SATT: Tailoring Code Metric Thresholds for Different Software Architectures

Maurício Aniche[1,3], Christoph Treude[2], Andy Zaidman[1], Arie van Deursen[1], Marco Aurélio Gerosa[3]
{m.f.aniche,a.e.zaidman,arie.vandeursen}@tudelft.nl, christoph.treude@adelaide.edu.au, gerosa@ime.usp.br
[1]Delft University of Technology - The Netherlands
[2]University of Adelaide - Australia
[3]University of São Paulo - Brazil

***Abstract*—Code metric analysis is a well-known approach for assessing the quality of a software system. However, current tools and techniques do not take the system architecture (e.g., MVC, Android) into account. This means that all classes are assessed similarly, regardless of their specific responsibilities. In this paper, we propose SATT (Software Architecture Tailored Thresholds), an approach that detects whether an architectural role is considerably different from others in the system in terms of code metrics, and provides a specific threshold for that role. We evaluated our approach on 2 different architectures (MVC and Android) in more than 400 projects. We also interviewed 6 experts in order to explain why some architectural roles are different from others. Our results shows that SATT can overcome issues that traditional approaches have, especially when some architectural role presents very different metric values than others.**

I. Introduction

The usage of tools such as PMD[1] and Sonar[2] to spot problematic pieces of code in software systems has become popular among software developers. Internally, these tools decide whether a class is problematic after performing a code metric analysis. Indeed, code metrics have been proven useful to assess the quality of object-oriented design [1], [2], [3], [4], as well as to code smell detection [5], [6].

These tools commonly rely on thresholds to distinguish between “good" and “bad” classes, *i.e.*, if a class has a metric value higher than the threshold, the tool reports that the class may have a problem. These thresholds are usually calculated after extensive analysis of many software systems, similar to benchmarking [7], [8]. Indeed, defining a good threshold is not a straightforward task [9], [10], [11], [12], and different approaches have been proposed [13], [8], [14], [15] over the years.

However, the impact of the system architecture on class level metrics is unclear. In practice, software developers rely on well-known architectural styles and design patterns [16], [17], [18] as common building blocks. These building blocks are typically composed of a number of classes, each carrying a specific *architectural role*. As an example, suppose a system follows the Model-View-Controller (MVC) pattern [19]: classes that play the CONTROLLER role are responsible for coordinating the process, while classes that play the MODEL role represent business concepts.

Thus, as some classes have specific responsibilities, we can raise interesting questions, such as “Do developers deal with coupling in CONTROLLERS the same way as they do in MODELS?” or “Do developers deal with complexity in CONTROLLERS the same way as they do in MODELS?”. If the answer to these questions is “no", then there might be room for improvement, as tools and techniques commonly use the entire distribution of the code metric values, no matter the architectural role a class plays in the system, as if all classes were equal to each other.

Adding context to code metrics is an emerging topic among researchers. Zhang *et al.* [20] showed that metric values can be affected by factors, such as programming language, age and lifespan. Recently, Gil and Lalouche [21] argued that metric values vary among projects, and “they mean nothing when examined out of their context”. Bouwers *et al.* [22] also warned the community about using a metric without a proper interpretation, or, as they call, the “metric in a bubble” pitfall.

To that end, we propose SATT (Software Architecture Tailored Thresholds). Our approach detects whether an architectural role is considerably different from others in the system in terms of code metric values, and provides a specific threshold that should be used for that role. We evaluate our approach by 1) applying it to two different software architectures (MVC and Android), 2) comparing the outcomes to one state-of-art approach, and 3) seeking an explanation from industrial experts in these architectures.

II. Motivational Example

SSP, or the Student Success Plan[3], is a tool that allows students to create plans for the completion of their academic goals. It is an open source project hosted on GitHub, with more than 4,000 commits, 26 releases, and 18 contributors. The application makes use of the Spring MVC framework, which means it contains CONTROLLERS to handle the requests, ENTITIES to represent the business model, SERVICES to implement business rules, and REPOSITORIES responsible to persist the data in a database.

