Using Machine Learning to Classify Role-Stereotypes of Classes

Author 1
Faculty1
University1
Email: author1@university1

Author2 & Author3
Department1
University2
Email: { author2, author3 }@university2

***Abstract*—Role stereotypes indicate generic roles that classes play in the design of systems, such as e.g. controller, information holder, or interfacer. Knowledge about the role-stereotypes can help in various tasks in software development and maintenance, such as program understanding, program summarization and quality assurance. This paper presents an automated machine learning-based approach for classifying the role-stereotype of classes in Java. We analyse the performance of this approach against a manually labelled ground truth for a sizable open source project (of 770+ Java classes) for the Android platform. Moreover, we compare our approach to an existing rule-based classification approach.**

The contributions of this paper include: an analysis of which machine learning algorithms and which features provide the best classification performance. This analysis shows that the Random Forest algorithm yields the best performance. We find however, that the performance of the algorithm varies a lot for classifying different role-stereotypes. In particular it performs poorly for rare role-types. Moreover, we illustrate how knowledge about role stereotypes unveils structural patterns about the anatomy of software design.

***Index Terms*—Software Design, Reverse Engineering, Program understanding**

I. INTRODUCTION

The concept of ”role stereotype” was introduced by Wirfs-Brock as a concept to denote ideal types of well-scoped responsibilities of classes [1]. Such role stereotypes indicate generic roles that classes play in the design of a system, such as e.g. controller, information holder, or interfacer. Knowledge about the role-stereotypes can help in various tasks in software development and maintenance, such as program understanding, program summarization and quality assurance.

Methods for automatically inferring the role-stereotype of classes have been studied by Dragan et al. [2] for C++. Moreno et al. [3] migrated this to Java. Both approaches are based on a collection of expert-designed rules that are applied to characteristics extracted from the source code of classes. This inference of role-stereotypes can be seen as an enrichment of reverse engineering, in particular in the area of uncovering design: Role-stereotypes indicate a generic type of responsibility which entails both the type of functions that a class should perform as well as how a class should interact with other classes. Hence a role-stereotype conveys part of the design intention of a class.

Several studies [4]–[9] have demonstrated the benefits of using stereotypes in various software development and maintenance activities, such as program comprehension, program design, quality assurance, and program summarization: Using knowledge about role-stereotypes in creating lay-outs of UML class diagrams improves the comprehensibility of the diagrams [4], [7]–[9]. Staron et al. [4] show the effectiveness of class stereotypes based in program comprehension. Alhindawi, et al. [10] suggests that enhancing the source code with stereotype information can help improve feature location in the software.

This paper has the following contributions:

1) In this paper, we present a machine learning approach to address the problem of automatically inferring the stereotype of classes in a software system. Using machine learning, we hope to produce a classifier that is more complete and more robust (i.e. covers as many of the classes of software systems as possible) than classifiers using thresholds-based rules. The robustness has been a point of improvement of the approach by Dragan en Moreno. Their rules consist of a collection of conditions on the (relative) quantities of *method stereotypes*; e.g. $\#mutators > 2 * \#accessors$. These conditions include thresholds that are based on a mix of theoretical argumentations and statistical techniques (based on [11]). However, Dragan states “The rules for stereotype identification are subjective and thresholds might vary depending on differences in subjects interpretations.”. Moreover, these rules leave a large number of classes in software systems unclassified. We suspect this is because these classes have characteristics that are not covered by any of the rules.
2) We analyse the performance of our approach against a manually labeled ground truth for a sizable open source project (of 770+ Java classes) for the Android platform. From this, we conclude which machine learning algorithm works best and which features are most important for different stereotypes. Moreover, we compare our approach to an existing rule-based classification approach. To our knowledge, this is the first publication of such dataset. The earlier work by Dragan used only 45 classes for validating their approach for C++. The paper by Moreno [3] does not include a validation of its performance.
3) We illustrate new uses of role-stereotypes: i) Discovery

of patterns of collaboration between role-stereotypes: We assert that this knowledge contribution fundamental insights regarding the anatomy of software design. ii) Introduce stereo-type specific rules for quality assurance of the design.

This paper has the following structure: We first explain the key concept of role stereotype and discuss related work. Next, we explain our research methodology. As part of this we explain the taxonomy of role-stereotypes that we use in our study and how this relates to other taxonomies. Then we analyze the performance of various machine learning algorithms for this classification task, and identify the most influential features. We illustrate new uses of role-stereotypes. We end with discussion, conclusions and future work.

II. Class Role Stereotypes

Wirfs-Brock [12] proposed an object-oriented design-approach based on the notion that each software object should have a well-defined responsibility in order to play one of a few generic roles in a system's design. Wirfs-Brock classified the roles of software objects into six stereotypes:

(CT) Controller: objects designed to make decisions and control complex tasks,

(CO) Coordinator: objects that do not make many decisions, but in a rote or mechanical way, delegate work to other objects

(IH) Information holder: object designed to know certain information and provide that information to others.

(IT) Interfacer: objects that transform information and requests between distinct parts of a system. It can be a user interfacer object that interacts with users. An interfacer can communicate with external systems or between internal subsystems.

(SP) Service provider: objects that perform work and offer services to others on demand.

(ST) Structurer: objects that maintain relationships between objects and information about those relationships. Structurers might pool, collect, and maintain groups of objects.

This taxonomy aims for orthogonal non-overlapping categories. However, there may be situations where a class can play different roles towards different collaborators.

For our study it is important to realize that Wirfs-Brock suggests to use role stereotypes while *designing* a system. In our study, we aim to establish the role-stereotypes based on the implementation of a system. In general, the classes that end up in an implementation are not ideal. In fact they may mix two (or more) responsibilities.

III. Related Work

Dragan et al. [13] proposed an automated tool to detect stereotypes of *methods* in the C++ programming language. They define a taxonomy of methods such as structural (accessor and mutator), collaborational, and creational. They propose several rules to determine method stereotypes based on the type of the method, the return type, and the way in which the method modifies the state of the class. On top of their method for classifying method stereo-types, Dragan et al. create rules to determine class stereotypes [2]. The proposed class stereotypes are: entity, minimal entity, data provider, commander, boundary, factory, controller, pure controller, large class, lazy class, degenerate, data class, and small class (see Table VI). While several of these seem close to Wirf-Brocks's, the Dragan classification is presented as being empirically derived from studying the source code of 21 open source projects. In section VII-A we will discuss the relation between these taxonomies in more detail.

