Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Two Sides of the Same Coin: A Study on Developers' Perception of Defects

Geanderson Santos[1] | **Igor Muzetti**[2] | **Eduardo Figueiredo**[1]

[1]Computer Science Department, Federal University of Minas Gerais, Belo Horizonte, Brazil
[2]Computer Science Department, Federal University of Ouro Preto, Ouro Preto, Brazil

Correspondence
Geanderson Santos. Email: geanderson@dcc.ufmg.br

Present Address
Av. Antônio Carlos, 6627, Prédio do ICEx, Pampulha, Belo Horizonte, Minas Gerais, Brasil, CEP: 31270-901

Summary

Software defect prediction is a subject of study involving the interplay of software engineering and machine learning. The current literature proposed numerous machine learning models to predict software defects from software data, such as commits and code metrics. Further, the most recent literature employs explainability techniques to understand why machine learning models made such predictions (i.e., predicting the likelihood of a defect). As a result, developers are expected to reason on the software features that may relate to defects in the source code. However, little is known about the developers' perception of these machine learning models and their explanations. To explore this issue, we focus on a survey with experienced developers to understand how they evaluate each quality attribute for the defect prediction. We chose the developers based on their contributions at GitHub, where they contributed to at least 10 repositories in the past two years. The results show that developers tend to evaluate code complexity as the most important quality attribute to avoid defects compared to the other target attributes such as source code size, coupling and documentation. At the end, a thematic analysis reveals that developers evaluate testing the code as a relevant aspect not covered by the static software features. We conclude that developers' perceptions are not aligned with the machine learning models. For instance, while machine learning models give high impact to documentation, developers tend to neglect the documentation and focus on the complexity of the code.

KEYWORDS:
defect prediction, software features for defect prediction, machine learning models

1 | INTRODUCTION

Due to the growth of software development in many areas, the reliability of software projects has become a critical concern for practitioners and stakeholders alike[1,2]. Therefore, a software system may contain defect-prone attributes in various stages of development, testing, and maintenance. One way to increase the reliability of software systems is to predict the likelihood of defects in the software system. Consequently, the current literature embraces strategies to detect and mitigate the impacts of defective code[3,4,5]. For instance, several previous studies have analyzed code metrics as predictors of software defects[6,7,8,9]. Various investigations focus on unique software features (or code metrics) extracted from source code that may indicate defects[1,10,11]. For instance, these software features could represent change metrics[4,8], class-level metrics[12,13], Halstead and McCabe metrics[7,14], entropy metrics[3,4], among others. Here, we refer to code metrics as software features, as previously adopted in the machine learning literature[15,16].

Software defect prediction usually relies on machine learning and statistical modeling. For this reason, it heavily depends on the quality of the data to effectively predict defects and provide insights about software features for the development team[14,17,18,19]. Therefore, we need to ensure that the data is reliable and that we have enough data to create models that can generalize to different contexts of software development. The different datasets available in the literature vary in size, software features, and how they define a defect[7,14,17,20]. Recently, a unified dataset merged the different sources available for the community[21,22]. This dataset contains 47,618 classes from 53 Java projects and 70 software features. These software features relate to different characteristics of the code, such as cohesion, complexity, coupling, documentation, inheritance, and size[23]. In contrast to other works in the current literature[24], the unified dataset presents numerous features and classes, which makes the analysis more comprehensive and reliable about software features.

Software features and machine learning models have shown promise in predicting defects, but their usefulness in real-world projects is still a challenge for the software engineering community[17,25]. To address this challenge, researchers are proposing new techniques to explain the predictions made by these models, which may help developers reason about the software features that are more problematic for the software system. However, it is crucial to understand developers' perception of these models and their explanations. Exploring developers' perception can help us understand how they evaluate each quality attribute for defect prediction and identify any differences between their perception and the actual machine learning models. This research is essential because understanding the phenomenon behind defect prediction can lead to better software development practices and improve software quality.

Furthermore, our work is relevant because it goes beyond simply explaining how a machine learning model works and instead seeks to understand the human side of the equation. While SHAP (SHapley Additive exPlanations) and other model interpretation techniques can provide valuable insights into how a model makes decisions, they only tell part of the story. Understanding how developers and other stakeholders perceive and use these models is equally important for building effective and trustworthy machine learning systems. By exploring the differences between developers' opinions and machine learning model performance, this paper provides a more comprehensive view of the challenges and opportunities associated with machine learning in software development. It highlights the importance of considering not only the technical aspects of machine learning but also the social and cultural factors that shape how these models are understood and used. Ultimately, this paper's contribution is to provide a more nuanced and holistic view of machine learning in software engineering. By bringing together perspectives from both the technical and human sides of the equation, it offers insights that can inform the design of more effective and ethical machine learning systems.

To accomplish this, we conducted a survey study with developers from different backgrounds, inviting them to evaluate different scenarios that use quality attributes such as documentation, coupling, complexity, and size. We also asked developers which static software features they perceived as defect predictors. Guided by these goals, our paper explores the following overarching research question: "Do machine learning models and developers agree on software features that indicate defects?" Our results showed that the developers' perceptions did not always match the model outputs. For instance, developers tended to overlook documentation and focus on code complexity. We also concluded that developers indicated that testing code is a relevant aspect that may reduce the likelihood of defects, although it is not a static software feature like the ones analyzed in the literature[15,16].

The remainder of this research paper is organized as follows. In Section 2, we present the study setup and methodology. Section 3 provides an analysis of the survey results obtained from developers. In Section 4, we discuss the potential threats to the validity of our study. Section 5 introduces relevant prior work related to our research. Finally, in Section 6, we present our concluding statements and suggest areas for further exploration.

2 | STUDY DESIGN

This section describes the research method we use to investigate the developers' perception of the defects in software projects. Figure 1 exemplifies the steps applied in the research. Therefore, we divide our method into four fundamental steps. Section 2.1 presents the background related to software defects. Section 2.2 describes the goal of the investigation with the main research question. Section 2.3 presents the dataset we used to predict software defects (i. Unified dataset). Moreover, Section 2.4 displays the data preparation applied to fit the data for the experiments (ii. Data Preparation). Then, Section 2.5 discusses the software features. Afterward, Section 2.6 describes the software features for defect prediction in Java projects (iii. Software Features). Finally, Section 2.7 briefly discusses the machine learning models (iv. Defect Prediction Models).

2.1 | Background

Defects. A software defect represents an error, failure, or bug[26] in a software project. In this paper, we specifically refer to errors, which are defined in[26] as the observable difference between the expected and actual results caused by a fault. These errors can harm the appearance, operation, functionality, or performance of the target software project[27]. Software defects may appear in various stages of software development. These software defects may interrupt the development progress and increase the planned budget of software projects[14].

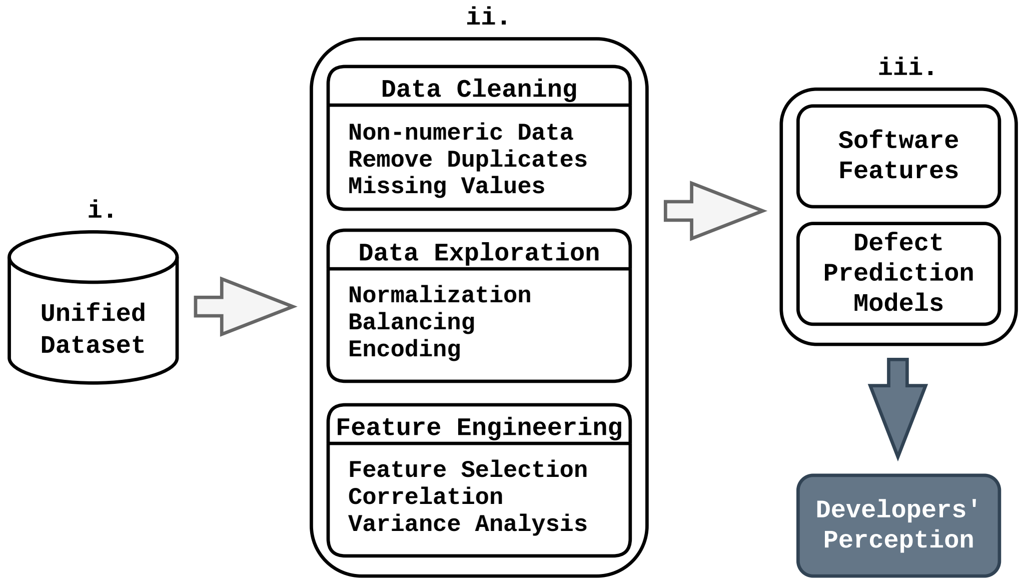

FIGURE 1 Method Overview (the colored boxes indicate the results of the present work).

Furthermore, a software team may discover software defects after code release generating significantly more effort to tackle these defects in production[9]. Moreover, if a company notices more software defects than expected in the testing and integration phase, it may impair its ability to meet customer deadlines or milestones. Therefore, it is hard to upgrade a software system that is already defective, as the software team continues to spend development time fixing the existing code instead of developing new features for their customers.

