Role Stereotypes in Software Designs and Their Evolution

Truong Ho-Quang[a], Arif Nurwidyantoro[b], Satrio Adi Rukmono[c,d], Michel R. V. Chaudron[d,a], Fabian Fröding[a], Duy Nguyen Ngoc[a]

[a] *Chalmers |Gothenburg University, Gothenburg, Sweden*
[b] *Monash University, Melbourne, Australia*
[c] *Institut Teknologi Bandung, Bandung, Indonesia*
[d] *Eindhoven University of Technology, Eindhoven, The Netherlands*

Abstract

Role stereotypes are abstract characterisations of the responsibilities of the building blocks of software applications. The role a class plays within a software system reflects its design intention. Wirfs-Brock introduced the following six role stereotypes: Information Holder, which knows information, Structurer, which maintains object relationships, Service Provider, which offers computing services, Coordinator, which delegates tasks to others, Controller, which directs other's actions, and Interfacer, which transforms information. Knowledge about class role stereotypes can help various software development and maintenance tasks, such as program understanding, program summarisation, and quality assurance. This paper presents an automated machine learning-based approach for classifying the role-stereotype of classes in Java projects. We analyse this approach's performance against a manually labelled ground truth for three open source projects that contain 1,500+ Java classes altogether. The contributions of this paper include: i) a machine learning (ML) approach to address the problem of automatically inferring role-stereotypes of classes in Object-Oriented Programming Languages, ii) the manually labelled ground truth, iii) an evaluation of the performance of the classifier, iv) an evaluation of the generalisability of the approach, and v) an illustration of new uses of role-stereotypes. The evaluation shows that the Random Forest algorithm yields the best classification performance. We find, however, that the performance of the ML-classifier varies a lot for different role stereotypes. In particular, its performance degrades when classifying rarer stereotypes. Among the 23 features that we study, features related to the classes' collaboration characteristics and complexity stand out as the best discriminants of role stereotypes.

Keywords: Class Role-Stereotypes, Machine Learning Classification, Program Analysis, Design Metrics, Software Engineering

[1] *Email addresses*: truongh@chalmers.se (Truong Ho-Quang), arif.nurwidyantoro@monash.edu (Arif Nurwidyantoro), sar@itb.ac.id (Satrio Adi Rukmono), m.r.v.chaudron@tue.nl (Michel R. V. Chaudron)

1. Introduction

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

Dragan et al. have proposed methods for automatically inferring the role-stereotype of classes in C++ [2]. Moreno et al. [3] migrated this approach to Java. Both approaches are based on a collection of expert-designed decision rules applied to classes' syntactic characteristics in the source code. This inference of role stereotypes can be seen as an enrichment of reverse engineering, particularly in uncovering design: Role stereotypes indicate generic types of responsibility that characterize the type of functionality that class should perform and the type of interactions a class can have with other types of classes. Hence role stereotypes are essential clues for the design-intention of classes.

Several studies [4, 5, 6, 7, 8] have demonstrated the benefits of using stereotypes in various software development and maintenance activities. These benefits include program design, program comprehension, quality assurance, and program summarisation. We mention the following as concrete examples of the usefulness of role-stereotypes as demonstrated in earlier work: *Using role-stereotypes in creating layouts of UML class diagrams improves the comprehensibility of the diagrams* [6, 7, 8]. In addition, Alhindawi et al. [9] show that enhancing the source code with stereotype information helps improve feature location in the source code.

We present a machine learning (ML) approach to infer the stereotype of Java classes automatically. Dragan [2] and Moreno [3] observe that the robustness of their rule-based approach leaves room for improvement: the rules of their approach are not 'complete': they do not classify a reasonably large portion of classes of a system. This paper presents an ML-based classifier that classifies all classes and thus is more robust than existing rule-based classifiers.

The remainder of this paper first explain the key concept of role stereotype (Section 2) and discuss related work (Section 3). Next, we explain our research methodology in Section 4. As part of this, we explain the taxonomy of role stereotypes used in our study and how they relate to other taxonomies (Section 4.2). We then discuss the following contributions:

1. We publish the first sizeable validated dataset of 1,547 Java classes and their role stereotypes. This dataset can serve as a (ground truth) resource for other researchers (Section 4.3).

2. We evaluate the performance of our approach (Section 5). We infer which features are most important for classifying stereotypes and which machine learning algorithm works best (Section 6).

3. We evaluate the generalisability of our approach (Section 7). The earlier work by Dragan

[2] used only 45 classes for validating their approach for C++. The paper by Moreno [3] does not include validation of its performance.

4. We use the perspective of role stereotypes to gain novel insights into software structure and its evolution (Sections 8 and 9).

We then follow up with the threats to validity in Section 10. Finally, Section 11 concludes our paper and provides potential future work.

2. Class Role Stereotypes

Wirfs-Brock [10] proposed an object-oriented design approach based on the notion that each software object should have a well-defined responsibility to play one of a few generic roles in a system's design. Wirfs-Brock classified the roles of software objects into six stereotypes:

(IH) *Information Holder*: objects designed to know certain information and provide that information to others.

(ST) *Structurer*: objects that maintain relationships between objects and information about those relationships.

(SP) *Service Provider*: objects that perform work and offer services to others on demand. *Structurers* might pool, collect, and maintain groups of objects.

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

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

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

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

It is crucial to recognize that Wirfs-Brock suggests using role stereotypes while *designing* a system. In our study, we aim to establish role stereotypes based on the *implementation* of a system. In general, the classes that end up in implementation are not ideal. For example, they may mix two or more responsibilities.

3. Related Work

Dragan et al.[11] proposed an automated tool to detect stereotypes of methods in the C++ programming language. They define a taxonomy of methods, including *Structural* (*Accessor* and *Mutator*), *Collaborative*, and *Creational*. They propose several rules to determine

method stereotypes based on the type of the method, the return type, and how the method modifies the class's state. On top of their method for classifying method stereotypes, 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*. While several of these seem close to Wirf-Brocks's, the Dragan classification is presented as being derived empirically from studying 21 open source projects.

The rules in Dragan's approach consist of a collection of conditions on the quantities of *method stereotypes*; e.g. $\#Mutators > 2 \times \#Accessors$. These conditions include thresholds based on a mix of theoretical arguments and statistical techniques [12]. Dragan states that "The rules for stereotype identification are subjective and thresholds might vary depending on differences in subject's interpretations." [2] Moreover, these rules leave a large number of classes in software systems unclassified. They do not cover the spectrum of possible combinations of method stereotypes that occurs in practice.

Moreno and Marcus propose a method for identifying class stereotypes in the Java programming language based on the work of Dragan [3]. They adapted their procedures [2] and provided additional method and class stereotypes. Both classifier approaches by Dragan and Moreno's rules are not disjoint: different rules each may assign different stereotypes to a single class. Unfortunately, Moreno's subsequent research has used this classifier chiefly for automatically generating summaries of classes in natural language [13] and seems not to have continued improving its performance.

Budi et al. [14] built an automated tool to detect design flaws based on design stereotypes. They use a taxonomy of 4 role-stereotypes, different from Wirfs-Brock: *Boundary*, *Control*, *Regular Entity*, and *Data Manager*. For these design stereotypes, some rules describe how these stereotypes should be allocated to the typical layers of 3-tiered software architectures and how they are supposed to collaborate. The authors used SVM to automatically labels classes into categories. They then used rules about the relationship between the stereotypes to detect potential design flaws in the system.

The Gang-of-Four (GoF) design patterns [15] also represent idealized software design patterns. However, there are essential differences between the GoF patterns and the design stereotypes that we consider: Firstly, only a tiny portion of classes in a system are part of GoF design patterns. Whereas in the design philosophy of Wirfs-Brock, each class in a system should play at least one of her proposed stereotype roles. Secondly, GoF design patterns are defined as specific ways in which individual types of classes collaborate. In contrast, Wirfs-Brock's stereotypes are the property of individual classes. Finally, we mention one approach by Fontana et al. [16] that uses machine learning to identify design patterns in source code.

4. Methodology

Figure 1 shows an overview of our research methodology. First, we select three case studies and collect their source code (Step 1). Second, we define and establish a ground truth (Steps 2 & 3). Then, we extract features from the source code that are to be used by

the machine learning algorithms (Step 4). After that, we experiment with various machine learning algorithms (Step 5). Finally, we evaluate the performance of the machine learning algorithms (Step 6) and then study the evolution of software structure concerning the role stereotypes (Step 7). In the following subsections, we elaborate on these steps in more detail.

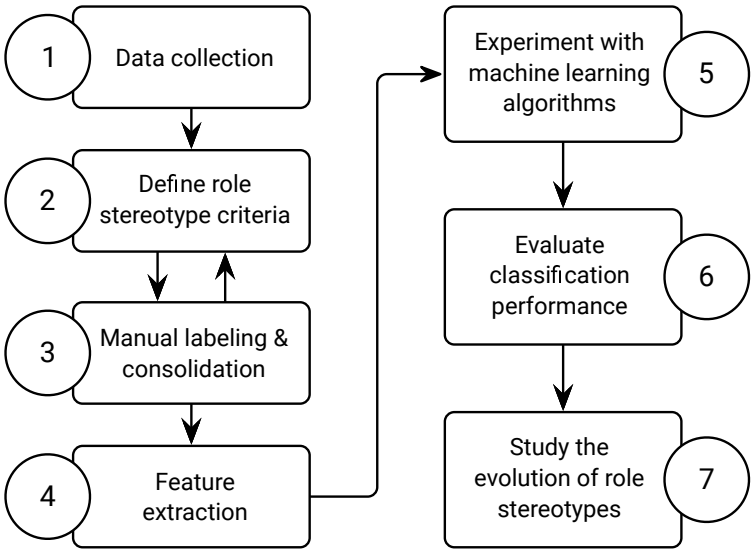


Figure 1: Research methodology

4.1. Data Collection

This research uses three open-source software systems as our case studies: K-9 Mail[2], Bitcoin Wallet[3], and SweetHome3D[4]. Table 1 shows descriptive information about the projects. K-9 Mail and Bitcoin Wallet are Android applications hosted in GitHub, while SweetHome3D is a pure-Java application hosted in SourceForge.

Table 1: Description of OSS projects used in this study
#Class is calculated at the studied version. #Release, #Contributor and #Star are retrieved at the time of writing this paper

#	OSS project	Version	Released	#Class	#Release	#Contributors	#Stars	Type
1	K-9 Mail	v5.304	Nov. 10, 2017	779	367	202	4276	mobile/Android
2	Bitcoin Wallet	v6.31	Oct. 1, 2018	222	274	26	1705	mobile/Android
3	SweetHome3D	v5.6	Oct. 25, 2017	546	46	n/a	4.7/5.0	desktop

These projects are chosen for this study because of the following reasons:

[2] https://github.com/k9mail/k-9/tree/1a12b18f0c4a452b74941340179735f0383bd1fb

[3] https://github.com/bitcoin-wallet/bitcoin-wallet/releases/tag/v6.31

[4] https://sourceforge.net/projects/sweethome3d/files/SweetHome3D-source/SweetHome3D-5.6-src

- They use Java as the primary programming language,
- They are active open source projects: they exist for many years, and usable released versions are available. These projects are not student projects.
- They vary in size (#Class), domain, and technology (Android & pure-Java), and the sizes are beyond 'toy'-projects.
- They are from domains that non-experts can understand.

We downloaded the source code of these projects from the corresponding GitHub and SourceForge links. We found a small number of "nested" classes that might interfere with their outer classes' feature extraction. Therefore, the next step was to extract these nested classes into independent classes. The extraction process was performed as follows. Firstly, the source code was parsed using srcML [17]. For a given source code, srcML creates a list of classes, including nested classes and their details, in a standardized XML representation. Then, we used XPath queries to generate all classes from the saved XML file. As a result, every nested class was extracted into a separate Java file. Finally, we used *checkstyle* [5] to remove unused import statements in every class to obtain the actual number of import statements used and the number of lines in the class. The scripts for automating this extraction process are included in the replication package of this paper [18]. After these steps, we obtained a total of 1,547 Java classes over three cases.

4.2. Ground Truth, part 1: Criteria for Role Stereotypes

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 authors obtained the initial criteria by studying the descriptions by Wirfs-Brock [10]. Then the criteria were refined and calibrated in follow-up meetings where the authors had assessed additional sets of classes (details in section 4.3). The criteria can be divided into three categories: i) Criteria regarding characteristics of classes, ii) Criteria regarding the relationship between role stereotypes, and iii) Other criteria. We discuss each of these next.

4.2.1. Criteria regarding characteristics of classes

These criteria focus on the intrinsic (static) properties of classes. We take the *Structurer* stereotype as the first example: Table 2 shows the criteria used to characterize *Structurer* classes. In this particular case, we look into data types of attributes, library use, and content of methods inside a class to get an impression of whether the class is capable of organizing/manipulating a collection of objects. As a second example, the *Information Holder* may include persistence mechanisms (files or databases). Other class properties such as class name and getter/setter methods 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 [18].

[5] `https://github.com/checkstyle/checkstyle`

Table 2: Properties of a *Structurer* class

What makes a class a *Structurer*?
- May contain "user defined object" type as attributes
- May extend 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.

Some of the criteria are somewhat similar to the rules developed by Dragan and Moreno. For example, we both consider getter and setter methods to indicate the *Information Holder* stereotype (in our classification) and *Entity/Data Provider*-type (in Dragan's classification). However, different from Dragan and Moreno, our criteria also consider other features, such as persistence functionality.

4.2.2. Criteria regarding the relationship between roles

Wirfs-Brock mentions that "the roles an object plays imply certain kinds of collaborations." [10] 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. The graph demonstrates common relationships between different role stereotypes. It shows, for example, that *Information Holders* are commonly used by *Controllers*, *Service Providers*, or *Structurers* but not by *Coordinators* or *Interfacers*. Hence, the presence or absence of relations can be used as a criterion in deciding a class's possible roles.

The following are additional examples of criteria on the relationships between roles: A *Structurer* may link to several *Information Holders*; a *Service Provider* can store information by collaborating with *Information Holder* and *Structurer* classes. As an intermediary between different layers of a system, an *Interfacer* might collaborate with *Coordinators* and *Service Providers* in each layer to conduct a cross-layer task; *Controller* classes often control this kind of task.

4.2.3. Other Considerations

We defined additional criteria aimed at essential differences in behaviour between different role stereotypes. For example, both *Service Provider* and *Controller* classes may include some control-flow logic. The design intention of these classes suggests that a *Controller*'s decision should affect a broader control flow of the system. In contrast, decisions made by a *Service Provider* should primarily affect the flow within the class itself.