Based on the work of Dragan, Moreno and Marcus propose a method for the identification of class stereotypes in the Java programming language [3]. For this, they adapted the procedures from [2] and provided additional method- and class stereotypes. In the classifier approach by Dragan and Moreno's rules are not orthogonal/disjoint: different rules each may assign different stereotypes to a single class. Moreno's subsequent research has mostly used this classifier for automatically generating summaries of classes in natural language [14], and seems not to have continued in improving its performance.

Budi et al. [15] built an automated tool to label class stereotypes and use this to detect design flaws based on the rules that associated with each stereotype. Their taxonomy consists of 4 categories: boundary, control, regular entity and data manager. These category are chosen because there exist design rules that describe how these stereotypes should be allocated to typical layers of 3-tiered software architectures and how they are supposed to collaborate. They used SVM to automatically labels classes into categories. Then, they use rules that explained about the relationship between the stereotypes namely robustness and well-formedness rules to detects the design flaws in the software system. This illustrates an example of using class stereotyping to perform quality assurance at the architecture level of design of system.

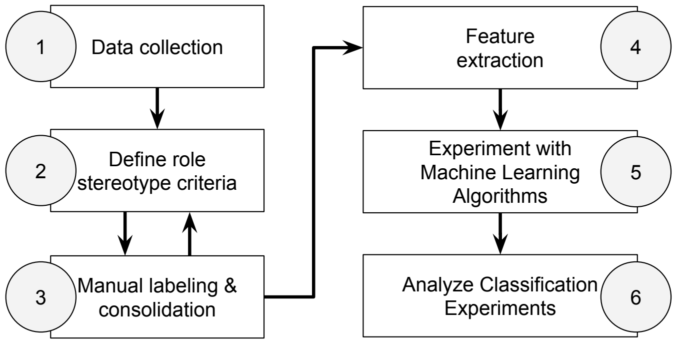
The GoF patterns also represent idealized patterns of software design. However, these patterns manifest themselves only in a small number of the classes of source code; i.e. not every class is involved in a design pattern. Hence this taxonomy is not suitable for reverse engineering of design intent for all classes of software systems. For completeness, we mention one approach by Fontana et.al. [16] that has used machine learning to identify design patterns in source code.

IV. Methodology

Figure 1 shows a high level overview of our research methodology. First, we select a case study and collect its source code. Second, we define a ground truth. Then, we extract features from the source code that are to be used by the machine learning algorithms. After that, we experiment with various machine learning algorithms and evaluate their performance. Next, we elaborate these steps in more detail.

A. Data Collection

In this research, we use the K-9 Mail application as our case study. This application is available in the Google Play Store. Reasons for choosing the K9 application are: i) it is an



1
Data collection
2
Define role stereotype criteria
3
Manual labeling & consolidation
Feature extraction
4
Experiment with Machine Learning Algorithms
5
Analyze Classification Experiments
6

Fig. 1. Proposed methodology

active open source project, ii) uses Java, iii) it is non-trivial size (750+ classes).

We use the K-9 Mail source code branch Master downloaded from its GitHub page[1] on November 22nd, 2017. We turned a small number of 'inner classes' we found in K9 into independent classes. Then, we removed unused import statements in every class (using the IntelliJ IDEA) to obtain the actual number of import statements used and the number of lines in the class. After these steps, we obtained 779 Java classes.

B. Ground Truth step 1: Criteria for Role Stereotypes

In order to produce a ground truth for machine learning, we first establish criteria to be used by human experts for manually classifying classes into role stereotypes. The initial criteria were obtained by the authors by studying the descriptions by Wirfs-Brock [12]. Then the criteria were refined and calibrated in follow up meetings where the authors had assessed additional sets of classes (details in section IV-C). The criteria can be divided into 3 categories: i) Criteria regarding characteristics of class; ii) Criteria regarding relationship between role stereotypes; and iii) Other criteria.

1) Criteria regarding characteristics of classes: These criteria focus on intrinsic (static) properties of classes. We take the structurer stereotype as first example: table I shows the criteria used to characterize Structurer classes. In this particular case we look into data types of attributes, library use, content of methods inside a class in order to get an impression whether the class is capable of organizing/manipulating collection of objects. As a second example, the *Information holder* may include persistence mechanisms (files or databases). Other class properties such as class name, getter/setter methods, etc. are used for other role stereotypes. A complete list of the criteria for all stereotypes can be found in the replication package of this study [17]. Some of the criteria are rather similar to the rules developed by Dragan and Moreno. For example, we both consider getter and setter methods as a sign for Information Holder (in our classification) and Entity/Data Provider (in Dragan's classification). However, different from Dragan and Moreno, our criteria look at absence or presence of features and not at the size or frequency of characteristics.

[1]https://github.com/k9mail/k-9

2) Criteria regarding relationship between roles: In [12] Wirfs-Brock mentions that "the roles an object plays imply certain kinds of collaborations". For this, we form a set of criteria that look at the collaborations stereotypes have with other classes. From Wirfs-Brock's theory of role stereotypes and collaborations, we come up with the graph in Figure 2. This demonstrates common relationship between different role stereotypes. This graph, for example, shows that information holders are used by controllers, service providers or structurers, but not commonly by coordinators or interfacers. Hence, the presence or absence of relations is used as criterion in the decision-making about the possible roles of a class.

The following are additional examples of criteria on the relationships between roles: *Structurer* may link to several *information holders*. A *service provider* can store information by collaborating with *information holder*- and *structurer* objects. An interfacer class, as an intermediate between different layers of a system, might collaborate with *coordinators* and *service providers* in each layer to conduct a cross-layer task. An *interfacer* class is often controlled by a *controller*.

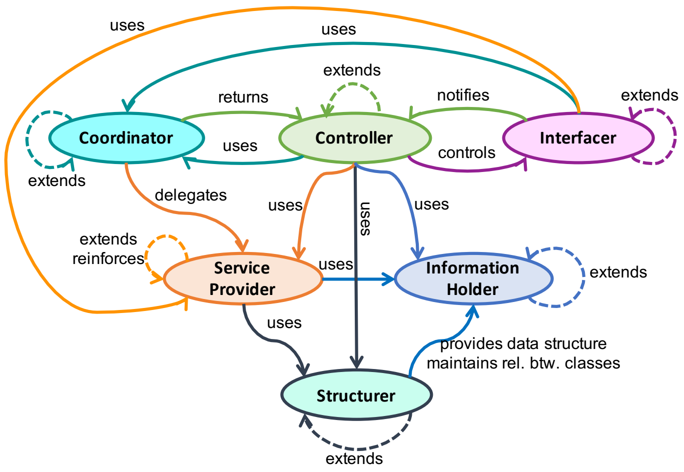


Fig. 2. Relationship between role stereotypes

Expert judgement is needed when multiple criteria apply: We have seen classes that carry characteristics of the service provider, but were ultimately classified as another role stereotype because their services were not used by any other classes.