To mitigate these issues with software development, defect prediction is a method for predicting defects in a software project[7,14]. Therefore, the use of defect predictors is valuable for anticipating issues in the source code. For instance, if a software team has limited resources for software inspection, a defect predictor may indicate which modules are most likely to be defective. To generate a machine learning model able to predict software defects, it is necessary to follow the expected steps. First, we need a dataset with instances of known labels (i.e., defective or clean)[28]. Since we are interested in identifying defective instances, it is generally ideal to have a dataset with only binary labels. Later, we employ feature extraction to the entire data source extracting the most relevant features from the existing ones. Furthermore, the complete pool of software features with the corresponding label (i.e., defective or clean) serves as the input of the machine learning classifier. Finally, the model can classify unseen software instances (i.e., classes, files, or modules) into defective or clean. More specifically, a group of datasets[17] defines a defect based on the following expression, where one or more errors change the status of a module to defective.

$$defective? = errorcount >= 1$$

Explaining Software Defects. In machine learning, complex predictions can be challenging to explain, which presents a central challenge in software defect prediction[29,30,31]. Understanding why a model made a specific prediction is essential. For instance, if the complexity of a software class affects the defectiveness of the target class, the software team can modify the class to address its complexity[32,33,34]. To understand defect prediction, feature impact, or feature importance, is calculated. In current literature, feature importance is defined as an increased error after permuting feature values. Permutation breaks the relationship between the feature and the outcome[35,36]. Therefore, a software feature is relevant if permuting its value increases the model error because the model relied on that feature for the prediction[33,37]. Conversely, a feature has little importance if permuting its values keeps the model error unchanged because the model ignored that feature for the prediction. Often, software features interact in many different ways to create models that provide accurate predictions. Thus, feature importance represents a function of the interaction between other software features. In this case, Shapley values are used to find a fair division design that distributes the total importance among features.

More specifically, we transform instances into a space of simplified binary features, and the explanation model g is a linear function of binary variables:

$$g(z) = \phi_0 + \sum_{i=1}^{m} \phi_i \times z_i, \tag{1}$$

The parameters ϕ_i, where $i = 0, 1, \ldots, m$, are known as Shapley values. The number of simplified input features is denoted by m, and the binary vector $z_i = z_1, z_2, \ldots, z_m$ represents the simplified input space, where each element of the vector z takes a binary value of either 0 or 1. Shapley values are used to measure the contribution of each feature to the prediction, and they are theoretically optimal, unique, consistent, and locally accurate attribution values. In this work, we approximate Shapley values using SHAP (SHapley Additive exPlanation) values[35,37,38] to compute the importance of each feature in the prediction.

2.2 | Goal and Research Question

This paper aims to compare developers' perceptions of software defects with the explanations provided by machine learning models. To achieve this goal, we conducted a survey with developers from diverse backgrounds to investigate how they perceive various software quality attributes in comparison to machine learning models. Based on this objective, we formulated the following overarching research question: "Do machine learning models and developers agree in terms of software features that indicate defects?" By addressing this research question, we aim to shed light on the accuracy and reliability of machine learning models in identifying and explaining software defects and to provide insights for software development teams to improve their defect detection and resolution processes.

2.3 | Unified Data

The dataset used in our machine learning experiments is a merged version of several resources available for the scientific community, including PROMISE[39], Eclipse Bug Prediction[40], Bug Prediction Dataset[4], Bugcatchers Bug Dataset[41], and GitHub Bug Dataset[23][1]. The dataset contains 47,618 classes from 53 Java projects and 70 software features related to different aspects of the code. More details about the dataset and the models generated in this paper are available in the replication package[2]. The original authors of the dataset, Ferenc et al.[42], provided a pre-processed version of the dataset, which we used for our experiments. The dataset is imbalanced, as only around 20% of the classes represent a defect[15]. Therefore, we had to apply a data preparation step to create the machine learning models to predict software defects, as we discuss next.

2.4 | Data Preparation

To prepare the training dataset for our study, we relied on previous works[15,16]. We used Pycaret to assist with the data preparation and training of the models[43]. Figure 1 ii provides an overview of the data preparation process undertaken in our research paper. These steps were applied only to the training set and were necessary not only to clean the data and avoid misinterpretation but also to discover a list of software features for use in the feature selection step. We have provided a description of the three steps illustrated in Figure 1 ii below.

Data Cleaning. First, we applied data cleaning to eliminate duplicate classes and non-numeric data (see Figure 1 (ii)). This process is particularly critical in the defect prediction task, as many datasets have incorrect entries gathered by automatic systems[44]. We then performed data imputation to track any missing values, although no missing data were found in the structured data. Furthermore, we reduced the horizontal dimension of the data from 70 to 65 features by eliminating the non-numeric features. Additionally, we removed four over-represented software features that gathered information about the exact line and column of the source code where a class started and ended.

Data Exploration. In the second step of data preparation (ii - Figure 1), we conducted data exploration. To encode the data, we applied one-hot encoding[45] and normalized it. The type feature, which stored information about the class type, was one-hot encoded to create new features for classes, enumerates, interfaces, and annotations. To avoid multicollinearity, we tested for Variance Inflation Factor (VIF) and concluded that the preparation was suitable for our purpose, with low VIF values. We then normalized the data using Standard Scaler and used Synthetic Minority Oversampling Technique (SMOTE)[46,47] to deal with the imbalanced nature of the dataset, but only on the training set. The unified data contained only around 20% of defective classes, so oversampling was necessary to generate models that could generalize to unseen data. It's important to note that these steps were executed on the training set only.

Feature Engineering. In the final step, we applied feature engineering to select the most relevant software features. This involved three key techniques: feature selection, correlation analysis, and setting multicollinearity thresholds. Feature selection techniques allow us to choose a subset of relevant software features to be used in the analysis. To encode the categorical variables in these features, we used a technique called one-hot encoding. One-hot encoding creates binary variables (0 or 1) for each category of the categorical variable,

[1] https://zenodo.org/record/3693686
[2] https://github.com/gesteves91/replication-package-unified

indicating whether or not it is present in the observation[45]. After one-hot encoding, we used a combination of permutation importance techniques, including Random Forest, Adaboost, and Linear correlation, to choose a subset of software features. Next, we checked the correlation between these features (using a threshold of 99%). Finally, we set a multicollinearity threshold of 85%, meaning that we removed any software features with a correlation higher than this threshold. These steps resulted in a final selection of 59 software features that were used in our analysis.

2.5 | Software Features

The current literature provides numerous techniques for explaining machine learning models across various domains. One of the most prominent and widely used techniques is SHAP (SHapley Addictive exPlanation) values[35]. These values are used to compute the importance of each feature in the prediction model, enabling us to understand why a machine learning model made specific decisions for a given domain. Therefore, SHAP is particularly suitable for explaining complex machine learning models that are hard to interpret[35].

Furthermore, since features interact in complex ways to create more accurate prediction models, understanding the logic behind a software class is crucial in identifying the reasons for a defect in the target class. Therefore, SHAP is an important tool that can help in explaining and understanding the decision-making process of machine learning models. However, it's important to note that the explanations provided by SHAP depend on the software features that are used to explain the phenomenon. Table 1 presents the complete list of features we initially used to predict and explain software defects, which fall into seven quality attributes: size, complexity, coupling, cohesion, inheritance, documentation, and clone. In the next subsection, we discuss in detail the most important features and their relevance for predicting software defects.

2.6 | Quality Attributes

Although the literature offers several quality attributes for classifying software features, we have chosen to focus on the top-10 features identified by SHAP. As a result, we have categorized these features into four groups: Complexity, Coupling, Size, and Documentation. Complexity refers to the measurement of code complexity based on source code metrics[48,49]. Coupling pertains to the degree of interdependencies among source code metrics[49,50]. Documentation denotes the number of comments and documents in the source code[51,52]. Lastly, Size relates to the fundamental properties of the analyzed system in terms of various cardinalities such as the number of lines of code, the number of classes and methods in a file, among others[51,1].

2.7 | Defect Prediction Models

In order to build our classifier, we utilized an ensemble machine learning model technique. This technique involves combining the predictions from multiple machine learning models to achieve better overall performance. By using this approach, we were able to leverage the strengths of multiple models to build a stronger classifier. To train the models, we divided our dataset into two sets: 70% of the data was used for training, and the remaining 30% was used for testing. We also employed k-fold cross-validation in the training data to assess the performance of our models. In this case, the training data is partitioned into k subsets, and the model is trained and validated k times, using a different fold for validation each time while the remaining folds are used for training. Specifically, we used K=10 as recommended in[53]. With this method, the data is divided into K partitions, and at each iteration, we use nine partitions for training and one partition for validation. We then permute these partitions on each iteration to ensure that each partition is used for training and validation at least once. This approach allows us to compare distinct models and avoid overfitting, as the training set varies on each iteration.