[1]http://www.sonarqube.org/.
[2]http://pmd.github.io.
[3]http://www.studentsuccessplan.org/.

A concept that exists in SSP is "Plan". Users create study plans. The class *PlanController* is responsible for coordinating actions such as exhibiting, creating, deleting, and printing the study plan. It has 48 dependencies and 25 methods. However, these methods are not complex in terms of branches (*i.e. ifs* and *fors*), as they mostly coordinate the process between the data from the user interface to the service layer, as expected in a CONTROLLER class. On the other hand, the class *PlanServiceImpl* is responsible for managing business rules. It contains 29 dependencies and 16 methods in which nested conditionals and loops are found. The *PlanDAO* (a Data Access Object [23]) class, responsible for the integration with the database, is also different. It is coupled to 20 classes, 7 of which are simple Data Transfer Objects [23]. The number of branch instructions is not high. The *Plan* entity class is also lowly coupled; it depends on 15 classes, of which 3 belong to the system (others are related to annotations required by the persistence framework). It contains attributes that are mapped to the database, and the majority of its methods are getters and setters.

Interestingly, these characteristics are similar in all other classes that play CONTROLLERS, SERVICES, REPOSITORIES, and ENTITIES in the system. For example, CONTROLLERS are, on average, more coupled than the rest of the system. The median of the number of dependencies in an SSP CONTROLLER is 17. REPOSITORIES are different; their median number of dependencies is 7 and the third quartile is at 8. These architectural roles are also different in terms of complexity: the third quartile of McCabe's complexity [24] in all ENTITIES is 17, while for REPOSITORIES the same number is 7.

As we saw in the example, some architectural roles can present considerably different metric values. Towards this issue, we propose SATT.

III. BACKGROUND

In this Section, we present the code metrics and the architectural roles in the two system architectures (MVC and Android) that we analyze in this study.

A. Code Metrics

We rely on the Chidamber & Kemerer (CK) metrics suite [25], as (i) it covers different aspects of object-oriented programming, such as coupling (CBO, RFC), cohesion (LCOM), and complexity (WMC, NOM), (ii) it has already proven its usefulness in earlier studies [1], [2], [26], (iii) both studied architectures are object-oriented. The class level metrics we used from the CK suite are the following:

- **Number of Methods (NOM)**. Number of methods in a class.
- **Weighted Methods Per Class (WMC)**. Sum of McCabe's cyclomatic complexity [24] for each method in the class.
- **Coupling Between Object Classes (CBO)**. The number of classes a class depends upon. It counts classes used from both external libraries as well as classes from the project.
- **Response for a Class (RFC)**. It is the count of all method invocations that happen in a class.
- **Lack of Cohesion of Methods (LCOM)**. Number of method pairs whose similarity in terms of used attributes is zero minus the count of method pairs whose same similarity is not zero.

Each metric has its own scale, varying from 0 or 1 to infinite. The higher the metric value, the larger is the presence of the measured effect.

B. State-of-the-Art Benchmarking Approach

In this section, we present Alves *et al.*'s [8] work, a state-of-art benchmarking technique, which our approach was based on. We chose this one as it (i) assumes the non-normality of the metric values distribution, (ii) uses LOC as a weight function, which emphasizes the metric variability, (iii) separates the thresholds into different risk categories.

The authors proposed a benchmarking approach that uses a weighted function. First, the approach extracts code metric values from a set of different systems. This extraction can be done at method or class level (which they generically call *entity*). Then, for each entity, the approach calculates its weight (lines of code). After ordering the entities by their weights in ascending way, the approach selects as thresholds the code metric values relative to the 70%, 80%, and 90% percentiles of the accumulated weight. In other words, classes in which metric values range in the 70%-80% percentiles have "moderate risk", while from 80%-90% the risk is "high", and "very high" between 90%-100%.

As we can see, their approach derives a unique threshold for the code metric that is studied. As a consequence, all classes will be assessed using this threshold. In Section IV, we explain our approach, which can provide different thresholds for different architectural roles.