TABLE I
PROPERTIES OF A STRUCTURER CLASS

What makes a class a Structurer?
- May contain user defined object type as attributes
- May extend/implement Java's Collection framework or equivalent
- Has method(s) to maintain relationships between objects
+ methods that manipulate the collection such as sort(), compare(), validate(), remove(), updates(), add(), etc.
+ methods that give access to a collection of objects such as get(index), next(), hasNext(), etc.

3) Other Considerations: We defined additional criteria aimed at essential difference in behaviour between different role stereotypes. For example, both Service Provider and Controller class may include some control-flow logic. The design intention of these classes suggest that decisions made by a controller should affect a broader control flow of the system, while decisions made by a service provider should mostly have effect on the flow within the class itself.

Secondly, sometimes classes may carry multiple roles. This possibility of multiple roles is also discussed by both Wirfs-Brock [12] (p.4) and Dragan [2]. In this case, experts discuss and choose one most prominent role for this class, and if any secondary role is identified, then this is also recorded.

C. Ground Truth step 2: Manual Labeling and Consolidation

We randomly chose 20 classes to label. Two of the authors and one additional Ph.D. student labelled these classes. They discussed any differences in classification, and refined the criteria based on this discussion. We repeated this steps three times until the criteria seemed sufficient/saturated. Next, two of these persons labeled all the remaining classes. A final discussion round took place between the two graders in order to resolve disagreement and hard-to-stereotype cases. The whole manual labeling process took about 60 hours in total (30 hours per person) spread out over 2 months. Ultimately, we established a ground truth of 779 labeled classes, which is all of K9 [17].

D. Feature Extraction

Our classification is based on static analysis of the Java source code of a system. We extract the features by creating the srcML [18] representation of the source code. The srcML tool outputs a list of classes in the source code in a standardized XML format. Then, we use multiple XPath queries to obtain the features of interest. We chose to use the following features because they correspond to the criteria used in the manual classification (see step 'Criteria for Role Stereotypes in IV-B).

1) loc: the number of lines in the class' source code. We assume that the controller and service provider stereotype will have more lines of code than the other.
2) numAttr: the number of attributes declared in the class. We assume that an information holder will have many attributes.
3) numImports: the number of import statements in the class. We assume that roles like controller, coordinator, interfacer, and service provider will have many import statements.
4) numMethod: the number of methods declared in the class. Constructors are excluded. We assume that a service provider will have many methods.
5) setters: the number of methods started with 'set' phrase. We assume that this method is a setter method, i.e., the method that modifies variable in information holder.
6) getters: the number of methods started with 'get' phrase. We assume that this method is a getter method, i.e., the method that accesses variable values in the information holder class.
7) isPersist: a boolean value that indicates whether a class has persistence features, i.e., implements a Serializable interface or importing database connectivity library. We assume that an information holder has persistence features.
8) isCollection: a boolean value that indicates whether a class is a subclass of Java's Collection library. We assume that a structurer will need it to maintain relations between objects.
9) numWordName: the number of words in the class name. We assume that information holder and structurer role stereotypes have a simple short name, while the others have a long name.
10) isOrEr: a boolean value that indicates whether the class name is ended with 'or' or 'er'. We assume that a controller or service provider name will end with that characters.
11) isController: a boolean value that indicates whether the class name is ended with 'controller' phrase. We assume that a controller will have controller phrase on their name.
12) numIfs: the number of conditional statements in the class body, i.e., if and switch statement. We assume that a controller has many conditional statements.
13) numParameters: the total number of parameters in all methods in the class. We assume that a service provider has many parameters.
14) numOutboundInv: the number of invocation to outside of the class methods. We assume that the coordinator and controller has many invocations outside of its class.
15) isStaticClass: a boolean value that indicates whether a class is a static class. We assume that a static class can represent a service provider or an information holder.
16) isInterface: a boolean value that indicates whether the source code file is a Java's interface. We assume that an interface can provide methods that must be implemented by a service provider or a structurer.
17) isInnerClass: a boolean value that indicates whether a class is an inner class. We assume that the inner class is an additional class that can provide functionality as an information holder or a service provider.
18) isClass: a boolean value that indicates whether the source code file is a class. We assume that a class can represent all of the role stereotypes.
19) isEnum: a boolean value that indicates whether the source code is a Java's enum. We assume that the enum type can represent an information holder.
20) numPublicMethods: the number of public methods inside the class. We assume that an information holder and an interfacer stereotype will have many public methods.
21) numPrivateMethods: the number of private methods inside the class. We assume that a controller, coordinator, and service provider can distribute the job on separate methods inside their class.
22) numProtectedMethods: the number of protected methods

inside the class. We assume that a controller, coordinator, and service provider can distribute the job on separate methods inside their class, but its subclass can still use or override the methods.

23) isAbstractClass: a boolean value that indicates whether a class is an abstract class. We assume that an abstract class can provide methods as a service provider and an abstract method that must be implemented by a service provider subclass.
24) classPublicity: the access modifier of the class. We assume that service providers and interfacer stereotypes offer public access so that other classes can access them.

From the list of features, we can see that most of the features are natural numbers. Several other features are boolean-valued, which are features started with 'is' on its name. We have a category variable, namely classPublicity, that we represent using one hot encoding technique into four additional features for each access modifier supported by Java language.

E. Machine Learning Classification Experiments

We experiment with 3 machine learning methods, namely Random Forest, Multinomial Naive Bayes, and Support Vector Machine. These three methods are widely used in machine learning research and provide good performance on various application. We use stratified 10-fold cross-validation to evaluate the performance of each method. The classification performance is measured using recall, precision, F1 score, and Matthews Correlation Coefficient (MCC).

We perform two experiments for machine learning algorithms. In the first experiment, we analyse which algorithm provides the best performance in classifying all role-stereotypes. For this we use each role-stereotype as a separate classification category. Hence this constitutes a multi-class classification. We explore the use of the SMOTE [19] resampling technique to handle the imbalanced distribution of role-stereotypes. We compare the performance between using the regular and the SMOTE resampling technique. We then will show the confusion matrix using 60:40 train test split.