To identify which models are suitable for our goal, we evaluated 15 machine learning algorithms: CatBoost Classifier, Random Forest, Decision Tree, Extra Trees, Logistic Regression, K-Neighbors Classifier (KNN), Gradient Boosting Machine, Extreme Gradient Boosting, Linear Discriminant Analysis, Ada Boost Classifier, Light Gradient Boosting Machine (LightGBM), Naive Bayes, Dummy Classifier, Quadratic Discriminant Analysis, and Support Vector Machines (SVM). In order to optimize the performance of our models, we used a technique called Optuna[54] to tune the hyperparameters of each individual model. Optuna uses Bayesian optimization to find the best set of hyperparameters for each model.

Furthermore, we have customized our machine learning pipeline to optimize the predictive modeling process for our specific dataset. Several key settings have been carefully configured to enhance both model performance and data preprocessing. For feature selection, we utilized the 'boruta' method[55], enabling the algorithm to select the most relevant features for modeling while eliminating redundant or less informative ones. To address multicollinearity, we set a threshold of 85%, which indicates that highly correlated features exceeding this limit will be removed to enhance model stability[43]. Additionally, we activated the automatic class imbalance correction using the

TABLE 1 Complete List of Software Features.

Acronym	Feature	QA	Acronym	Feature	QA
CC	Clone Coverage	Clone	CCLOC	Logical Lines of Code	Size
CCL	Clone Classes	Clone	LOC	Lines of Code	Size
CCO	Clone Complexity	Clone	NA	Number of Attributes	Size
CI	Clone Instances	Clone	NG	Number of Getters	Size
CLC	Clone Line Coverage	Clone	NLA	Number of Local Attributes	Size
CLLC	Clone Logical Line Coverage	Clone	NLG	Number of Local Getters	Size
LDC	Lines of Duplicated Code	Clone	NLM	Number of Local Methods	Size
LLDC	Logical Lines of Duplicated Code	Clone	NLPA	Number of Local Public Attributes	Size
LCOM5	Lack of Cohesion in Methods 5	Cohesion	NLPM	Number of Local Public Methods	Size
NL	Nesting Level	Complexity	NLS	Number of Local Setters	Size
NLE	Nesting Level Else-If	Complexity	NM	Number of Methods	Size
WMC	Weighted Methods per Class	Complexity	NOS	Number of Statements	Size
CBO	Coupling Between Object Classes	Coupling	NPA	Number of Public Attributes	Size
CBOI	Coupling Between Object Classes Inv.	Coupling	NPM	Number of Public Methods	Size
NII	Number of Incoming Invocations	Coupling	NS	Number of Setters	Size
NOI	Number of Outgoing Invocations	Coupling	TLLOC	Total Logical Lines of Code	Size
RFC	Response for a Class	Coupling	TLOC	Total Lines of Code	Size
CD	Comment Density	Documentation	TNG	Total Number of Getters	Size
CLOC	Comment Lines of Code	Documentation	TNLA	Total Number of Local Attributes	Size
DLOC	Documentation Lines of Code	Documentation	TNLG	Total Number of Local Getters	Size
PDA	Public Documented API	Documentation	TNLM	Total Number of Local Methods	Size
PUA	Public Undocumented API	Documentation	TNLPM	Total Number of Local Pub. Methods	Size
TCD	Total Comment Density	Documentation	TNLS	Total Number of Local Setters	Size
TCLOC	Total Comment Lines of Code	Documentation	TNM	Total Number of Methods	Size
DIT	Depth of Inheritance Tree	Inheritance	TNOS	Total Number of Statements	Size
NOA	Number of Ancestors	Inheritance	TNPM	Total Number of Public Methods	Size
NOC	Number of Children	Inheritance	TNS	Total Number of Setters	Size
NOD	Number of Descendants	Inheritance	TNA	Total Number of Attributes	Size
NOP	Number of Parents	Inheritance	TNPA	Total Number of Local Pub. Attributes	Size

fix imbalance parameter to ensure that the model does not exhibit bias toward the majority class (non-defective classes). This correction is crucial for reliable predictions, especially in imbalanced datasets. Our configuration has been meticulously designed to streamline the model-building process while upholding data integrity and ensuring high predictive accuracy

After experimenting with all the baseline models, we observed that five models consistently achieved good performance. These models are Random Forest[56], LightGBM[57], Extra Trees[58], Gradient Boosting Machine[59], and KNN[60]. We selected these models to form the ensemble model. To evaluate the performance of our models, we focused on one key metric: F1. F1 represents the harmonic mean of precision and recall, which is useful for evaluating the trade-off between precision and recall. We also used accuracy, AUC, recall, and precision to evaluate the performance of our models. Table 2 presents the average performance of the machine learning models.

TABLE 2 Performance of the Machine Learning Model.

Target	Accuracy	AUC	Recall	Precision	F1
Defect	0.811	0.841	0.639	0.487	0.556

2.8 | Selected Software Features

In Figure 1, we present the top 10 features selected by our explainability technique, which will play a pivotal role in our forthcoming survey. These selected features, along with their respective acronyms and full names, can be found in Table 3. It is worth noting that all the selected features fall under the class-level category, aligning with the focus of our investigation[15]. As detailed in Table 3, a majority of these features are closely tied to code size metrics, including NLG, TLOC, NG, and TNLA[21]. However, we've also incorporated elements related to documentation, such as CLOC, AD, and CD, in addition to those linked with code coupling (CBOI and CBO) and code complexity (NL)[21,23]. Importantly, these features have been derived from the best-performing models, which we discussed in depth in Section 2.7.

TABLE 3 Selected Software Features.

Acronym	Description	Quality Attribute
TLOC	Total Lines of Code	Size
NLG	Number of Local Getters	Size
NG	Number of Getters	Size
TNLA	Total Number of Local Attributes	Size
CLOC	Comment Lines of Code	Documentation
AD	API Documentation	Documentation
CD	Comment Density	Documentation
CBOI	Coupling Between Objects classes Inverted	Coupling
CBO	Coupling Between Objects classes	Coupling
NL	Nesting Level	Complexity

3 | DEVELOPERS' SURVEY

To gain a deeper understanding of developers' perceptions of the quality attributes, we conducted a two-part analysis. First, we asked developers four questions about each of the quality attributes, which group the software features discussed in Section 2.6. Each question was designed to allow developers to analyze each category separately from the others, and we included an example software feature for each quality attribute to illustrate the relationship between the group and software development. This approach enables us to identify which quality attributes developers consider most important for detecting software defects, and how these attributes may be related to specific software features. For example, by using WMC as an example of code complexity, we can examine how developers' perceptions of this quality attribute may influence their approach to managing complexity in their code.

Second, we used a Likert scale to ask developers to rate the importance of each software feature within its respective quality attribute. This analysis provides a more detailed understanding of how developers perceive the relative importance of individual software features for detecting defects within each quality attribute. By combining these two types of analysis, we can gain a more comprehensive understanding of which quality attributes and software features developers consider most critical for detecting defects in their code. This information can be used to guide the design of more effective tools and techniques for software defect detection and prevention, as well as to inform best practices for software development more broadly. Table 5 presents each of the questions developers had to evaluate.

In this section, we present the findings of a survey study conducted with fifty-four developers, which focused on their perceptions of the quality attributes discussed in Section 2.6 (Quality Attributes). We divide this section into three parts. First, in Section 3.1, we provide an overview of the developers' backgrounds who participated in the survey. Next, in Section 3.2, we present the main results of the survey, which include specific questions related to the target quality attributes and software features. In Section 3.3, we report the thematic analysis of open-ended responses provided by developers about static software features not included in the survey. Then, in Section 3.4, we discuss the implications of our findings, and in Section 3.5, we provide a detailed analysis of the results.

3.1 | Developers Background

To compare the developers' perspective with the machine learning models presented in Section 2.7, we apply a survey with developers from various backgrounds. However, before sending the survey to developers, we conducted a pilot study with members of our research group. In total, 10 members of the research group helped to enhance the survey with valuable feedback. The pilot survey was helpful to improve several aspects of the investigation, both related to the developers' background questions and the developers' perspective about the software features. Thus, we divide the study into two major groups: (i) developers' background and (ii) developers' understandability about the quality attributes and their impact on software defects.

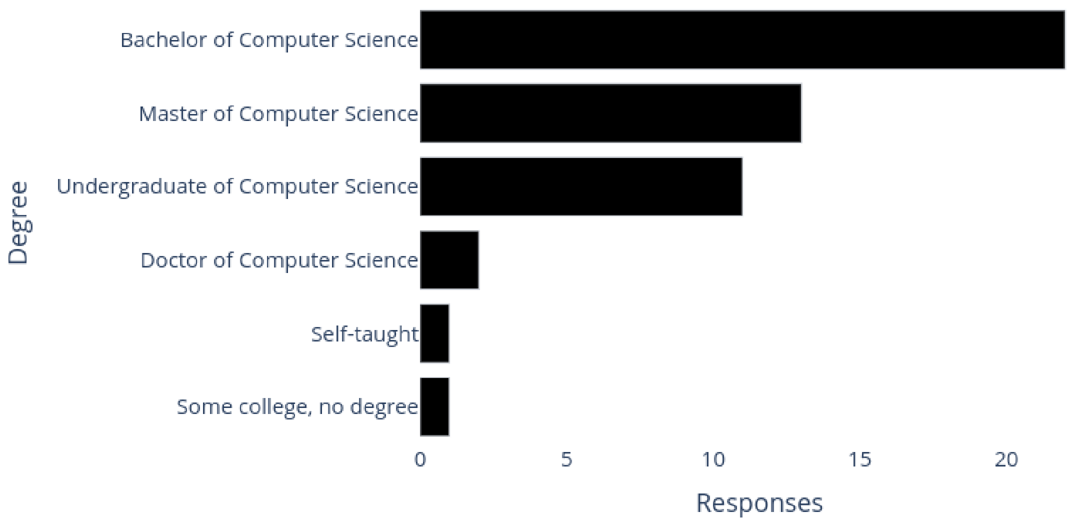

FIGURE 2 Developers' Background Degree.