Secondly, we draw special attention to Android case projects. In particular, Android applications are built upon Android frameworks which encapsulate low-level functionalities of the Android OS. Thus, some *Controller*, *Structurer*, and *Interfacer* classes at the UI and

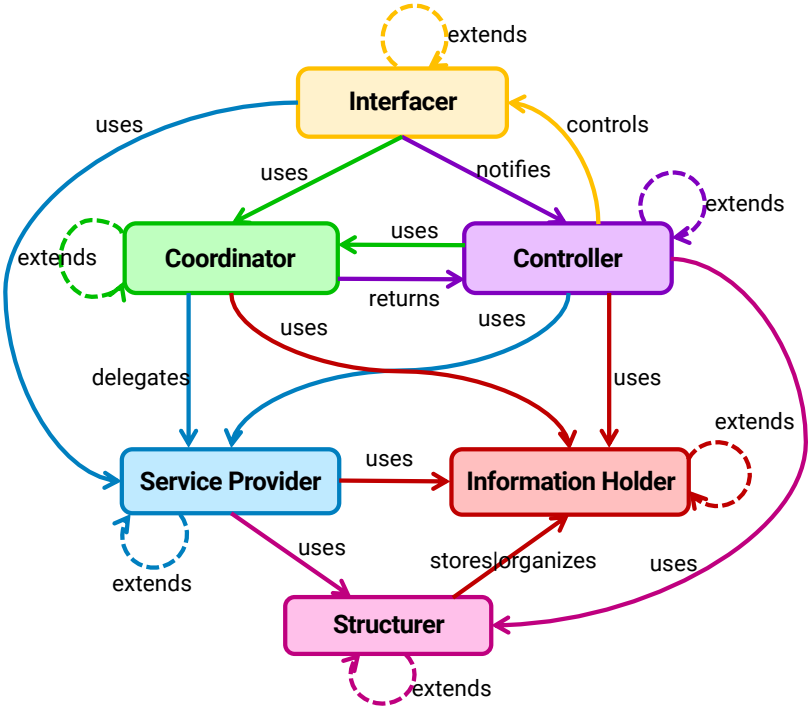


Figure 2: Relationship between role stereotypes

activity management level and collaboration between them might be hidden away. The roles and responsibilities of those classes that extend or implement these base functionalities might be overlooked. The experts pay extra attention by reviewing roles and collaborators of its ancestor (Android) classes, mainly from Android's API reference[6].

Lastly, sometimes a class may carry more than one role. This possibility of multiple roles is also discussed by both Wirfs-Brock [10] (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, this is also recorded.

4.3. Ground Truth, part 2: Manual Labelling and Consolidation

We used an iterative process to establish agreement on criteria and how to apply them. Initially, we randomly chose 20 classes for each project. Then, two of the authors and one non-author PhD student manually labelled these classes independently. They scrutinised each other's work, discussed any differences in classification, and refined the criteria based on this discussion. These steps were repeated two more times on K-9 Mail, as this is the case with the most significant number of classes, until the criteria seemed sufficient or saturated. Next, two of these people labelled all the remaining classes. A final discussion round took place between the two graders to resolve disagreements and hard-to-stereotype cases. Afterwards, two master students, both are authors, reviewed the resulting data. These

[6] `https://developer.android.com/reference/`

authors did not participate in earlier manual labelling tasks. The whole manual labelling process took about 100 hours in total—approximately 45 hours by each primary grader and 10 hours by the extra grader—spread out over three months. Ultimately, we established a ground truth of 1,547 labelled classes, covering all three cases [18]. Table 3 summarises the distribution of all classes in the ground truth by projects and by role-prototype.

Comparing to the approach proposed by Dragan et al. [2] and Budi et al. [14], some of their features are similar to our features, i.e., *numMethod*, *setters*, *getters*, and *numOutboundInv*. However, they only considered a smaller number of features, and most of our features were not considered in their work.

Table 3: The distribution of role-stereotypes in each project under study

Project	**CO**	**CT**	**IH**	**IT**	**SP**	**ST**	**Total**
K-9 Mail	79	20	231	77	323	49	779
Bitcoin Wallet	2	5	83	62	57	13	222
SweetHome3D	21	38	227	63	159	38	546
Total	102	63	541	202	539	100	1547

4.4. Feature Extraction

Our classification is based on a static analysis of the Java source code of a system. First, we extract the features using the srcML tool [17]. For a given source code, srcML creates a list of classes and their details in a standardised XML representation. Then, the values of features are calculated by using XPath queries. We chose to use 23 source code features that correspond to the criteria used in the manual classification (see Section 4.2). In Tables 4 to 8, we list the features (**F**) and a short subjective opinion on their relevance to classifying different role stereotypes. The features are categorised into five categories (**C**), i.e., *Accessibility*, *Complexity*, *Functionality*, *Naming Convention*, and *Collaboration*.

Table 4: **C1: Accessibility features**, representing how accessible a class and its content is.

№	Feature Name	Data Type	Explanation
F01	*classPublicity*	string ('default', 'private', 'protected', or 'public')	The access modifier of the class. We assume that *Service Provider* and *Information Holder* classes might offer public access so that other classes can collaborate with them.
F02	*numPublicMethods*	integer	The number of public methods inside the class. We assume that an *Information Holder*, an *Interfacer*, a *Service Provider*, or an *Interfacer* class might have many public methods.
F03	*numPrivateMethods*	integer	The number of private methods inside a class. We assume that a *Controller*, *Coordinator*, and *Service Provider* might distribute the job on separate methods inside the class.
F04	*numProtectedMethods*	integer	The number of protected methods inside a class. We assume that a *Controller*, *Coordinator*, and *Service Provider* might distribute the job to separate methods inside the class. These methods might still be used or overridden by any sub-classes.

Table 5: **C2: Complexity features**, representing the complexity of a class.

№	Feature Name	Data Type	Explanation
F05	*loc*	integer	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.
F06	*numIfs*	integer	The number of conditional statements in the class body, i.e., the `if`/`if-else` and `switch` statements. *Controller* classes might use lots of conditional statements to make decisions and to control workflows.
F07	*numParameters*	integer	The total number of parameters in all methods in the class. We assume that methods in a *Service Provider* or a *Coordinator* class might have many parameters.
F08	*numAttr*	integer	The number of attributes declared in the class. We assume that an *Information Holder* class might have many attributes.
F09	*numMethod*	integer	The number of methods declared in the class (constructors are excluded). We assume that *Service Provider* and *Coordinator* classes have many methods.
F10	*setters*	integer	The number of methods which names start with `set-`. We assume that this method is a setter method, i.e., the method that modifies a variable in an *Information Holder*.
F11	*getters*	integer	The number of methods which names start with `get-`. We assume that this method is a getter method, i.e., the method that accesses variable values in an *Information Holder* class.

Table 6: **C3: Functionality features**, aiming at detecting specific functions that a class may have.

№	Feature Name	Data Type	Explanation
F12	*isPersist*	boolean	Indicates whether a class has persistence features, i.e., implements a `Serializable` interface or importing database connectivity libraries. We assume that *Information Holder* classes are more likely to employ persistence features.
F13	*isCollection*	boolean	Indicates whether a class is a subclass of Java's collection library. We assume that a *Structurer* might need it to maintain relations between objects.
F14	*isClass*	boolean	Indicates whether the source code file is a class. We assume that a class can represent all of the role stereotypes.
F15	*isEnum*	boolean	Indicates whether the source code is a Java enum. We assume that the enum type can represent an *Information Holder*.
F16	*isAbstractClass*	boolean	Indicates whether a class is an abstract class.
F17	*isStaticClass*	boolean	Indicates whether a class is a static class. We assume that a static class can represent a *Service Provider* or an *Information Holder*.
F18	*isInterface*	boolean	Indicates whether the source code file is a Java interface. We assume that an Interface can provide methods that a *Service Provider* or a *Structurer* must implement.

Table 7: **C4: Naming Convention features**, for detecting specific naming conventions in the class name.

№	Feature Name	Data Type	Explanation
F19	*numWordName*	integer	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 might have a longer name.
F20	*isOrEr*	boolean	Indicates whether the class name ends with `-or` or `-er`. We believe a *Controller* or a *Service Provider* class is more likely to have a name that ends with `-or` or `-er`.
F21	*isController*	boolean	Indicates whether the class name ends with `-Controller`. We believe that those classes are more likely to be *Controller* classes.

Table 8: **C5: Collaboration features**, indicating the level of collaboration between a class and other classes.

№	Feature Name	Data Type	Explanation
F22	*numImports*	integer	The number of `import` statements in the class. Classes carrying the roles like *Controller*, *Coordinator*, *Interfacer*, and *Service Provider* might need to collaborate with many other classes. Thus, we assume that they might have many `import` statements.
F23	*numOutboundInv*	integer	The number of invocations to methods outside of itself. We assume that the *Coordinator*, *Controller*, and *Interfacer* invoke many methods outside of itself.

4.5. Machine Learning Classification Experiments

We experiment with three machine learning algorithms: Random Forest (RF), Multinomial Naïve Bayes (MNB), and Support Vector Machine (SVM). These algorithms are widely used in machine learning research and provide good performance on various applications. We do not conduct experiment with neural network techniques because they require very large training sets (10's of thousands of objects) which we do not yet have. We use stratified 10-fold cross-validation to evaluate the performance of each algorithm, measured using Precision, Recall, F1-Score, and Matthews Correlation Coefficient (MCC).

We perform two experiments for machine learning algorithms. In the first experiment, we analyze 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. Furthermore, we explore using the SMOTE [19] resampling technique to handle the imbalanced distribution of role stereotypes. Recent software engineering studies have used SMOTE to handle their imbalanced dataset (e.g., [20, 21]). When applied, the oversampling technique resamples all role-stereotypes but the majority one. We use SMOTE exclusively on training sets for all features. Finally, we compare the performance between using the regular and the SMOTE resampling technique.

In the second experiment, we examine which features are essential for classifying each role stereotype. For this, we use only one machine learning algorithm: the one that came out best in the first experiment. Here, 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 [22]. We can get this score by calculating a node's importance 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.

4.6. Generalizability of Trained ML Classifier

We study the trained machine learning classifier's generalizability by evaluating the use of different software projects as training data to classify the other remaining software projects. We use the most performant machine learning algorithm in the previous experiment and use different software projects as the training data and measure their performance.

4.7. Studying the Evolution of Role Stereotypes

To understand the evolution of a software structure concerning role stereotypes, we select a sizeable number of versions from our three cases. We apply the classifier to the source code from these versions and perform quantitative analyses and measurements on the resulting data. We also employ exploratory data visualisation techniques to discover insights on the dynamics of these software projects.

5. Experiment Results

This section presents the results of our experiment on the CRI classifier.

5.1. Multi-role Classification of all Stereotypes

In the first experiment, we evaluated the dataset of aggregated data (of 1,547 classes) from our three cases K-9 Mail, SweetHome3D, and Bitcoin Wallet. In particular, we calculate Precision, Recall, F1 Score, and MCC using 10-fold cross-validation. Then we compare this result with the performance reported in our previous work that used only the K-9 Mail dataset [23]. We use 1000 trees in the Random Forest classifier to handle the large number of features that we used. Table 9 demonstrates the performance result in these two datasets.

Table 9: Performance comparison of the additional dataset

Dataset	Classifier Method	Precision	Recall	F1-Score	MCC
All	RF	0.64±0.07	0.67±0.07	0.64±0.06	0.55±0.09
	MNB	0.56±0.04	0.56±0.06	0.55±0.05	0.40±0.07
	SVML	0.60±0.05	0.64±0.06	0.60±0.06	0.49±0.09
	RF (SMOTE)	0.66±0.07	0.62±0.08	0.63±0.08	0.50±0.11
	MNB (SMOTE)	0.58±0.04	0.55±0.07	0.55±0.06	0.40±0.08
	SVML (SMOTE)	0.63±0.06	0.58±0.09	0.59±0.08	0.45±0.10
K-9	RF	0.64±0.09	0.67±0.06	0.63±0.06	0.52±0.08
	MNB	0.53±0.10	0.51±0.09	0.49±0.10	0.33±0.12
	SVML	0.54±0.07	0.60±0.06	0.55±0.06	0.41±0.08
	RF (SMOTE)	0.65±0.08	0.62±0.08	0.62±0.09	0.49±0.10
	MNB (SMOTE)	0.54±0.11	0.46±0.08	0.45±0.10	0.31±0.10
	SVML (SMOTE)	0.56±0.08	0.51±0.07	0.52±0.08	0.36±0.10

Table 9 shows that Random Forest outperforms MNB and SVML and remains the best classification algorithm. There is an increase in the performance of the Random Forest using

all three projects compared to the performance using only K-9 Mail. This increase is possibly due to the addition of training data for less frequently appearing role-stereotypes, such as *Controller* and *Structurer*. On the other hand, we also observed no significant differences in applying SMOTE resampling technique in both cases. We argue that, by using the SMOTE resampling technique, the number of the dataset of each role in both cases will still be equal. Moreover, the increase of the dataset's size from using only K-9 Mail to using all three projects may not be significant enough to increase the classifier's performance.

5.2. Single Role (Binary) Classification

In the second experiment, we run a binary classification on each role stereotype. From our original dataset, we create six new datasets (one for each role-stereotype) that use exactly two labels: one label for the role-stereotype at hand and one label *Other* representing the combination of all remaining role-stereotypes. As for the machine learning algorithm, we use Random Forest, which offered the best performance in the multi-role classification experiment from Section 5.1. Table 10 shows the results for 10-fold cross-validated evaluation for each role-stereotype classification using the imbalanced K-9 Mail dataset following the previous work [23] and both imbalanced and SMOTE-resampled of our three cases extended dataset.

Comparing imbalanced K-9 Mail dataset (Table 10a) and imbalanced three cases dataset (Table 10b), we can see the increase of performance in the least frequent role-stereotypes, namely, *Controller*. The addition of two other projects in the dataset increased the number of least frequent stereotypes that led to the performance increase. On the other hand, the addition of the two projects did not increase the performance of the already most frequent dataset, namely, *Information Holder* and *Service Provider*.

On the imbalanced dataset of all cases (Table 10b), based on an interpretation of the MCC score in previous studies [24, 25], the classifier has a very good performance (MCC score = 0.70) in detecting *Information Holders*, moderate (MCC scores between 0.40 and 0.50) at detecting *Controllers*, *Interfacers*, and *Service Providers*, fair (MCC score = 0.21) at detecting *Structurers*, and low (MCC score = 0.14) at detecting *Coordinators*. We suspected that the different number of training sets for each classifier contributed to this varying performance.

