How do Design Decisions Affect the Distribution of Software Metrics?

Marcos Dósea
Department of Information Systems
Federal University of Sergipe
Itabaiana, Sergipe - Brazil
Department of Computer Science
Federal University of Bahia
Salvador, Bahia - Brazil
dosea@ufs.br

Cláudio Sant'Anna
Department of Computer Science
Federal University of Bahia
Salvador, Bahia - Brazil
santanna@dcc.ufba.br

Bruno C. da Silva
Department of Computer Science & Software Engineering
California Polytechnic State University
San Luis Obispo, CA - USA
bcdasilv@calpoly.edu

ABSTRACT

Background. Source code analysis techniques usually rely on metric-based assessment. However, most of these techniques have low accuracy. One possible reason is because metric thresholds are extracted from classes driven by distinct design decisions. Previous studies have already shown that classes implemented according to some coarse-grained design decisions, such as programming languages, have different distribution of metric values. Therefore, these design decisions should be taken into account when using benchmarks for metric-based source code analysis. *Goal.* Our goal is to investigate whether other fine-grained design decisions also influence over distribution of software metrics. *Method.* We conduct an empirical study to evaluate the distributions of four metrics applied over fifteen real-world systems based on three different domains. Initially, we evaluated the influence of the class design role on the distributions of measures. For this purpose, we have defined an automatic approach to identify the design role played by each class. Then, we looked for other fine-grained design decisions that could have influenced the measures. *Results.* Our findings show that distribution of metrics are sensitive to the following design decisions: (i) design role of the class (ii) used libraries, (iii) coding style, (iv) exception handling, and (v) logging and debugging code mechanisms. *Conclusion.* The distribution of software metrics are sensitive to fine-grained design decisions and we should consider taking them into account when building benchmarks for metric-based source code analysis.

CCS CONCEPTS

• **General and reference → Metrics**; • **Software and its engineering →** *Software creation and management*;

KEYWORDS

Design Decisions, Design Role, Metrics, Empirical Study

ACM Reference Format:
Marcos Dósea, Cláudio Sant'Anna, and Bruno C. da Silva. 2018. How do Design Decisions Affect the Distribution of Software Metrics?. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196321.3196337

1 INTRODUCTION

Software engineers sometimes have to carry out software maintenance activities to improve software design or to enhance program comprehension. To support these activities, there are source code analysis techniques that usually rely on metric-based assessment. For instance, techniques for identifying code smells [20] rely on metric-based detection strategies [7, 8, 33, 37]. A detection strategy uses logical operators to combine metrics and theirs respective thresholds to identify source code elements (usually classes or methods) with structural characteristics that correspond to a certain code smell [33]. A number of tools (e.g. PMD[1], Checkstyle[2], SonarQube[3] and NDepend[4]) support metric-based analysis of source code. However, the accuracy of metric-based assessment is heavily influenced by the calibration of thresholds for the used metrics [44].

Most of the existing metric-based assessment techniques are based on generic thresholds. We have a generic threshold for a given metric when we use the same single value for classifying into categories (such as low or high) every class (or every method) of one or more systems. For instance, Lanza and Marinescu [30] classify as *long* any method that has more than 20 lines of code (LOC) in Java systems. In this case, 20 is used as a generic threshold for LOC.

Different approaches for calculating generic thresholds have been proposed [2, 6, 19, 30, 48]. These approaches calculate thresholds based on the distribution of metrics obtained from measurement data over sets of software systems as benchmarks. However, recent studies have shown that the distribution of metrics can vary according to some design decisions made over classes or the whole system [3, 23, 53]. Thus, a generic threshold may not make sense for the entire set of classes in a system [31]. Zhang *et al.* [53], for instance, show that application domain and programming language are some of the design decisions that influence on the distribution of metric values. Aniche *et al.* [3] show that metric distributions are different for sets of classes playing different architectural roles. For instance, in an MVC-based system, a generic threshold value might

[1]http://pmd.sourceforge.net
[2]http://checkstyle.sourceforge.net/
[3]https://www.sonarqube.org/
[4]http://www.ndepend.com/

be too low for classes playing the *View* architectural role or too high for classes playing the *Controller* role. This might, for instance, lead to false code smell alarms or hide potential code smells. Both situations may hinder maintenance activities and developers' perception about the quality of the source code [26, 27, 39, 46, 51].

We hypothesize, however, that the design decisions studied so far are too coarse-grained to explain the differences in the metrics distribution. For instance, classes implemented with same programming language (e.g. Java) and playing the same architectural role (e.g. *Repository*) in different systems may use distinct persistence libraries (e.g. JDBC or JPA for the Java platform). Thus, the source code of classes playing the same architectural role and using the same programming language in different systems may have different distribution of metrics. Moreover, several classes are not bound to any reference architecture (e.g. Spring MVC, Android). For instance, in MVC-based systems, there are many classes that do not assume the roles defined by this architectural pattern (e.g. *Controller* or *View*). Therefore, guiding the source code analysis based only on architectural roles might not cover a reasonable number of classes.

In this paper, we carried out a study to investigate whether fine-grained design decisions affect metrics distributions and, therefore, should be taken into account when building benchmarks for metric-based analysis of source code. We analyze the source code of fifteen real-world open-source systems from three distinct domains (Eclipse plugins, Android Applications and Web-based Systems). Over the selected systems, we compute four metrics commonly used to assess method maintainability and then we evaluate the effect of design decisions on their distributions. We analyze the distributions over class methods grouped according to what we call as *design roles*. Design roles include architectural roles, but also include classes whose responsibility is application-specific and not bound to any particular reference architecture. For instance, *buffering* is a design role for classes responsible for buffering data of a song before playing it in music player applications.

Our investigation has three main perspectives. First, we investigate whether metric distributions vary between different design roles of the same system. Second, we compare the metrics distributions of classes from different systems but having the same design role. Our goal here is to verify whether other design decisions, besides the design role, also affect metrics distributions. Finally, we compare metric distributions of classes from the same design role but over different stable releases of the same system. Since releases of the same system tend to comprise the same design decisions, our hypothesis here is that the distributions would not vary significantly. We summarize our findings as follows:

- We found that different design roles from the same system drive different metrics distributions. These findings extend the results obtained by Aniche *et al.*, which considers only architectural roles. Using the design role concept (our approach) increases the number of classes that could be covered and assessed, for instance, by different thresholds.
- We then investigated whether the distribution of metric values of the same design role were similar across different systems of the same domain. If this occurs, we should consider building benchmarks with different systems that have similar sets of design roles. However, we found that the same design role (e.g. Persistence) can drive different distributions of software measures in different systems. Then, we conducted a manual and deep source code analysis to identify what design decisions made such distributions different.
- Finally, we investigated if the distribution of metric values of the same design role were similar across different releases of a system. If this occurs, we should consider building benchmarks with previous releases that underwent quality review. The results we obtained show that in most of the cases the same design role in a system did not vary significantly throughout different releases.