In the second experiment, we analyse which features are important for classifying each individual role-stereotype. For this we use only 1 machine learning algorithm: the one that came out best in the first experiment. In this second experiment, we perform binary classifications for each stereotype; i.e. we use two categories: i) that specific stereotype, and ii) all other stereotypes together. We then evaluate the importance of the features in this classification. For this, we use the Scikit-toolkit for machine learning and its built in method to compute feature-importance based on Gini scores [20]. We can get this score by calculating the importance of the node for each feature split divided by the importance of all nodes in the tree, then normalize it by the sum of all feature importance values.

V. Experiment Results

In this section, we present experiment results on our classifier Class Role Identifier (CRI).

A. Multi-role Classification of all Stereotypes

CRI must classify each Java class of the K9 application into one of the six role stereotypes that we defined earlier. We use 1000 trees in the Random Forest classifier in response to the large number of features that we use. As for the Support Vector Machines, we use the linear kernel approach. Table II shows the average and standard deviation of precision, recall, and F1 score for this experiment.

TABLE II
Multi-roles classification result

	Precision	**Recall**	**F1-Score**
RF	0.63 ± 0.08	0.66 ± 0.05	0.62 ± 0.05
MNB	0.39 ± 0.25	0.41 ± 0.28	0.39 ± 0.26
SVML	0.32 ± 0.24	0.34 ± 0.26	0.32 ± 0.24
RF (SMOTE)	0.89 ± 0.06	0.89 ± 0.06	0.89 ± 0.06
MNB (SMOTE)	0.51 ± 0.06	0.49 ± 0.05	0.48 ± 0.06
SVML (SMOTE)	0.69 ± 0.03	0.69 ± 0.03	0.68 ± 0.03

From the F1 score in table II, we see that the recall and precision score are in the same range for each classifier. The F1 score is not too different from the recall and precision score. However, the performance of the classifier on the (imbalanced) dataset shown in the first three rows of the table is not very good. The best score for the imbalanced dataset is achieved by the random forest classifier.

The last three rows of the table show the result of the experiment using SMOTE resampling technique. We see a significant increase in performance on all classifier. Besides that, we can also see lower standard deviation on all scores compared to the imbalanced data experiment. The performance of the SVM classifier even becomes better than the Multinomial Naive Bayes. The best score for the SMOTE-resampled dataset is also achieved by the Random Forest classifier. From this experiment, we conclude that the imbalanced nature of the dataset has a significant effect on the performance of the classifiers.

TABLE III
Confusion matrix using RF on imbalanced data
(P=0.69, R=0.68, F1=0.65)

		Predicted stereotype					
		CT	*CO*	*IH*	*IT*	*SP*	*ST*
Actual stereotype	*CT*	0	0	1	1	8	0
	CO	0	14	1	4	13	0
	IH	1	1	80	0	19	2
	IT	1	0	0	15	19	0
	SP	0	0	5	6	100	0
	ST	0	1	4	0	13	3

Table III shows the confusion matrix of the random forest classifier on imbalanced data with a proportional split of 467 classes as training data and 312 classes as testing data from the entire dataset. We can see that the classifier failed to classify Controller because there is only a little portion of a Controller in the dataset. Actually for all roles, the classifier seems to have a tendency to classify them as a Service Provider. This is typical phenomenon in machine learning when the dataset is unbalanced.

B. *Single Role (Binary) Classification*

For the single role classification, we run 6 experiments - one for each stereotype. For each stereotype we organize our dataset into two: one for that specific role-stereotype and one other set for all other stereotypes combined ('Others'). In this experiment, we only use the Random Forest algorithm which gave the best result in the multi-roles classification experiment. For each stereotype we calculate the precision, recall, F1 score, and MCC. Additionally, we identify the five features that appear the most important for classifying a specific stereotype. For each performance metric, we compute its average and standard deviation. These results are shown in table IV.

TABLE IV
SINGLE ROLE (BINARY) CLASSIFICATION RESULT

	Precision	**Recall**	**F1-Score**	**MCC**
CT	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
CO	0.98 ± 0.08	0.29 ± 0.12	0.43 ± 0.14	0.50 ± 0.10
IH	0.89 ± 0.07	0.75 ± 0.10	0.81 ± 0.08	0.75 ± 0.11
IT	0.50 ± 0.43	0.14 ± 0.13	0.21 ± 0.17	0.23 ± 0.20
SP	0.72 ± 0.08	0.65 ± 0.11	0.68 ± 0.08	0.47 ± 0.12
ST	0.55 ± 0.44	0.19 ± 0.15	0.27 ± 0.21	0.30 ± 0.25

From table IV, we can see that the classifier failed to classify controller role-stereotype, showed by zero scores on all performance metrics. We think this happened because there is only a tiny number of controller compared to the other role-stereotypes, i.e., 20 controllers out of 779 classes. Other role-stereotypes that have a small proportion like coordinator, interfacer, and structurer have high precision but low recall. We think this happened because the classifier is good for classifying 'Other' rather than the role-stereotype itself, simply because the proportion of 'Other' labels is higher compared to the role-stereotype in the dataset. MCC [21], which is known for measuring the performance of imbalanced binary classification [22], supports this evidence by showing a lower score on those three role-stereotypes. This low score means that the classifier is good at predicting 'Other' label, but not so good at predicting the role-stereotypes.

The information holder binary classifier shows the highest score in all performance metrics. The interesting thing, service provider classifier that has the highest proportion in the dataset does not perform better than the information holder classifier. The information holder classified has more misclassified label than the information holder. We think that the features that we use have something to do with it. We will discuss this further in the feature importance section.

VI. DISCUSSION

A. *Feature Importance in Determining Role-Stereotype*

In this section, we study how important each feature is for classifying each of the role-stereotypes. Table V shows the average score of each feature on each role-stereotype binary classification. We omit some features that have only very low scores such as isStaticClass, isAbstract, and classPublicity. The top five features (highest score) for each role-stereotype are marked with '$*$', while features that we are expected to determine a role-stereotype explained in IV-D are highlighted.