Meanwhile, the use of SMOTE resampling technique (Table 10c) increased the performance of the three role-stereotypes, namely, *Coordinator*, *Interfacer*, and *Structurer*. These three role-stereotypes could be found moderately frequent in the dataset among other role-stereotypes. We also can see a substantial drop in the least frequent role-stereotypes, i.e., *Controller*. For the remaining two frequent role-stereotypes, namely, *Information Holder* and *Service Provider*, there is no substantial change of performance. These results demonstrated that the effects of using SMOTE resampling technique are varied across different role-stereotypes.

Table 10: Single role (binary) classification result

Role	Precision	Recall	F1-Score	MCC
CO	0.42±0.41	0.23±0.28	0.28±0.31	0.28±0.31
CT	0.00±0.00	0.00±0.00	0.00±0.0	0.00±0.00
IH	0.86±0.11	0.71±0.11	0.78±0.09	0.70±0.13
IT	0.32±0.41	0.08±0.10	0.12±0.16	0.13±0.20
SP	0.70±0.09	0.64±0.15	0.65±0.10	0.44±0.13
ST	0.36±0.42	0.18±0.29	0.21±0.27	0.22±0.29

(a) Imbalanced K-9 Mail dataset (rerun from [23])

Role	Precision	Recall	F1-Score	MCC
CO	0.17±0.33	0.14±0.29	0.15±0.30	0.14±0.30
CT	0.68±0.38	0.42±0.37	0.46±0.33	0.49±0.32
IH	0.87±0.08	0.72±0.09	0.79±0.07	0.70±0.10
IT	0.64±0.30	0.37±0.26	0.44±0.26	0.43±0.26
SP	0.71±0.09	0.61±0.13	0.65±0.08	0.49±0.11
ST	0.55±0.46	0.11±0.14	0.17±0.18	0.21±0.21

(b) Imbalanced three cases dataset

Role	Precision	Recall	F1-Score	MCC
CO	0.26±0.17	0.24±0.26	0.24±0.21	0.20±0.22
CT	0.27±0.20	0.42±0.36	0.30±0.23	0.29±0.25
IH	0.82±0.09	0.78±0.09	0.79±0.06	0.69±0.09
IT	0.58±0.18	0.58±0.21	0.56±0.15	0.50±0.17
SP	0.67±0.10	0.68±0.12	0.67±0.07	0.49±0.11
ST	0.22±0.13	0.20±0.15	0.20±0.13	0.16±0.13

(c) SMOTE-resampled three cases dataset

The Random Forest classifier gives the best performance in classifying role-stereotypes. The addition of classes from two more projects to the dataset increases the performance of the least frequent role-stereotype binary classifiers, namely Controller. The use of the SMOTE resampling technique seems to increase the performance of the binary classifier for moderately frequent role-stereotypes. The performance of the most frequent role-stereotypes is not substantially different when using SMOTE resampling. As a whole, it cannot be concluded that the use of SMOTE resampling always provide better performance.

5.3. Comparison to Existing Classifiers

In this section we compare our approach to the main previous work by Dragan and Moreno. This comparison was extracted from our previous work [23] to keep this paper self-contained. To enable comparison, we first mapped the categories from Dragan's [2] and Moreno's [3] taxonomy onto Wirfs-Brock's [1]. It is worth noting that Moreno's stereotypes were adapted from Dragan's. For the mapping, we consider the equivalence of Moreno's JStereoCode[7] results with our own manually labeled ground-truth. We also take into account how the definitions of the categories correspond to each other. This was not obvious because JStereoCode categories concern implementation characteristics while Wirfs-Brock's involve conceptual responsibilities. Table 11 shows the descriptions of Dragan's & Moreno's stereotypes [2, 3] and their mapping to Wirfs-Brock's.

Table 11: Class stereotype taxonomy of Dragan & Moreno [2, 3]

Class Stereotype	**in[2]**	**in[3]**	**Description**	**WB-Role-stereotype**
Entity	✓	✓	Encapsulated data and behavior. Keeper of the data model and business logic.	Information Holder
Minimal Entity	✓	✓	Trivial Entity that consists entirely of accessor and mutator methods.	Information Holder
Data provider	✓	✓	Entity that consists mostly of accessor methods.	Information Holder
Data class	✓	✓	Degenerate behavior—it has only get and set methods.	Information Holder
Pool		✓	Consists mostly of class constants and a few or no methods.	Information Holder
Commander	✓	✓	Entity that consists mostly of mutator methods	Service Provider
Boundary	✓	✓	Communicator that has a large percentage of collaboration methods,,a low percentage of controller- and not many factory methods.	Service Provider
Boundary+Data Provider		✓	Boundary class that provides access to its state.	Service Provider
Boundary+Commander		✓	Boundary class that provides access for modifying its state.	Service Provider
Factory	✓	✓	Consists mostly of factory methods.	Service Provider
Controller	✓	✓	Controls external objects—the majority of its methods are controllers and factories.	Controller
Pure Controller	✓	✓	Consists entirely of controller and factory methods.	Coordinator
Large class	✓	✓	Contains a high number of methods that combine multiple roles, i.e., it consists of accessors, mutators, collaboration and factory methods.	Not suitable to any role
Lazy class	✓	✓	Its functionality cannot be easily determined. It consists mostly of incidental- and get- or set methods.	Not suitable to any role
Degenerate class	✓	✓	Very trivial class that does very little - it consists mostly of empty- and get- or set methods.	Not suitable to any role
Small class	✓		A class that only has one or two methods.	Not suitable to any role

[7]Archived at `http://web.archive.org/web/20160207063920/http://www.cs.wayne.edu/~severe/JStereoCode/`

5.3.1. Mapping of Taxonomies

We applied the JStereoCode tool to K-9 Mail. Out of the 779 classes of our K-9 Mail dataset, 193 were successfully labelled according to the taxonomy by Dragan & Moreno. The remaining classes were either not classified successfully by JStereoCode or (Java) interfaces, which do not provide functionalities and, thus, do not correspond with Wirfs-Brock's stereotypes. Next we looked up in our ground truth which stereotypes these classes belong to in our taxonomy. Table 12 shows the summary of this. Cells in this table are shaded if we map them onto each other based on theoretical grounds.

Table 12: Empirical comparison between JStereoCode [3] and Wirfs-Brock [1] based on our ground truth for K-9

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 11 shows that Moreno provides more specific stereotypes for data and entity classes: *Entity*, *Minimal Entity*, *Data Provider*, *Data Class*, and *Pool*. Table 12 shows that the empirical- (highest numbers) and theoretical (shaded cells) approaches agree on that these stereotypes map onto *Information Holder*.

Next, we establish correspondence between *Commander*, *Boundary* (and its refinements *+ Data Provider* 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, as 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* stereotypes respectively 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 design smell "God Class" by combining too much functionality. This characteristic of lumping functionality is mostly present in the *Coordinator*.

The 43 classes that are labelled Degenerate by JStereoCode only hold information without providing many methods. For this reason, these classes were classified as Information Holders in our taxonomy.

There are no theoretical grounds to map the categories *Large Class* and *Lazy Class* to any of our stereotypes. These categories are design smells rather than role stereotypes[8]. Their paper does not explain why these categories are used. Our hypothesis is that their reasoning is that classes that have been poorly designed (i.e., are design smells) do not properly represent any role-stereotypical responsibility. Hence, these categories are used as the 'other'-bucket which does not clutter up the classification of properly designed classes. In our approach, we do not use such an 'other'-category and will leave these classes out of statistical comparisons.

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

5.3.2. Role Stereotype Classification Performance

In this section, we attempt to compare the classification performance between our classifier and Dragan's StereoClass [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. The authors do not report any standard performance metrics (such as precision or recall). However, precision can be calculated from the data provided by Table IV of paper [2].

We compare this to the result of our classifier on the K-9 dataset. Our classifier successfully classified all 779 classes compared to their 193, and hence is more robust and complete. Table 13 shows precision rates of StereoClass and our classifier grouped by Wirfs-Brock'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 the column "Tool" in Dragan's original table (Table IV in [2]).

The fourth column "Precision Rate (%) – StereoClass", 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). Other measures such as recall, MCC, F1 Score can not be computed because of unavailability of the data.

[8]It is also unclear to us how *Large Class* and *Lazy Class* are different from *Degenerate*.

Table 13: Precision rates for StereoClass (on HippoDraw) and our classifier (on K-9)

Dragan's stereotypes	**Wirfs-Brock's stereotypes**	**Num. classes**	**Precision rate (%)**	
			StereoClass	**Our classifier**
Entity	Information Holder	13	58	89
Minimal Entity		3		
Data Provider		8		
Data Class		2		
Commander	Service Provider	7	88	72
Boundary		15		
Factory		5		
Controller	Controller	6	78	0
Pure Controller	Coordinator	2	33	98

The fifth column shows the precision rates (in percentage) of our classifier in classifying single role-stereotype. Data in this column is referenced to data in the column "Precision" in Table 10a. It is worth noting that the precision rates of StereoClass (in the fourth column) and our classifier (in the fifth column) are derived from different evaluation settings, using different ground truths and on different datasets (HippoDraw and K-9). Therefore, it is impossible to draw an absolute performance comparison between 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*. Our classifier performs 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 our classifier in classifying *Controller* classes, at the precision rate of 78% compared to our 0%. Both of these numbers may be exaggerated by the small number of controller-type classes present in both datasets.

6. Classification Feature Importance

In this section, we discuss the importance of our classification features regarding their performance in classifying each role-stereotype. Table 14 shows the average Gini scores of each feature (represented in a row) on every role-stereotype (represented in columns) obtained from the above-mentioned binary classification experiment. We omit the feature *isStaticClass* for having a poor score of less than 0.001. In model training, the feature *classPublicity* was split into four mutually exclusive boolean features representing each possible value ('default', 'private', 'protected', and 'public') due to its categorical nature. In Table 14, for each role-stereotype, the top five features (highest score) are marked with "$*$", while the features that were expected to determine a role-stereotype explained in Section

4.4 are underlined. These qualities are also reflected by table cell colors: blue cells indicate top five features, yellow cells indicate expected distinctive features, and green cells indicate an intersection of both qualities, i.e., we expected them to be important and our results confirmed it. For each feature, the number of times where it ends up in the Top 5 and the number of times where it is expected to be determinant are computed and represented in columns "#Top." and "#Exp.", respectively. The rows are then sorted from highest to lowest #Top. value.

Table 14: Feature importance for each role-stereotype

Feature	**# Top.**	**#Exp.**	**CO**	**CT**	**IH**	**IT**	**SP**	**ST**
loc	6	2	0.121*	0.155*	0.083*	0.095*	0.112*	0.134*
numImports	6	4	0.118*	0.112*	0.128*	0.183*	0.079*	0.089*
numAttr	3	1	0.060	0.058	0.105*	0.042	0.156*	0.072*
numMethod	3	1	0.065*	0.091*	0.052	0.040	0.055	0.070*
numIfs	3	1	0.068*	0.101*	0.064	0.037	0.062*	0.048
numParameters	2	2	0.061	0.054	0.111*	0.063	0.064*	0.058
numOutboundInvocation	2	3	0.059	0.084*	0.059	0.070*	0.054	0.051
isOrEr	2	2	0.035	0.018	0.073*	0.106*	0.031	0.044
getters	1	1	0.042	0.035	0.040	0.051	0.045	0.067*
numWordName	1	2	0.069*	0.035	0.038	0.038	0.045	0.063
isInnerClass	1	2	0.031	0.046	0.018	0.081*	0.028	0.042
classPublicity_public	0	0	0.043	0.021	0.019	0.014	0.027	0.050
numPublicMethods	0	4	0.057	0.063	0.052	0.036	0.055	0.048
numPrivateMethods	0	3	0.016	0.030	0.021	0.024	0.021	0.029
classPublicity_private	0	0	0.036	0.012	0.009	0.019	0.011	0.026
setters	0	1	0.018	0.027	0.017	0.016	0.042	0.023
classPublicity_default	0	0	0.030	0.020	0.012	0.034	0.011	0.023
numProtectedMethods	0	3	0.015	0.017	0.016	0.016	0.023	0.015
isEnum	0	1	0.008	0.000	0.023	0.004	0.037	0.009
isClass	0	0	0.012	0.002	0.015	0.006	0.011	0.008
isCollection	0	1	0.001	0.000	0.002	0.001	0.004	0.008
isInterface	0	1	0.013	0.000	0.030	0.009	0.013	0.006
isController	0	1	0.001	0.012	0.002	0.004	0.005	0.006
isPersist	0	1	0.011	0.004	0.005	0.004	0.004	0.005
isAbstract	0	0	0.008	0.003	0.007	0.006	0.006	0.004
classPublicity_protected	0	0	0.001	0.000	0.001	0.000	0.001	0.001

It can be seen from Table 14 that most of the scores are greater than 0.000 (except the three cells in column **CT** and one in column **IT**), meaning that all features in the list affect identifying a role stereotype from others. In terms of prediction power, features *loc* (number of lines of code) and *numImports* (number of import statements) stand out as always being in the top five most predictive features for every role stereotype. The significance of *numImports* implies that the level of collaboration of a class could reveal the

role stereotype the class plays in the design of a software system. Interestingly, Wirfs-Brock mentioned the causal relationship between roles and collaboration in her book, that "*the roles an object plays imply certain kinds of collaborations*" (p.159 [10]). With this finding, which is drawn from analysing three realistic software systems, we can confirm the statement in the other direction "*collaborations of a class could characterise its roles*".

Features *numAttr* (number of attributes), *numMethod* (number of methods), and *numIfs* (number of *if*-statements) ranked third to fifth places in the list as the most predictive features for three out of six role stereotypes. It is interesting that our (theoretical) expectation did not identify any of the top five distinguishing features for Structurers. Furthermore, we expected neither *numImports* nor *numParameters* to play a major role in identifying Information Holders. In hindsight, the significance of *numParameters* in an Information Holder makes perfect sense because methods in an Information Holder are most likely accessor-types that require little or no parameters.