C. System Architecture and Architectural Roles

We define "architectural role" as a particular role that classes can play in a system architecture. When a class plays an architectural role in the system, its task is well-defined, and usually classes are focused only on that. As an example, CONTROLLERS in Spring MVC applications coordinate the flow between the user interface and the domain layer. One can note the difference between architectural roles and design patterns: while some design patterns can be optional in the system, architectural roles are fundamental to that system architecture, *e.g.*, an MVC-based architecture requires the existence of CONTROLLERS, while a Strategy design pattern [18] can be optionally applied in the system.

We chose the Spring MVC and Android application architectures, as both require software engineers to use classes with specific architectural roles in their applications. We had several reasons to select Spring MVC and Android: (i) they have well-defined architectural roles, (ii) they are frequently used (in a survey with more than 2,000 respondents [27], Spring MVC was used by 40% of developers that use a web framework; in July 2015, there were 1.6 million applications in the Google

Play Store, the official Android application repository), (iii) their domains are different (web vs. native mobile).

1) Spring MVC-based application architecture: Spring MVC is a Java framework that supports developers in building web applications. A Spring application must have classes playing different architectural roles [28]:

- **Controller**: Control of the flow between the domain layer and the view layer.
- **Service**: Offer an operation that stands alone in the model, with no encapsulated state.
- **Repository**: Encapsulate persistence, retrieval, by emulating a collection of objects.
- **Entity**: Represent a lightweight persistence domain object. They may or may not contain business rules.
- **Component**: Represent some isolated component in the application. Practical examples are UI formatting or data conversion. REPOSITORIES and SERVICES are a special kind of component.

2) Android-based application architecture: Android is a rich application framework that allows developers to build apps and games for mobile devices in a Java language environment. In the following, we describe 3 of the main roles when developing applications in Android.

- **Activity**: Provide a screen with which users interact in order to do something, such as dial a number, take a photo, send an email, or view a map.
- **Fragment**: Represent a behavior or a portion of a user interface in an Activity. It can be reused in many different ACTIVITIES.
- **AsyncTask**: Perform background operations and publish results to the UI thread without having to manipulate threads and/or handlers.

IV. THE SATT APPROACH

The SATT (**S**oftware **A**rchitecture **T**ailored **T**hresholds) approach derives a threshold for an architectural role when its code metric values distribution is considerably different from the distribution of other classes in the system.

According to Alves *et al.* [8], a benchmarking technique should present three characteristics, which ours follow: 1) it should be driven by the empirical data instead of experts' opinion, 2) should be robust to the distribution of the code metric values, and 3) should be repeated, transparent, and straightforward.

In the following, we present our approach step-by-step. Suppose we want to define the McCabe threshold for CONTROLLER classes in MVC systems. The approach can be repeated for any other architectural role and code metric.

1) **Dataset creation.** We select systems that follow the analyzed architecture, *e.g.,* Spring MVC applications. We perform this step only once and use the same benchmark to calculate the thresholds for all other architectural roles.
2) **Architectural roles extraction.** We identify each class' architectural role in the benchmark. In case of Spring MVC, CONTROLLER classes are always annotated with `@Controller`.
3) **Metrics calculation.** We calculate code metrics for all classes in the benchmark, regardless of their architectural role. In this example, the McCabe number of all classes.
4) **Statistical measurement.** We perform a statistical test to measure the difference between the code metric values in that architectural role (group 1) and the other classes (group 2). As metric values distributions tend not to follow a normal distribution (discussed in Section VII), we suggest the use of non-paired Wilcoxon test and Cliff's Delta between the two groups. Bonferroni correction should be applied, as the approach is performed for all combinations of architectural roles and code metrics.
5) **Analysis of the statistical tests.** If the difference is significant and the effect size ranges from medium to large, we continue the approach. Otherwise, we stop. We use Romano *et al.*'s [29] classification to describe the effect. Supposing D as the effect size, ranging from -1 to 1, $|D|<0.147$ means "negligible effect", $|D|<0.33$ means "small effect", $|D|<0.474$ means "medium effect", and $|D|>=0.474$ means "large effect".
6) **Weight ratio calculation.** From now on, we only look to the classes of the analyzed architectural role. Following the original approach, we use lines of code (LOC) as a weight of all classes. Thus, we calculate LOC for all classes and normalize it for all classes that belong to that architectural role in the benchmark. Normalization ensures that the sum of all weights will be 100%. In the example, suppose that our benchmark contains 100,000 lines of code in CONTROLLER classes, and a class A with 100 lines of code. Thus, A's weight is 0.001.
7) **Weight ratio aggregation.** We order classes according to their metric values in an ascending way. For each class, we aggregate the weights by summing up all the weights from classes that have smaller metric values, *i.e.*, classes that are above the current class.
8) **Thresholds derivation.** We extract the code metric value from the class that has its weight aggregation closest to 70% (moderate), 80% (high), and 90% (very high).

