]. Some of them are situated in the repository of Tera-PROMISE, and five datasets on the repository of D'Ambros [34].

The publicly available inheritance metrics includes Depth of the Inheritance Tree {dit}, Inheritance Coupling {ic}, Number of Children {noc}, Functional Abstraction Measure {mfa}, Inherited Method Number {nomi}, Inherited Attribute Number {noai}, Dependent of the Son {doc}, number of methods called per class {fanOut},number of classes that are called class methods {fanIn)}, and number of immediate base classes{ifanin}. A total of 78 public datasets found, which have inheritance metrics. Unluckily, all ten metrics have not existed collectively in any dataset. Nevertheless, groups of inheritance metrics found is different datasets, which are explained as under:

1) FOUR INHERITANCE METRICS [dit, noc, ic, mfa]

It is found that about 63 public datasets having four inheritance metrics namely Depth of the Inheritance Tree

TABLE 3. Public Data sets with 4 Inheritance Metrics.

Dataset Name	# Ins	% Falty	Dataset Name	# Ins	% Falty	dit	noc	ic	mfa
ant-1.7	745	22	prop-1	18471	15	✓	✓	✓	✓
arc	234	11	prop-2	2314	10	✓	✓	✓	✓
berek	43	37	prop-3	10274	11	✓	✓	✓	✓
camel-1.2	608	36	prop-4	8718	10	✓	✓	✓	✓
camel-1.4	872	17	prop-5	8516	16	✓	✓	✓	✓
camel-1.6	965	19	prop-6	660	10	✓	✓	✓	✓
ckjm	10	50	redaktor	176	15	✓	✓	✓	✓
e-learning	64	9	serapion	45	20	✓	✓	✓	✓
forrest-0.6	7	14	skarbonka	45	20	✓	✓	✓	✓
forrest-0.7	29	17	sklebagd	20	60	✓	✓	✓	✓
forrest-0.8	32	6	synapse-1.0	157	10	✓	✓	✓	✓
iny-1.1	111	57	synapse-1.1	222	1	✓	✓	✓	✓
iny-1.4	241	7	synapse-1.2	256	34	✓	✓	✓	✓
iny-2.0	352	11	systemdata	65	13	✓	✓	✓	✓
jedit-3.2	272	33	szybkafucha	25	56	✓	✓	✓	✓
jedit-4.0	306	25	tempoproject	42	30	✓	✓	✓	✓
jedit-4.1	312	25	tomcat	858	8	✓	✓	✓	✓
jedit-4.2	367	13	velocity-1.4	196	75	✓	✓	✓	✓
jedit-4.3	492	2	velocity-1.5	214	69	✓	✓	✓	✓
Kalkulator	27	22	velocity-1.6	229	34	✓	✓	✓	✓
log4j-1.0	135	25	workflow	39	51	✓	✓	✓	✓
log4j-1.1	109	34	wspomaganiepi	18	67	✓	✓	✓	✓
log4j-1.2	205	91	xalan-2.4	723	15	✓	✓	✓	✓
lucene-2.0	195	47	xalan-2.5	803	48	✓	✓	✓	✓
lucene-2.2	247	58	xalan-2.6	885	46	✓	✓	✓	✓
lucene-2.4	340	60	xalan-2.7	909	98	✓	✓	✓	✓
nieruchomosci	27	37	xerces-1.2	440	16	✓	✓	✓	✓
pdftranslator	33	45	xerces1.3	453	15	✓	✓	✓	✓
poi-1.5	237	59	xerces-1.4	588	74	✓	✓	✓	✓
poi-2.0	314	12	xerces-init	162	47	✓	✓	✓	✓
poi-2.5	385	64	zuzel	29	44	✓	✓	✓	✓
poi-3.0	442	64				✓	✓	✓	✓

TABLE 4. Public Datasets [dit,noc,ic]Metrics.

Dataset Name	No of Instances	Percentage of Fault	dit	noc	ic
jEdit_4.0_4.2	274	49	✓	✓	✓
jEdit_4.2_4.3	369	55	✓	✓	✓

{dit}, the Number of Children {noc}, Inheritance Coupling {ic}, and Functional Abstraction Measure {mfa}. These are available at the repositories of D'Ambros, and Tera-PROMISE [93]. The Table 3 shows the public dataset name in 1st column, the total number of instances in the second column, the percentage of faults in the third column, and the availability of metrics are mentioned with ✓mark.

2) THREE INHERITANCE METRICS [dit, noc, ic]

It is found that about 2 public datasets having three inheritance metrics namely Depth of the Inheritance Tree {dit}, the Number of Children {noc}, and Inheritance Coupling {ic}. These are available at the repositories of D'Ambros, and Tera-PROMISE. Table 4 illustrations the explanation of these datasets where the name of the dataset in the first column of the table, in second contains the number of instances, and the fault percentage in the third column. The remaining columns marked ✓to show the availability of the metric data in the associated dataset.

TABLE 5. Public Datasets[dit,noc,ic,doc,fanIn] Metrics.

Dataset Name	# Ins	% Falty	dit	noc	ic	doc	fanIn
Kc1-binary	145	41	✓	✓	✓	✓	✓
Kc1-class-binary	145	41	✓	✓	✓	✓	✓
kc1-numericdefect	145	41	✓	✓	✓	✓	✓
kc1-numeric	145	41	✓	✓	✓	✓	✓
kc1-top5	145	5	✓	✓	✓	✓	✓

3) FIVE INHERITANCE METRICS [dit,noc,ic,doc,fanIn]

It is found that about 5 public datasets having five inheritance metrics namely Depth of the Inheritance Tree {dit}, the Number of Children {noc}, Inheritance Coupling {ic}, Dependent on Child{DOC}, and number of classes that are called class methods {fanIn}. These are available at the repositories of D'Ambros, and Tera-PROMISE. Table 5 illustrations the explanation of these datasets where the name of the dataset in the first column of the table, in second contains the number of instances, and the fault percentage in the third column. The remaining columns marked ✓to show the availability of the metric data in the associated dataset.

4) FIVE INHERITANCE METRICS[dit,noc,noai,nomi,ifanin]

It is found that about one public dataset having five inheritance metrics namely Depth of the Inheritance Tree {dit}, the Number of Children {noc}, Number of Attribute Inherited {noai}, Number of Method Inherited {nomi}, and Number of immediate base classes of hierarchy {ifanin}. These are available at the Tera-PROMISE repository. Table 6 illustrations the explanation of these datasets where the name of the dataset in the first column of the table, in second contains the number of instances, and the fault percentage in the third column. The remaining columns marked ✓to show the availability of the metric data in the associated dataset.