The remainder of this paper is organized as follows. Section II presents a motivating example. Section III presents our approach to automatically identify and assign design roles to classes. Section IV describes the settings of our empirical study to identify the impact of design decisions on the distribution of metrics. Section V presents the results of the study. Section VI discusses threats to validity. Section VII discusses related work. Finally, Section VIII presents conclusions and discusses implications of our research.

2 MOTIVATING EXAMPLE

The first goal of this motivating example is to illustrate that classes playing the same design role, but in distinct systems, may have different metric distributions. This example involves two open source MVC-based web applications: LibrePlan[5] and WebBudget[6]. LibrePlan is a project management, monitoring, and controlling tool, whereas WebBudget is a personal financial management tool. We downloaded the source code of both system from Github: WebBudget on October 20th 2016 and LibrePlan on November 9th 2016.

Firstly, we computed the Lines of Code per Method (LOC/Method) metric for methods of classes playing the Persistence design role in both systems. We manually identified those classes. Both systems use the Repository design pattern [1] and the Hibernate framework [10] for implementing persistence. We observed that all Persistence classes: (i) have the @Repository annotation or (ii) extend classes or implement interfaces with the "Repository", "DAO" or "Store" tokens in their names.

Afterwards, we applied the Mann-Whitney U statistical test with 5% confident level and Bonferroni correction [32] to compare the distribution of LOC/Method metric from WebBudget and LibrePlan samples. The test result showed differences between the two samples. We then conducted a manual analysis on both samples and noticed that methods from WebBudget use at least 50% more lines of code than similar methods in LibrePlan system.

Listing 1: Query Example in Libreplan system.

```java
public List<MaterialAssignment> getByMaterial(Material material) {
        return (List<MaterialAssignment>) getSession().
            createCriteria(MaterialAssignment.class).
            add(Restrictions. eq("materialInfo.material",
            material)).list();
}
```

However, why does this happen if they use the same framework (Hibernate) and design pattern (Repository)? We manually analyzed their source code and found another design decision responsible

[5]https://github.com/LibrePlan/libreplan
[6]https://github.com/arthurgregorio/web-budget

for that difference: coding style. To illustrate that, on the one hand, we show on Listing 1 a method extracted from LibrePlan with a single-line statement to query a database and return a list of objects. On the other hand, Listing 2, extracted from WebBudget, shows a similar query, but using five lines of code. One may argue about the quality of both source code fragments, but both coding styles are quite common in applications that use the Hibernate framework and thus represent a design decision usually obeyed by other classes on the same system [52]. In summary, this is an example that different choices related to a single design decision (coding style) may contribute to different metric distributions of the same design role across different systems.

Listing 2: Query Example in WebBudget system.

```java
public List<Movement> listByCardInvoice(CardInvoice cardInvoice) {
        final Criteria criteria = this.getSession().
            createCriteria(this.getPersistentClass());
        criteria.createAlias("cardInvoice", "ci");
        criteria.add(Restrictions.eq("ci.id",
            cardInvoice.getId()));
         criteria.addOrder(Order.desc("inclusion"));
        return criteria.list();
}
```

This motivating example also aims to illustrate that only considering architectural or design roles bound to reference architectures, as Aniche *et al.* [4] do, may leave most of the classes of a system out of metric distribution analysis. For instance, we identified that 83.3% of LibrePlan classes and 79,8% of WebBudget classes do not play any of the MVC architectural roles (e.g. Model, View, or Controller). They are classes designed to solve specific problems of each system specific context. For instance, Libreplan has a set of 40 classes with the design role of providing additional functionalities to HTML components. All these classes extend the HtmlMacroComponent abstract class. This group of classes may have specific design characteristics and, as a consequence, their metric distributions would be different from other system classes. Therefore, it may be important to consider this design role when grouping classes to analyze metric distribution.

3 HEURISTIC FOR IDENTIFYING DESIGN ROLES

Our main hypothesis is that design role is an important design decision that may impact on the distribution of metric values. Therefore, our empirical study (Section 4) takes into account the design role of every class of the analyzed systems. Many different role concepts have been proposed and discussed from different viewpoints [18, 40, 49, 54]. We consider the concept Wirfs-Brock *et al.* defined for software component design [50]. According to them, design role is a set of related responsibilities assumed by an object to fit into a community, such as, a framework or an enterprise architecture. We decided to use this concept because a large number of modern object-oriented systems are developed based on reference architectures [9]. In such systems, design roles are assigned to one or more classes through inheritance, interface implementation, or class annotations. In this context, we defined a heuristic to automatically identify the main design role played by each class. Our heuristic uses a customizable token-based method which considers the syntactic structures of the class. In Section 6, we discuss possible threats to validity to our decision of assign only a single design role to each class.

Table 1: Keywords and Proposed Predefined Design Roles

Keywords	Standardized Design Role
DTO, Entity	ENTITY
DAO, Repository, Storage	PERSISTENCE
Activity	ACTIVITY
Controller	CONTROLLER

The automatic identification of the design role a class plays is not a trivial task for the following reasons:

a) the same design role, even in the same system, can be assigned to a class by different mechanisms. For instance, in Spring MVC-based systems, a class playing the *Controller* design role can get the @Controller annotation or extend the *AbstractController* class.
b) the same design role can be assigned to a class by means of different levels of inheritance in the same hierarchy of classes. For instance, in Android applications, a class that extends the Activity superclass plays the *Activity* design role. Consider that the name of this class is *PreferenceActivity*. In addition, classes that extend the class *PreferenceActivity* also play the *Activity* design role.
c) the same design role may follow different naming patterns, usually related to used design patterns or frameworks. For instance, Repository and DAO (Data Access Object) are two common design patterns used to implement the *Persistence* design role. Superclasses or interfaces defining this design role usually have the keywords "DAO" or "Repository" in their names.

Based on these assumptions, we propose a keyword-based heuristic to assign design roles to classes with six steps.

Step 1: Preparing the table of keywords and corresponding predefined design roles. The heuristic receives as input a table that associates keywords with corresponding design roles. We call the design roles in this table as *predefined design roles*. Predefined design roles are design roles that, based on our previous knowledge of the domain, we know they are presented in a system and also we know keywords related to them. Table 1 shows examples of keywords and their corresponding design roles. For instance, classes implementing interfaces that contain keywords such as "Repository", "DAO" or "Storage" in their names usually play the *Persistence* design role. Based on the analysis of the three studied domains and our previous experience as developers of systems based on them, we built a table which is available on our website [7]. This table can be reused as it is or refined before starting the next steps, which are, in fact, the automatic ones. It is important to note in next steps that the predefined design roles are not the only ones assigned to classes. The heuristic discovers other non-predefined design roles during the process.

Step 2: Assigning design roles by means of annotations. Some architectures use class annotations to define the design role a class plays. For example, in MVC-based systems, developers use the

[7]https://sites.google.com/site/designdecisions2018/

@Service annotation to define that a class implements the *Service* design role. However, class annotations can be placed with other goals. For instance, the *@deprecated* annotation is used to indicate a deprecated class. For this reason, our heuristic considers class annotation to assign a design role to a class only if the annotation is included in the set of keywords defined in Step 1.