After conducting a pilot survey, we proceeded with a consolidated version of the study, inviting a different set of developers to participate. To ensure our focus on active GitHub developers, given GitHub's prominent role in the software community[61,62], we developed a Python script to collect information from the GitHub API about a random sample of developers who met specific criteria. More precisely, we sent invitational emails exclusively to developers who had contributed to a minimum of 10 repositories on GitHub within the past two years. This criterion allowed us to select more experienced developers, enabling us to gather pertinent insights about defects and their correlation with software features. We distributed the survey to this random pool of developers over a three-week period in April 2021, resulting in 54 responses out of 735 invitations sent, corresponding to an acceptance rate of 7.35%. Following an initial analysis, we excluded four responses that were invalid or incomplete, leaving us with 50 valid survey responses for analysis and comparison with the machine learning model's results.

In the first part of the survey, we aimed to collect more information about the background of the developers. Therefore, we asked three questions about their experience with software development. First, we asked the developers about their highest level of education in the present moment. Figure 2 shows the background of developers related to their education. In total, 22 developers (44% of the developers) hold a degree in some computer science-related field. Complementary, 13 developers (26%) hold a master's degree in a computer science-related field, and 11 developers (22%) are undergraduates in computer science-related courses. Finally, 2 developers (4%) hold a Ph.D. in some computer science-related field. As we allow developers to add more responses as they reply to the survey, two developers included unexpected degrees. One participant (2%) answered that they did some college, but they did not finish it, and one participant (2%) affirm that they are a self-taught learner.

Finally, to complement the developers' background, we checked how many years of experience the developers had in software development. In this case, the developers could choose only one of three options. Table 4 reveals that twenty-two developers (44% of developers) have between five and ten years of experience in software projects. In addition, nineteen developers (38% of developers) hold more than ten years of experience in developing code. To conclude, only nine developers (18% of developers) have less than five years of experience in the software industry. We conclude from this part of the survey that most of the developers have a consistent experience in software development, as 41 developers (82% of developers) hold more than five years of experience.

TABLE 4 Developers' Years of Experience with Software Development.

Years of Experience	# Developers
Less than 5 years	9
Between 5-10 years	22
More than 10 years	19

3.2 | Developers' Software Features Perception

For the second part of the survey, we questioned developers about how they perceived a list of software features and their relationship to the defects in the source code. To do so, we present a scenario of use that focuses on API development not specifically on any programming language. Although we provide insights into the behavior of the API and specific classes that relate to each of the quality attributes explored in our investigation, we supply the developers with little information about the structure of the project. Thus, the main goal of this part of the survey is to capture developers' perceptions about the possible impact of each feature category on software defects. These software features are originally grouped into four categories[21,22,42]: documentation, size, complexity, and coupling. Figure 3 shows how the developers rank each quality attribute comparing their impact to cause defects in the software class. The stacked bar chart represents how many developers ranked each quality attribute as the most important. In this case, each developer had to pick a first, second, third, and fourth option.

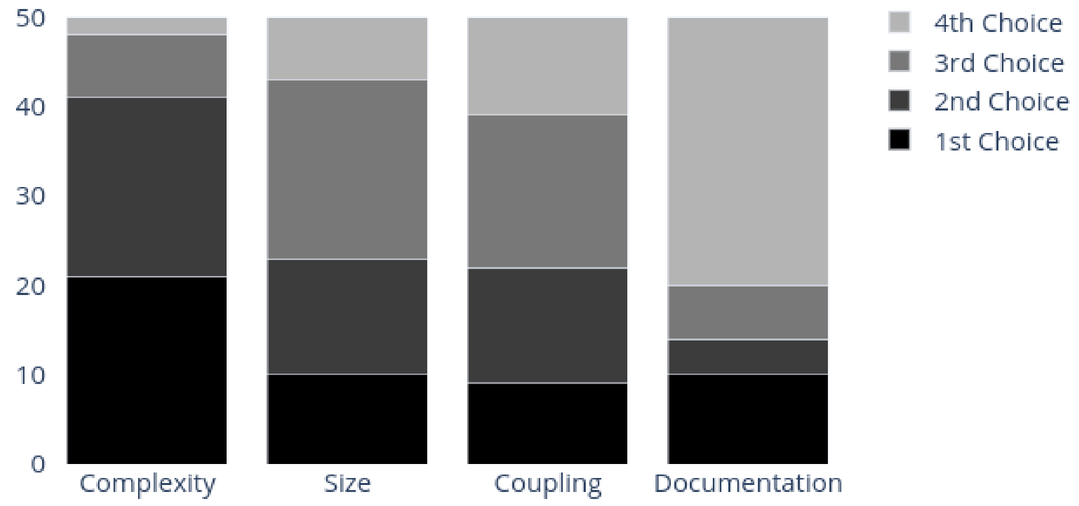

FIGURE 3 Developers' Perception about the Quality Attributes.

As stated previously, fifty-four developers participated in the survey, although we receive four invalid responses. Among them, twenty-one developers (42% of the developers) consider the code complexity the main quality attribute to contribute to defects in the source code compared to the other quality attributes (documentation, size, and coupling). Another 10 developers (20% of the developers) classify the code documentation and size as the most relevant quality attribute to indicate software defects in the source code. In addition, only nine developers (18% of the developers) think that coupling is the main category to predict software defects. This result differs from the machine learning models we discussed in Section 2.7, since the documentation and size quality attributes are the most relevant categories to predict defects in the source code according to our models. As a result, we conclude that the machine learning model presented in Section 2.7 contradicts developers' common sense.

To further explore the developers' perception of the quality attributes. We ask the developers four questions concerning each of the quality attributes that group the software features discussed in Section 2.6. Hence, we employ a software feature included in the quality attribute to exemplify the relationship between the group and software development. In this case, we opt to introduce the software feature with the most influence on the defect prediction. For instance, we rely on WMC as an example of code complexity. We designed these questions to allow developers to analyze each category separately from the remaining ones. Besides, the software feature introduction in each question may help developers to examine the effects of the quality attribute in their source code. Table 5 presents each of the questions developers had to evaluate.

After analyzing each question, the developers rank the quality attributes based on a 5-point scale. In this case, the scale ranged from "Very Important" to "Unimportant". Figure 4 shows the Likert scale[63] for evaluating the four questions. For the first question (Q1), developers rated the impact of modularization very highly compared to the other three questions. Thus, twenty-two developers (44% of developers) estimate the complexity category (represented by the WMC) as very important. Complementary, twenty-one developers (representing 42% of the developers) rank the code complexity as important. Only 7 developers (14% of the developers) believe the WMC is moderately important for the defects. As a result, the complexity category is the most important category to cause defects in the code based on developers' perceptions.

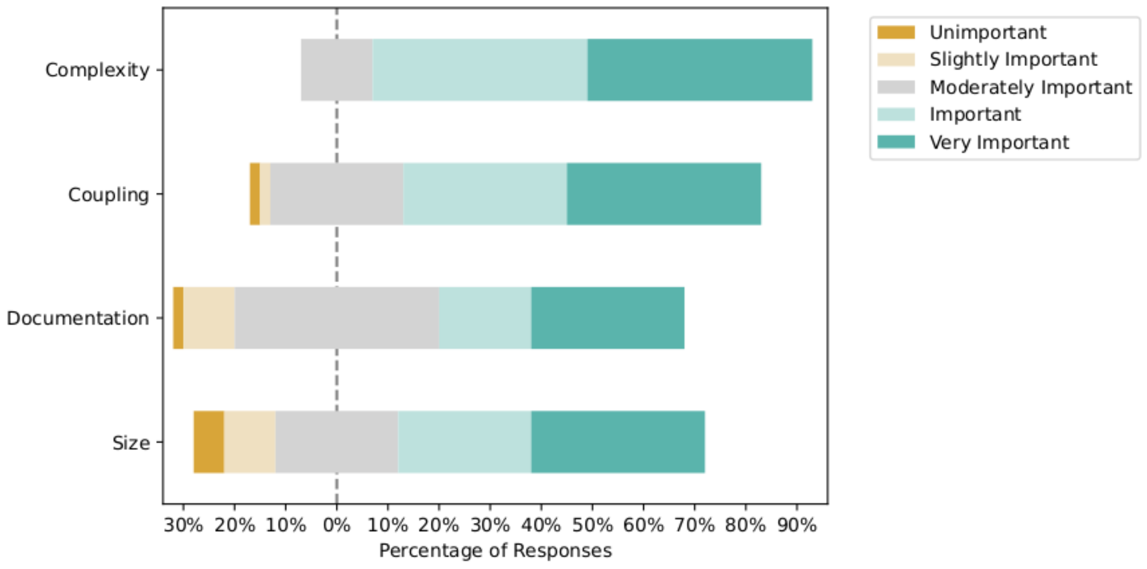

FIGURE 4 Likert Distribution of Research Questions.