TABLE V
FEATURE IMPORTANCE FOR EACH ROLE-STEREOTYPE

Feature	**CT**	**CO**	**IH**	**IT**	**SP**	**ST**
loc	0.098*	0.172*	0.083*	0.118*	0.124*	0.133*
numAttr	0.072	0.069	0.139*	0.078*	0.128*	0.047
numMethod	0.085*	0.066	0.077*	0.062	0.055	0.056
setters	0.030	0.019	0.019	0.048	0.027	0.032
getters	0.035	0.054	0.042	0.050	0.040	0.056
isPersist	0.003	0.013	0.005	0.002	0.019	0.003
isCollection	0.000	0.000	0.003	0.000	0.004	0.100*
numWordName	0.039	0.043	0.036	0.06	0.055	0.052
isOrEr	0.021	0.027	0.035	0.034	0.045	0.025
isController	0.060	0.000	0.000	0.000	0.001	0.000
numIfs	0.087*	0.079*	0.051	0.069	0.072*	0.056
numParameters	0.101*	0.079*	0.101*	0.075*	0.064*	0.067*
numImports	0.090*	0.097*	0.109*	0.132*	0.088*	0.083*
numOutbndInv	0.082	0.076*	0.057	0.088*	0.056	0.061*
isInterface	0.000	0.001	0.030	0.004	0.016	0.006
isInnerClass	0.017	0.027	0.017	0.014	0.024	0.020
isClass	0.001	0.004	0.022	0.002	0.010	0.004
isEnum	0.001	0.001	0.057	0.000	0.020	0.001
numPblcMthd	0.060	0.057	0.056	0.058	0.061	0.081*
numPrvtMthd	0.059	0.034	0.016	0.037	0.024	0.028
numPrtctdMthd	0.017	0.018	0.008	0.028	0.019	0.024

In general, there are some features that always comes up as top five in every role-stereotype experiments, namely the number of lines in the source code, parameters, and import statements. We think that these features represent the complexity and the relationship of a class with the others. Other features have a very low score. The features that represent an indicator of some values, i.e., whether the source code is a class, an interface, or an enum, does not have a significant score in predicting role-stereotypes, except for the collection indicator. We argue that this happened because the type and publicity of a class do not correlate tightly with the role that the class has. So do the number of methods based on its access modifier, i.e., public, private, or default. We also think that the low number of some role-stereotypes has a significant effect on the score of each feature.

For the controller, the top five features are the number of parameters, lines in the source code, import statements, conditional statements, and methods. The number of conditional statements, import statements, and lines in the source code is as our expected assumption. Our other hypothesis, the number of outbound invocations, is not listed in the top five but still has a significant score. We did not expect the number of parameters to be substantial; we think this is because a controller needs to manage some information passing to other class that collaborate with it.

In the coordinator experiments, the top five features that have the highest score are the number of lines in the source code, import statements, parameters, conditional statements, and outbound invocations. The number of import statements and outbound invocations are as our expectation. The number of parameters and conditional statements showed up again in the coordinator experiments. We think this shows that in a way, a coordinator might have similarities with a controller, but a coordinator has fewer methods than a controller.

For the information holder, the top five significant features are the number of attributes, import statements, parameters, lines in the source code, and the number of methods. From those five features, only the number of attributes is as our expectation. We think that the other higher score features, such as the number of imports, parameters, and methods have strong predictive power because they also have a direct relation to the number of attributes in information holder. For instance, if an information holder class has many attributes, it may also use a lot of other classes as their attributes, hence increasing the number of imports. The high number of attributes in an information holder class can also lead to a large number of setter and getter methods, hence also increasing the number of methods and its parameters.

In the interfacer experiments, the top five features are the number of import statements, lines in the source code, outbound invocations, attributes, and parameters. Only the number of import statements is as our assumption. We think that the number of outbound invocations and parameters has a high score because an interfacer may acts as an abstraction to the client so that it will call and sends many parameters to other internal methods. We also think that a class that provides a user interface as an interfacer so that the attributes in the class is the component of the user interface.

For the service provider, the top five features with the highest score are the number of attributes, lines in the source code, import statements, conditional statements, and parameters. Three features out of that top five, namely the number of lines in the source code, import statements, and parameters are as our expected assumption. We did not expect the number of attributes to determine a service provider. We think that this is the reason why the performance of this classifier is quite low compared to information holder.

In the structurer experiments, the top five features are the number of lines in the source code, import statements, public methods, parameters, outbound invocations, and whether the class implementing Java's Collection interfaces. The relation between structurer stereotype and Collection interface is as our expected. For the other features, we think that they are useful for determining the 'Other' stereotype.

VII. Comparison to Existing Classifiers

We would like to compare our approach to the main previous work by Dragan and Moreno. However, before we can compare results we need to map the categories from their taxonomy onto ours. Hence, next, we compare the role-stereotypes from [1] with the class stereotypes presented in [3] (which are adapted from [2]). The class stereotypes taxonomy of [3] is shown in table VI. We explore mapping these taxonomies via 2 approaches: i) empirically: We apply JStereocode to K9. Then we take all classes that are classified in one role by JStereocode and look as which classes these are labbeled in our ground truth (according to the Wirf-Brocks-taxonomy). ii) theoretically: we look whether the definitions of the categories correspond to each other. This is not obvious because Jstereocode characterizes categories based on implementation characteristics and Wirf-Brocks based on conceptual responsibilities. For completeness, we include the descriptions from Dragan & Moreno [2], [3] in Table .

TABLE VI
Class stereotype taxonomy of Dragan & Moreno [2], [3]

Stereotype	Description
Entity	Encapsulated data and behavior. Keeper of the data model and business logic.
Minimal Entity	Trivial Entity that consists entirely of accessor and mutator methods.
Data provider	Entity that consists mostly of accessor methods.
Data class	Degenerate behavior - it has only get and set methods
Pool	Consists mostly of class constants and a few or no methods.
Commander	Entity that consists mostly of mutator methods
Boundary	Communicator that has a large percentage of collaboration methods, a low percentage of controller- and not many factory methods.
Factory	Consists mostly of factory methods
Boundary + Data Provider	Boundary class that provides access to its state.
Boundary + Commander	Boundary class that provides access for modifying its state.
Controller	Controls external objects - the majority of its methods are controllers and factories.
Pure Controller	Consists entirely of controller and factory methods
Large class	Contains a high number of methods that combine multiple roles i.e. it consists of accessors mutators collaborational and factory methods.
Lazy class	Its functionality cannot be easily determined. It consists mostly of incidental- and get- or set methods.
Degenerate	Very trivial class that does very little - it consists mostly of empty- and get- or set methods.
Small Class	A class that only has one or two methods

A. Mapping of Taxonomies

We applied the JStereocode tool[2] provided by [3] to K-9 Mail. Out of the 779 classes of our K-9 Mail dataset, JStereoCode could not label 505 classes. From the 274 remaining classes, JStereoCode labelled all the interfaces (81 interfaces) as Interface stereotype. But Interface is not considered a stereotype in [3]. Moreover because interfaces do not contain any functionality they are conceptually different from Interfacer Objects in the Wirf-Brocks taxonomy, hence these interfaces are left out of our dataset. In the end, this leaves 193 classes labelled according to the taxonomy by Dragan & Moreno. Next we looked up in our ground truth which stereotypes these classes belong to in our taxonomy. Table VIII shows the summary of this. Cells in this table are shaded if we map them onto each other based on theoretical grounds.