Step 3: Assigning design roles by means of inheritance: This step only applies for classes without a design role assigned to them in the previous step. It assigns to a class the design role associated to the name of the superclass at the top of the inheritance tree where that class is. This step considers two possibilities. The first possibility holds if the name of the superclass contains a keyword that matches with a keyword in the table defined in Step 1. In this case, the heuristic selects the corresponding predefined design role to assign to all subclasses. For instance, suppose a superclass called *AbstractController* with different levels of subclasses. And suppose that *"Controller"* is a keyword corresponding to the *Controller* predefined design role (Step 1). All direct or indirect subclasses of *AbstractController* would have the *Controller* design role assign to them. When there is no keyword matching with the name of the superclass, the second possibility holds: our heuristic creates a new design role (non-predefined) named after the superclass and associates it to its subclasses. Considering the same example, this step would create an *AbstractController* design role. This step does not consider Java platform classes as superclasses.

Step 4: Assigning design roles by means of implemented interfaces: Again this step applies only for classes without a design role assigned to them in the previous steps. It considers that classes implementing the same set of interfaces should be grouped in the same design role, as each possible set of interfaces can assign different responsibilities to a class. Again, there are two possibilities. First, if the name of at least one of the interfaces contains a keyword that corresponds to a predefined design role (Step 1), then the predefined design role is assigned to the all classes implementing the set of interfaces. On the other hand, if there is no keyword matching with the name of any interface, this step creates a new design role and names it by using the names of the implemented interfaces separated by comma and bounded by square brackets. For instance, this step assigns the design role named [IRepository, Comparable] to classes that implement *IRepository* and *Comparable*.

Step 5: Assigning the Entity design role: When the previous steps fail to identify a class' design role, this step applies. It assign the *Entity* design role to classes that has non-static attributes and has at least 90% of its methods starting with "get" or "set". These classes are responsible to encapsulate business model, including rules, data, relationships, and sometimes persistence behavior. This percentage can also be adjusted according to the evaluated system.

Step 6: Assigning the Undefined design role: When all previous steps fail to define a class' design role, the heuristic assigns to it a general design role called *Undefined*. This step is performed when the class does not contain any structural elements that allows the heuristic to associate other design role to it. For instance, the *Undefined* design role is usually assigned to utility classes because they generally do not use structural elements like inheritance or annotations. In fact, classes with the *Undefined* design role are classes our heuristic was not able to cover. In Section 7 we discuss how our heuristic increases the number of covered classes in comparison to previous works. We also make some suggestions for future works that might reduce the number of *Undefined* classes.

To support the proposed heuristic we developed a tool called DesignRoleMiner[8], also available in our website. The tool extends the MetricMiner tool [47]. It mines project versions on GitHub to identify the design roles of their classes. In addition, our tool calculates the metrics used in our study, which are described in the next section.

We carried out a preliminary evaluation of the heuristic and tool. This evaluation involved five developers and five governmental Web systems of a Secretary of State Treasury. Due to confidentiality reasons we did not make the data obtained from these systems available in our website. We selected this organization for convenience because one of the authors of this research had already worked on it. We invited all 40 developers of the organization to take part on the evaluation. Five of them accepted the invitation. They have at least 12 years of experience as software developers and 10 years working with Java. In addition, each of them is the most experienced and leading developer of one of the five systems. They have been leading maintenance and evolution tasks concerning the design of the systems for at least 5 years. Each developer evaluated the results of the heuristic for only one system, the one he leaded. Here we call the systems as S1, S2, S3, S4 and S5. They have 47, 70, 99, 181 and 808 classes respectively.

Each developer received a worksheet with the design role identification results generated by DesignRoleMiner. Each row of the worksheet included: class name, the design role the heuristic assigned to the class, and a field for the developer to answer if he agreed or not with the assignment. Each developer did that for all classes of the system, except for classes the heuristic classified as *Undefined.* They took between 30 to 120 minutes to finish it. The number of classes of each system influenced the time each developer spent to conduct the analysis. For instance, the analysis of S5, which is the largest system, required the longest period of time. During the evaluation, the developers had access to the source code of the system. They went to check the source code of some classes but they did not find it necessary for every class.

The developers agreed with the design role the heuristic assign to 1039 classes, which represents 86.2% of the total number of classes (1205 classes). On the other hand, according to them, the heuristic failed only for 15 classes (1.2%). The other 12.5% of the classes (151 classes) received the *Undefined* design role. Figure 1 shows the results per system. All the 15 misclassified classes belong to the system S5. Some misclassification occurred due to programmer mistakes on the use of the enterprise architecture. For instance, some classes were misclassified because they extended a class responsible to define the application constants. Programmers use this approach, which is not recommended, as a shortcut to access constants. Other errors could be avoided with adjustments in the table of keywords.

The design roles assigned to the highest number of classes were: Transaction (67.14%), Persistence (4.23%), Entity (5.98%), BackgroundProcess (3.40%) and Abas (2,07%). Transaction is an implementation of the Command design pattern [21], in which each

[8]https://github.com/marcosdosea/DesignRoleMiner

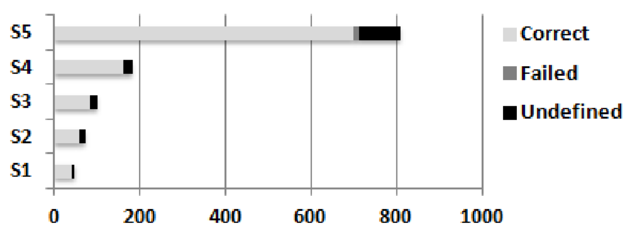

Figure 1: Evaluation of the Proposed Design Roles by System

action provided by the system is implemented in a class. This explains the high number of *Transaction* classes. Most of the classes with the *Undefined* design role are concerned with the application business logic. In fact, these classes did not use annotations, inheritance or implement interfaces.

Although we need to perform a broader study to generalize these results, we know that the structural mechanisms used by the proposed heuristic are widely used by systems based on reference architectures. We also tried to perform similar study with the 15 open source systems selected for our study on metric distributions. We sent the results obtained with the heuristic to four active developers of each system (60 requests). We sent two e-mails for each selected developer containing a short message explaining the purpose of the research and a form listing the design roles assigned to each class. However, after 90 days, only two developers replied us promising to answer the survey but we did not get any response. For this reason, we restricted this initial evaluation of the heuristic to the five governmental systems. However, we were not allowed to use them in our metric distribution study due to confidentiality reasons.

4 STUDY SETTINGS

The main *goal* of this study is to evaluate the impact of fine-grained design decisions over distribution of metric values in systems of the following domains: Web, Android and Eclipse Plugins. A high impact may suggest that we should not overlook fine-grained design decisions when building metric-based benchmarks for assessing source code quality. For this purpose, we conceived the following research questions (RQs) to guide our study.