The second question (Q2) is similar to the first question (Q1) presented to developers. The only difference is that the developers rate the impact of loosely coupled objects lesser than the code complexity. In this case, nineteen developers (38% of the developers) rank the coupling quality attribute (represented by the CBO) as very important for the defectiveness of their code. Sixteen developers (32% of the developers) rank the coupling category as important. Another, thirteen developers (26% of developers) believe the CBO is moderately important for the defects. One developer (2% of developers) believes the CBO is slightly important for the software defects in the code. Complementary, one developer (2% of developers) considers the CBO unimportant for the software defects. Hence, we conclude that the coupling category is the second most important category to cause defects in the code based on the developers' perception.

The third question (Q3) is similar to the first two questions (Q1 and Q2). Developers rate the impact of smaller classes (represented by NOA) less important than the code complexity and coupling between objects. The number of attributes is the most controversial quality attribute as many developers ranked the category as moderately important to cause a defect. In this case, twenty developers (40% of the developers) rank the code size as moderately important for the defectiveness of their code. Another thirty developers (30% of developers) ranked the size as very important, and nine developers (18% of developers) ranked the category as important. On the other hand, five developers (10% of developers) ranked the category as slightly important. Finally, only one developer (2% of developers) ranked the category as unimportant. As a result, the code size is not as important as the other two categories (complexity and coupling).

The final question (Q4) is similar to the remaining three questions (Q1, Q2, and Q3). In this case, developers rate the impact of the lack of proper documentation as the least relevant quality attribute to cause defects in the code. Thus, seventeen developers (34% of the developers) rank the documentation as very important for the defectiveness of their code. Thirteen developers (26% of the developers) rank this quality attribute as important. Twelve developers (24% of developers) believe the documentation is moderately important for the software defects in the code. Another five developers (10% of developers) consider the documentation is slightly important for the software defects in the code. Finally, three developers (6% of the developers) rank the documentation as unimportant for the defectiveness of the code. As a result, the documentation category is the least relevant quality attribute to cause defects in the code based on

TABLE 5 Questions for Developers about the Quality Attributes.

Q1	You notice that major classes have many responsibilities (WMC). Based on your experience, how would you classify the impact of modularization to create precise responsibilities for each class as a contributor to the defectiveness of the API?
Q2	You then notice that the coupling between the objects is also high (CBO). How would you classify the impact of loosely coupled objects to avoid defects in the API?
Q3	In the end, you notice that some classes have too many attributes (NOA). How would you classify the impact of smaller classes to avoid defects in the API?
Q4	You also noticed that the API has little to no documentation. How would you classify the impact of the lack of proper documentation to contribute to the defectiveness of the API?

the developers' perceptions, although 60% of the developers consider this quality attribute as very important or important. Therefore, we rely on these survey questions to answer the main research question. Do machine learning models and developers agree in terms of software features that generate defects?

The survey results do not always agree with machine learning models since developers believe that code complexity is the most relevant quality attribute to prevent software defects, while machine learning models indicate documentation as the most relevant quality attribute.

3.3 | Thematic Analysis

The survey includes an open question that allowed us to collect information about quality attributes that are not present in the study. As a result, developers could enter specific software features that they believe are relevant to the defectiveness of software classes. Among the total number of developers (fifty developers), twenty-two (representing 44% of the developers) entered the software features that they believe are relevant to the defectiveness of the code and are not available in the static software features. As the field is open, we got all sorts of comments from the developers' perceptions. To analyze the open-field, two research members conducted the thematic analysis over the software features.

In the first stage of the coding analysis, the researchers individually classified each comment into several categories. They then discussed the quality attributes with a third member of the research team to consolidate the comments into reliable categories that can enhance future interactions in this study. Consequently, we developed a list of ten quality attributes based on the comments of twenty-two developers. The list of quality attributes includes: (i) testing, (ii) cohesion, (iii) coupling, (iv) code guidelines, (v) documentation, (vi) team practices, (vii) inheritance, (viii) size, (ix) complexity, and (x) invalid. We also determined that the research team responsible for the categorization could use up to three labels (i.e., quality attributes) per comment.

Finally, in the second stage of the coding analysis, the researchers analyzed the comments and categorized the quality attributes based on the developers' comments. Figure 5 shows the quality attributes created after the thematic analysis. The main quality attribute that the developers selected is testing. In this case, eight developers pointed out that testing is important to avoid defects, and it was not included in the quality attributes. Although we are aware of the impact of testing the source code to guarantee high levels of software quality, this characteristic of the code is not a static software metric. Testing the code is a dynamic aspect of the source code that we have not used to predict defects. Second, four developers think that both cohesion and coupling are important quality attributes not present in the quality attributes. These quality attributes are static and, in fact, they were included in the unified dataset[21,22]. However, the SHAP analysis did not include them in the top-10 features that we focused on in this investigation.

Complementary, three developers believe that the Documentation and Code Guidelines are relevant to avoid defects in the source code. The code documentation is one of the quality attributes included in the survey. As an example, one developer (P1) pointed out that "Documentation should be intelligible around all development stages (QA, design, product owner). For me, is the most important issue raised on this survey". On the other hand, we did not include the code guidelines in the static feature set because they are not a

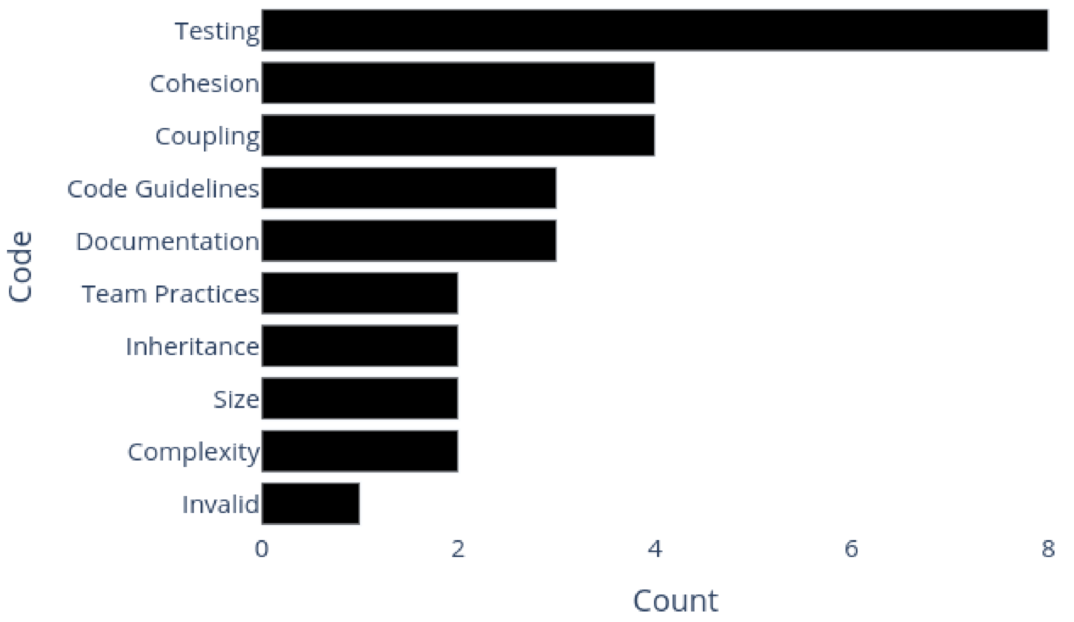

FIGURE 5 Thematic Coding of the Open-Field.

static metric available on the dataset. We consider the code guidelines related to design choices, such as design patterns. For instance, a developer (P2) commented that "Some well-defined design pattern for the project is also a very important thing".

Finally, two developers believed that Team Practices, Inheritance, Size, and Complexity are important for the defectiveness of their code. Among these options, inheritance is not included in the study, although we could include some software features representing them. For instance, DIT (Depth of Inheritance Tree) and NOC (Number of Children) are common software features representing this quality attribute. We consider team practices as code agreements between the team's members (i.e., developers, project managers, quality assurance, among others). As an example, a developer (P3) pointed out that "I would consider testing (at least unit), clean code and language standard conventions". Note that the comment also mentions testing the code. We consider one invalid response in the open field as the developer did not answer the question about the additional quality attributes.

In conclusion, the open-field thematic analysis was instrumental in identifying quality attributes that were not originally considered in our study. While some quality attributes are associated with dynamic code aspects (e.g., testing) or team agreements (e.g., coding guidelines and practices), we found that cohesion and coupling are two static attributes that may be worth exploring in future investigations.