TABLE 6. Five Inheritance Metrics [dit, noc, noai, nomi, ifanin].

Dataset Name	# Ins	% Falty	dit	noc	noai	nomi	IFANIN
eclipse34_debug	1065	25	✓	✓	✓	✓	✓

5) FOUR INHERITANCE METRICS[dit, noc, noai, nomi]

It is found that about one public dataset having four inheritance metrics namely Depth of the Inheritance Tree {dit}, the Number of Children {noc}, Number of Attribute Inherited {noai}, and Number of Method Inherited {nomi}. These are available at the tera-PROMISE repository. These are available at the repositories of D'Ambros, and Tera-PROMISE. Table 7 illustrations the explanation of these datasets where the name of the dataset in the first column of the table, in second contains the number of instances, and the fault percentage in the third column. The remaining columns marked ✓to show the availability of the metric data in the associated dataset.

TABLE 7. Public Datasets [dit, noc, noai, nomi] Metrics.

Dataset Name	No of Instances	Percentage of Fault	dit	noc	noai	nomi
eclipse34_swt	1485	44	✓	✓	✓	✓

TABLE 8. Public Datasets [dit, noc, noai, nomi, fanIn, fanOut]Metrics.

Dataset Name	# Ins	% Falty	dit	noc	noai	nomi	fanIn	fanOut
churn	997	21	✓	✓	✓	✓	✓	✓
Eclipse JDT Core	997	21	✓	✓	✓	✓	✓	✓
Eclipse PDE UI	1497	14	✓	✓	✓	✓	✓	✓
Equinox Framework	324	40	✓	✓	✓	✓	✓	✓
Lucene	691	9	✓	✓	✓	✓	✓	✓
mylyn	1862	13	✓	✓	✓	✓	✓	✓
single-version-ck-oo	997	20	✓	✓	✓	✓	✓	✓

6) SIX INHERITANCE METRICS [dit, noc, noai, nomi, fanIn, fanOut]

It is found that about six public datasets having six inheritance metrics namely Depth of the Inheritance Tree {dit}, the Number of Children {noc}, Number of Attribute Inherited {noai}, number of classes that call class methods{fanIn}, and number of methods called per class{FanOut}. These are available at the repositories of D'Ambros, and Tera-PROMISE. Table 8 illustrations the explanation of these datasets where the name of the dataset in the first column of the table, in second contains the number of instances, and the fault percentage in the third column. The remaining columns marked ✓to show the availability of the metric data in the associated dataset.

In the literature review, it is perceived that researchers utilize public, and private datasets for the validation of their research works. In this regard, Table 9 shows the author's name, year of publication, and public/private datasets used in studies.

The utmost significant difficulties for software fault prediction experiments are the use of non-public (private) datasets. Numerous companies developed fault prediction models utilizing patented data and displayed these models in conferences. Nevertheless, it is not probable to match the outcomes of such studies with outcomes of our own models since their datasets may not be grasped [7]. The distribution of datasets is shown in Figure 2.

Table 10 showed the fault distribution in the publicly available data sets. The dataset is grouped according to the percentage of fault they contain. The first column shows the range of percentage of fault, the second column number of public datasets fall in this group, and the third column shows the name of datasets with usage in studies.

TABLE 9. Data usage by studies.

Author	Year	Dataset
Briand et al.	2000	Hypothetical video rental business
Cartwright et al.	2000	Large European telecommunication industry, which consists of 32 classes and 133KLOC.
Emam et al.	2001	Used two versions of Java application: Ver 0.5 and Ver 0.6 consisting of 69 and 42 classes.
Gyimothy et al.	2005	Source code of Mozilla with the use of Columbus framework
Nachiappan et al.	2005	Open source eclipse plug-in
Zhou et al.	2006	NASA consisting of 145 classes, 2107 methods and 40 KLOC
Olague H.M et al.	2007	Mozilla Rhino project
Kanmani et al.	2007	Library management system consists of 1185 classes
Pai et al.	2007	Public domain dataset consists of 2107 methods, 145 classes, and 43 KLOC
Tomaszewksi et al.	2007	Two telecommunication project developed by Ericsson
Shatnawi et al.	2008	Eclipse project: Bugzilla database and Change log
Aggarwal et al.	2009	Student projects at University School of Information Technology
Singh et al.	2009	NASA consists of 145 classes, 2107 methods and 40K LOC
Cruz et al.	2009	638 classes of Mylyn software
Burrows et al.	2010	iBATIS, Health watcher, Mobile media
Singh et al.	2010	NASA consists of 145 classes, 2107 methods, and 40K LOC
Zhou et al.	2010	Three releases of Eclipse, consisting of 6751, 7909, 10635 java classes and 796, 988, 1306 KLOC
Fokaefs et al.	2011	NASA datasets
Malhotra et al.	2011	Open source software
Mishra et al.	2012	Eclipse and Equinox datasets
Malhotra et al.	2012	Apache POI
Heena	2013	Open Source Eclipse System
Rinkaj Goyal et al.	2014	Eclipse, Mylyn, Equinox and PDE
Yeresime et al.	2014	Apache integration framework (AIF) Ver 1.6
Ezgi Erturk et al.	2015	Promise software engineering repository data
Golnoush Abaei et al.	2015	NASA datasets
Saiqa Aleem et al.	2015	PROMISE data repository
Santosh et al.	2015	10 Datasets from PROMISE repository
Yohannese et al.	2016	AEEEM Datasets & Four datasets PROMISE repository
Ankit Pahal et al.	2017	Four projects from the NASA repository
Bartłomiej et al.	2018	Github Projects: Flask, Odoo, GitPython, Ansible,Grab
Patil et al.	2018	Real-time data set, Attitude Survey Data
Bahman et al.	2018	Five NASA datasets
Hiba Alsghaier et al.	2019	12-NASA MDP and 12-Java open-source projects
Balogun et al.	2019	NASA and PROMISE repositories
Alsaeedi et al.	2019	10 NASA datasets
Wasiur Rhmann et al.	2020	GIT repository, Android-4 & 5 versions
Deepak et al.	2020	Open source bug metris dataset
Razu et al.	2020	3 open source datasets from PROMISE

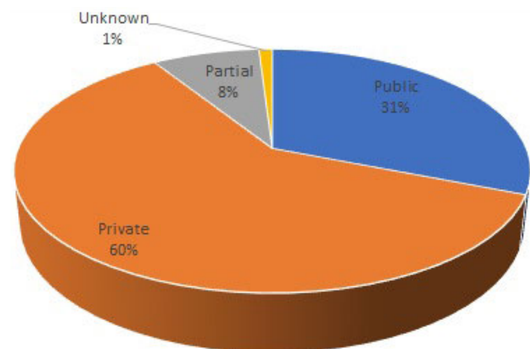

FIGURE 2. Data Sets Distribution.