To understand how values of these features spread over different role-stereotypes, we create boxplots of the values of the top 5 predictive features, as shown in Fig. 3. In each boxplot graph, role-stereotypes are sorted from the lowest to the highest median value of the corresponding feature. It can be seen from Fig. 3 that the value of the ranges of the features differs across role-stereotypes: some ranges are greater than others. For example, *Information Holder* classes are likely to have smaller *loc*, *numImports*, *numMethod*, and *numIfs*, while having a generally higher *numAttr* than *Coordinators* and *Service Providers*. This finding supports our expectation when selecting these features as mentioned in Section 4.4. There is also a case where our expectation goes otherwise: we expected *Service Provider* classes to have high *numImports* and *loc*. However, the graphs show that these numbers in fact belong to the lower end of the spectrum. A consistent trend across the boxplots is that *Controllers* and *Interfacers* almost always have the largest number of *loc*, *numImports*, *numAttr*, *numMethod*, and *numIfs*.

Moreover, 4 out of 5 most predictive features (i.e., *loc*, *numAttr*, *numMethod*, and *numIfs*) are *Complexity* features (**C2** in Section 4.4). This suggests that the complexity of a class is an important characteristic that relates to its role-stereotype. We further discuss the relation between role-stereotypes and a system's complexity in Section 9.1.

We analysed which features have significant contributions to identifying roles. Some features seem to be more predictive than others. Among them, *loc* and *numImports* stand out as the best discriminants.
The relation between the role stereotype of a class and the collaborations it has is a bidirectional relationship: the role a class plays can be inferred by its collaborations(s) and vice versa.

7. Generalizability of the Classifier

In this section, we study the generalizability of our approach. To this end, we explore different choices of training- and testing-sets. In particular, we study using one or two projects for training and then testing the classifier on the remaining project(s).

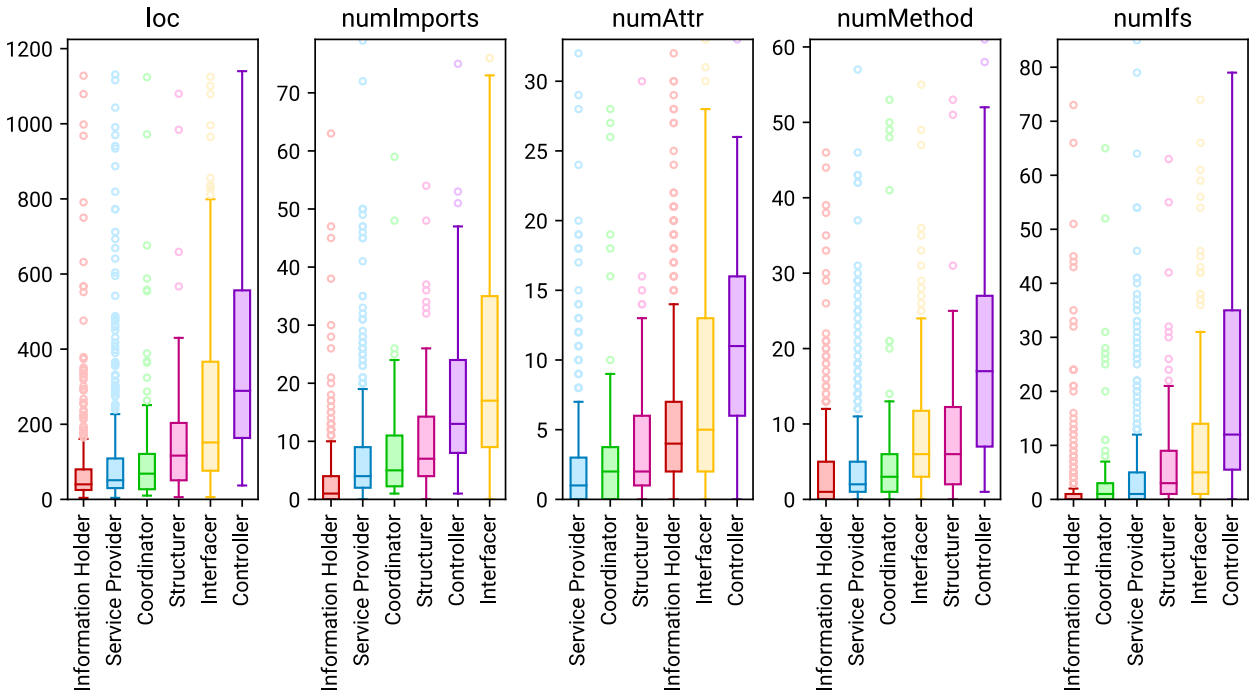


Figure 3: Distribution of loc, numImports, numAttr, numMethod, and numIfs across role-stereotypes
In each graph: Role-stereotypes are sorted from lowest to highest median value of the corresponding feature

Table 15: Performance of the classifier trained on K-9 Mail

Project	Precision	Recall	F1 Score	MCC
Bitcoin Wallet	0.65	0.52	0.56	0.38
SweetHome3D	0.72	0.73	0.72	0.62

7.1. Generalizability Experiment 1: Single Case Training

We start by applying the classifier trained with data from K-9 Mail from our previous work [23] to the other two cases. In this experiment, we use the Random Forest classifier with SMOTE resampling.

Table 15 demonstrates the classifier's performance trained on K-9 Mail on the other two cases, Bitcoin Wallet and SweetHome3D. Table 15 shows that the classifier performs average on classifying role-stereotypes of Bitcoin Wallet and slightly better on SweetHome3D. We investigate this further by the confusion matrix of the classifier for both cases presented in Table 16.

The confusion matrix for both cases (Table 16) shows that the classifier misclassified all *Coordinators* (2 classes) and *Controllers* (5 classes) in the Bitcoin Wallet project (Table 16a) but managed to classify some of these two role-stereotypes correctly for SweetHome3D (Table 16b). We think the poor classification happens more in Bitcoin Wallet because the number of *Coordinators* and *Controllers* in Bitcoin Wallet (2 CO and 5 CT) is much smaller than their numbers in SweetHome3D (21 CO and 38 CT). We believe that this is the main

Table 16: Confusion matrix of the classifier trained on K-9 Mail

Actual Label \ Predicted Label	CO	CT	IH	IT	SP	ST
CO	**0**	0	1	1	0	0
CT	1	**0**	0	1	2	1
IH	5	1	**42**	7	25	3
IT	18	0	1	**36**	7	0
SP	7	5	4	8	**32**	1
ST	0	0	1	3	4	**5**

(a) Bitcoin Wallet (222 classes)

Actual Label \ Predicted Label	CO	CT	IH	IT	SP	ST
CO	**3**	4	3	0	10	1
CT	0	**32**	0	3	3	0
IH	1	6	**200**	4	8	8
IT	0	2	2	**43**	13	3
SP	1	6	15	12	**116**	9
ST	1	2	13	7	10	**5**

(b) SweetHome3D (546 classes)

reason why the performance of the classifier on SweetHome3D is better than the performance on Bitcoin Wallet. Another concern for the classification of Bitcoin Wallet was that there was frequent misclassification of *Information Holder* as *Service Provider* (25 cases).

7.2. Generalizability Experiment 2: Double Cases Training

Next, we investigate the generalizability of our approach using the combination of two projects as training data and the remaining case as the testing data. This experiment aims to study the effect of having more training data than the previous experiment (i.e., using only K-9 Mail as the training data). Table 17 summarizes the performance of the classifier in this experiment.

Table 17: Performance of the classification trained on two cases and tested on the remaining case

Training Data	Testing Data	Precision	Recall	F1-Score	MCC
K-9 Mail and SweetHome3D	Bitcoin Wallet	0.65	0.55	0.58	0.41
K-9 Mail and Bitcoin Wallet	SweetHome3D	0.68	0.71	0.69	0.59
Bitcoin Wallet and SweetHome3D	K-9 Mail	0.58	0.60	0.57	0.42

When comparing Table 17 and Table 15, it seems that the performance of the classifier does not differ significantly. The classifier still has the same medium performance in classifying role-stereotypes of Bitcoin Wallet and SweetHome3D. In other words, the addition of another project to K-9 Mail as the training set did not give a significant impact on the classifier. Using two new projects, i.e., Bitcoin Wallet and SweetHome3D, as the training set and tested it on the K-9 Mail dataset also gave similar performance.

We then investigated the confusion matrix of the classification result using two projects training data presented in Table 18. Table 18 shows a lot of misclassification of *Coordinators* in all three cases. We think this is due to the low number of *Coordinators*, especially in the classification of K-9 Mail (Table 18c) which has the smallest total number of *Coordinators* from two cases in the training set (23 CO).

In the case of classifying *Controller*, the classifier failed to classify all *Controllers* in Bitcoin Wallet (Table 18a), even though the total number of *Controllers* from the other two cases in the training set was the highest (58 CT). We think this is because the number of

Table 18: Confusion matrix of the classifier trained on two cases and tested on the remaining case

Actual Label \ Predicted Label	CO	CT	IH	IT	SP	ST
CO	**1**	0	1	0	0	0
CT	1	**0**	0	0	3	1
IH	6	0	**44**	7	24	2
IT	12	0	1	**38**	11	0
SP	8	0	5	7	**37**	0
ST	0	1	1	3	5	**2**

(a) K-9 Mail and SweetHome3D (train) on Bitcoin Wallet

Actual Label \ Predicted Label	CO	CT	IH	IT	SP	ST
CO	**1**	3	6	1	9	1
CT	1	**32**	0	2	3	0
IH	1	8	**202**	3	9	4
IT	1	1	5	**38**	13	5
SP	1	6	26	7	**109**	10
ST	1	2	15	7	9	**4**

(b) K-9 Mail and Bitcoin Wallet (train) on SweetHome3D

Actual Label \ Predicted Label	CO	CT	IH	IT	SP	ST
CO	**1**	1	14	16	37	10
CT	0	**4**	3	4	8	1
IH	0	1	**170**	7	44	9
IT	0	1	4	**40**	25	7
SP	2	4	27	24	**242**	24
ST	0	1	7	4	29	**8**

(c) Bitcoin Wallet and SweetHome3D (train) on K-9 Mail

Controllers in Bitcoin Wallet is too small (5 CT) due to different coding styles in which the developer decided to have only a small number of *Controllers*.

To see how the addition of another project to the training set affects the classification of each role-stereotypes, we also compared the confusion matrix of classifying Bitcoin Wallet and SweetHome3D using K-9 Mail as training set (Table 16a and Table 16b) with the confusion matrix of two cases training set (Table 18a and Table 18b). In classifying Bitcoin Wallet, the addition of SweetHome3D in the training set increases the correct classification of *Coordinator*, *Information Holder*, *Interfacer*, and *Service Provider* but reduces the correct classification of *Structurer*, resulted in a little increase of performance. On the other hand, adding Bitcoin Wallet to the training set for classifying SweetHome3D reduces the correct classification of all role-stereotypes except *Controller* resulting in a performance decrease of the classifier.

On a different angle, comparing the combination of Android applications (K-9 Mail and Bitcoin Wallet) and pure Java application (SweetHome3D) as the training and test set led to an interesting finding. The combination of two Android applications in the training set improved performance in classifying pure Java applications (Table 18b). Meanwhile, combining an Android application with a pure Java application in the training set to classify the other Android application gave almost equal performance (Table 18a and Table 18c) but less than the previous combination. However, we cannot make any further conclusion without studying more Android and pure Java application cases.

The Random Forest classification model trained with data from one or two projects shows a medium performance when classifying the other project(s). Learning from two projects does not lead to a significant increase in classification performance compared to using one.

8. Studying the Evolution of the Structure of Software

For studying the evolution of the structure of software, we first apply the classifier against a more extensive dataset: the source code of multiple versions of each of our case subjects, K-9 Mail, Bitcoin Wallet, and SweetHome3D. We then perform data analysis and exploratory visualisations to discover facts and confirm assumptions on the role stereotypes in software design.

8.1. Selecting a relevant number of versions for each project

Data was gathered from multiple versions of each of the projects. For each project, we used the version described in Table 1 as a pivot. We selected versions relative to this pivot by selecting versions before and after the pivot with an approximate interval of 3 months. We chose this interval to get good coverage of the significant changes to the project of interest while also getting a reasonable number of versions to study the longitudinal evolution. The interval might vary slightly depending on whether commits were available in the projects' version control system on the exact date based on the interval. This way, we obtained 36 versions for Bitcoin Wallet over its lifetime. To keep things comparable, we selected 37 versions for K-9 Mail and SweetHome3D. The period for the selected versions spans from the beginning of 2011 to the beginning of 2020.

Test classes such as JUnit-classes were removed since they do not represent the structure of the actual software. We manually identified and removed those classes from each version of the three projects.

To identify the class role stereotypes in the source code, we apply our role stereotype classifier to all classes of all selected versions of all three projects. We used the model trained using Random Forest method on SMOTE-resampled ground truth data from the pivot versions of all three projects for this study.

One of the selected versions of Bitcoin Wallet, specifically the version from 2011-10-03, was removed. This version contains many Java classes that were not included in any other subsequent versions of Bitcoin Wallet. When included, this version caused a massive spike at that time point to the degree that the rest of the data became insignificant. Our interpretation is that this was a trial to include some library into the project that was already discarded in the next version that was committed. We choose to remove the data from this version for the sake of readability. The removal does not affect the overall picture of the evolution of the Bitcoin Wallet project.

SweetHome3D had two separate sub-projects, "FurnitureLibraryEditor" and "TextureLibraryEditor", included in the main project directory at the pivot version. However, these subprojects were not part of earlier versions of the project, and they are not considered

parts of the main application. Therefore, we choose to focus on the main SweetHome3D application in our study and exclude the sub-projects.

All the raw data extracted from the tools can be found in a Github repository[9].

8.2. Graphs & Analyses

Our initial step is to create graphs showing the numbers of each stereotype over time, as shown in Figure 4. For the sake of legibility, we name the versions we selected in integers starting from 1. This numbering does not reflect actual release version of each project. For completeness, we include a version of the graph for Bitcoin Wallet that does not ignore the problematic version number 3 (from 2011-10-03) in Figure 5. Note that the high frequencies of *Information Holders* and *Service Providers* in versions 4 to 7 of Figure 4c are likely to be remnants of the removed version 3 and should be interpreted as such.