Table VI shows that Moreno distinguishes more categories of stereotypes for data and entity classes: entity, minimal entity, data provider, data class, and pool. Table VII shows that the empirical- (highest numbers) and theoretical (shaded cells) approaches agree on that these map onto Information Holder.

[2]http://www.cs.wayne.edu/ severe/jstereocode/

TABLE VII
EMPIRICAL COMPARISON BETWEEN JSTEREOCODE [3] AND WIRFS-BROCK [1] (BASED ON OUR GROUND TRUTH FOR K9)

Stereotype	**CT**	**CO**	**IH**	**IT**	**SP**	**ST**
Entity	0	0	1	0	1	0
Minimal Entity	0	0	3	1	0	1
Data Provider	1	0	4	0	1	1
Data Class	0	0	10	0	0	0
Pool	0	0	9	0	2	0
Commander	0	0	1	0	6	2
Boundary	1	20	0	0	39	2
Boundary + Data Provider	0	1	0	0	3	2
Boundary + Commander	2	0	0	0	5	1
Factory	0	0	2	0	10	2
Controller	0	0	0	0	0	0
Pure Controller	0	0	0	0	0	0
Large Class	0	0	0	0	0	0
Lazy class	0	2	0	0	5	1
Degenerate	0	0	43	0	7	1

TABLE VIII
MAPPING BETWEEN DRAGAN & MORENO STEREOTYPES [2], [3] AND WIRFS-BROCK STEREOTYPES [1]

Class Stereotype	**in [2]**	**in [3]**	**WB-Role-stereotype**
Entity	✓	✓	Information Holder
Minimal Entity	✓	✓	Information Holder
Data provider	✓	✓	Information Holder
Data class	✓	✓	Information Holder
Pool		✓	Information Holder
Commander	✓	✓	Service Provider
Boundary	✓	✓	Service Provider
Boundary + Data Provider		✓	Service Provider
Boundary + Commander		✓	Service Provider
Factory	✓	✓	Service Provider
Controller	✓	✓	Controller
Pure Controller	✓	✓	Coordinator
Large class	✓	✓	Not suitable to any role
Lazy class	✓	✓	Not suitable to any role
Degenerate class	✓	✓	Not suitable to any role
Small class	✓		Not suitable to any role

Next, we establish correspondence between Commander, Boundary (and its refinements +Data and +Commander) to Service Provider. This is suggested by the empirical numbers, and there is reasonable theoretical match to justify this mapping: both aim to do offer some service. However, this may not be a perfect correspondence: 20 out of 59 (39%) of the Boundary classed are mapped onto Coordinators.

Given that JStereocode does not find any Controllers or Pure Controllers, we map these onto Controller and Coordinator based on theoretical grounds. For Controller the definitions seem to match. For the Pure Controller stereotype, Dragan [2] explained that this stereotype is a candidate for the bad-smell God class by combining too much functionality. This characteristic of lumping functionality is mostly present in the Coordinator.

There are no theoretical grounds to map the categories Large Class, Lazy Class and Degenerate to any of our stereotypes. Their naming and description are not related to any type of responsibility. We will leave these classes out of later statistical comparisons.

We looked into the 43 classes that are labelled degenerate by JStereocode. When we reviewed these classes, they only hold information without providing many methods. That is the reason these classes were classified as information holder in our taxonomy.

Now that we have establish a correspondence of the stereotypes in both taxonomies, we use this in the next section to perform more statistical comparisions.

B. Role Stereotype Classification Performance

In this section, we attempt to compare the classification performance between CRI and StereoClass by Dragan et.al. [2]. To the best of our knowledge, there is no evaluation on the classification performance of JStereoCode [3].

Dragan et.al. conducted an assessment study on their tool StereoClass. In particular, they recruited 3 subjects (doctoral students) to label in total 45 classes (15%) of the HippoDraw source code using their role stereotypes. Table IV in paper [2] reports agreement between labels generated by StereoClass and labels given by the subjects. Even though the authors do not report any standard performance metric (such as precision or recall), precision can still be calculated from the data in this table.

We compare this to CRI applied to K9. CRI produces a classification for all 779 classes, and hence is more robust and complete. Table IX shows precision rates of StereoClass and CRI grouped by CRI's stereotypes. The first and second columns of the table show the mapping between Dragan's stereotypes and the stereotypes we use in this study. The third column "Num.Classes" shows number of classes labelled by StereoClass. Data in this column is taken from column "Tool" in the original table (Table IV in [2]).

TABLE IX
PRECISION RATES OF STEREOCLASS (ON HIPPODRAW) AND CRI (ON K9)

Dragan's stereotypes	**CRI's stereotypes**	**Num. Classes**	**StereoClass's precision(%)**	**CRI's precision(%)**
Entity	Information Holder	13	57.69	89
Minimal Entity		3		
Data Provider		8		
Data Class		2		
Commander	Service Provider	7	87.65	72
Boundary		15		
Factory		5		
Controller	Controller	6	77.78	0
Pure Controller	Coordinator	2	33.33	98

The fourth column "StereoClass's precision(%)", which is also derived from Table IV in [2], shows *average precision rates* (in percentage) across three ground truths obtained from the three subjects in the assessment study. The following example demonstrates how we calculate the precision rate for each subject. The precision rate of StereoClass in classifying IH classes with ground truth from Subject 1 (S1) is number of classes that are labelled as IH-related by both S1 and StereoClass (i.e. $8 + 1 + 7 + 1 = 17$ classes) divided by number of classes labelled as IH-related by StereoClass (i.e. $13+3+8+2=26$ classes).

The fifth column shows the precision rates (in percentage) of CRI in classifying single role stereotype. Data in this column is referenced to data in column "Precision" in table IV of this paper. It is worth noting that the precision rates of StereoClass (in fourth column) and CRI (in fifth column) are derived from different evaluation settings, using different ground truths and on different cases (HippoDraw and K9). Therefore, it is impossible to draw an absolute comparison between the performance of the tools. The following observations are made in order to get an impression which tool is good in classifying what stereotype.