V. ANALYSIS OF THE APPROACH

In order to analyze the proposed approach, we answer the following research questions:

RQ_1. What differences in metric values distributions does SATT find for common architectural styles such as MVC and Android? First, it is important to determine whether differences between architectural roles in terms of code metrics are significant, *e.g.*, if a CONTROLLER presents a similar metric values distribution of any other classes, then, we would not need a specific threshold.

RQ_2. Can the differences in distributions thus found be explained from the architectural constraints imposed on classes fullfilling dedicated architectural roles? As we will see in RQ_1, some architectural roles do differ from others in terms of code metric values. In this RQ, we provide explanations on why these differences happen.

RQ$_3$. What impact do these differences have on the use of thresholds for quality assessments and smell detection? In this RQ, we compare and explain the differences in the outcomes of both the state-of-the-art benchmarking approach and SATT.

To answer these questions, we conducted a case study in two different software architectures (MVC and Android). To that end, we collected 120 Spring MVC and 301 Android systems in Github and performed both Alves *et al.*'s [8] and our approach. We also relied on a qualitative analysis of interviews with 6 different experts in both architectures.

A. Data Collection

To select Spring MVC and Android projects, we made use of BOA [30], a domain-specific language and infrastructure that eases mining software repositories and currently contains extensive data from Github. Using its DSL, we developed a query[4] that specifies that: *(i)* the project should have more than 500 commits in its history, *(ii)* the project should contain at least 10 classes with architectural roles. Although the constants 500 and 10 were chosen by convenience, we conjecture that they filter out pet projects and small experiments that developers store on GitHub. We also manually inspected the sample to make sure they are stand-alone systems. We eliminated the ones that were part of Spring or Android itself or were libraries.

To determine the architectural role for classes in Spring MVC applications, we analysed their annotations. If a class contains one of the following annotations, we consider that class as playing that role. The name of the annotation matches with the name of the architectural role: @CONTROLLER, @SERVICE, @ENTITY, @REPOSITORY, and @COMPONENT. Android applications make use of inheritance to determine the roles. Thus, we applied the same idea. If the class inherits from one of following classes (or its sub-classes), we consider that class to play a specific role: ASYNCTASK, ACTIVITY, and FRAGMENT. In both architectures, developers are required to follow these conventions. If they do not, that class may not work as expected in the system. Other classes in the system were considered "unindentified".

We obtained 120 Spring MVC projects and 301 Android projects. Together, they have more than 127,000 classes with 14 million lines of code. In Table II, we describe the numbers of each analysed architectural role, as well as the median of the number of classes in each role per project. Full data and scripts used in this study can be found in our online appendix [31].

B. RQ$_1$. *What differences in metric values distributions does SATT find for common architectural styles such as MVC and Android?*

1) Method: We performed the SATT approach in both Spring MVC and Android systems until step 4 (which compares whether architectural roles present different metric values distributions when compared to other classes). All the source code and analysis scripts that we used are open source and available for inspection [31].

[4]Job IDs in BOA: 11947 and 14071.