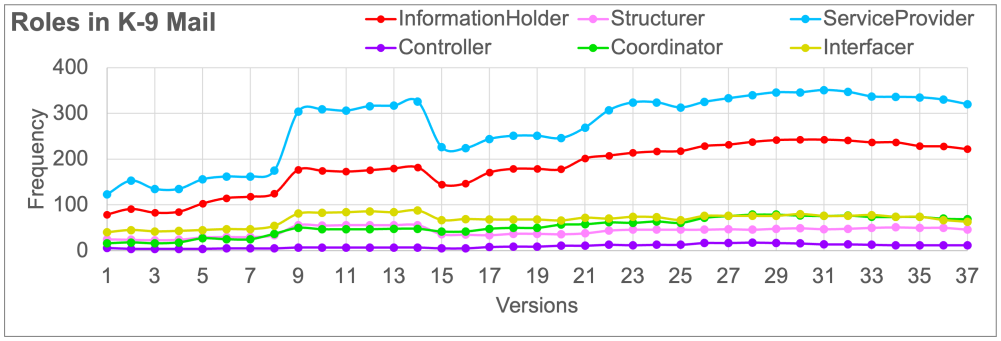
Changes in the distribution of role stereotypes over time. In K-9 Mail, the most common role is *Service Provider*. *Information Holder* is also very common. All roles except for *Controller* make a noticeable increase at the beginning of 2013 (version 9) and then decreases again halfway through 2014 (version 15). In SweetHome3D, *Information Holder* is the most common role. *Service Provider* is also very common. The distribution of roles in SweetHome3D remains mostly unchanged throughout the time period. In Bitcoin Wallet, the most common role is *Service Provider*. *Information Holder* and *Interfacer* are also very common, both growing steadily throughout the time period. The number of *Information Holders* increase drastically at the beginning of 2018 (around versions 29 to 30) and causes the distribution of roles to make a noticeable change, making *Information Holder* the most common role instead of *Service Provider*.

The three projects share some characteristics but also have some significant differences in terms of the distribution of roles over time. Bitcoin Wallet and K-9 Mail both have *Service Provider* as the most common role, while SweetHome3D has *Information Holder* as the most common role. In K-9 Mail and SweetHome3D, the distribution of roles is mostly unchanging throughout the entire time period, while the roles in Bitcoin Wallet seems to fluctuate a bit. Bitcoin Wallet has an unusually high amount of *Interfacers* in relation to the other roles compared to K-9 Mail and SweetHome3D.

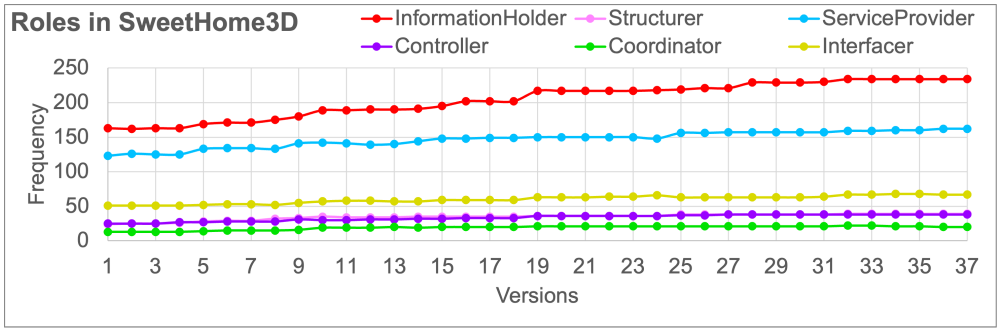
All three projects have a somewhat low numbers of *Structurers*, *Controllers*, and *Coordinators* compared to other roles. In SweetHome3D, the *Coordinator* is the least common role, which is not the case in Bitcoin Wallet and K-9 Mail that instead has the *Controller* as the least common role.

The graphs from Figure 4 do not show the dynamics of each class. Instead, we visualise the same data in a different format (Figure 6), presenting each class within a project as a horizontal line that spreads across versions (depicted as vertical dark lines). The colour of a line segment denotes its role stereotype in each specific version. We use the same colour coding throughout this section. The classes are grouped by the role stereotype of their first appearance.

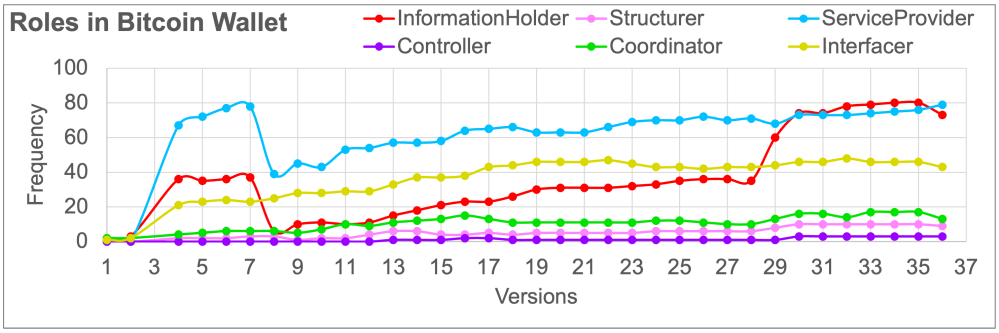
[9] `https://github.com/fabianfroding/apcrm/tree/master/Resources/raw-data`



(a) K-9 Mail



(b) SweetHome3D



(c) Bitcoin Wallet

Figure 4: Distribution of the roles in each version of the three cases.

These graphs provide insight on the stability of a software project at a glance. We can see that classes in SweetHome3D tend to exist for longer period of time and that there are less new classes introduced during the selected window compared to the other two projects. In both K-9 Mail and Bitcoin Wallet we can see new classes introduced that are later moved or removed. (It should be noted that classes are identified based on their fully qualified Java class names, e.g., `com.fsck.k9.Account`, and as such, if a class were moved to a different package or renamed, it is considered a separate class in this representation.)

These graphs also show how some classes change stereotypes during their lifetime. For example, in the topmost of the *Service Provider* group of SweetHome3D (coloured blue, between the 350 and 400 class marks), we can see a couple of classes that switched roles to

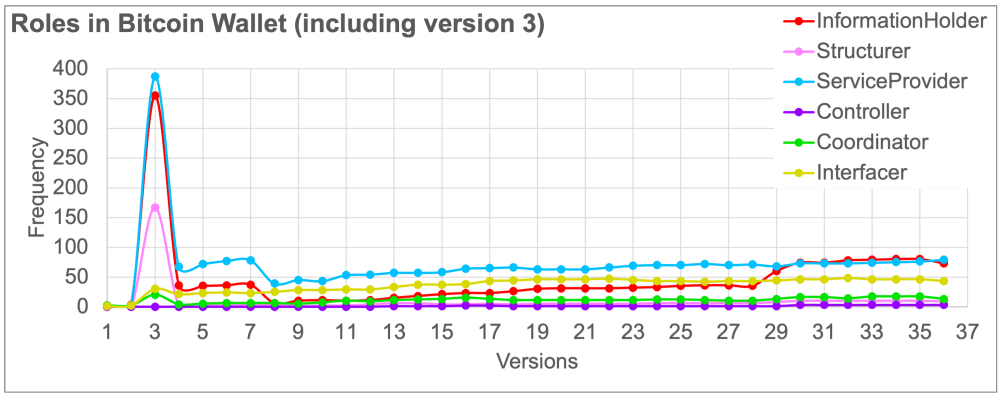


Figure 5: Distribution of the roles in Bitcoin Wallet (including version 3 from 2011-10-03)

Controller (purple) early in their lifetime. A few classes down, we see a class that turned to *Interfacer* (yellow) but quickly changed again and spent most of its life as a *Coordinator* (green). A notable number of classes that appear or disappear together, such as those around the 300 mark and 900–1000 marks in K-9 Mail (Figure 6a) are likely subjects to refactoring events. To explore further into this direction, we analyse two specific cases where class role change has happened in two projects K9-Mail and Bitcoin Wallet in Section 8.3.

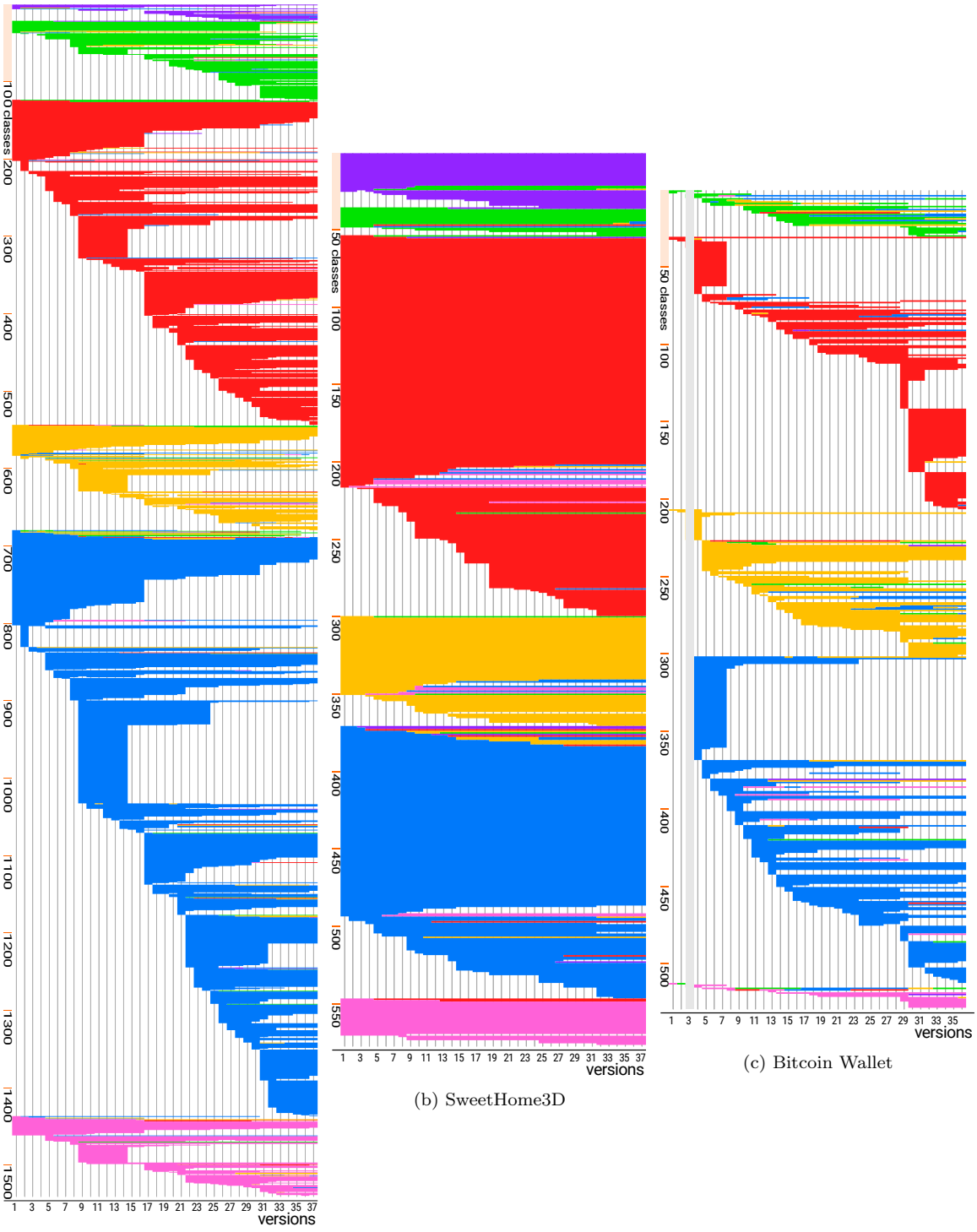
Figure 7 shows the number of individual changes in role stereotypes. These numbers do not represent the actual number of classes that switch role stereotypes, as some classes changed stereotypes more than once in their lifetime.

K-9 Mail underwent substantial number of changes in role-stereotypes, concretely at 161 instances. A majority amount of changes are from *Service Provider*, *Interfacer*, and *Coordinator* to other roles, with 47, 41 and 23 instances respectively. Moreover, a majority of classes changes from other roles to *Service Provider* and *Interfacer*, with 47 and 33 instances respectively.

SweetHome3D has 62 occurrences of classes changing roles throughout the time period, with 14 instances derive from classes switching from *Service Provider* to other roles, and another 14 from *Structurer*. Additionally, 20 classes changed from other roles to *Service Provider*.

In Bitcoin Wallet, there have been 85 instances of classes changing roles throughout all selected versions. Most changes are seen in *Service Provider* and *Interfacer*. Specifically, there are a total of 29 instances of classes switching from *Service Provider* to other roles and a total of 23 instances of classes switching from other roles to *Service Provider*. Furthermore, 20 classes changed from *Interfacer* to other roles, and conversely, 21 classes changed from other roles to *Interfacer*.

Most classes that change stereotypes only changed once in the selected time window. To be precise, 83, 53, and 44 classes from K-9 Mail, SweetHome3D, and Bitcoin Wallet respectively did so. However, some other classes changed stereotypes more than once. We call these classes "chameleons" and found 32, 7, and 17 of them in each respective case. The largest number of changes that a class experienced is five (`FolderInfoHolder`, K-9 Mail),

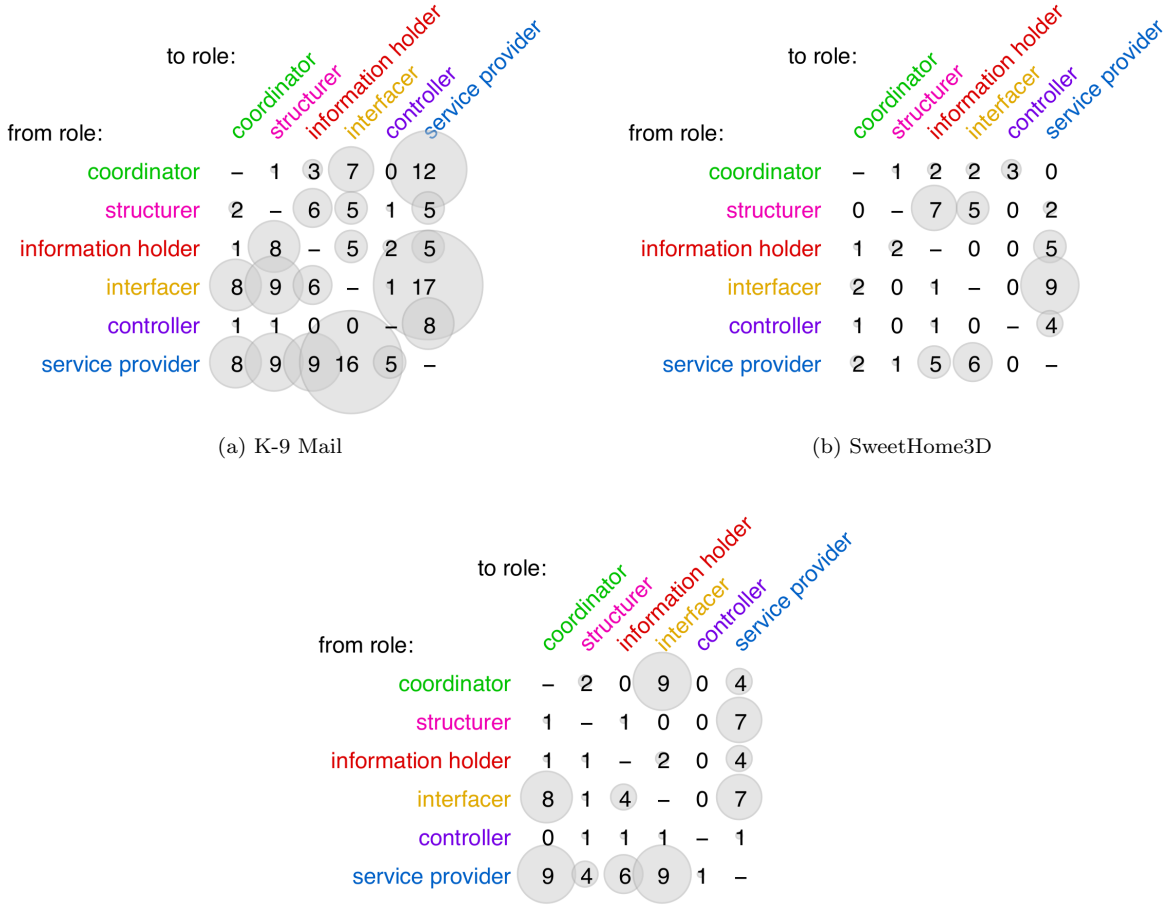


(a) K-9 Mail

(b) SweetHome3D

(c) Bitcoin Wallet

Figure 6: Roles of each class in each version of the three cases. The x-axis represents versions. The charts differ in the y-axis scale due to the significant difference in the number of classes



(a) K-9 Mail

(b) SweetHome3D

(c) Bitcoin Wallet

Figure 7: Frequency of role stereotype changes in three cases.

but the largest number of unique role stereotypes that a class assumed is four. Figures 8, 9, and 10 depict these classes and all the assumed role stereotypes.