3.4 | Implications

The finding that developers prioritize different quality attributes than what actually matters for machine learning models can have significant implications. First, it highlights a potential gap between the expectations of developers and the actual requirements for building effective machine learning models for defect prediction. This gap can lead to suboptimal models that do not perform as well as expected. Another implication is that there may be a need for better collaboration between developers and data scientists or machine learning experts. By working together, these professionals can ensure that all important aspects of machine learning models are being considered and prioritized appropriately.

Furthermore, the finding that models prioritize documentation and developers prioritize complexity suggests that there may be a need for more automated tools and processes that can help address these differences in priorities. For example, automated documentation tools can help ensure that important documentation is generated and maintained, while automated complexity analysis tools can help identify potential issues related to model complexity. Therefore, this finding highlights the importance of considering all aspects of machine learning models, even those that may not be intuitive to developers. By doing so, we can improve the quality and effectiveness of these models.

3.5 | Discussion

Our study aimed to explore the differences between developers' perceptions of important features for predicting defects in machine learning models and the actual importance of these features as determined by the models. To achieve this, we first computed the most important categories of features for predicting defects and then surveyed developers to understand their perceived importance of these categories of features. While our results showed that there are notable differences between developers' opinions and model predictions, we acknowledge that our study does not provide a quantitative measure of the difference between these two perspectives. This is an important limitation of our study, and we recognize the need for future research to explore this issue further using more sophisticated quantitative methods. Our study highlights the need for further research to understand the reasons for these discrepancies and to explore ways to bridge the gap between developers' perceptions and the actual requirements for machine learning models. One potential direction for future research is to conduct in-depth interviews with developers to gain a more nuanced understanding of their perceptions and priorities when building machine learning models.

4 | THREATS TO VALIDITY

This study has some limitations that could potentially threaten our results presented in Section 3. These threats are divided into four categories[64]: internal, external, construct, and conclusion validities. Next, we discuss in detail each threat to validity and how they could impact our conclusions.

External Validity. Threats to external validity concern the extent to which the results of a study can be generalized beyond the specific context of the study[64]. In our case, the primary threat to external validity is related to the programming language we used to explore the software features and machine learning models. Our study focused solely on the Java programming language, which limits the extent to which our findings can be generalized to other programming languages.

Although our survey had a general-purpose approach and did not require respondents to have experience with Java, we cannot fully evaluate the impact of the programming language on the machine learning models or the perceptions of the developers. It is worth noting, however, that Java is the third most commonly used programming language among developers who participated in our survey. This suggests that our findings may be more applicable to programming languages that share similarities with Java. Overall, the limited scope of our investigation in terms of programming languages is a potential limitation that must be acknowledged[11]. Further research could explore the effects of different programming languages on software features and machine learning models to improve the external validity of our findings.

Internal Validity. Threats to internal validity concern the extent to which we can accurately attribute the observed outcomes to the treatment used in an experiment[64]. In our study, the primary threat to internal validity is related to the software features used to generate the main explanation that guided our research[15]. While we applied an extensive data cleaning process to address most of the dataset problems[44], we cannot be certain that the dataset we collected is completely error-free. As with any stored data, there is always a risk of erroneous information being introduced into the dataset, especially in the complex context of software development. Another issue with our data is the imbalanced distribution of defects, with only around 20% of instances containing defects. To mitigate the impact of this imbalance, we applied an undersampling technique called SMOTE[65]. However, this technique is not without its own limitations and may not fully address the issue.

Furthermore, the number of survey respondents does not represent the broader population of software engineers. This could introduce bias into our results and limit the generalizability of our findings. To address this, we could consider increasing the number of survey participants or exploring alternative methods for collecting data, such as interviews or focus groups. Overall, while we made efforts to mitigate threats to internal validity in our study, there are still several limitations that must be acknowledged. These include potential errors in our dataset, the imbalanced distribution of defects, and the limited number of survey respondents. Future research could explore ways to address these limitations and improve the internal validity of our findings.

Another internal limitation of our study is the lack of an explanation for the main finding, which showed that developers do not agree with machine learning models on the importance of certain features. For example, our findings showed that developers thought complexity was more important than documentation, while the opposite was true for machine learning models. One possible explanation for this discrepancy is that there may be a mismatch between what developers think complexity is and what the metrics actually measure. It is possible that developers perceive complexity in terms of factors that are not captured by the metrics used in our study. Alternatively, it is possible that the machine learning models are using different types of data or metrics to assess complexity compared to what developers consider when building software.

Construct Validity. According to Wohlin (2012), construct validity refers to the extent to which an experiment's design and measures align with the underlying theory[64]. In our study, the primary threat to construct validity pertains to the questions we posed. Specifically, we asked developers to assess the impact of specific quality attributes as defect predictors using a predefined set of categories. While this approach enabled us to derive a ranking of developers' perceptions within a particular scenario, it may have constrained the depth and accuracy of their responses. To address this limitation, we recommend augmenting our investigation with quantitative research that delves more deeply into developers' opinions and experiences. Furthermore, enhancing the qualitative analysis, such as conducting interviews or surveys with open-ended questions, would allow developers to express their views on a broader spectrum of software features beyond those predefined in our study.

An issue that surfaced within the open field was the inclusion of software features that fell outside the intended scope, such as code guidelines, team practices, and testing. We acknowledge that our instructions may not have sufficiently emphasized the importance of concentrating on static software attributes, such as coupling and cohesion. Consequently, we were unable to incorporate certain responses provided by developers, potentially impacting the overall quality of our dataset.

Conclusion Validity. According to Wohlin (2012), threats to conclusion validity concern the extent to which we can accurately determine whether the treatment used in an experiment is responsible for the observed outcome[64]. In our study, we used the feature WMC to represent the complexity quality attribute. However, after retraining our models and using SHAP to identify the top-10 features, we discovered that WMC was not among them. This raises concerns about the conclusion validity of our study, as we cannot be certain that NL (Nesting Level), which did appear in the top-10, is an accurate representation of WMC as used in our survey. To address this threat to conclusion validity, we may need to revisit our survey design and consider alternative ways of measuring complexity. For example, we could explore other code metrics that are more strongly correlated with software complexity, or we could conduct a more extensive literature review to identify established measures of complexity that have been used in previous studies.

Alternatively, we could use our current results as a starting point for further investigation. By analyzing the relationship between NL and WMC in more detail, we may be able to identify patterns or factors that contribute to the observed differences. This could lead to new insights into how different aspects of code complexity are related and how they can be effectively measured and managed. In summary, the absence of WMC in the top-10 features identified by our models is a significant threat to the conclusion validity of our study. To address this, we may need to explore alternative measures of complexity or conduct further analysis to identify the underlying factors contributing to this discrepancy.

5 | RELATED WORK

This section presents a review of relevant literature on the application of machine learning models to predict defects in source code. We begin by introducing the concept of software defects as defined in the literature (Section 2.1). Next, we discuss previous studies that utilized source code metrics to predict defects in software projects (Section 5.1). Furthermore, we delve into how the current literature explains software defects (Section 5.2). Finally, we examine research on measuring developers' perceptions towards software defect prediction (Section 5.3).

5.1 | Learning from Source Code Metrics

Among the many efforts to build machine learning models to predict software defects, the use of code metrics are considered a great source of defect models. Several studies[7,2,1,66] share the ability of applying code metrics for the defect prediction. However, they vary in terms of accuracy, complexity, target programming language, the input prediction density, and the machine learning models. Research studies also rely on source code metadata[1] and software metrics[7,2] as features to machine learning-based algorithms. For instance, Wang et al.[1] studied the impact of using the program's semantic as the prediction model's features. The authors used deep learning networks to automatically learn semantic features from token vectors obtained from abstract syntax trees[1]. In a similar approach, Xu et al.[66] employed a non-linear mapping method to extract representative features by embedding the original data into a high-dimension space. Their results achieved average F-measure, g-mean, and balance of 0.480, 0.592, and 0.580[66], respectively.

The current literature applies several software metrics for defect prediction. As an example, Menzies et al.[7] presented defect classifiers using code attributes defined by McCabe and Halstead metrics. They concluded that the choice of the learning method is more important than which subset of the available data we use for learning the software defects. The study also discusses the usefulness of defect models for software development[7]. From a different perspective, Jing et al.[2] used a dictionary learning technique to predict software defects by using characteristics of software metrics mined from open source software projects. They used datasets from NASA projects as test data to evaluate the proposed method, which achieved a recall value of 79%, improving the recall by 15% compared to other methods previously employed to predict defects in the same dataset[2].

Some studies investigate cross-project defect prediction with cross-company defect prediction[56,67]. For instance, Fukushima et al.[56] explored cross-project prediction models within the context of just-in-time prediction. Their results indicate no relationship between project prediction performance and cross-project prediction performance. Furthermore, they conclude that just-in-time prediction models built using projects with similar characteristics or applying ensemble methods usually perform well in a cross-project context[56]. In a similar approach, Turhan et al.[67] used cross-company data for building localized defect predictors. They used principles of analogy-based learning to cross-company data to fine-tune these models for localization. The authors used static code features extracted from the source code, such as complex software features and Halstead metrics. The paper concludes that cross-company data are useful in extreme cases and when within-company data is not available[67].