F. ALGORITHMS AND PERFORMANCE MEASURES

Statistical methods are reflected as black-box solutions since these are extremely dependent on data, it is encouraging to perceive that more researchers are discovering the capability of machine learning methods to predict fault-prone modules. The literature review revealed that 59% used machine learning, and 22% used statistical methods [21]. Different machine learning methods and performance measures are examined which use object-oriented metrics for fault prediction. These are grouped into tables where studies from 1990-2003

TABLE 10. Datasets used by studies.

Range of % Fault Distribution	# Datasets	Datasets with Studies
0-4	9	PC2(10), MC1(3), SP3(1) , SP4(1) , PC5(4), ARGO(1) , Camel 1.0 (1), Inventory System 13 (1), SP2 (1)
5-9	17	Inventory System 14 (1), CCCS12 (1), Inventory System 17 (1), Inventory System 16 (1), SP1 (1), Ivy 1.4 (1), PC1 (25), Inventory System 12 (1), Inventory System 15 (1), Inventory System 11 (1), AR1 (2), MW1 (13), Inventory System 10 (2), Datatrieve (2), KC3 (13),CM1 (20), CCCS8 (1)
10-14	17	Argo uml 0.26.2 , Synapse 1.0 , PC3 (10), Arc (1), Inventory System 7 (1), Eclipse 2.1a (1), Ant 1.5 (1), Ivy 2.0 (2), Inventory System 9 (1), POI 2.0 (1), PC4 (12), Inventory System 8 (1), AR3 (7), Inventory System 6(1), Eclipse 2.0a (1), Eclipse 3.0a (1), AR6 (2).
15-19	15	Xalan 2.4 (1), Xerces 1.3 (1), Inventory System 4 (1), KC1 (19), Inventory System 2 (1), Ant 1.3 (1), Xerces 1.2 (1),Inventory System 5 (1), Camel 1.4 (1), AR4 (7), JM1 (16), AR5 (6), Camel 1.6(1), CCCS4 (1), Inventory System 3 (1).
20-24	7	KC2 (13), Ant 1.7 (2), Jedit 4.3 (1), Ant 1.4 (1), Eclipse 2.1 (4), Jedit 4.0 (2), Eclipse Component: SWT (1).
25-29	3	Ant 1.6 (59), Symapse 1.1 (59), CCCS2 (43).
30-34	6	Eclipse 2.0 (7), Eclipse 3.0 (5), MC2 (6), Jedit 3.2 (2), Synapse 1.2 (2), Velocity 1.6 (2).
35-39	2	Camel 1.2 (1), Inventory System 1 (1).
40-44	2	KC1 (9), Library Management System (1).
45-49	2	Xalan 2.6 (1), Lucene 2.0 (1), Xerces init (1), Xalan 2.5 (2), KC4 (6).
50-54	5	Cos (2), Eclipse Component: JDT Core (1)
55-59	1	Ivy 1.1 (1), Mozilla (1), Lucene 2.2 (1), POI 1.5 (1), Lucene 2.4 (3)
60-64	5	POI 2.5 (1)
65-70	2	Velocity 1.5 (1), POI 3.0 (3)
>70	5	Xerces 1.4 (1), Velocity 1.4 (1), Tomcat 6.0 (2), Jedit1 (2), Jedit2 (2)

are depicted in Table 11, studies in between 2004-2007 in Table 12 and studies between 2008-2020 mentioned in Table 13 [7], [94]–[104].

IV. DISCUSSION

In this section, findings of the literature review mentioned in Section III will be discussed, and highlight the potential resources available to make use of inheritance aspects in SFP arena.

It is also important to mentioned here that many solutions have been developed to solve the class imbalance problem such as sampling, cost-sensitive, and ensemble methods [4], [105]. However, these solutions are not equally effective as most empirical studies do not take into consideration the impact of class imbalance on prediction models and which, imbalance method works well or help to learn capabilities in software defect prediction. Selecting models, which are stable and efficient with class imbalance will give a better result.

A. METRICS

Many object-oriented metrics are used in experiments where the C&K metrics suite is a widely used set of metrics in SFP [106].

The literature review reveals that about 54 inheritance metrics are defined in the literature by the researcher to address the inheritance aspect. Out of these only two inheritance metrics are used in SFP namely `{dit}`, and `{noc}`. The reason for their extensive use is that these metrics are the part of C&K metrics suite, which is widely utilized in research experiments by the researchers. There are fever studies, which address the inheritance aspect exclusively.

TABLE 11. SFP studies (1990-2003).