RQ1 *Is there a significant difference in the distribution of metric values of different design roles in the same system*

RQ2 *Is there a significant difference in the distribution of metric values of the same design role across different systems of the same domain?*

RQ3 *Is there a significant difference among metric distributions of the same design role across different releases of the same system?*

Through RQ1, we aim to investigate whether design role is a design decision that affects metric distributions. With RQ2, our goal is to evaluate whether other design decisions, besides design roles, also influence the distribution of metrics values. Finally, through RQ3, we aimed to verify whether there are decisions along the releases of each system that change the design in a way that significantly impact metrics values. The goal with RQ3 is also verifying whether previous stable releases of the same system would be good candidates to compose benchmarks. In some cases, modules of a previous version, implemented or reviewed by a senior and experienced developer, for instance, may be the only source of design decisions developers consider that fit to their context. To answer these research questions we designed a study composed of three major steps described in the following subsections.

4.1 Selecting Target Systems

Firstly, we searched on GitHub and selected fifteen real-world systems developed in Java from three distinct domains: (i) Web Applications; (ii) Mobile Applications for Android platform; and (iii) Eclipse plugins. We chose three distinct domains implemented in the same language (Java) because application domain and programming language are recognized factors that impacts the distribution of metrics [53]. The authors' experience and knowledge on the selected domains was also a requirement to allow the manual analysis of the code planned for our empirical study. Moreover, the selected domains are popular in the software development industry[9] (mainly the first two) and follow well-defined reference architectures [36], which is an essential requirement to apply the heuristic proposed in Section 3.

To select the systems from GitHub we used the following search strings: "Eclipse plugin language:java", "android language:java" and "Web language:java". We ordered the resulting lists according to the number of repository forks. The goal was to select systems with as many contributions as possible. Additionally, we excluded frameworks, libraries, and systems not updated since January 2016. Frameworks and libraries are out of scope of our study because they usually follow very particular design decisions seldom shared with other systems. We also excluded systems with no release available because of RQ3. In order to select widely used Android applications, we also only considered systems with at least 1,000 reviewers and 1,000 downloads. This information is only available for Android applications at Google Play store. Then, we selected the first five systems from the resulting list of each domain. The sample corresponded to approximately 5% of the resulting list of each domain.

Table 2 summarizes the main characteristics of the fifteen selected systems. The #classes and #methods columns show the number of classes and methods in each system. The selected systems have between 12 and 282 thousand lines of code (#LOC column) and 08 and 224 contributors contributors (#contributors column). The #releases column shows the number of stable releases available in each system. The last column shows the commit date corresponding to the source code we used in our study. Finally, the #design roles column shows the number of design roles identified by our heuristic (Section 3). For instance, SMS Backup+ system has six predefined design roles (eg. *Activity* and *Persistence*), three non-predefined design roles (eg. *BroadCastReceiver*) and the *Undefined* design role, totaling 10 design roles.

Design roles are usually domain-specific. For instance, *Activity* is a design role typically found only in Android applications. Some design roles are typically found in all systems of a domain. For instance, *Fragment* and *Service* are common in Android applications.

[9]https://octoverse.github.com/

However, each system is a unique design solution and normally has some particular design roles. For instance, Exoplayer has the *Buffer* and *SimpleDecoder* design roles for manipulation of audio files. These design roles are hardly found in other Android systems. This situation illustrates why the number of design roles are distinct even among applications of the same domain.

Compared to previous works (see Section 7), our heuristic improved the number of covered classes, i.e. classes the heuristic was able to assign a design role different of the *Undefined* one. In fact, our heuristic works better for systems whose classes are structurally bound to a reference architecture. The higher is the number of classes structurally bound to the reference architecture, the smaller is the number of classes associated to the *Undefined* design role. This is the reason for the differences on the number of *Undefined* classes among the systems. For example, the Qalingo system has only 5.6% of its classes associated with the *Undefined* design role. However, the SMSBackup system has 40.5% of its classes as *Undefined.* In Section 7, we detailed our plan to conduct future studies to further reduce the number of classes assigned to the *Undefined* design role.

4.2 Design Role Identification and Metric Computation

In this step, we used our heuristic (Section 3) for identifying and automatically assigning a design role to each class of the fifteen systems. We used the DesignRoleMiner to do that as well as to compute method-level metrics. It is important to highlight that classes are grouped by design roles. As a consequence, methods are grouped by their classes' design roles. Therefore, each design role constitutes a sample of method-level metric values. Our study considered the following four metrics:

- **McCabe's Cyclomatic Complexity (CC) [35]**: It counts number of branching points of each method.
- **Number of Method Parameters (NMP) [20]**: It counts the number of parameters of each method.
- **Lines of Code (LOC) [30]**: It counts the number of executable statements of each method, excluding comments and blank lines.
- **Efferent Coupling (EC) [34]**: It counts the number of classes from which each method calls methods or accesses attributes.

We selected these method-level metrics because we can manually compute them without tool support. This criterion is essential for conducting the manual analysis planned for our study and identifying the factors that impact on the distribution of metrics. Also, these metrics are available in many tools [38] and have been successfully used for fault-proneness prediction[5, 12, 24], for instance.

4.3 Comparing Distributions of Metric Values

In this step, we compare the distribution of values of each metric according to the following configuration: (i) to answer RQ1, we compare different design roles within each system, (ii) to answer RQ2, we compare the same design role across different systems, and (iii) to answer RQ3, we compare the same design role across releases of each system.

To do that, we initially apply the Kruskal-Wallis test [45] using the 5% significance level (i.e. p-value < 0.05). Kruskal-Wallis is a non-parametric statistical test used to evaluate whether three or more samples have similar distribution of values. When the null hypothesis is rejected, the test indicates that at least one of the samples has distribution of values different to the others. However, it does not indicate what is that sample.

Therefore, if Kruskal-Wallis test rejects the null hypothesis, we additionally apply a multiple comparison procedure to identify pairs of samples with significant differences using the Mann-Whitney U test with 5% confident level and Bonferroni correction [32]. The result of this procedure is a table ordered according to the distance between samples. In this way, the first and the last rows of the table contain the most distant samples.

Finally we apply Cliff's δ [15] to quantify the importance of the difference between distribution values of pairs of samples. To avoid excessive comparisons, we only compare the most distant samples. This is enough to verify whether there are at least two groups of methods that has distinct distribution of metric values. We use Romano *et al.* [41] approach to interpret the effect size based on Cliff's δ. Supposing δ as effect size, ranging from -1 to 1, $|\delta| < 0.147$ means negligible effect, $|\delta| < 0.33$ means small effect, $|\delta| < 0.474$ means medium effect, and $|\delta| >= 0.474$ means large effect. Cohen [16] states that a small effect size is noticeably smaller than medium but not so small as to be trivial, a medium effect size represents an effect likely to be visible to the naked eye of a careful observer, while large effect is noticeably larger than medium.

Therefore, we decided to use small, medium and large effect sizes to consider that two samples of methods should be manually analyzed. Effect sizes must be judged according to the context and even small effects might be of practical importance [28].