From these charts and those of Figure 7, we identified frequently occurring changes such as *Interfacer* to *Service Provider*, *Service Provider* to *Interfacer*, *Service Provider* to *Information Holder*, and, in fact, *Service Provider* to *any role* reverting to *Service Provider*, and to a lesser extent *Interfacer* to *any role* reverting to *Interfacer*. However, there are indications that some of these patterns are the result of misclassification by our ML model. An obvious one is K-9 Mail's `FolderInfoHolder` class—this class name strongly suggests an intended role of *Information Holder*, however our classifier predicted variants of this class as *Structurer* and *Service Provider* in addition to *Information Holder*. We discuss the patterns of role stereotype evolution further in 9.3.

Figure 8: "Chameleons" in K-9 Mail.

Figure 9: "Chameleons" in SweetHome3D.

8.3. Examples of classes changing stereotypes

In this section, we look at two cases where the classifier identifies a change of role of a class (from one version to the next version that we look at). We look at one case from Bitcoin Wallet and one case from the K9-Mail app. We create a visualisation that shows the class that changes and its dependencies to other classes. The diagram also shows the package containment. In both cases, analysis of the source code supports the conclusion

Figure 10: "Chameleons" in Bitcoin Wallet.

that the role of a class has indeed changed. This finding confirms our view that the classifier can be used to detect changes to the system's design, including refactorings.

Example 1. Class `BlockchainService` (Bitcoin Wallet project). Figure 11 shows the change of role of class `BlockchainService` from *Interfacer* at version 4 (commit id fe59b12 in Fig. 11) to *Information Holder* at version 5 (commit id 77fc50b). At first, class `BlockchainService` contains methods and the logic needed to response to a number of events at `WalletApplication`, thus playing the *Interfacer* role between the main app interface and internal block chain services. Here, this class has a dependency to class `WalletApplication`.

In the next version, the class is refactored to become an *Information Holder* that only contains constants and definitions of services. Looking into the diagram, we realize that a new class called `BlockchainServiceImpl` was created. This class contains the implementation of the methods that previously resided within class `BlockchainService`. In addition, all the dependencies that were from the class `BlockchainService` to class `WalletApplication` are now moved to class `BlockchainServiceImpl` (and therefore do not show in the picture because this visualisation only shows direct dependencies to and from the class 'in focus', i.e., `BlockChainService`). Hence, class `BlockchainServiceImpl` now plays the *Interfacer* role. Here we observe the splitting off of responsibility into two classes: definition of the interface is done by `BlockchainService` and the implementation of the services resides in `BlockchainServiceImpl`.

Example 2. Class `MessageWebView` (K9-Mail project). In this example, the role of class `MessageWebView` was changed from *Interfacer* to *Coordinator* as shown in Figure 12. In the version 22 (commit id 28232ed in Fig. 12), the role of class `MessageWebView` was *Interfacer*. The code of this class contains the configuration of the message to load into a

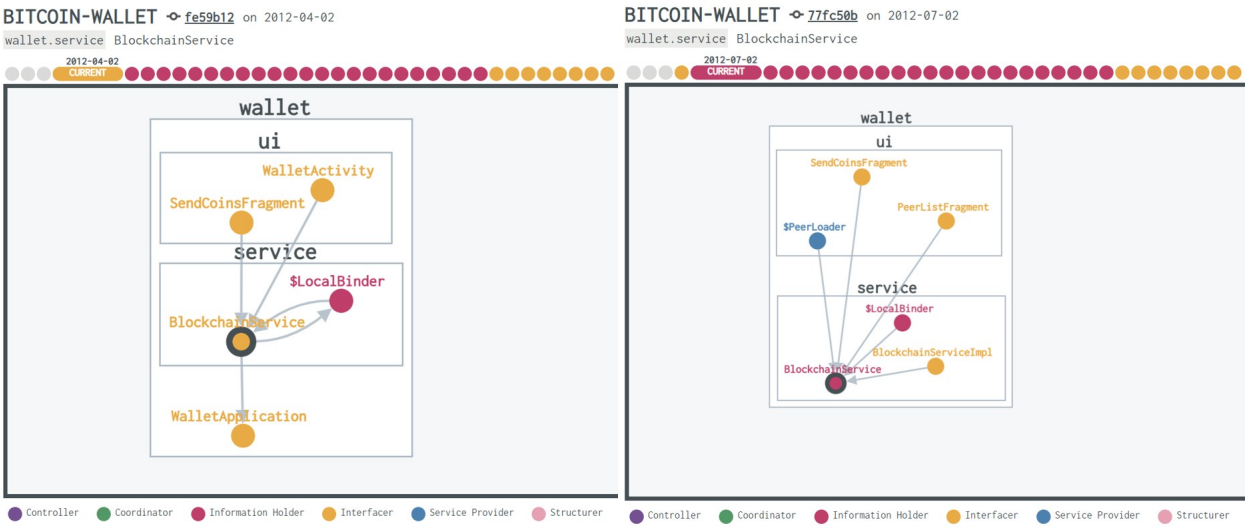


Figure 11: Role change at class `BlockchainService` of the Bitcoint Wallet project.

web view (which allows displaying web pages as a part of the Android *activity* layout). In particular, it contains logic to wrap the message text in an HTML header and footer so that the message can be appropriately displayed in the web view.

In version 23 (commit id d276bbd), the role of class `MessageWebView` changed into *Coordinator*. It coordinates the task of formatting message text to a newly created class called `K9WebViewClient`. This class also extended the basic HTML text display to handle URIs within the message body via *intent* (an Android mechanism that allows starting another app from within a running app). Besides, it delegates the work of handling attachments to another newly created class called `AttachmentResolver`.

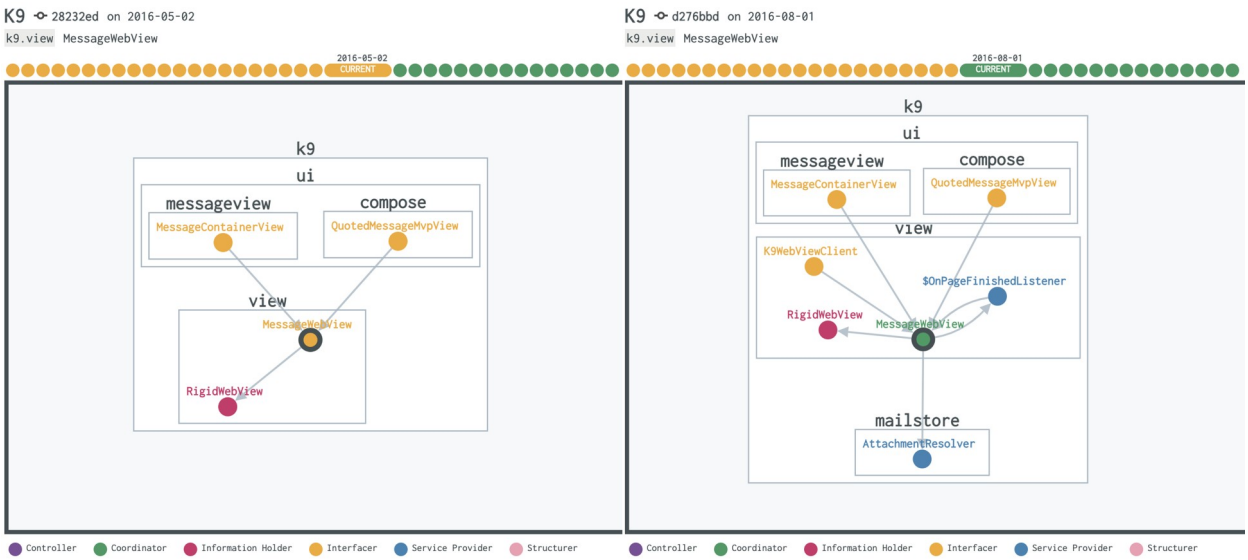


Figure 12: Role change at class `MessageWebView` of the K9-Mail project.

9. Discussion: Finding New Uses of Role Stereotypes

In this section, we reflect on the results described in the previous sections. The automatic classification into role stereotypes opens new directions for analysing and understanding software designs. Having complete identification of role stereotypes for all classes in a software system enables analysis of software structure, leading to software design comprehension and summarisation. Where versioned source code of the software is available, complete role stereotypes classification also enables the analysis of the evolution of a software structure, potentially identifying refactorisation history and patterns (and anti-patterns) in software design evolution.

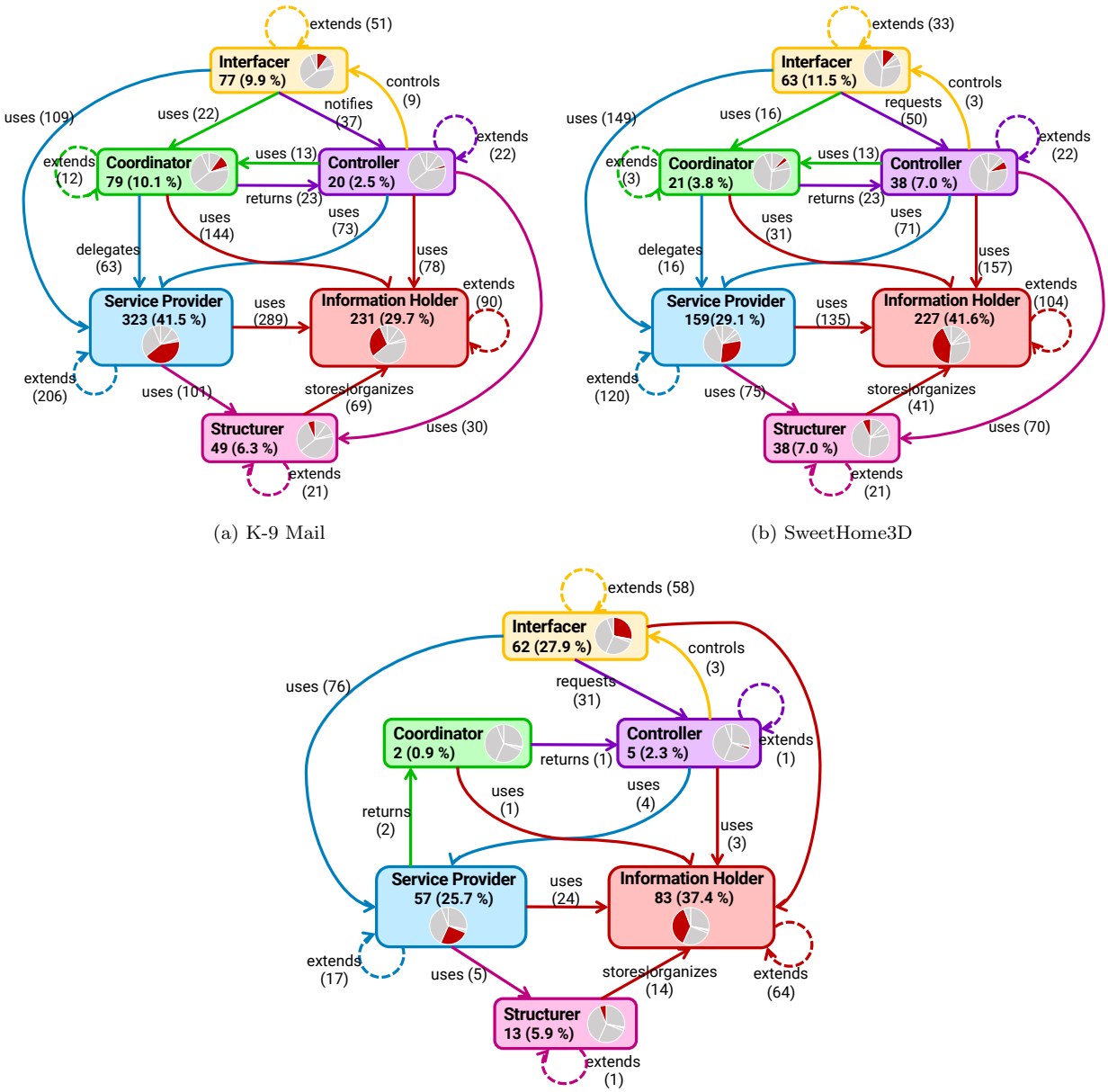
9.1. Analysing the Dominant Role Stereotypes within Software Project

This subsection explores whether there is any regularity concerning the occurrence of role stereotypes across the multiple cases that we consider. Fig. 13 contains three diagrams that show the frequency of occurrence of the role stereotypes and the relationships between them for our three cases. The numbers below the role stereotype names in each diagram indicate the absolute and relative number of occurrences of the role stereotypes. The labels on the edges indicate the type and frequencies of occurrence of relations between these role stereotypes. Looking at these diagrams side by side, we can see similarities and differences between the three cases. Next, we will look into these in more detail.