It can be seen from the table that StereoClass performs quite well ($[75\%, 85\%)$) and very well ($[85\%, 100\%]$) in classifying stereotypes that correspond to Controller and Service Provider. CRI can perform well ($[50\%, 75\%)$) in classifying Service Provider classes and very well in classifying Information Holder and Coordinator classes. It is also observable that StereoClass performs better than CRI in classifying Controller classes, at the precision rate of 77.78% comparing to CRI's 0%. Both of these numbers may be exaggerated by the small number of controllers present in both cases.

VIII. Patterns in the Anatomy of Software Design

In this section we illustrate how the classification of stereotypes can be used to find patterns in the anatomy of software designs. First we explore the frequencies of occurrence of individual stereotypes, and then the occurrence of graph-patterns, i.e. typical collaborations between role-stereotypes. Next, we explore whether different stereotypes have different characteristics by looking at them from a design-metrics perspective.

A. *Collaboration Pattern between Stereotypes*

Fig. 3 shows the frequencies of occurrence of the role stereotypes and also the relationship between them (based on our manual classification of K9). The numbers below the role-stereotype names are the frequencies of the role-stereotype in K9. The labels on the edges indicate the frequencies of occurrence of relations between these stereotypes. We can see that some stereotypes occur often (such as Information Holder, Service Provider), and others only rarely (such as Controller). Indeed, in [2], by applying StereoClass on 5 open source systems, Dragan also finds that the stereotypes that they considers differ in frequency of occurrence. Surprisingly, there is quite similar amount of stereotypes related to Information Holder and Controller (26.6% and 1.9%). comparing to that amounts of Information Holders and Controllers in K-9Mail (29.6% and 2.5% respectively). The fact that this distribution of occurrence of stereotypes is very unbalanced has a negative effect on the performance of the machine learning algorithms.

To find patterns, we created a visualisation of the K9 design where we used different colours to highlight different role-stereotypes. The visualisation and its source (in *plantuml* format) can be found in the replication package of this study [17]. For now, these findings are based on the researchers' visual assessment of patterns in a visualisation of K9.

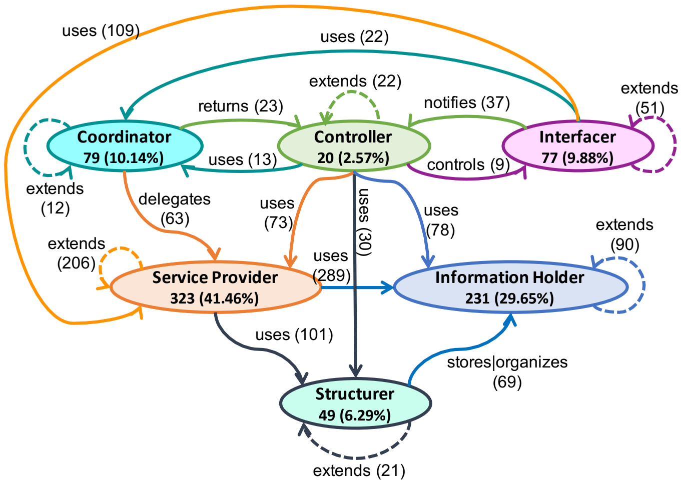


Fig. 3. Relationship between role stereotypes in K-9 Mail case study

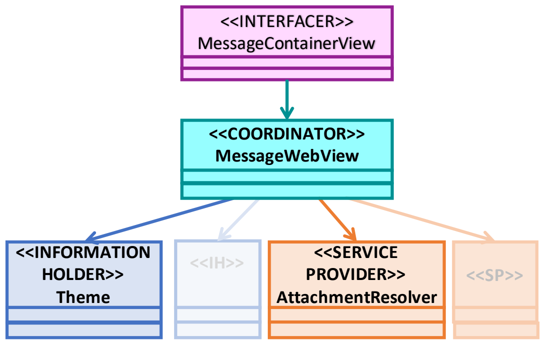

<<INTERFACER>> MessageContainerView
<<COORDINATOR>> MessageWebView
<<INFORMATION HOLDER>> Theme
<<IH>>
<<SERVICE PROVIDER>> AttachmentResolver
<<SP>>

Fig. 4. Typical Fragment of Collaboration between Stereotypes

Fig 4 shows a typical pattern of collaboration between role stereotypes: on the top, there is an interfacer which receives UI-events and reacts to the events by sending messages to an associated coordinator. This coordinator contains logic to break down the task and to pass requests to one or more associated service providers and one or more associated information holders. The coordinator is also responsible for gathering the results from the service provider(s) and information holder(s) and finally returning the results to the interfacer. Via our visualisation, we have found multiple occurrences of this pattern in K9. This pattern empirically confirms the architectural control style for user events described by Wirfs-Brock (p. 207 [12]).

B. *Design Characteristics of Role Sterotypes*

One application we foresee is to use role stereotypes for more contextualized quality assurance [23]. More specifically, on theoretical grounds we expect that information holders contain little complicated logic, hence generally should have a low WMC (Weighted Method per Class). In contrast controllers contain complicated decision logic and hence generally should have higher WMC. We also expect that controller, interfacer, and coordinator to have a higher CBO (Coupling between Object Classes), because these stereotypes will communicate with many other classes. Currently, tools aimed at detecting design smells/antipatterns typically work by computing design metrics for all classes and then either check these against fixed thresholds or look for outliers. Figure 5 confirms the aforementioned theoretical predictions about these design metrics for these stereotypes. This implies that it makes little sense to detect design smells by applying the same thresholds on design metrics uniformly for all classes in a system. Instead, Figure 5 shows that design metrics for different role stereotypes have very different ranges.