5 RESULTS

In this section, we report and discuss the main findings of our study guided by each research question.

RQ1*: Is there a significant difference in the distribution of metric values of different design roles in the same system?*

Motivation: If design roles affect the distribution of metric values, we should consider taking them into account when building metric-based benchmarks.

Method: To examine the overall impact of design roles on each metric and system, we test the following null hypothesis.

$H0_1$*: there is no difference in the distributions of metric values among all design roles in the same system.*

For each system, we execute the steps described in Section 4.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant design roles, this means that, for the analyzed system and metric, there are at least two design roles with significantly different metric distributions. So, we can answer "yes" to RQ1. Our website provides R scripts for replication purposes.

Findings: All 60 executions of the Kruskal-Wallis test (four metrics times fifteen systems) reject the null hypothesis ($H0_1$). This means that at least one design role has distribution of values distinct from the others. In addition, when comparing the two most distant pairs of design roles, we obtained Cliff's δ corresponding to large effect size for 57 of the 60 combinations of metrics and systems (95%). For the other 3 combinations (5%), we obtained Cliff's δ

Table 2: Target Systems Summary

Domain	System	Description	#classes	#methods	#LOC	#releases	#contributors	#design roles	#Commit Date
Android Applications	Bitcoin Wallet	Bitcoin payment system	111	1407	25K	138	21	9	2016-10-07
	Exoplayer	Media player	429	4532	84K	63	41	27	2016-10-06
	K-9 Mail	Email client	488	6913	109K	344	150	23	2016-04-13
	SMS Backup+	SMS backup tool	118	921	12K	78	39	10	2016-07-08
	Talon for Twitter	Twitter client	312	3391	83K	1	8	12	2016-04-03
Eclipse Plugins	Activiti-Designer	BPMN editor	704	3768	74K	19	11	51	2016-08-18
	AngularJS Eclipse Plugin	Editor to AngularJS	106	653	12K	19	10	14	2016-07-03
	Arduino IDE for Eclipse	IDE for Arduino hardware	124	1180	24K	5	25	12	2016-04-04
	Drools and jBPM	IDE for Drools and jBPM	796	6936	107K	76	43	54	2016-12-09
	SonarLint Eclipse Plugins	Feedback about quality issues.	200	1178	18K	45	12	20	2016-12-12
Web Applications	BigBlueButton	Conferencing system for on-line learning	1537	11377	176K	20	65	56	2016-10-18
	OpenMRS	Patient-based medical record system	1066	12195	210K	115	224	37	2016-10-12
	Heritrix	Portal for web-crawling	567	4950	90K	2	22	34	2016-07-21
	Qalingo	E-commerce system	956	13836	147K	4	9	18	2016-09-25
	LibrePlan	Project management tool	1541	23278	282K	32	28	35	2016-11-09

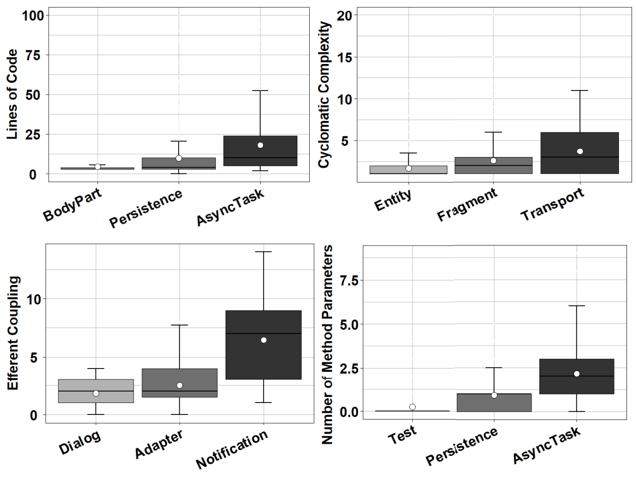

Figure 2: Distributions of Metrics of K-9 Mail System

corresponding to medium effect size. These results mean that for all systems and metrics there are significant differences between the distributions of metric values of at least two design roles.

Most of the design roles compared with Cliff's δ comprise more than 15 methods. Therefore, we consider them as representative. For the Android application domain, 85% of the design roles taken into account have from 15 to 958 methods. For the Web application domain, 84.6% have between 25 to 4088 methods. Finally, for the Eclipse Plugin domain, 79.1% have from 15 and 223 methods.

Figure 2 illustrates some differences between the distributions of values of design roles from the K-9 Mail Android application. It shows four graphs with three box plots each. Each graph is about one of the four metrics. The box plots on the ends of each graph correspond to design roles with distribution largely different from each other (large effect size). The box plot on the middle shows the distribution of values of a design role with medium or small difference to the other two (medium or small effect size.)

Regarding the LOC metric, Figure 2 shows *BodyPart*, *Persistence* and *AsyncTask* design roles. *BodyPart* involves methods ranging from 3 to 6 lines of code. *Entity* and *Exception* are other design roles (not shown in Figure 2) with similar distributions. In fact, methods from these design roles have a low number of lines of code because they are usually only responsible for encapsulating data. In the *Persistence* and *AsynkTask* design roles, methods have maximum value of 21 and 53 lines of code, respectively. In fact, classes assigned to *AsynkTask* are responsible for more complex tasks, such as automatically updating mail folders.

Figure 2 shows the *Entity*, *Fragment* and *Transport* design roles to illustrate differences regarding the CC metric. These three design roles present, respectively, 3, 6 and 11 as CC maximum value. In fact, implementing domain entities (*Entity* design role) is quite simpler than implementing message transport following security protocols (*Transport* design role). Using the same threshold for assessing methods of both design roles might lead to false negatives or false positives.

Regarding the efferent coupling metric, the maximum values obtained for the three design roles shown in Figure 2 are 4, 7, 14, respectively. Methods associated to the *Notification* design role have higher efferent coupling because they call other parts of the mobile device to notify changes in the email box state. Finally, regarding number of parameters, Figure 2 shows design roles with 0, 2 and 6 as maximum values. The methods associated to the *Test* design role, for instance, have a very low number of parameters because each method usually is responsible to test only one method. Methods associated with *AsyncTask* design role have more parameters because they implement application business logic, which requires more information as input parameters.

Another interesting point, is that we noticed a large number of outliers in the *Undefined* design role. An outlier is an observation that appears to deviate from other observations in a sample. This finding is expected because, in fact, the *Undefined* design role ends up accommodating classes for which our heuristic was not able to identify a design role using syntactic structures. Classes associated with *Undefined* design role are more likely to be developed following distinct design decisions.

In summary, the significant differences we observed on metric distributions of distinct design roles, allow us to answer RQ1 as follows:

Design roles affect the distribution of metric values. Therefore, we should consider taking design roles into account when using benchmarks for metric-based source code analysis.