Role stereotypes occurrence in systems is imbalanced. In Fig 13, we can see that some stereotypes occur often, and some are rare. Nevertheless, the numbers suggest that there may be regularities across software systems. For example, the *Information Holder*-stereotype covers 30% - 40% of the classes for all systems. Another example is that the *Controller* stereotype seems rare: it represents only between 2.3% - 7.0% in all systems. This finding aligns with Dragan's: by applying *StereoClass* on five open source systems, Dragan also finds that the stereotypes that they consider differ in frequency of occurrence, but the relative frequencies of occurrence are somewhat regular across systems [2].

We note that the categories of stereotypes used by Dragan have similarities and differences to our categories. For example, both our and their approaches use categories for *Information Holder* and *Controller*, for which their semantics seem to match. Furthermore, we find similar percentages of occurrence for these categories in Dragan's work (29.6% on average for *Information Holder* and 1.9% for *Controller*) compared to the numbers across our three cases (34.0% and 4.0%, respectively). This ties with the very definition of some role stereotypes. For example, *Coordinators* provide service to *Interfacers* by coordinating several *Service Providers*, so, logically, there would be more *Service Providers* than *Coordinators*. Indeed, this highly unbalanced distribution of stereotypes harms the performance of the machine learning algorithms.

On the occurrence of stereotypes and the complexity of systems. Coordinators are present in systems designs when there is a need for coordinating or delegating jobs from one class to another class. This need arises when a class becomes too big in size and complicated. It follows that *Coordinators* are seen less often in small systems and more often in large



(a) K-9 Mail

(b) SweetHome3D

(c) Bitcoin Wallet

Figure 13: Occurrences of role stereotypes in the three cases

systems. From our 3 cases, we observe that the frequency of *Coordinators* increases with the size of the systems. In particular, there are only two *Coordinator* and five *Controller* classes in the case of Bitcoin Wallet—together, less than 4% of all classes. On a closer look, these classes contain little logic for the controlling and coordinating of workflows from *Interfacer* to *Service Provider* and *Information Holder* classes. In fact, in Bitcoin Wallet, most user requests are distributed directly to *Information Holders* and *Service Providers* by *Interfacers*. It results in a high frequency of collaboration between these role stereotypes.

This pattern is different from the SweetHome3D and K-9 Mail cases, where a large amount of work is coordinated via *Coordinators* and *Controllers*.

Indeed, the reasonably small amount of *Coordinators* and *Controllers* in Bitcoin Wallet can partly be explained by looking into the design intention of the system. That is, Bitcoin Wallet focuses on providing wrapper functions to the *bitcoinj* library for Android devices. It relies on *bitcoinj* to maintain a wallet and send/receive transactions in Bitcoin protocols. Consequently, the logic and workflows defined in Bitcoin Wallet are primarily used for monitoring the process and adapting it for Android users to use. Given this characteristic of the design, the direct contact from *Interfacers* (where user requests are made) to *Service Providers* (as execution units) and *Information Holders* (as data storage) can be considered as efficient.

Compared to Bitcoin Wallet, K-9 Mail and SweetHome3D can be considered more complex because the main logic/workflows are defined internally. In contrast, Bitcoin Wallet can build on much logic that is contained in an external library. In particular, K-9 Mail handles the mailing process from the level of mail protocols to end-user management level (such as multiple account management, scheduling services), and SweetHome3D provides services for designing home plans which might include creating/joining walls, arches, insertion of windows, doors, et cetera. Thus, the complexity of the systems/services requires more fine-grained coordination mechanisms, i.e. via employing *Coordinators* and *Controllers* in between *Interfacers* and *Service Providers* or *Information Holders*.

The presence and distribution of role stereotypes in a system reflects its architectural characteristics. Our analysis of three different systems from the perspective of role stereotypes has led us to understand that such analysis can uncover how a system's architectural design follows some architectural style or design principles. In this section, we illustrate some of the insights that can be obtained from such analyses. For understanding these analyses, recall that Bitcoin Wallet and K-9 Mail are built on the Android framework, while SweetHome3D is a pure Java desktop application.

Smaller numbers of *Controllers* (2.5% and 2.3%) are observed in the two Android apps than the pure Java app (7%). The nature of Android applications provides a partial explanation: they are built upon Android frameworks that encapsulate the Android OS's low-level functionalities. Thus, some *Controller*, *Structurer*, and *Interfacer* classes at UI and activity management level, as well as collaborations between them, might be hidden away; fewer control logics need to be created as part of an overall application. Meanwhile, in SweetHome3D, persistence tasks are implemented via basic java.io functions, and the user interfaces are implemented mainly by using and customising Swing components. Moreover, being a pure Java app, extra control logic are needed for handling portability across different execution environments (such as different operating systems). Implementing these could result in extra control units and data units. For this, we can expect a higher occurrence of *Controller* and *Information Holder* classes.

Examining the two Android applications, on the one hand, we observe that the portion of *Interfacers* in Bitcoin Wallet is much higher than in K-9 Mail. On the other hand, K-9 Mail contains more *Service Providers* (41.5%) than Bitcoin Wallet (25.7%). This number

suggests that Bitcoin Wallet must handle more user interaction and a smaller amount of actual transactional services, whereas K-9 Mail has a greater focus on building business mailing services. This finding aligns with that of Bagheri et al. [26]. By studying 200 Android applications in the Google Play store, the authors found that Android applications in different domains have different architecture characteristics regarding the type and number of components. In particular, "finance" applications, such as Bitcoin Wallet and other banking or payment systems, provide richer user-interface than other applications. On the other hand, applications for which "communication" is the key feature, such as K-9 Mail, largely depend on listening, receiving, and handling system events. Such events, in turn, are typically handled by a *Service Provider*-type of class.

While in the Android projects *Service Provider* is the dominant stereotype, in SweetHome3D *Information Holder* appears most frequently. Reflecting the nature of this software system, this makes sense since SweetHome3D is a data-intensive software that manages various information regarding three-dimensional objects in an environment.

The role a class plays within a software system reflects design intention. The design intention is, in its turn, affected by architecture style, choices of technology/library use, and domain-specific requirements. Therefore, it is possible to use role stereotypes as a tool for profiling/capturing software systems' design styles and intentions. It also enables the possibility to compare designs of different software systems via their role stereotypes.

9.2. Feature Importance and Detection of Anti-patterns

Looking back at the importance of class features as discussed in Section 6, we have shown the top five determining features to differentiate the role stereotype of a class: *loc*, *numImports*, *numAttr*, *numMethod*, and *numIfs*. We have also shown the value trends of each feature in Figure 3 and how some of our assumptions when selecting features in Section 4.4 were confirmed or debunked. For example, we can see that *Information Holders* tend to have fewer *loc*, *numImports*, *numMethods*, and *numIfs* but have a moderate *numAttr* since it correspond with the *information* they hold. We propose an additional use of these features in relation to role stereotypes, to detect anti-patterns in program source code.

Sharma and Spinellis [27] presented a catalogue of *design smells* (which, for our purpose, are considered equal to *design anti-patterns*) including smell detection techniques. Be it metrics-based, heuristic-based, history-based, ML-based, or optimisation-based, all smell detection techniques act on software metrics. To illustrate our point, we take, for example, anti-patterns *Large Class* and *Lazy Class* [28, 29], which directly correlate with some features that we use in this study.

A *Large Class* is defined as "a class that has grown too large in term of LOCs." It correlates directly with our top-defining feature *loc*. It can be arbitrary to decide how many lines of code is too many, but by examining Figure 3 we can say that, for example, it is quite normal for a *Controller* to have 200 to 500 lines of code, while 150 may already be too large for an *Information Holder*. Taking the distribution of *loc* in different roles at face

value may introduce bias toward our studied case, but at the very least, we can rank each role stereotype in terms of its average *loc* and derive some heuristics to tailor the detection of *Large Class* specifically according to a class' role stereotype. Our data suggest that *Service Providers* and *Coordinators* can have similar *loc* that are generally fewer than those of *Interfacers* and *Controllers*.

Contrast to the *Large Class*, a *Lazy Class* is "a class that has few fields and methods." As this description suggests, this anti-patterns is related with our features *numAttrs* and *numMethods*. Figure 3 suggests that *Service Providers* and *Coordinators* cannot be considered lazy even if they have no attributes. Thus, for these role stereotypes, detecting *Lazy Class* anti-pattern should involve only *numMethods*. Conversely, *Information Holders* are characterised by their attributes, and therefore *numAttr* may have larger impact than *numMethods* in detecting *Lazy Class* in *Information Holders*.

Furthermore, the evolution of role stereotypes over time can also contribute in identifying anti-patterns. While we cannot say that a specific pattern of role change correlates with a specific anti-patterns, we suppose that the existence of an uncommon stereotype change in a class should raise a flag and encourage the programmer to examine the class for anti-patterns. We repeat for clarity that the common changes are *Interfacer* to *Service Provider*, *Service Provider* to *Interfacer*, *Service Provider* to *Information Holder*, *Service Provider* to *any role* reverting back to *Service Provider*, and *Interfacer* to *any role* reverting back to *Interfacer*.

We also propose that the ratio of role stereotypes within a software project can detect architectural smells. For example, we have shown that in both Android applications, most classes are either *Information Holders* or *Service Providers*. When an Android application project has a more significant number of *Interfacers* compared to *Information Holders*, for instance, we suspect that sub-optimal architectural design may be in place. However, as we only examined two Android applications and one desktop application, we cannot stake a claim on the validity of this method. A more extensive study on role stereotypes in different classes of software projects would benefit this direction of thought.

Role stereotypes can be used to aid in anti-pattern detection, either by tailoring design metrics thresholds to specific role-stereotypes, or by finding undesirable patterns of role stereotypes.

9.3. Patterns in the Evolution of Role Stereotypes

Refactoring and the change in the distribution of role stereotypes. In section 8, we saw a dramatic rise in both *Information Holder* and *Service Provider* classes in Bitcoin Wallet version 4, with virtually no change in other stereotypes (Fig. 4c). These additions appear to be reverted in version 8. We noted that this change is likely to result from the experimental addition of an external library into the codebase. These couple of changes happen very early in the development of Bitcoin Wallet, where architectural instability can be expected.

There is also a high increase of *Information Holders* around version 29. Since none of the other roles decreases at this time-point, this indicates that the developers added a significant amount of *Information Holder*-classes. We suspect that the developers attempted to extend

the application by adding *domain models*. Domain models are classes that represent real-world objects or entities and hold information related to these objects[30]. Considering that the responsibility of an *Information Holder* is to hold and provide information[10], it is likely that new domain models were added to the project, ergo the drastic increase of *Information Holders* at the beginning of 2018. However, inspection and evaluation of the source code are necessary to confirm this suspicion.

We saw an increase in several roles in K-9 Mail in version 9. Similarly to what happened in Bitcoin Wallet, this increase did not occur in parallel with a decrease of any other roles, indicating an extension of the application. However, since this increase concerns all roles except the *Controller*, it becomes difficult to speculate what the source of this increase could have been. These classes were removed not long after, coinciding with the addition of a small number of classes (Fig. 6a), which suggests the possibility of merging functionalities into new classes. Again, source code inspection is crucial to confirm this.

In the results for SweetHome3D, we did not see any significant changes in the distribution of the roles. Instead, we can see a slow and steady increase in all roles, indicating a stable and professional evolution of the software design.

All three projects share high frequencies of *Information Holders* and *Service Providers*. Based on the responsibilities of each role, we suspect that the demand for classes the hold and provide information, and compute tasks and perform work are high for these types of applications.

Based on the definition of the responsibility of an *Interfacer* as *"handling and transforming requests and information between different parts of a system"*, it seems likely that the variety and number of parts that a system has might increase the occurrence of *Interfacers*. In our cases, we can see that all three projects have more or less the same number of *Interfacers*, specifically around 60 to 70. It is interesting to note, even when the three projects differ in size and number of classes, that despite their difference in size, they still have a similar demand of classes to perform the work of the *Interfacer* stereotype.

To summarise, we saw three significant changes in the distribution of role stereotypes in Bitcoin Wallet, two in K-9 Mail, and none in SweetHome3D. A change in the ratio of role stereotypes indicates a major change to the design. In other words, we postulate that *refactoring events* can be characterised by such significant changes. Furthermore, the relatively low number of refactoring events suggest architecture design-stability in all three projects.

We also ponder the question of whether "chameleon" classes are possible results of refactoring events. Close observation on Fig. 6 present no such correlation in K-9 Mail. However, a small number of classes did switch role stereotypes in the second refactoring event of Bitcoin Wallet.

On role stereotype evolution patterns. From the interpretation of the presented data in Section 8, it is evident that the majority of classes classified as *Service Provider* are more prone to change roles over time in all of the selected projects, and conversely, most classes also tend to switch from other roles to *Service Provider*.

An interesting finding is that there exists interchanging relationships between certain

pairs of role stereotypes. This can be seen most evidently in Bitcoin Wallet, for example, 8 classes changed from *Interfacer* to *Coordinator*, and conversely, 9 classes changed from *Coordinator* to *Interfacer*. Accordingly, 7 classes changed from *Interfacer* to *Service Provider*, and 9 classes changed from *Service Provider* to *Interfacer* (Figure 7c). More examples can be seen with other pairs of roles in all of the projects (Figure 7). These changes could be an issue of inaccurate classification in our ML model, as it may have classified a class as one role and the same class as another role at a later point in time. To confirm such speculation, we compare the confusion matrix table of our classifier (Table 18) with Figure 7. Our analysis indicates that the classifier tends to misclassify *Interfacers* as *Service Providers* and vice versa. Similar tendencies apply to the roles of *Information Holder* and *Service Provider*. Accordingly, in all projects, most changes in roles do involve *Interfacer* changing from and to *Service Provider* as well as *Information Holder* changing from and to *Service Provider*.

Although there are some correlations between the misclassification of certain pairs of role-stereotypes and the number of changes in those pairs, we cannot confirm whether interchanging relationships between the pairs or the relationships are the results of inaccurate classification without performing thorough code inspections. For example, if a class switch roles from *Interfacer* to *Service Provider* with only a few changes in the source code, or with changes that are insufficient to be labelled as *Service Provider*, we can suspect that the interchanging relationships are derived from the inaccurate classification of role-stereotypes.