Reference	Algorithms	Performance Measure
Porter and Selby(1990)	Classification Tree	Accuracy
Briand, Basili and Hetmanski (1993)	Logistic Regression, Classification Tree,Optimized Set Reduction	Correctness, Completeness
Lanubile, Lonigro, and Visaggio (1995)	PCA, Discriminant Analysis, Logistic Regression, Logical Classification	Misclassification Rate
Cohen and Devanbu (1997)	Foil, Flipper on IPL	Error Rate
Khoshgoftaar, Allen Hudepohl, Aud 1997	ANN and Discriminant	Type-I,Type-II,Misclassification Rate
Evett, Khoshgoftar, Chien, Allen (1998)		Ordinal Evaluation Procedure
Ohlsson, Zhai, Helander 1998	PCA,Discriminant Analysis for Classification, Multivariate Analysis	Misclassification Rate
Binkley, Schach 1998	Spearman Rank Correlation Test	
De Almeida and Matwin 1999	C4.5, CN2, FOIL,NewID	Correctness, Completeness,Accuracy
Kaszycki 1999		TN, TP
Yuan, Khoshgoftaar, Allen, Ganesan 2000	Fuzzy Subtractive Clustering	Type-I,Type-II,Overall Misclassification Rate,Effectiveness,Efficiency
Denaro 2000	Logistic Regression	R2
Khoshgoftaar, Allen Bushboom 2000		Type-I,Type-II
Xu, Khoshgoftaar, Allen 2000	PCA,FNR	Average Absolute Error
Guo, Lyu 2000	Finite Mixture Model Analysis, Expectation Maximization (EM)	Type-II Error
Khoshgoftaar, Gao, Szabo 2001	ZIP	AAE,ARE
Schneidewind	BDF,LRF	Type-I,Type-II,Misclassification Rate
Emam, Melo,Machado 2001	Logistic Regression	J-Coefficient
Khoshgoftaar,Geleyn,Gao 2002	PRM,ZIP,Module Order Modeling	Average Relative Error
Khoshgoftar 2002	GBDF	Type-I,Type-II
Mahaweerawt,Sohasathit, Lursinsap 2002	Fuzzy Clustering, RBF	Type-I,Type-II,Misclassification Rate
Khoshgoftaar,Seliya 2002a	SPRINT(Classification Tree),CART(Decision Tree)	Type-I,Type-II,Misclassification Rate
Pizzi,Summers, Pedrycz 2002	Median-Adjusted Class Labels(Pre-Processing),Multilayer Perception	Accuracy
khoshgoftaar,Seliya 2002b	CART-LS,S-PLUS,CART-LAD	AAE,ARE
Reformat 2003	Classification Models	Rate Change
Koru,Tian 2003	Tree Base Models	U-Test
Denaro, Lavazza,Pezze 2003	Logistics Regression	
Thwin, Quah 2003	GRNN	R2, R, ASE, AAE, Min, AE, Max, AE
Khoshgoftaar, Seliya 2003	CART-LS, S-PLUS, CART-LAD	
Guo, Cukic,Singh 2003	Dempster-Shafer Belief Networks	Probability of Detection,Accuracy
Denaro, Pezze,Morasca 2003	Logistic Regression	R2,Completeness

Recently a experiment study highlight the potential of inheritance in SFP where they have used `{noc}`, `{dit}`, `{mfa}`, `{ic}`, `{nomi}`, `{noai}`, `{ifanin}`, `{fanOut}`, `{fanIn}`, and `{doc}` inheritance metrics.

The literature review also indicated that the use of method-level metrics is superior to then class level, which is 60, and 24 percent respectively.

B. DATASETS

The literature review reveals that experimental studies used public, and private datasets where the ratio to use private datasets is higher than public datasets, which is 60 to 31 percent respectively. The bug data are available at PROMISE, D'Admros, and NASA repositories.

The literature review indicate a total of 78 public datasets containing 10 inheritance metrics, which includes `{noc}`, `{dit}`, `{mfa}`, `{ic}`, `{nomi}`, `{noai}`, `{ifanin}`, `{fanOut}`, `{fanIn}`, and `{doc}`.

Unluckily, all ten inheritance metrics have not discovered within a single dataset. But, the various combination of inheritance metrics is found in public datasets.

- A group of four inheritance Metrics `{dit, noc, ic, {mfa}` found in 63 public datasets available at PROMISE, and D'Adam repositories.

TABLE 12. SFP studies (2004-2007).

Reference	Algorithms	Performance Measure
Menzies, Disterfano,Orrego,Chapman 2004	NaÃŕve Bayes, J48	PF
khoshgoftaar, Seliua 2004	CART, S-PLUS, SPRING-Sliq, C4.5	Misclassification Rate
Wang,Yu,Zhu 2004	CGA,ANN	Accuracy
Mahaweerawat, Sophatsathit, Lursinsap, Musilek 2004	RBF	Accuracy ,Type-I,Type-II
Menzies,Distefano 2004	LSR, Model Trees, ROCKY	Accuracy,Sensitivity,Precision
Kaminsky, Boetticher 2004	Genetic Algorithm	T-Test
Kanmani, Uthariaraj, Thambidurai 2004	GRNN, PCA	R,R2, ASE, AAE, Max, AE, Min, AE
Zhong, Khoswhgoftaar, Seliya 2004	K-means,Neural-Gas clustering	MSE, FPR, FNR, Misclassification Rate
Xing, Guo, Lyu 2005	SVM, QDA, Classification Tree	Type-I,Type-II Error
Koru, Liu 2005	J48, Kstar, Bayesian Netowrks, ANN,SVM	F-measure
Khoshgoftaar, Seliya, Gao 2005	C4.5, Decision Tree, Discriminant Analysis,Logistic Regression	Misclassification Rate
Koru, Lin 2005	J48,K-Star,Random Forests	F-measure
Challagulla, Bastani,Yen,Paul 2005	Linear Regression, SVM, NaÃŕve Bayes,J48	AAE
Gyimothy, Ference, Siket 2005	Logistic Regression,Linear Regression, Decision trees, NN	Completeness,Correctness, Precision
Ostrand, Weyuker,Bell 2005	Negative Binomial Regression Model	Accuracy
Tomaszewski, Lundberg, Grahn 2005	Regression Technique, PCA	R2
Hassan,Holt 2005		Hit Rate,APA
Ma, Guo, Cukic 2006	Random Forests,LR,DA,NaÃŕve Bayes, J48,ROCKY	G-mean I,G-mean II,F-measure,ROC,PD,Accuracy
Challagulla, Bastani 2006	MBR	PD, Accuracy
Khoshgoftar, Seliya, Sundaresh 2006	MLR,CBR	ARE,AAE
Nikora, Munson 2006	Rules for Fault Definition	
Zhou, Leung 2006	Logistic Regression, NaÃŕve Bayes, Random Forest	Correctness,Completeness,Precision
Mertik, Lenic, Stigili, Kokol 2006	C4.5, SVM, RBF	PD,PF,Accuracy
Bibi, Tsoumakas, Stamelos, Vlahvas 2008	RvC	AAE,Accuracy
Gao, Khoshgoftaar 2007	Poisson Regression, Negative Binomial Regression, Hurdle Regression	AAE, ARE
Li,Reformat 2007	SimBoost	Accuracy
Mathaweerawat, Sophatsathit, Lursinsap 2007	RBP,Self-Organizing Map Clustering	MAR
Menzies et al. 2007	NaÃŕve Bayes, J48	PD,PF,Balance
Ostrand, Weyuker, Bell 2007	Negative Binomial Regression Model	accuracy
Pai,Dugan 2007	Linear Regression,Poisson Regression, Logic Regression	Sensitivity, Specificity, Precision, FP, FN
Wang,Zhu,Yu 2007	S-PLUS, TreeDisc	Type-I,Type-II
Seliya, Khoshgoftaar 2007	EM Techniques	Type-I, Type-II
Tomaszewski, Hakansson, Grahn, Lundberg 2007	Univariate Liner regression Analysis	Accuracy
Seliya, Khoshgoftaar 2007	Semi-Supervised Clustering, K-means Clustering	Type-I,Type-II
Olague, Gholston, Quattlebaum 2007	UBLR, Spearman Correlation	Accuracy