Similarly, He at al.[68] investigate defect prediction based on data selection. The authors propose a brute force approach to select the most relevant data for learning the software defects. To do so, they experiment with three large-scale experiments on 34 datasets obtained from ten open source projects. They conclude that training data from the same project does not always help to improve the prediction performance[68]. In the same direction, the study of Turhan et al.[69] evaluate the effect of mixing data from different projects stages. In this case, the authors use within and cross-project data to improve the prediction performance. They show that mixing project data based on the same project stage does not significantly improve the model performance. For this reason, they conclude that optimal data for defect prediction is still an open challenge for researchers[69].

On the other hand, our investigation differs from the aforementioned studies. We employed an ensemble of the top 5 machine learning models to create an optimized model capable of effectively predicting defects. Our findings suggest that Random Forest, LightGBM, Extra Trees, Gradient Boosting Machine, and KNN perform marginally better than other baseline models. Utilizing this model, we were able to explain the defects and compare the model results with the developers' perception.

5.2 | Explaining Software Defects

Software defect explainability is a relatively recent topic in the literature[70,33,71]. Mori and Uchihira[70] analyzed the trade-off between accuracy and interpretability of various models. The experimentation displays a comparison between the balanced output that satisfies both accuracy and interpretability criteria. Likewise, Jiarpakdee et al.[33] empirically evaluated two model-agnostic procedures, Local Interpretability Model-agnostic Explanations (LIME)[72] and BreakDown techniques. They improved the results obtained with LIME using hyperparameter optimization, which they called LIME-HPO. This work concludes that model-agnostic methods are necessary to explain individual predictions of defect models. Finally, Pornprasit et al.[71] proposed a tool that predicts defects using the Python programming language. The input data consists of software commits, and the authors compare its performance with the LIME-HPO[33]. They conclude that the results are comparable to the state-of-the-art technology to explain models.

Another approach to explainable defect prediction involves a model that selects features for predicting defects[34]. This model enables us to interpret defect models using a minimal set of software features that span several programming languages. Our research concludes that explaining software defects is a challenging problem due to their variation depending on the internal characteristics of the project. In a similar vein, we employed machine learning techniques to predict defects using classic metrics like Halstead and McCabe metrics[73]. After conducting a survey with developers of diverse backgrounds, we found that the model was readily understandable, as the majority of respondents were able to use the model to reason about defects in their source code. In contrast, our study involves a comparison of developers' perceptions with machine learning models for predicting defects, and we utilize SHAP to explain the model results.

5.3 | Developers' Perception

Pantiuchina et al. aims to bridge the gap between the use of code quality metrics and developers' perceptions of code quality improvement[74]. The paper investigates whether quality metrics are able to capture code quality improvement as perceived by developers, which is a critical assumption for the effective use of code quality metrics in identifying design flaws and recommending refactoring. The authors use commit messages from developers that clearly state their aim of improving specific quality attributes and then assess the change in those attributes using state-of-the-art metrics. The study found that, in many cases, quality metrics were not able to capture the quality improvement as perceived by developers, highlighting the limitations of relying solely on code quality metrics to assess code quality. This approach differs from previous studies that relied on surveys to investigate whether metrics align with developers' perception of code quality, providing a more objective and empirical assessment of the effectiveness of quality metrics in capturing code quality improvement.

Wan et al. explores the potential value of defect prediction in practice and investigates the perceptions and expectations of practitioners in contrast to research findings[75]. Through a mixed qualitative and quantitative study, the authors collected hypotheses from open-ended interviews and literature reviews and conducted a validation survey with 395 responses from practitioners across five continents. The study found that the majority of respondents are willing to adopt defect prediction techniques, but there is a disconnect between practitioners' perceptions and well-supported research evidence regarding defect density distribution and the relationship between file size and defectiveness. Additionally, the study revealed that some practitioners exhibit inconsistencies between their behavior and perception regarding defect prediction, and that defect prediction at the feature level is the most preferred level of granularity. The study also identified reasons why practitioners are reluctant to adopt defect prediction tools and noted features that practitioners expect from such tools.

On the other hand, our work focuses on exploring developers' perceptions of machine learning models used for software defect prediction. Specifically, we conducted a survey with experienced developers to understand how they evaluate different quality attributes for defect prediction. The survey targeted a random sample of developers who have contributed to at least 10 repositories on GitHub in the past two years. Our study reveals a misalignment between the perceptions of developers and the focus of machine learning models, which tend to give higher importance to documentation. This research has implications for improving the effectiveness of software defect prediction models and highlights the importance of understanding the developers' perspective in developing such models.

6 | CONCLUSION

This study compared developers' perceptions of software defects with a machine learning model. To explain the defects, we used a model-agnostic technique known as SHAP, which analyzed features related to software characteristics like code complexity, size, coupling, and documentation. We then used these features to propose a survey to developers and compared their perceptions with the machine learning model's predictions. Our results showed that developers' perceptions were quite different from the machine learning models. While developers believed that the complexity of software features was the main cause of defects, the machine learning models identified documentation as the most significant quality attribute related to defects. This finding is interesting because it contradicts the common understanding of quality attributes and their impact on defects. In conclusion, this study highlights the importance of using machine learning to supplement developers' perceptions and enhance the accuracy of software defect prediction. By incorporating both human expertise and machine learning techniques, we can gain a more comprehensive understanding of software quality and improve our software development processes.

In future steps of this research, we aim to explore different techniques for explaining defects in Java projects. While we used a black-box concept to explain software defects in this paper (represented by SHAP values), the current literature offers alternative techniques, such as the glass-box concept. Glass-box models may produce different predictions based on the software features used in this paper, but they tend to achieve lower accuracy than black-box models.

Therefore, our future work will involve comparing these techniques. We will investigate how glass-box models perform in explaining defects and compare their accuracy with that of black-box models. Another possibility for this research is to classify software features before explaining their impact on software defects. By doing this, we can determine which features are more critical for software quality and prioritize them in our software development processes. In addition, we plan to join dynamic and static features to improve the accuracy of defect prediction. Our results indicate that developers consider a lack of testing to be a significant indicator of defects in their code, and we aim to incorporate this insight into our analysis. Overall, our goal is to develop a more comprehensive understanding of software quality and defect prediction by exploring various techniques for explaining software defects. Through this research, we hope to enhance the accuracy of defect prediction and improve the quality of software development.

ACKNOWLEDGMENTS

This research was partially supported by Brazilian funding agencies: CNPq, CAPES, and FAPEMIG.

Financial disclosure

None reported.

Conflict of interest

The authors declare no potential conflict of interests.

References

1. Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. International Conference of Software Engineering (ICSE) 2016.
2. Jing X, Ying S, Zhang Z, Wu S, Liu J. Dictionary learning based software defect prediction. International Conference of Software Engineering (ICSE) 2014.
3. Hassan AE. Predicting faults using the complexity of code changes. International Conference of Software Engineering (ICSE) 2009.
4. D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. 7th IEEE Working Conference on Mining Software Repositories (MSR) 2010.
5. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. The Impact of Automated Parameter Optimization on Defect Prediction Models. Transactions on Software Engineering (TSE) 2019.
6. Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. International Conference on Software Engineering (ICSE) 2005.
7. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. Transactions on Software Engineering (TSE) 2007.
8. Moser R, Pedrycz W, G. S. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. International Conference on Software Engineering (ICSE) 2008.
9. Levin S, Yehudai A. Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes. Proceedings of the 13rd International Conference on Predictor Models in Software Engineering (PROMISE) 2017.
10. Nagappan N, Ball T, Zeller A. Mining Metrics to Predict Component Failures. International Conference on Software Engineering (ICSE) 2006.
11. Cui C, Liu B, Wang S. WIFLF: An approach independent of the target project for cross-project defect prediction. Journal of Software: Evolution and Process (JSEP) 2022.
12. Jureczko M, Madeyski L. Towards Identifying Software Project Clusters with Regard to Defect Prediction. Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE) 2010.
13. Herbold S. CrossPare: A Tool for Benchmarking Cross-Project Defect Predictions. International Conference on Automated Software Engineering Workshop (ASEW) 2015.
14. Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A. Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering (ASE) 2010.
15. Santos G, Veloso A, Figueiredo E. The Subtle Art of Digging for Defects: Analyzing Features for Defect Prediction in Java Projects. International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) 2022.
16. Santos G, Veloso A, Figueiredo E. Understanding Thresholds of Software Features for Defect Prediction. 36th Brazilian Symposium on Software Engineering (SBES) 2022.

17. Ghotra B, McIntosh S, Hassan AE. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. International Conference on Software Engineering (ICSE) 2015.

18. Tantithamthavorn C, Hassan AE. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) 2018.

19. Ni C, Chen X, Xia X, Gu Q, Zhao Y. Multitask defect prediction. Journal of Software: Evolution and Process (JSEP) 2019.

20. Menzies T, Distefano J, Orrego A, Chapman R. Assessing predictors of software defects. In Proceedings, Workshop on Predictive Software Models (PROMISE) 2004.

21. Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T. A Public Unified Bug Dataset for Java. Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE) 2018.

22. Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. Software Quality Journal (SQJ) 2020.