Table 3: Cliff's δ Interpretation

Effect	LOC	CC	EC	NMP
Large	**02**	**01**	**09**	**09**
Medium	**12**	**05**	**11**	**05**
Small	**16**	**14**	**09**	**10**
Negligible	**02**	**04**	**03**	**03**

***RQ2**: Is there a significant difference in the distribution of metric values of the same design role across different systems of the same domain?*

Motivation: In RQ1, we found that design roles affect metric distributions. However, can we group together classes of the same design role but from different systems when building benchmarks? Or do different systems have other design decisions for the same design role that make metric distributions distinctive for different systems?

Method: To address RQ2, we test the following null hypothesis.

$H0_2$: there is no difference among distributions of metric values of the same design role across systems of the same domain.

For that, we only took into account design roles present in at least two systems of the same domain. We restrict the comparison among systems of the same domain, because systems of different domains barely have design roles in common developed with similar design decisions. We compared the distributions of the same design role in different systems. For each design role, we execute the steps described in Section 4.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant systems, this means that, for that design role and metric, there are at least two systems with significantly different metric distributions. When this occurs, we manually investigate the source code trying to identify if any design decision in particular is responsible for that difference.

Findings: For Android applications, we evaluated 10 predefined design roles (eg. *Activity* and *Service*) and one non-predefined (*BroadcastReceiver*). For the Eclipse plugins domain, we considered nine predefined design roles (eg. *Dialog* and *View*) and nine non-predefined (eg. *Plugin* and *AbstractHandler*). Finally, for the Web application domain, we analyzed 11 predefined design roles (eg. *View* and *Persistence*) and five non-predefined (eg. *Validator* and *DispatcherServlet*). Also, we examined the *Undefined* design roles for the three domains. In total, we analyzed 46 design roles (the *Undefined* design role counts three times, one for each domain). Some design roles are present on all five systems of the domain. For instance, this is the case of the *Activity* and *Persistence* design roles in the Android domain. Other design roles are present in some of the systems. For example, *BroadCastReceiver* is present in only three systems of the Android domain.

For each design role, we executed the statistical tests four times, one for each metric, totaling 192 tests (48 times 4). The null hypothesis ($H0_2$) was rejected in 115 of the 192 tests. For these cases, we applied the multiple comparison procedure and the Cliff's δ to quantify the size of the difference between the two systems with highest difference among the samples. Table 3 summarizes the Cliff's δ results. It shows, for each metric, the number of design roles per effect size found. For instance, for the EC metric, we found large effect size for nine design roles and medium effect size for eleven design roles. The full table with individual results for each design role and metric is available on our website. Then, we manually analyzed the source code of all design roles for which we found small, medium or large effect sizes. The goal was to find out which design decisions contributed to the difference between the systems. In the following subsection, we discuss the design decisions we identified. In some cases, more than one of them contribute to the difference regarding the same design role.

A. Used Libraries: We found the use of distinct libraries as one design decision that makes metric distributions of the same design role significantly different when comparing different systems. The *Persistence* design role is a clear example of this. We identified the use of distinct persistence mechanisms or libraries across systems of the three domains. This affected the distributions of the metrics in the three domains. In the Android application domain, for instance, we found medium effect size for LOC and EC and large effect size for NMP when comparing the *Persistence* design role of Bitcoin Wallet and SMS Backup+. The Bitcoin Wallet application uses both the *ContentProvider* Android native library and *SQLiteDatabase* to share and persist data, respectively. We observed that 90% of the *Persistence* methods in Bitcoin Wallet range from 3 to 29 lines of code. On the other hand, the SMS Backup+ only uses native Android libraries to open, read, and store SMS and MMS messages. This mechanism is simpler because it does not use database libraries. Then, we observed that 90% of the *Persistence* methods in SMS Backup+ range from 2 to 11 lines of code. Another example of the use of different libraries occurs with the *Test* design role. For instance, we found medium effect size for LOC and EC metrics when comparing the LibrePlan and BigBlueButton systems. The former uses libraries for implementing integration test while the latter uses libraries for unit testing. Implementing unit tests is usually simpler than implementing integration tests.

B. Coding Style: We also identified coding style as a design decision that affected the distribution of metric values. For instance, we found medium effect size for the EC and LOC metrics when comparing the *Persistence* design role of Qalingo and OpenMRS Web applications. Both systems use the Hibernate framework to implement persistence. However, developers of Qalingo decided to use the *Criteria* mechanism, a type-safe way to express queries in Hibernate. Although some methods in OpenMRS also use the *Criteria* mechanism, most methods use Hibernate Query Language (HQL), a non-type-safe way to express queries. Both mechanisms are common in systems using Hibernate, but source code using HQL usually needs fewer lines of code and uses fewer external classes. These results complement the study of Higo *et al.* [25] that reports the effect of coding style on the LOC metric.

C. Exception Handling, Logging and Debugging Code: We also observed cases in which decisions related to Exception Handling, Logging or Debugging affected metric distributions. For instance, we found medium effect size for the EC metric when comparing the *Service* design role of Bitcoin Wallet and SMS Backup+ Android applications. Methods in Bitcoin Wallet contain try-catch blocks to handle exceptions, while most of SMS Backup+ methods throw exceptions rather than handle them. The efferent coupling is higher in Bitcoin Wallet methods due to references to other classes within catch blocks.

Regarding Logging and Debugging, we found significant differences of EC and LOC when comparing the *Service* design role of K-9 Mail and SMS Backup+. This occurs due to code snippets used to log actions when the application is executed in debug mode. This is a common mechanism developers use to debug Android applications. However, in K-9 Mail developers placed this logging code in *Service* methods, while SMS Backup+ developers placed it in methods related to the *Activity* design role, causing differences on metric distributions.

In summary, the significant differences we observed on metric distributions when comparing design roles common to different systems, allow us to answer RQ2 as follows:

Design decisions related to a design role may make metric distributions for this design role different in distinct systems. Therefore, we should be aware of other design decisions, not only design roles, when building benchmarks for metric-based source code analysis.

RQ3*: Is there a significant difference among metric distributions of the same design role across different releases of the same system?*

Motivation: In RQ2, we found that different systems may have different metric distributions for the same design role due to different design decisions. This means that only considering design roles to group classes when building or using benchmarks with different systems may not be enough to have accurate metric-based source code analysis.

A possible alternative is to use previous system releases as a benchmark. Evidently, the idea is to use well designed releases or releases that underwent source code quality review. Following this idea, it is important to investigate if there are design decisions along system releases that change the design in a way that significantly affect metric values.

Method: To address RQ3, we test the following null hypothesis.

$H0_3$*: there is no difference on the distribution of metric values of the same design role across different releases of the same system.*

For that, we decided to compare the release considered in the investigation of RQ1 and RQ2 with the three most recent preceding releases. We did that for each system. We did not make an exhaustive analysis of all releases because very old releases are likely to be very different from recent ones. All releases we used in our study are available on our website. We also decided to only consider design roles with variation on the number of lines of code higher than 1% between at least two of the analyzed releases. Variations smaller than 1% of LOC hardly imply differences on the distribution of metric values. Finally, for each design role, we executed the steps described in Section 4.3 four times, one for each metric. If, using Cliff's δ, we find a large, medium or small difference between the most distant releases, this means that, for that design role and metric, there are at least two releases with significantly different metric distributions. In this case, we manually investigate the source code to identify the reasons behind the difference.