- A group of three Inheritance Metrics `{dit,noc,ic}` found in 2 public datasets available at PROMISE, and D'Adam repositories.
- A group of five Inheritance Metrics `{dit, noc, ic, doc,fanIn}` found in 5 public datasets available at PROMISE, and D'Adam repositories.
- A group of five Inheritance Metrics `{dit, noc, noai, nomi, ifanin}`found in one public dataset available at PROMISE, and D'Adam repositories.
- A group of four Inheritance Metrics `{dit, noc, noai, nomi}` found in one public dataset available at PROMISE, and D'Adam repositories.
- A group of six Inheritance Metrics `{dit, noc, noai, nomi, fanIn, fanOut}` found in six public datasets available at PROMISE, and D'Adam repositories.

TABLE 13. SFP studies (2008-2020).

Reference	Algorithms	Performance Measure
Bingbing, Qian, Shengyong, Ping 2008	K-means, Affinity Propagation	Type-I,Type-II
Marcus, Poshyvanyk, Ference 2008	Univariate Logistic Regression	Precision, Correctness
Shafi, Hassan, Arshaq, Shamail 2008	Classification Via Regression	Precision, Recall, Accuracy
Catal, Diri 2009	Random Forests(Artificial Immune Systems),NaÃŕve Bayes	AUC
Turhan, Kocak, Bener 2009	CBGR, Nearest Neighbor Sampling	
Catal 2009	X-means Clustering,	
Catal, Diri 2009	NaÃŕve Bayes, YATSI	
Yeresime Suresh, Lov Kumar (2014)	Linear Regression, Logistic Regression,ANN	Precision,Correctness,Completeness, MAE, MARE, RMSE, SEM
Saiqa Aleem,Luiz Fernando (2015)	NaiveBayes, MLP, SVM, AdaBoost, Bagging, Decision Tree, Random Forest, J48, KNN, RBF and K-means	Accuracy, Mean absolute error and F-measure
Santosh, Sandeep (2015)	Genetic Programming(GP)	Error rate, Recall, Completeness
Yohannese, Tianrui Li (2016)	NB, NN, SVM, RF, KNN,DTr, DTa, and RTr	ROC
Ankit Pahal, R. S. Chillar(2017)	ANN, SSO	Accuracy
Yogomaya Mohapatra (2018)	GSO-GA,SVM	Fitness Value,Accuracy
BartÅĆomiej WÄşjcicki,Robert (2018)	Machine Learning	Recall, False Positive Rate
Patil, Nagaraja (2018)	Linear Regression,FCM	Coefficients, Standard Errors And T-Values
Bahman Arasteh(2018)	Naive Bayes, ANN, SVM	Accuracy, Precision
Hiba Alsghaier, Akour(2019)	GA,SVM	Accuracy, Sd, Error Rate, Specificity, Precision, Recall, And F-Measure
Balogun,Shuib Basri (2019)	RIPPER, Bayesian Network, Random Tree, and Logistic Model Tree	Area Under Curve (AUC)
Rajkumar N, Viji C (2019)	SVM,ANN,KNN	Accuracy, Sensitivity, Specificity, Precision
Alsaeedi,Zubair (2019)	SVM, DS, RF	Accuracy, Precision, Recall, F-Score, ROC-AUC
Wasiur Rhmann, Babita Pandey (2020)	Random Forest, J48	Precision, Recall
Deepak, Pravin (2020)	Factor Analysis (FA)	R2, Adjusted R2
Razu, Asraf (2020)	SVM	Precision, Recall, Specificity, F 1 Measure, Accuracy

C. ALGORITHMS

The literature review reveals that experimental studies used statistical, and machine learning models for their experiments. Machine learning models have widely used a comparison to statistical methods in the context of SFP. Recent studies show the use of Support Vector Machine, Naive Bayes, Decision Tree, and ANN models for fault prediction.

D. PERFORMANCE MEASURES

The literature review reveals that experimental studies use many performance measures to validate the results of machine learning models. Frequently used performance measures include Accuracy, Precision, Recall, F1-Score, AUC, and ROC [107].

V. THREAT TO VALIDITY

Many inheritance metrics are described in the literature, however it is possible that other metrics might be a better indicator of faults. Since we only concentrate on inheritance metrics that were published in the various researches and these are accessible in the certain datasets. The survey results shown are basing on the inheritance metrics only. These datasets might not be characteristic of all business sectors. These datasets might not be excellent representatives in terms of class numbers and sizes.

The main threat to the validity of this survey is related to bias in selecting papers to inclusion and exclusion criteria.

There might be a slight risk that some important papers might have been missed in the search process. All the selected papers were extracted and quality assessed by the author which is reassessed twice a time specifically focusing inheritance aspect.

While performing a survey, it is essential to be mindful of possible threats to the validity of the acquired outcomes and drawn conclusions. A first potential cause of bias associates with the data being utilized, whether the data is representative of the domain in question and can result be generalized. Since the data utilized in this survey branches from the public domain, so our findings can be matched with others and can be subjected to reproduction if required. Also, numerous authors have debated in support of the suitability of the PROMISE data repository and eclipse datasets and/or applied some of its data sets for their experimentations. Thus, we believe the acquired findings to be suitable for the SFP community.

VI. CONCLUSION AND FUTURE WORK

In this paper, a literature review is conducted to see the efficacy of inheritance metrics on SFP is validated. The review is conducted by collecting, organizing, categorizing, and investigate published fault prediction studies. The outcomes of the study show 54 inheritance metrics defined so far by the researchers. Only two inheritance metrics are being used in SFP. Besides, 78 public datasets identified, which contain ten inheritance metrics with various combinations. The usage of method-level metrics is 60%, and a similar number is for the use of private datasets. The use of machine learning approaches is increasing, which uses many performance measures.

This study will assist scholars to examine the earlier studies from metrics, methods, datasets, performance evaluation metrics, and experimental outcomes viewpoints in an easy, and efficient way exclusive attention on the Inheritance aspect.