23. Tóth Z, Gyimesi P, Ferenc R. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. Computational Science and Its Applications (ICCSA) 2016.

24. Jureczko M, D. SD. Using Object-Oriented Design Metrics to Predict Software Defects. Models and Methods of System Dependability (MMSD) 2010.

25. Eken B, Tufan S, Tunaboylu A, Guler T, Atar R, Tosun A. Deployment of a change-level software defect prediction solution into an industrial setting. Journal of Software: Evolution and Process (JSEP) 2021.

26. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 1990. doi: 10.1109/IEEESTD.1990.101064

27. Haskins B, Stecklein J, Dick B, Moroney G, Lovell R, Dabney J. Error Cost Escalation Through the Project Life Cycle. INCOSE International Symposium 2004.

28. Turhan B, Bener A. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. Data & Knowledge Engineering 2009.

29. Jiang T, Tan L, Kim S. Personalized defect prediction. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2013.

30. Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead Jr EJ. Does bug prediction support human developers? findings from a google case study. International Conference of Software Engineering (ICSE) 2013.

31. Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C. Mining Software Defects: Should We Consider Affected Releases?. IEEE/ACM 41st International Conference on Software Engineering (ICSE) 2019.

32. Tantithamthavorn C, McIntosh S, Hassan AE, Ihara A, Matsumoto K. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. International Conference on Software Engineering (ICSE) 2015.

33. Jiarpakdee J, Tantithamthavorn C, Dam HK, Grundy J. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. Transactions on Software Engineering (TSE) 2020.

34. Santos G, Figueiredo E, Veloso A, Viggiato M, Ziviani N. Understanding machine learning software defect predictions. Automated Software Engineering Journal (ASEJ) 2020.

35. Lundberg SM, Lee S. A unified approach to interpreting model predictions. Conference on Neural Information Processing Systems (NIPS) 2017.

36. Lundberg SM, Erion GG, Lee S. Consistent Individualized Feature Attribution for Tree Ensembles. Computing Research Repository (CoRR) 2018.

37. Lundberg SM, Erion G, Chen H, et al. From local explanations to global understanding with explainable AI for trees. Nature Machine Intelligence 2020.

38. Lundberg SM, Nair B, Vavilala MS, et al. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. Nature Biomedical Engineering 2018.

39. Sayyad S. J, Menzies T. The PROMISE Repository of Software Engineering Databases.. http://promise.site.uottawa.ca/SERepository; 2005.

40. Zimmermann T, Premraj R, Zeller A. Predicting Defects for Eclipse. International Workshop on Predictor Models in Software Engineering (PROMISE) 2007.

41. Hall T, Zhang M, Bowes D, Sun Y. Some Code Smells Have a Significant but Small Effect on Faults. Transactions on Software Engineering and Methodology (TOSEM) 2014.

42. Ferenc R, Tóth Z, Ladányi G, Siket I, Gyimóthy T. Unified Bug Dataset. https://doi.org/10.5281/zenodo.3693686; 2020

43. Ali M. PyCaret: An open source, low-code machine learning library in Python. Read the Docs 2020.

44. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. The Jinx on the NASA Software Defect Data Sets. International Conference on Evaluation and Assessment in Software Engineering (EASE) 2016.

45. Lin Z, Ding G, Hu M, Wang J. Multi-Label Classification via Feature-Aware Implicit Label Space Encoding. International Conference on International Conference on Machine Learning (ICML) 2014.

46. Tantithamthavorn C, Hassan AE, Matsumoto K. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. Transactions on Software Engineering (TSE) 2019.

47. Agrawal A, Menzies T. Is better data better than better data miners?: on the benefits of tuning SMOTE for defect prediction. International Conference of Software Engineering (ICSE) 2018: 1050–1061.

48. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 1996; 22(10).

49. Stroulia E, Kapoor R. Metrics of Refactoring-based Development: An Experience Report. 7th International Conference on Object Oriented Information Systems2001 2001.

50. Abdullah AlOmar E, Wiem Mkaouer M, Ouni A, Kessentini M. Do Design Metrics Capture Developers Perception of Quality? An Empirical Study on Self-Affirmed Refactoring Activities. Journal of Machine Learning Research (JMLR) 2019.

51. Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley . 1999.

52. Aghajani E, Nagy C, Linares-Vásquez M, et al. Software Documentation: The Practitioners' Perspective. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE) 2020.

53. Cawley GC, Talbot NL. On Over-Fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. Journal of Machine Learning Research (JMLR) 2010.

54. Akiba T, Sano S, Yanase T, Ohta T, Koyama M. Optuna: A Next-Generation Hyperparameter Optimization Framework. International Conference on Knowledge Discovery & Data Mining (SIGKDD) 2019.

55. Kursa MB, Jankowski A, Rudnicki WR. Boruta - A System for Feature Selection. Fundamenta Informaticae 2010.

56. Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N. An empirical study of just-in-time defect prediction using cross-project models. Working Conference on Mining Software Repositories (MSR) 2014.

57. Ke G, Meng Q, Finley T, et al. LightGBM: A Highly Efficient Gradient BoostingDecision Tree. 31st Conference on Neural Information Processing System 2017.

58. Bui XN, Nguyen H, Soukhanouvong P. Extra Trees Ensemble: A Machine Learning Model for Predicting Blast-Induced Ground Vibration Based on the Bagging and Sibling of Random Forest Algorithm. Proceedings of Geotechnical Challenges in Mining, Tunneling and Underground Infrastructures (ICGMTU) 2022.

59. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. IEEE Transactions on Software Engineering (TSE) 2017.

60. Xuan X, Lo D, Xia X, Tian Y. Evaluating Defect Prediction Approaches Using a Massive Set of Metrics: An Empirical Study. Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC) 2015.

61. Thung F, Bissyandé TF, Lo D, Jiang L. Network Structure of Social Coding in GitHub. 17th European Conference on Software Maintenance and Reengineering (CSMR) 2013.

62. Gousios G, Vasilescu B, Serebrenik A, Zaidman A. Lean GHTorrent: GitHub Data on Demand. Proceedings of the 11th Working Conference on Mining Software Repositories (MSR) 2014.

63. Liu Q, Basu D, Goel S, Abdessalem T, Bressan S. How to Find the Best Rated Items on a Likert Scale and How Many Ratings Are Enough. Database and Expert Systems Applications (DEXA) 2017.

64. Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A. Experimentation in Software Engineering. Springer . 2012.

65. Guo S, Dong J, Li H, Wang J. Software defect prediction with imbalanced distribution by radius-synthetic minority over-sampling technique. Journal of Software: Evolution and Process (JSEP) 2021.

66. Xu Z, Liu J, Luo X, Zhang T. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. International Conference on Software Analysis, Evolution and Reengineering (SANER) 2018.

67. Turhan B, Menzies T, Bener AB, Di Stefano J. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering (EMSE) 2009.

68. He Z, Shu F, Yang Y, Li M, Wang Q. An investigation on the feasibility of cross-project defect prediction. Automated Software Engineering (ASE) 2012.

69. Turhan B, Tosun A, Bener A. Empirical Evaluation of Mixed-Project Defect Prediction Models. Proceedings of the 37th Conference on Software Engineering and Advanced Applications (SEAA) 2011.

70. Mori T, Uchihira N. Balancing the trade-off between accuracy and interpretability in software defect prediction. Empirical Software Engineering (EMSE) 2018.

71. Pornprasit C, Tantithamthavorn C, Jiarpakdee J, Fu M, Thongtanunam P. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. International Conference on Automated Software Engineering (ASE) 2021.

72. Ribeiro MT, Singh S, Guestrin C. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. International Conference on Knowledge Discovery and Data Mining (KDD) 2016.

73. Santos G, Figueiredo E, Veloso A, Viggiato M, Ziviani N. Predicting Software Defects with Explainable Machine Learning. Brazilian Symposium on Software Quality (SBQS) 2020.

74. Pantiuchina J, Lanza M, Bavota G. Improving Code: The (Mis) Perception of Quality Metrics. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) 2018.

75. Wan Z, Xia X, Hassan AE, Lo D, Yin J, Yang X. Perceptions, Expectations, and Challenges in Defect Prediction. IEEE Transactions on Software Engineering 2020.

76. Chen X, Mu Y, Qu Y, et al. Do different cross-project defect prediction methods identify the same defective modules?. Journal of Software: Evolution and Process (JSEP) 2020.

77. Gong L, Jiang S, Jiang L. An improved transfer adaptive boosting approach for mixed-project defect prediction. Journal of Software: Evolution and Process (JSEP) 2019.

78. Fayyad U, Irani K. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. International Joint Conferences on Artificial Intelligence (IJCAI) 1993.

79. Ma B, Zhang H, Chen G, Zhao pY, Baesens B. Investigating Associative Classification for Software Fault Prediction: An Experimental Perspective. International Journal of Software Engineering and Knowledge Engineering 2014.

AUTHOR BIOGRAPHY

How to cite this article: Santos G., Muzetti I., and Figueiredo E. (2022), Two Sides of the Same Coin: A Study on Developers' Perception of Defects, Q.J.R. Meteorol. Soc., 2017;00:1–6.