Findings: For the Android application domain, we evaluated 36 design roles. Thus, considering the four studied metrics, we executed 144 Kruskal-Wallis tests (36 times 4). The null hypothesis ($H0_3$) was only rejected in 11 tests. For these 11 pairs of design role and metric, we performed the multiple comparison procedure and calculated the effect size between the most distant releases. The effect size was negligible in 10 tests. Only EC metric for the *Persistence* design role presented small effect size. The reason was that the persistence mechanism underwent some refactoring changes along the releases, such as the extract class refactoring.

For the Eclipse Plugins, we evaluated 51 design roles. We performed 204 Kruskal-Wallis tests (51 times 4), one for each metric. The null hypothesis $H0_3$ was only rejected in 7 tests. Then, we obtained negligible effect size for five of these seven pairs of design role and metric. Only two of them presented small effect size. This occurred for the *Trackable* design role in Sonarlint and the LOC metric due to some simpler classes created from one release to the other within this design role. Similar reason affected the NOP metric for the *Undefined* design role. Some new simpler methods were also created and associated to this design role.

Finally, for Web Applications, we evaluated 59 design roles and executed the Kruskal-Wallis test for each metric, totaling 236 tests (59 times 4). The null hypothesis $H0_3$ was only rejected in 31 tests. We calculated the effect size using Cliff's δ and found six cases with small effect size and four cases with medium effect size. Three cases of small effect size involved the LOC, CC and EC metrics and the *Entity* design role in the OpenMRS system. The differences occurred because the developers decided to exchange the library for generating reports. The other three cases of small effect size involved LOC, CC and EC and the *View* design role in the OpenMRS system. In this case, the developers also decided to use a simpler library for implementing entity searches. Cases of medium effect size also occurred due to changes of libraries. For instance, developers of BigBlueButton modified the *MessageHandler* design role to change the message handling mechanism, which affects the distribution of EC from one release to the other. Also, developers of the Libreplan system decided to use a different library to deal with time, which affects the distribution of NOP of the *TimeTrackerState* design role.

In summary, we only observed very few cases of significant differences on metric distributions when comparing design roles across different releases. This allow us to answer RQ3 as follows:

Differences on the distribution of metrics across different stable releases may not be frequent. Therefore, we should consider taking previous releases into account when building benchmarks for metric-based source code analysis.

6 THREATS TO VALIDITY

This section discusses the threats to validity of our study following common guidelines [29].

Internal validity. There might be a threat associated with the correctness of the tool we used to calculate metrics and identify design roles. DesignRoleMiner extends the MetricMiner tool [47], which was already used in other studies [4, 42]. Also, we manually checked many metric values and identified design roles. Moreover, we evaluated our tool and heuristic for design role identification by means of a study with developers and Web-based governmental systems, as described in Section 3. Other possible threat is that some classes may accommodate more than one design role, but we assign only the most prominent design role according to our proposed heuristic. Although this overlapping of responsibilities is not considered adequate in object-oriented systems [11], future studies could assess the impact of having classes with multiple design roles and their effect on metric distribution.

Construction validity. There is a possible threat related to metrics we selected for our study. The selected method-level metrics cover important aspects of source code quality and are widely used for software fault prediction [14, 22]. Errors in calculating metrics may also occur [2]. However, these errors are usually small and to minimize these interferences we use the Kruskal-Wallis and Cliff's δ statistical tests.

External validity. Some of the findings might be specific to the selected software systems and domains assessed. To minimize this bias, we discussed in Section 4.1 some well-defined and replicable criteria for selecting representative systems in each application domain. Although other domains use similar mechanisms to implement the architecture, we still intend to extend this investigation other systems and domains. We also do not claim that the design decisions considered in this study are the only design decisions that impact metric distributions. However, they were the most evident in the systems involved in our study. Future studies with other systems may evidence new design decisions impacting on the distribution of metric values. So, although we are restricted to the systems and domains analyzed, this is an important step toward improving the accuracy of metric-based assessment of source code.

7 RELATED WORK

Some studies have assessed the effect of coarse-grained design decisions on the distribution of software metrics. Zhang *et al.* [53] discuss that distribution of metric values depends greatly on the context of the project. They found six context factors that affect the distribution of at least 20 metrics. Programming language, application domain, and lifespan are three most important factors impacting over distribution values of 80% of the metrics. Therefore, our work complements it as we found fine-grained design decisions that also influence the distribution of metric values.

Aniche *et al.* [3] show that architectural roles affect the distribution of metric values. For example, a class playing the architectural role Controller, in an MVC-based system, has a different distribution of metric values from other architectural roles. However, they were able to identify and associate architectural roles to only 17.5% of the classes in MVC-based systems and 10.5% of the classes in Android applications. Consequently, metric-based assessments following such approach would disregard the design roles played by the rest of the classes. We deepen this discussion by showing that other design decisions also impact on metrics. We propose a heuristic that could correctly identify the class design role of 86.2% of the 1039 analyzed classes from five governmental systems. Considering the 15 selected systems used in our study, our heuristic was able to propose design role to 62.1% from Android classes, 73.5% for Eclipse classes and 77.2% for Web Classes. Therefore, the *Undefined* design role was assigned to 37.9% from Android classes, 26.5% for Eclipse classes and 22.8% for Web ones. As future work, we plan to investigate whether integrating other techniques, such as concern mining [49], to our heuristic improves the coverage and accuracy of design role identification. Additionally, we identified that others design decisions also affect the distribution of metric values.

Budi *et al.* [13] propose a classification framework using a machine learning technique that predicts a stereotype for each class. The heuristic identifies three class stereotypes (Entity, Control, and Boundary) introduced as an extension to the standard UML [43]. Dragan *et al.* [17] extends this set of class stereotypes to C++ systems. The approach uses patterns of the method stereotype distributions at the class level. Both approaches propose predefined stereotypes used in the analysis phase. Our approach to identify the design role played by classes could be used to define stereotypes focused on the design and implementation phases. We propose to use the class hierarchy and a customizable token-based method that is not limited to a predefined set of design roles. In the future, we plan to assess whether the stereotypes benefits, related to program comprehension, design recovery, and identification of code smells could be obtained with the design role information.

8 CONCLUSION

We performed an empirical study to assess whether fine-grained design decisions affect the distribution of four method-level metrics. Our analysis was driven by the concept of design role. We consider design role itself as a design decision in the sense that the developer decide to assign a responsibility to a class in the context of a reference architecture. To support our study, we defined a heuristic to automatically identify the design role played by the classes of a system. The study involved fifteen real-world systems, from three different domains. The results and their implications for research and practice can be summarized as follow.

Design roles impact the distribution of metrics. Initially, our results showed that design roles affected the distribution of metrics (RQ1). A potential implication of this is that future researches should propose and evaluate metric-based analysis methods that take design roles into account. In fact, the major reason for the occurrence of false positive and negatives on smell detection methods is the lack of context for metric thresholds[44]. Design roles might be considered to define this context. For instance, methods that use system benchmarks to calculate metric thresholds could derive thresholds according to design roles.