In the context of future work, we anticipate some scholars would make use of the inheritance aspect mentioned in this paper, and attempt to assess other inheritance metrics than the ones we utilized. Besides, utilizing regression using machine learning techniques for faults predict with inheritance metrics will be significant work to be done.

REFERENCES

[1] M. H. Halstead, *Elements of Software Science*, vol. 7. New York, NY, USA: Elsevier, 1977.
[2] A. J. Albrecht, "Measuring application development productivity," in *Proc. Joint Share, Guide, IBM Appl. Develop. Symp.*, 1979.
[3] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[4] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Appl. Soft Comput.*, vol. 33, pp. 263–277, Aug. 2015.
[5] R. Jayanthi and L. Florence, "Software defect prediction techniques using metrics based on neural network classifier," *Cluster Comput.*, vol. 22, pp. 77–88, Feb. 2018.
[6] S. A. Sherer, "Software fault prediction," *J. Syst. Softw.*, vol. 29, no. 2, pp. 97–105, May 1995.
[7] C. Catal, "Software fault prediction: A literature review and current trends," *Expert Syst. Appl.*, vol. 38, no. 4, pp. 4626–4636, Apr. 2011.
[8] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 414–423.
[9] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, Feb. 2015.
[10] T. Bayes, "An essay towards solving a problem in the doctrine of chances. [facsimil]," *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales*, vol. 95, no. 1, pp. 11–60, 2001.
[11] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
[12] B. Kröse, B. Krose, P. van der Smagt, and P. Smagt, "An introduction to neural networks," *J. Comput. Sci.*, vol. 48, pp. 1–52, Jan. 1993.
[13] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, Feb. 2015.
[14] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *J. Syst. Softw.*, vol. 82, no. 6, pp. 981–992, Jun. 2009.
[15] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Inf. Sci.*, vol. 179, no. 8, pp. 1040–1058, Mar. 2009.
[16] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[17] L. Son, N. Pritam, M. Khari, R. Kumar, P. Phuong, and P. Thong, "Empirical study of software defect prediction: A systematic mapping," *Symmetry*, vol. 11, no. 2, p. 212, Feb. 2019.
[18] K. Rajnish and V. Bhattacherjee, "Applicability of Weyuker property 9 to object-oriented inheritance tree metric—A discussion," in *Proc. 10th Int. Conf. Inf. Technol. (ICIT)*, Dec. 2007, pp. 234–236.
[19] S. Chawla and R. Nath, "Evaluating inheritance and coupling metrics," *Int. J. Eng. Trends Technol.*, vol. 4, no. 7, pp. 2903–2908, 2013.
[20] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Comput. Electr. Eng.*, vol. 67, pp. 15–24, Apr. 2018.
[21] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Softw. Eng.*, vol. 20, no. 3, pp. 199–206, Mar. 1994.
[22] G. Pascoe, "Elements of object-oriented programming," *Byte*, vol. 11, no. 8, pp. 139–144, 1986.
[23] M. Kölling, "The problem of teaching object-oriented programming, Part 1: Languages," *J. Object-Oriented Program.*, vol. 11, no. 8, pp. 8–15, 1999.
[24] S. D. Conte, H. E. Dunsmore, and Y. Shen, *Software Engineering Metrics and Models*. New York, NY, USA: Benjamin, 1986.
[25] L. C. Briand, J. Wüst, and H. Lounis, "Replicated case studies for investigating quality factors in object-oriented designs," *Empirical Softw. Eng.*, vol. 6, no. 1, pp. 11–58, 2001.
[26] S. R. Aziz, T. A. Khan, and A. Nadeem, "Experimental validation of inheritance Metrics' impact on software fault prediction," *IEEE Access*, vol. 7, pp. 85262–85275, 2019.
[27] K. M. Breesam, "Metrics for object-oriented design focusing on class inheritance metrics," in *Proc. 2nd Int. Conf. Dependability Comput. Syst. (DepCoS-RELCOMEX)*, Jun. 2007, pp. 231–237.
[28] Shivam, "A study on inheritance using object oriented programming with C++," *Int. J. Adv. Res. Comput. Sci. Manage. Stud.*, vol. 1, no. 2, pp. 10–21, Jul. 2013.
[29] S. S. Rathore and S. Kumar, "A decision tree logic based recommendation system to select software fault prediction techniques," *Computing*, vol. 99, no. 3, pp. 255–285, Mar. 2017.
[30] S. Beecham, T. Hall, D. Bowes, D. Gray, S. Counsell, and S. Black, "A systematic review of fault prediction approaches used in software engineering," in *Proc. Irish Softw. Eng. Res. Centre*, Limerick, Ireland, 2010.
[31] O. Gómez, H. Oktaba, M. Piattini, and F. García, "A systematic review measurement in software engineering: State-of-the-art in measures," in *Proc. Int. Conf. Softw. Data Technol.* Setúbal, Portugal: Springer, 2006, pp. 165–176.
[32] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs: An initial report," in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Qual. Eval. (MaLTeSQuE)*, Feb. 2017, pp. 21–26.
[33] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[34] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, May 2010, pp. 31–41.
[35] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Comput.*, vol. 21, no. 24, pp. 7417–7434, Dec. 2017.
[36] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.
[37] H. Kapila and S. Singh, "Bayesian inference to predict smelly classes probability in open source software," *Int. J. Current Eng. Technol.*, vol. 4, no. 3, pp. 1724–1728, 2014.
[38] R. Mahajan, S. K. Gupta, and R. K. Bedi, "Design of software fault prediction model using BR technique," *Procedia Comput. Sci.*, vol. 46, pp. 849–858, Jan. 2015.
[39] S. M. Jamali, "Object oriented metrics (a survey approach)," Dept. Comput. Eng., Sharif Univ. Technol., Tehran, Iran, Tech. Rep., 2006.
[40] X. Huang, L. Shi, and J. A. K. Suykens, "Support vector machine classifier with pinball loss," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 5, pp. 984–997, May 2014.
[41] S. Huang, N. Cai, P. P. Pacheco, S. Narrandes, Y. Wang, and W. Xu, "Applications of support vector machine (SVM) learning in cancer genomics," *Cancer Genomics Proteomics*, vol. 15, no. 1, pp. 41–51, Jan./Feb. 2018.
[42] A. Taherkhani, "Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms," *Comput. J.*, vol. 54, no. 11, pp. 1845–1860, Nov. 2011.
[43] A. Chaddad, P. O. Zinn, and R. R. Colen, "Brain tumor identification using Gaussian mixture model features and decision trees classifier," in *Proc. 48th Annu. Conf. Inf. Sci. Syst. (CISS)*, Mar. 2014, pp. 1–4.