The same principle applies to classes that have changed role-stereotypes multiple times throughout the time period (Figures 8, 10, and 9). We can see that the largest number of roles a class has been assigned is four, and thus, we are uncertain whether the changes come from the developers' intention or the accuracy error of the classifier. We only suspected that it is unlikely for a class to have been assigned four different role stereotypes. Further code inspection is required to validate the suspicion.

Furthermore, we also suspect that if the classifier had been trained on a more broad range of projects, it might have reduced the classifier's sensitivity and consequently reduced the number of role changes.

A change in the distribution of role stereotypes indicates a refactoring event. A significant change in the ratio of role stereotypes reflects a major change in the design. "Chameleon" classes could be appropriately caused by refactoring events, or the result of misclassification due to the corner cases of the classifier.

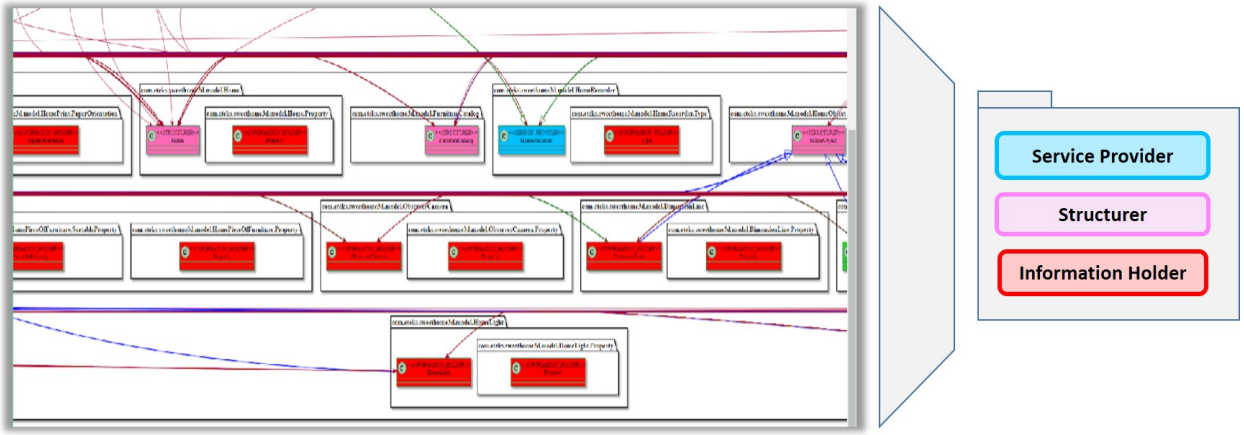
9.4. Use in Program Comprehension

Another use we foresee of using role-stereotype classification is in program comprehension and design summarisation. Knowing the role stereotypes of classes can aid in program comprehension: the role-stereotype suggests a type of responsibility that a component has, and thus the types of functionality that it should contain and the types of collaboration(s) that it can engage in. This information can be inferred by an IDE and cued to the developer. Also, for understanding patterns in the design, knowledge of role stereotypes is useful.

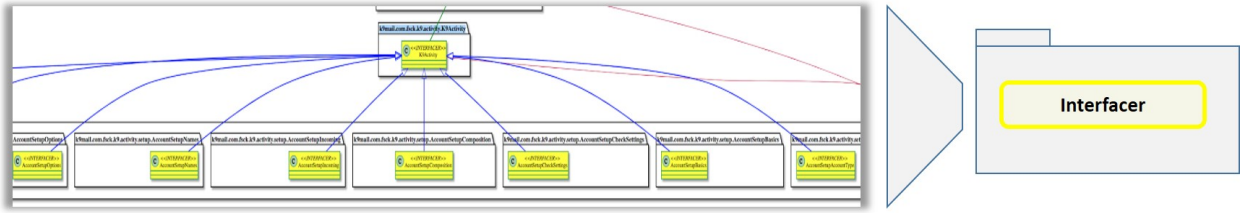
One method used for program comprehension is reverse engineering. Often, reverse engineering produces visualisations that contain much information and much detail. The

knowledge of role stereotypes can aid in producing good layouts of reverse-engineered diagrams. Moreover, knowledge of role stereotypes can aid in reducing the information presented to developers through design summarisation. Initial studies into summarisation have been described by Osman [31]. These studies showed that additional semantic information about components in the design could be useful. Figure 14 shows two examples of design summarisation. These are examples of actual patterns found in the cases studied in this paper.

Various researchers work in program summarisation in different levels. Hu et al., for instance, uses NLP techniques to summarise Java methods [32]. We call this "method-level summarisation". On the other hand, Moreno uses NLP techniques, static code analysis, and software repositories mining to provide summaries of Java classes ("class-level summarisation") and automatically generate release notes, i.e., the most important changes between releases ("application-level summarisation") [33]. The *design summarisation* we refer to in the previous paragraph lies between class-level and application-level summarisation. We believe this kind of summarisation, where we identify how classes interact or collaborate, supports the direction of architecture recovery [34]. Furthermore, the significance of features we identify in our work may help both method-level and class-level summarisation by providing context to the method or class in question.



(a) Example of summarization into a Data Structure-concept



(b) Example of summarization into an Interfacer-concept

Figure 14: Example of summarization of a collection of classes into a higher level conceptual representation

10. Threats to Validity

Threats to Internal Validity. An OOP class may be responsible for multiple roles and responsibilities [10]. In this study, however, we have chosen to build a machine learner that captures only one role for each class. This choice might cause an incomplete view of the roles and responsibilities of a single class. However, given that only a low number of classes carry multiple roles (68 out of 1547 classes, $\approx 4.4\%$), we consider this threat acceptable and the trade-off to keep our classification model relatively simple.

Threats to External Validity. Our machine-learning classification model was trained and evaluated on two Android projects and one pure Java project in this study. There might be threats to the generalization of the classification model to other projects. In the future, we plan to extend the ground truth and possibly retrain the classification model proposed in this study with more projects, e.g. more pure Java projects or projects in other languages than Java.

We believe the methods used in this study can be generalized to other OOP systems in various programming languages for the following reasons: i) the notion of class role-stereotype applies to OOP in general, regardless of implementation programming languages; ii) scripts [18] used in this study can be used to extract source code features from different languages than Java. The XPath queries that were used to extract features from parsed srcML files can be adapted to other languages such as C#, C/C++ by following srcML language and grammar rules[10].

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

11. Conclusion and Future Work

Role Stereotypes have been introduced as a conceptual aid in designing and especially scoping components. In this paper, we show that the role responsibilities can for a large degree be automatically inferred from the source code. Knowledge about the role-stereotype of classes can be useful in i) facilitating program understanding, ii) architecture recovery, and iii) improving the detection of design smells.

A key contribution of this paper is the construction of a machine learning-based approach for the automatic classification of role-stereotypes of classes in Java. We find that the Random Forest algorithm without SMOTE resampling yields the best performance for the multi-class classification with an F1-Score = 0.64. For the binary classification, we experienced challenges with an imbalanced dataset, i.e., some role-stereotypes are rare compared to others. In the imbalanced data, the classifier has varied MCC performance in detecting each role-stereotypes (0.14 to 0.70). This can be partially explained by the low frequency of occurrence of these roles in the design (offering few training examples). The SMOTE resampling technique increases the number of rare role-stereotypes in the dataset, resulting in more balanced training data as shown by the change in MCC scores. However, the slight

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

gain at detecting Coordinators (MCC score = 0.20, a mere 0.06 gain) was achieved at the expense of a considerable performance loss in most other stereotypes.

We used the classifier to analyse 3 medium-size open source systems. Our findings suggest that the ratio of different role-stereotypes exposes characteristics of the type of application. Also, we studied the evolution of role-stereotypes over time. These analyses showed that large changes in the ratio of role-stereotypes coincide with major refactorings of the design.

Future Work

At this point in the research, it seems further improvement to the performance of the classifier would be valuable. In particular attention needs to be paid to improving the classification of 'rare' role-stereotypes (esp. Controller). We identified some directions that could potentially help to improve the performance of the classifier in general: i) using a probabilistic iterative approach, i.e., roles with high-precision can swing the classification of other classes based on the relationship between these classes, ii) normalising the values of several features, e.g., by considering the ratio of public and private methods rather than using the absolute numbers, iii) combining machine learning with the rule-based approach in an 'ensemble method', thereby exploiting that rules are not sensitive to small numbers of training data, iv) combining individual binary classifiers, and v) using deep learning methods, although this probably requires a much more extensive training set.

We also mentioned several times in our discussion the importance of more extensive study on role stereotypes in different types of software project. Indeed, further study involving systems such as web applications, embedded systems, and microservice applications, to name a few, would help us advance the understanding of role stereotypes in software architecture.

The fact that the classifier works in an automated way enables the rapid labelling of role-stereotypes for extensive collections of classes and practically the entire source code of systems. In turn, this enables novel analysis that sheds light on the anatomy of software designs, such as analyses of the frequency of particular collaboration patterns between different stereotypes. We believe there are yet undiscovered regularities in the anatomy of software designs that can be uncovered through further studying these role-stereotypes for larger sets of projects. In particular, we suggest applying role stereotype analyses together with tools that aim to recognise refactoring events—especially at the level of architecture design. It would be interesting to see whether role stereotype perspective contributes additional understanding into the nature of refactoring.

Moreover, now that we have a robust automatic classifier for role-stereotypes, we can study advanced questions: How do the distributions of role-stereotypes differ across different types of architectures or business domains? Can role-stereotypes help us to better understand the evolution of software over time? Can role-stereotypes be used for tailoring test-generation strategies?

References

[1] R. Wirfs-Brock, Characterizing classes, IEEE Software 23 (2) (2006) 9–11. doi:10.1109/MS.2006.43.

[2] N. Dragan, M. L. Collard, J. I. Maletic, Automatic identification of class stereotypes, in: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609703.

[3] L. Moreno, A. Marcus, Jstereocode: automatically identifying method and class stereotypes in java code, in: IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, 2012, pp. 358–361. doi:10.1145/2351676.2351747.

[4] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, 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. 36 (1) (2010) 96–118. doi:10.1109/TSE.2009.69.

[5] M. Genero, J. A. Cruz-Lemus, D. Caivano, S. Abrahão, E. Insfran, J. A. Carsí, Does the use of stereotypes improve the comprehension of UML sequence diagrams?, in: 2nd Symposium on Empirical Software Engineering and Measurement, ESEM '08, ACM, USA, 2008, pp. 300–302. doi:10.1145/1414004.1414059.

[6] O. Andriyevska, N. Dragan, B. Simoes, J. I. Maletic, Evaluating UML class diagram layout based on architectural importance, in: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005, pp. 1–6. doi:10.1109/VISSOF.2005.1684296.

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

[8] B. Sharif, J. I. Maletic, The effect of layout on the comprehension of UML class diagrams: A controlled experiment, in: 5th IEEE Int. WS. on Visualizing Software for Understanding and Analysis (VISSOFT 2009), IEEE, 2009, pp. 11–18.

[9] N. Alhindawi, N. Dragan, M. L. Collard, J. I. Maletic, Improving feature location by enhancing source code with stereotypes, in: 29th Int Conf on Software Maintenance (ICSM), 2013, IEEE, 2013, pp. 300–309.

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

[11] N. Dragan, M. L. Collard, J. I. Maletic, Reverse engineering method stereotypes, in: 22nd IEEE ICSM 2006, 24-27 September 2006, Philadelphia, Pennsylvania, USA, 2006, pp. 24–34. doi:10.1109/ICSM.2006.54.

[12] M. Lanza, 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.

[13] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, K. Vijay-Shanker, Automatic generation of natural language summaries for java classes, in: IEEE 21st ICPC (2013) San Francisco, 2013, pp. 23–32. doi:10.1109/ICPC.2013.6613830.

[14] A. Budi, Lucia, D. Lo, L. Jiang, 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), USA, July 7-9, 2011, 2011, pp. 613–618.

[15] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[16] M. Zanoni, F. A. Fontana, F. Stella, On applying machine learning techniques for design pattern detection, Journal of Systems and Software 103 (2015) 102–117. doi:10.1016/j.jss.2015.01.037.

[17] 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 (IWPC'05), 2005.

[18] Replication package of this study, http://oss.models-db.com/Downloads/JSS2019_SCAM_ReplicationPackage/, accessed: March 23, 2018.

[19] K. W. Bowyer, N. V. Chawla, L. O. Hall, W. P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, CoRR abs/1106.1813. arXiv:1106.1813.
URL http://arxiv.org/abs/1106.1813

[20] J. E. Montandon, L. L. Silva, M. T. Valente, Identifying experts in software libraries and frameworks among github users, in: 2019 IEEE/ACM 16th International Conference on Mining Software Reposi-

tories (MSR), IEEE, 2019, pp. 276–287.

[21] G. Catolino, F. Palomba, A. Zaidman, F. Ferrucci, Not all bugs are the same: Understanding, characterizing, and classifying bug types, Journal of Systems and Software 152 (2019) 165–181. arXiv:arXiv:1907.11031v1, doi:10.1016/j.jss.2019.03.002.

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

[23] A. Nurwidyantoro, T. Ho-Quang, M. R. V. Chaudron, Automated classification of class role-stereotypes via machine learning, to be in proceedings of the Evaluation and Assessment in Software Engineering conference (EASE 2019) (2019).

[24] Y. Ma, S. Fakhoury, M. Christensen, V. Arnaoudova, W. Zogaan, M. Mirakhorli, Automatic classification of software artifacts in open-source applications, in: Proceedings - International Conference on Software Engineering, 2018, pp. 414–425. doi:10.1145/3196398.3196446.
URL https://ieeexplore-ieee-org.ezproxy.lib.monash.edu.au/abstract/document/8595225

[25] J. Cohen, Statistical power analysis for the behavioral sciences, Academic press, 2013.

[26] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, N. Medvidovic, Software architectural principles in contemporary mobile software: from conception to practice, Journal of Systems and Software 119 (2016) 31–44.

[27] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158–173.

[28] W. H. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[29] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[30] S. Millett, N. Tune, Patterns, principles, and practices of domain-driven design, John Wiley & Sons, 2015.

[31] M. H. Osman, M. R. V. Chaudron, P. van der Putten, Interactive scalable abstraction of reverse engineered UML class diagrams, in: S. S. Cha, Y. Guéhéneuc, G. Kwon (Eds.), 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers, IEEE, 2014, pp. 159–166. doi:10.1109/APSEC.2014.34.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7090773

[32] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), IEEE, 2018, pp. 200–20010.

[33] L. Moreno, Summarization of complex software artifacts, in: Companion Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 654–657.

[34] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 486–496.