Fine-grained design decisions impact the distribution of metrics. Our results also showed that, due to different design decisions (for instance, coding style or used libraries) the same design role might have different metric distributions on different systems (RQ2). A potential implication of this is that we should select systems with similar design decisions when building benchmarks for metric-based source code analysis. In addition, our results showed that differences on metric distributions across different releases were not frequent (RQ3). A practical implication of this is that companies should consider building their benchmarks from system releases that underwent design quality reviews.

In this context, as future work, we plan to investigate if deriving thresholds based on design roles and previous releases would improve the accuracy for detecting source code elements with design problems. Also, we suggest extending this study with more systems, programming languages, metrics and domains.

ACKNOWLEDGMENTS

This work was supported by CNPq (grant 312153/2016-3).

REFERENCES

[1] Deepak Alur, John Crupi, and Dan Malks. 2003. *Core J2EE Patterns.* 650 pages.

[2] Tiago L. Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10).* IEEE Computer Society, Washington, DC, USA, 1–10.

[3] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie Van Deursen, and Marco Aurélio Gerosa. 2016. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM).* Raleigh, North Carolina.

[4] M. F. Aniche. 2015. Detection strategies of smells in web software development. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 598–601.

[5] F Arcelli Fontana, V Ferme, A Marino, B Walter, and P Martenka. 2013. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *29 IEEE International Conference on Software Mainteinance.* IEEE, 260–269.

[6] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics.* IEEE, 44–53.

[7] Roberta Arcoverde, Isela Macia, Alessandro Garcia, and Arndt von Staa. 2012. Automatically Detecting Architecturally-relevant Code Anomalies. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE '12).* IEEE Press, Piscataway, NJ, USA, 90–91.

[8] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, Piscataway, NJ, USA, 931–940.

[9] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[10] Christian Bauer, Gavin King, and Gary Gregory. 2015. *Java Persistence with Hibernate, Second Edition.* Manning Publications. 608 pages.

[11] Grady Booch. 1986. Object-oriented development. *IEEE transactions on Software Engineering* 2 (1986), 211–221.

[12] Alexandre Boucher and Mourad Badri. 2018. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology* 96 (2018), 38 – 67.

[13] Aditya Budi, David Lo, Lingxiao Jiang, Shaowei Wang, et al. 2011. Automated Detection of Likely Design Flaws in Layered Architectures. In *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE).*

[14] Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. *Expert systems with applications* 36, 4 (2009), 7346–7354.

[15] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.

[16] Jacob Cohen. 1992. A power primer. *Psychological bulletin* 112, 1 (1992), 155.

[17] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2010. Automatic identification of class stereotypes. In *Software Maintenance (ICSM), 2010 IEEE International Conference on.* IEEE, 1–10.

[18] Marc Eaddy, Alfred Aho, and Gail C. Murphy. 2007. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07).* IEEE Computer Society, Washington, DC, USA.

[19] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. 2012. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software* 85, 2 (feb 2012), 244–257.

[20] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, Boston, MA, USA.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming.* Springer, 406–431.

[22] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement.* ACM, 171–180.

[23] Joseph Yossi Gil and Gal Lalouche. 2016. When do Software Complexity Metrics Mean Nothing?-When Examined out of Context. *Journal of Object Technology* 15, 1 (2016), 2–1.

[24] Yossi Gil and Gal Lalouche. 2017. On the correlation between size and metric validity. *Empirical Software Engineering* 22, 5 (01 Oct 2017), 2585–2611.

[25] Y. Higo and S. Kusumoto. 2017. Flattening Code for Metrics Measurement and Analysis. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 494–498.

[26] Mario Hozano, Henrique Ferreira, Italo Silva, Baldoino Fonseca, and Evandro Costa. 2015. Using Developers' Feedback to Improve Code Smell Detection. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15).* ACM, New York, NY, USA, 1661–1663.

[27] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. 2018. Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology* 93 (2018), 130 – 146.

[28] Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11 (2007), 1073–1086.

[29] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. 2006. Evaluating Guidelines for Empirical Software Engineering Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06).* ACM, New York, NY, USA, 38–47.

[30] Michele Lanza and Radu Marinescu. 2006. *Object-Oriented Metrics in Practice.* Springer Berlin Heidelberg, Berlin, Heidelberg. 205 pages.

[31] Luigi Lavazza and Sandro Morasca. 2016. An Empirical Evaluation of Distribution-based Thresholds for Internal Software Measures. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering.* ACM, New York, NY, USA, Article 6, 10 pages.

[32] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[33] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of IEEE International Conference on Software Maintenance.*

[34] Robert Cecil Martin. 1995. *Designing object-oriented C++ applications.* Prentice Hall.

[35] Thomas J. McCabe. 1976. A Complexity Measure. In *Proceedings of the 2Nd International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, USA.

[36] Nenad Medvidovic and Richard N. Taylor. 2010. Software Architecture: Foundations, Theory, and Practice. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10).* ACM, New York, NY, USA, 471–472.

[37] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code anomalies flock together. *Proceedings of the 38th International Conference on Software Engineering - ICSE '16* (2016), 440–451.

[38] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (06 Oct 2017), 7.

[39] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on.* IEEE, 101–110.

[40] Dirk Riehle and Thomas Gross. 1998. Role Model Based Framework Design and Integration. (1998), 117–133.

[41] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research.* Citeseer.

[42] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy. 2016. Comparing Repositories Visually with RepoGrams. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR).* 109–120.

[43] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *The Unified modeling language reference manual.* Pearson Higher Education.

[44] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173.

[45] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.

[46] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (aug 2013), 1144–1156.

[47] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, Marco Gerosa, et al. 2013. MetricMiner: Supporting researchers in mining software repositories. In *Source Code Analysis and Manipulation (SCAM).* IEEE, 142–146.

[48] Gustavo Andrade Do Vale and Eduardo Magno Lages Figueiredo. 2015. A Method to Derive Metric Thresholds for Software Product Lines. In *29th Brazilian Symposium on Software Engineering.* 110–119.

[49] S. Wang, D. Lo, Z. Xing, and L. Jiang. 2011. Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel. In *2011 18th Working Conference on Reverse Engineering.* 92–96.

[50] Rebecca Wirfs-Brock and Alan McKean. 2003. *Object design: roles, responsibilities, and collaborations.* Addison-Wesley Professional.

[51] Aiko Yamashita. 2013. How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study. In *2013 IEEE International Conference on Software Maintenance.* IEEE, 566–571.

[52] Daoqi Yang. 2010. *Java Persistence with JPA.* Outskirts Press.

[53] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan. 2013. How Does Context Affect the Distribution of Software Maintainability Metrics?. In *IEEE International Conference on Software Maintenance*. 350–359.
[54] H. Zhu and M. Zhou. 2008. Roles in Information Systems: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38, 3 (May 2008), 377–396.