[44] T.-K. An and M.-H. Kim, "A new diverse AdaBoost classifier," in *Proc. Int. Conf. Artif. Intell. Comput. Intell.*, vol. 1, Oct. 2010, pp. 359–363.
[45] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[46] M. J. Islam, Q. M. Jonathan Wu, M. Ahmadi, and M. A. Sid-Ahmed, "Investigating the performance of Naive-Bayes classifiers and K-nearest neighbor classifiers," in *Proc. Int. Conf. Converg. Inf. Technol. (ICCIT)*, Nov. 2007, pp. 1541–1546.
[47] A. K. Luhach, D. Singh, P.-A. Hsiung, K. B. G. Hawari, P. Lingras, and P. K. Singh, *Advanced Informatics for Computing Research: Second International Conference, ICAICR 2018, Shimla, India, July 14–15, 2018, Revised Selected Papers*, vol. 955. Shimla, India: Springer, 2018.
[48] D.-L. Miholca, G. Czibula, and I. G. Czibula, "A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks," *Inf. Sci.*, vol. 441, pp. 152–170, May 2018.
[49] G. M. Foody, "Status of land cover classification accuracy assessment," *Remote Sens. Environ.*, vol. 80, no. 1, pp. 185–201, Apr. 2002.
[50] J. S. Shirabad and T. J. Menzies, "The PROMISE repository of software engineering databases," School Inf. Technol. Eng., Univ. Ottawa, Ottawa, ON, Canada, Tech. Rep., 2005, vol. 24. [Online]. Available: http://promise.site.uottawa.ca/SERepository
[51] D. N. Card and W. W. Agresti, "Measuring software design complexity," *J. Syst. Softw.*, vol. 8, no. 3, pp. 185–197, Jun. 1988.
[52] K. El Emam, "A primer on object-oriented measurement," in *Proc. 7th Int. Softw. Metrics Symp.*, Apr. 2001, pp. 185–187.
[53] D. Mishra and A. Mishra, "Simplified software inspection process in compliance with international standards," *Comput. Standards Interface*, vol. 31, no. 4, pp. 763–771, Jun. 2009.
[54] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proc. 4th Int. Conf. Softw. Qual.*, vol. 186, 1994, pp. 1–8.
[55] D.-W. E, "Analysis and implementation of software metric for object-oriented," in *Proc. Int. Conf. Comput. Intell. Softw. Eng.*, Dec. 2009, pp. 1–4.
[56] S. Mal and K. Rajnish, "New quality inheritance metrics for object-oriented design," *Int. J. Softw. Eng. Appl.*, vol. 7, no. 6, pp. 185–200, Nov. 2013.
[57] J. Gao, H.-S. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. Norwood, MA, USA: Artech House, 2003.
[58] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *Proc. 1st Int. Softw. Metrics Symp.*, 1993, pp. 52–60.
[59] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. SE-7, no. 5, pp. 510–518, Sep. 1981.
[60] D. E. Monarchi and G. I. Puhr, "A research typology for object-oriented analysis and design," *Commun. ACM*, vol. 35, no. 9, pp. 35–48, 1992.
[61] W. Li, "Another metric suite for object-oriented programming," *J. Syst. Softw.*, vol. 44, no. 2, pp. 155–162, Dec. 1998.
[62] K. Rajnish and V. Bhattacherjee, "Class inheritance metrics-an analytical and empirical approach," *J. Comput. Sci.*, vol. 7, no. 3, pp. 25–34, 2008.
[63] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," *Decis. Support Syst.*, vol. 13, nos. 3–4, pp. 241–262, Mar. 1995.
[64] A. Lake and C. Cook, "Use of factor analysis to develop OOP software complexity metrics," in *Proc. 6th Annu. Oregon Workshop Softw. Metrics*. Silver Falls, OR, USA: Citeseer, 1994.
[65] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Software design metrics for object-oriented software," *J. Object Technol.*, vol. 6, no. 1, pp. 121–138, 2007.
[66] K. Rajnish and Y. Singh, "An empirical and analytical view of new inheritance metric for object-oriented design," *Int. J. Comput. Appl.*, vol. 65, no. 12, pp. 44–50, 2013.
[67] K. Rajnish, "Theoretical validation of inheritance metrics for object-oriented design against Briand's property," *Int. J. Inf. Eng. Electron. Bus.*, vol. 6, no. 3, pp. 28–33, Jun. 2014.
[68] S. Mal and K. Rajnish, "Applicability of Weyuker's property 9 to inheritance metric," *Int. J. Comput. Appl.*, vol. 66, no. 12, 2013.
[69] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Softw. Eng.*, vol. 3, no. 1, pp. 65–117, 1998.
[70] F. T. Sheldon, K. Jerath, and H. Chung, "Metrics for maintainability of class inheritance hierarchies," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 14, no. 3, pp. 147–160, 2002.
[71] K. Rajnish, A. K. Choudhary, and A. M. Agrawal, "Inheritance metrics for object-oriented design," *Int. J. Comput. Netw. Commun.*, vol. 2, no. 6, pp. 13–26, Dec. 2010.
[72] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.
[73] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Softw. Eng.*, vol. 1, no. 2, pp. 109–132, 1996.
[74] G. S. Krishna and R. K. Joshi, "Inheritance metrics: What do they measure?" in *Proc. 4th Workshop Mech. SPEcialization, Generalization inHerItance (MASPEGHI)*, 2010, p. 1.
[75] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Aug. 2002.
[76] P. Gulia and R. S. Chillar, "New proposed inheritance metrics to measure the software complexity," *Int. J. Comput. Appl.*, vol. 58, no. 21, pp. 1–4, Nov. 2012.
[77] D. Mishra and A. Mishra, "Object-oriented inheritance metrics in the context of cognitive complexity," *Fundam. Informaticae*, vol. 111, no. 1, pp. 91–117, 2011.
[78] J.-Y. Chen and J.-F. Lu, "A new metric for object-oriented design," *Inf. Softw. Technol.*, vol. 35, no. 4, pp. 232–240, Apr. 1993.
[79] Y. Lee, "Measuring the coupling and cohesion of an object-oriented program based on information flow," in *Proc. Int. Conf. Softw. Qual.*, 1995.
[80] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.
[81] C. Catal, "Performance evaluation metrics for software fault prediction studies," *Acta Polytechnica Hungarica*, vol. 9, no. 4, pp. 193–206, 2012.
[82] R. Kumar and D. Gupta, "A heuristics based review on CK metrics," *Int. J. Appl. Eng. Res.*, vol. 7, no. 11, p. 2012, 2012.
[83] B. M. Goel and P. K. Bhatia, "Investigation of reusability metrics for object–oriented designing," in *Proc. NCETCIT*, May 2012, pp. 104–110.
[84] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003.
[85] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.
[86] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[87] R. Bender, "Quantitative risk assessment in epidemiological studies investigating threshold effects," *Biometrical J., J. Math. Methods Biosci.*, vol. 41, no. 3, pp. 305–319, Jun. 1999.