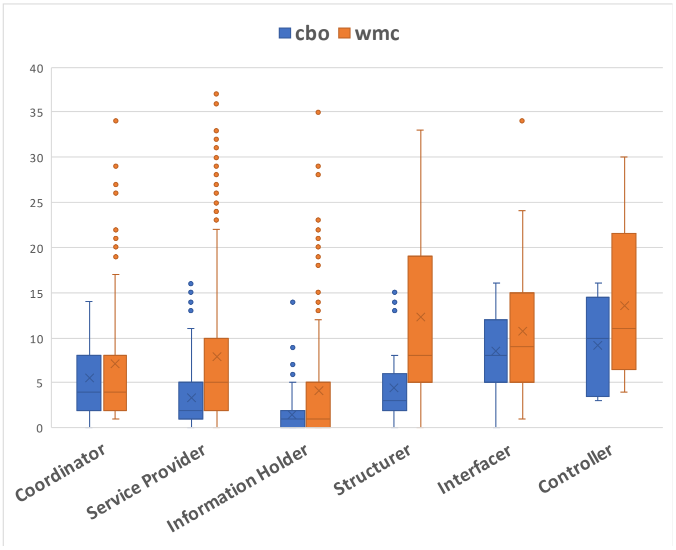


Fig. 5. Distribution of CBO and WMC for all stereotypes

IX. Threats to Validity

Threats to Internal Validity. An OOP class may be responsible for multiple roles and responsibilities [12]. In this study, we however have chosen to build a machine learner that captures only one role for each class. This choice might cause an incomplete view of roles and responsibilities of a single class. However, given that only a low number of classes carry multiple roles (13 out of 779 classes, $\approx 1.7\%$), we consider this threat is acceptable and is a trade-off to be made to keep our classification model rather simple.

The choice of studying an Android application (i.e. K-9 Mail) might also be a source of threats to the internal validity of this study. In particular, Android applications are built upon Android frameworks which encapsulate low-level functionalities of the Android OS. Thus, a number of Controller, Structurer and Interfacer classes at UI and activity management level and collaboration between them might be hidden away. Roles/responsibilities of those classes that extend/implement these base functionalities might possibly be overlooked. As to mitigate the threat, when labeling a sub-class, we pay extra attention by reviewing roles and collaborators of its ancestor (Android) classes (mainly from Android's API reference [3]). In the future, it is interesting to study pure OOP systems in comparison with K-9 Mail in this study.

Threats to External Validity. In this study, we try to identify class role stereotypes of K-9 Mail classes written in Java. Firstly, the training was done on a single case project. It should be studied whether using more cases leads to more robust classifier. However, we believe result of this study can be generalized to other OOP system in various programming languages because of the following reasons. Firstly, the notion of class role stereotype applies to OOP in general, regardless implementation programming languages. Secondly, scripts [17] used in this study can be used to extract source code features from different languages than Java. The XPath queries that was used to extract features from parsed srcML files can be easily adapted to other languages such as C#, C/C++ by following srcML language and grammar rules[4].

We however would not generalize the result of this study to programming languages other than OOP.

X. Conclusion and Future Work

The overall context of this research is to develop automated methods that can aid in the maintenance software, especially though increased understanding of the design of software. We presented a machine learning based approach for the automatic classification of role-stereotypes of classes of Java software. We find that the Random Forest algorithm enhanced by SMOTE resampling to address imbalanced data yields the best performance. It turns out however, that the classifier is good at detecting some role stereotypes (coordinator, information holder), medium good at detecting others (service provider, structurer, interfaces) and poor at detecting a third category (controllers). Partially this can be explained by the low frequency of occurrence of these roles in the design (offering few training examples). It may also be that other features we have not yet considered may improve classification performance. One direction in which we aim to look is to use OO-design metrics as additional features.

In this study, as an attempt to compare our classifier CRI with previous work by Moreno and Dragan, we establish a mapping between our stereotype taxonomies and finally come up with an approximate precision comparison. We observe that CRI performs better in classifying Information Holder and Coordinator classes while Dragan's tool StereoClass performs best in identifying Service Provider and Controller classes.

In this study we classified all 779 classes of the open source K-9 Mail application. This number is three times more than the number of classes that can be labelled by JStereoCode. This labeled dataset enabled us to perform a novel analysis that sheds light on the anatomy of software designs: we analyzed how frequently particular stereotypes collaborate with other stereotypes. We believe there are yet undiscovered regularities in the anatomy of designs that can be uncovered through studying these role-types for larger sets of projects.

We illustrated that there are various uses of stereotype classification in software quality assurance.

[3]https://developer.android.com/reference/

[4]srcML grammar rules: https://www.srcml.org/documentation.html

References

[1] R. Wirfs-Brock, "Characterizing classes," *IEEE Software*, vol. 23, no. 2, pp. 9–11, 2006.

[2] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pp. 1–10, 2010.

[3] L. Moreno and A. Marcus, "Jstereocode: automatically identifying method and class stereotypes in java code," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pp. 358–361, 2012.

[4] M. Staron, L. Kuzniarz, and C. Wohlin, "Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments," *Journal of Systems and Software*, vol. 79, no. 5, pp. 727–742, 2006.

[5] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: A series of four experiments," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 96–118, Jan. 2010.

[6] M. Genero, J. A. Cruz-Lemus, D. Caivano, S. Abrahão, E. Insfran, and J. A. Carsí, "Does the use of stereotypes improve the comprehension of UML sequence diagrams?," in *2nd ACM-IEEE Int. Symposium on Empirical Software Engineering and Measurement*, ESEM '08, (New York, NY, USA), pp. 300–302, ACM, 2008.

[7] O. Andriyevska, N. Dragan, B. Simoes, and J. I. Maletic, "Evaluating UML class diagram layout based on architectural importance," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 1–6, Sept 2005.

[8] S. Yusuf, H. Kagdi, and J. I. Maletic, "Assessing the comprehension of UML class diagrams via eye tracking," in *15th Int. Conf.on Program Comprehension. ICPC'07.*, pp. 113–122, IEEE, 2007.

[9] B. Sharif and J. I. Maletic, "The effect of layout on the comprehension of UML class diagrams: A controlled experiment," in *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pp. 11–18, IEEE, 2009.

[10] N. Alhindawi, N. Dragan, M. L. Collard, and J. I. Maletic, "Improving feature location by enhancing source code with stereotypes," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 300–309, Ieee, 2013.

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

[12] R. Wirfs-Brock and A. McKean, *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003.

[13] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pp. 24–34, 2006.

[14] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *IEEE 21st Int. Conference on Program Comprehension, ICPC San Francisco*, pp. 23–32, 2013.

[15] A. Budi, Lucia, D. Lo, L. Jiang, and S. Wang, "Automated detection of likely design flaws in n-tier architectures," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*, pp. 613–618, 2011.

[16] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[17] "Replication package of this study." http://oss.models-db.com/Downloads/ICSE2019_ReplicationPackage/. Accessed: Aug.22, 2018.

[18] M. L. Collard, "Addressing source code using srcml," in *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC05)*, 2005.

[19] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *CoRR*, vol. abs/1106.1813, 2011.

[20] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[21] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[22] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using matthews correlation coefficient metric," *PloS one*, vol. 12, no. 6, p. e0177678, 2017.

[23] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.