[88] A. Kaur and I. Kaur, "An empirical evaluation of classification algorithms for fault prediction in open source projects," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 30, no. 1, pp. 2–17, Jan. 2018.

[89] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proc. 6th Int. Softw. Metrics Symp.*, 1999, pp. 242–249.

[90] R. Malhotra and A. Jain, "Fault prediction using statistical and machine learning methods for improving software quality," *J. Inf. Process. Syst.*, vol. 8, no. 2, pp. 241–262, Jun. 2012.

[91] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using OO metrics. An industrial case study," in *Proc. 6th Eur. Conf. Softw. Maintenance Reeng.*, 2002, pp. 99–107.

[92] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, Sep. 2013.

[93] S. S. Rathore and S. Kumar, "An approach for the prediction of number of software faults based on the dynamic selection of learning techniques," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 216–236, Mar. 2019.

[94] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 32, no. 4, pp. 419–424, May 2020.

[95] H. Alsghaier and M. Akour, "Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier," *Softw., Pract. Exper.*, vol. 50, no. 4, pp. 407–427, Apr. 2020.

[96] D. Sharma and P. Chandra, "Linear regression with factor analysis in fault prediction of software," *J. Interdiscipl. Math.*, vol. 23, no. 1, pp. 11–19, Jan. 2020.

[97] Y. Mohapatra and M. Ray, "Software fault prediction based on GSO-GA optimization with kernel based SVM classification," *Int. J. Intell. Eng. Systems.*, vol. 11, no. 5, p. 152, 2018.

[98] M. R. Ahmed, M. A. Ali, N. Ahmed, M. F. B. Zamal, and F. M. J. M. Shamrat, "The impact of software fault prediction in real-world application: An automated approach for software engineering," in *Proc. 6th Int. Conf. Comput. Data Eng.*, Jan. 2020, pp. 247–251.

[99] C. W. Yohannese and T. Li, "A combined-learning based framework for improved software fault prediction," *Int. J. Comput. Intell. Syst.*, vol. 10, no. 1, pp. 647–662, Jan. 2017.

[100] I. Gondra, "Applying machine learning to software fault-proneness prediction," *J. Syst. Softw.*, vol. 81, no. 2, pp. 186–195, Feb. 2008.

[101] N. Rajkumar and C. Viji, "An efficient software fault prediction scheme to assure qualified software implementation using improved classification methods," *Int. J. Innov. Technol. Exploring Eng.*, vol. 8, no. 8S, Jun. 2019.

[102] A. Pahal and R. Chillar, "A hybrid approach for software fault prediction using artificial neural network and simplified swarm optimization," *IJARCCE*, vol. 6, no. 3, pp. 601–605, Mar. 2017.

[103] S. Patil, A. N. Rao, and C. S. Bindu, "Class level software fault prediction using step wise linear regression," *Int. J. Eng. Technol.*, vol. 7, no. 4, pp. 2552–2557, 2018.

[104] A. Alsaeedi and M. Z. Khan, "Software defect prediction using supervised machine learning and ensemble techniques: A comparative study," *J. Softw. Eng. Appl.*, vol. 12, no. 05, pp. 85–100, 2019.

[105] V. López, A. Fernández, S. García, V. Palade, and F. Herrera, "An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics," *Inf. Sci.*, vol. 250, pp. 113–141, Nov. 2013.

[106] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013.

[107] S. S. Rathore and S. Kumar, "A study on software fault prediction techniques," *Artif. Intell. Rev.*, vol. 51, no. 2, pp. 255–327, Feb. 2019.

SYED RASHID AZIZ received the M.S. degree in software engineering from COMSATS University, Islamabad, Pakistan, in 2008, and the M.Sc. degree in computer science from Al-Khair University, Islamabad, in 1998. He is currently pursuing the Ph.D. degree in software engineering with Bahria University, Islamabad. He has been engaged in many national, and enterprise-level business application projects, since 1986, and provide consultancy to private, public military, and government organization for automation alongside teaching courses to students at various tiers. His research interests include big data, software fault tolerance, software reliability, software testing, the Internet of Things, service-oriented computing, and data warehousing.

TAMIM AHMED KHAN received the B.E. degree (Hons.) in software engineering from Sheffield University, U.K., in 1995, the M.S. degree in computer engineering from CASE, Texila University, Pakistan, in 2006, the M.B.A. degree in finance and accounting from Preston University, Islamabad, Pakistan, in 1997, and the Ph.D. degree in software engineering from Leicester University, U.K., in 2012. He is currently serving as a Professor with the Department of Software Engineering, Bahria University Islamabad. His research interests include service-oriented architecture, E-learning, and software quality assurance.

AAMER NADEEM received the M.Sc. degree in computer science from Quaid-i-Azam University (QAU), the M.S. degree in software engineering from the National University of Sciences and Technology (NUST), and the Ph.D. degree in computer science from Mohammad Ali Jinnah University (MAJU). He is currently the Head of the Software Engineering Program at the Capital University of Science and Technology (CUST). He is also the Head of the Center for Software Dependability (CSD—a research group at CUST, working in the areas of software reliability, software fault tolerance, formal methods, and safety-critical systems. During his Ph.D., he worked as a Visiting Scholar with the Chinese University of Hong Kong (CUHK) under a research collaboration. He has over 30 years of teaching, research, and industry experience in computer science and software engineering. He has supervised 46 master's and two Ph.D. research theses in the areas of software testing, fault tolerance, and formal methods. He has authored or coauthored over 90 articles in reputable international journals and conferences. He is a reviewer or editorial board member of several international peer-reviewed journals and conferences. He is an Approved Ph.D. Supervisor for scholars funded by indigenous fellowship schemes of the Higher Education Commission (HEC) of Pakistan. He is a professional member of the Association for Computing Machinery (ACM).