Table II: Descriptive numbers of the sample

	Total classes	**Avg classes per proj**	**Total SLOC**	**Median class size**
Spring MVC				
Controller	3,126	20	365,274	79
Repository	1,325	14	105,842	46
Service	2,845	16	326,778	59
Entity	1,666	20	169,838	78
Component	2,167	12	158,975	43
Others	52,397	269	3,654,035	39
Android				
Activity	7,455	13	1,036,645	95
AsyncTask	381	2	38.680	69
Fragment	2,004	7	307,074	108
Others	83,542	142	8,285,671	51

We used a significance level of 95% and applied Bonferroni correction for each system architecture. In Spring MVC, we performed our approach in 25 combinations (5 architectural roles times 5 metrics). In Android, we performed it 15 times (3 architectural roles times 5 metrics). Thus, we adjusted the p-values to 0.002 and 0.003, respectively.

2) Findings: In step 3, our approach checks whether code metric values distributions are different among architectural roles. In Table I, we show both the Wilcoxon test and Cliff's Delta for all architectural roles and code metrics. Stars represent the result of the statistical test, numbers are the measured effect size, and grey cells highlight medium and large effect sizes. We also analyzed the boxplots of the distribution of metric values for each architectural role. Due to space constraints, both boxplots and quantile plots of distributions can be seen in our online appendix [31].

In Spring MVC, 24 out of the 25 comparisons were statistically different, while 8 of them had an effect size from medium to large. Only NOM in CONTROLLERS did not present a statistically significant difference. In Android, 13 out of 15 comparisons show statistically significant differences, 6 of which had medium to large effect sizes. Only NOM was not significantly different in ACTIVITY and ASYNCTASKS.

We now discuss the results for each metric in detail:

CBO. This coupling metric presents medium and large effect sizes in almost all architectural roles, with the exception of REPOSITORIES and ASYNCTASKS, which present small effect size. We highlight CONTROLLERS, which have a large effect size, and in the boxplot, we can see that their median is higher than that of other classes. The same happens in Android, as FRAGMENTS and ACTIVITIES have higher medians than others. This indicates that classes fulfilling architecural roles tend to be more coupled than other classes.

LCOM. In Spring MVC, we see that effect size for lack of cohesion in ENTITIES is large. When we observe the boxplot, we notice that their median is higher than other classes. Thus, ENTITIES are less cohesive than regular classes, which makes sense especially as they usually contain a larger number of

Table I: Cliff's Delta effect size of the comparison between architectural roles and population in Spring MVC and Android. $*$ significant difference according to Wilcoxon test, highlighted cells = medium or large effect size.

	CBO		LCOM		NOM		RFC		WMC	
Spring MVC										
Controller	0.6591	*	-0.1279	*	-		0.3540	*	0.0925	*
Repository	0.2862	*	-0.1279	*	-0.0864	*	0.0823	*	-0.0791	*
Service	0.4538	*	-0.1059	*	0.0351	**	0.3047	*	0.0972	*
Entity	0.4945	*	0.5769	*	0.5600	*	-0.3969	*	0.3088	*
Component	0.3738	*	-0.1946	*	-0.2157	*	0.205	*	-0.1078	*
Android										
Activity	0.4983	*	0.1246	*	-		0.3769	*	0.0900	*
AsyncTask	0.3181	*	-0.1658	*	-		0.1782	*	0.1504	*
Fragment	0.6136	*	0.2598	*	0.1709	*	0.3942	*	0.1933	*

fields. In Android, although FRAGMENTS have larger effect size in LCOM than others, we can see in the boxplot that they are similar. As the effect sizes are negligible to small, we can state that Android's architectural roles are not different in terms of cohesion from other classes.

NOM. In Spring MVC, ENTITIES present a large effect size when compared to other classes. In the boxplot, we notice that their median, as well as their first and third quartiles, are higher than those of other classes. This can be explained by the fact that ENTITIES commonly not only contain methods with business logic, but also getters and setters for most of their fields, and therefore have a larger total number of methods. In Android, only FRAGMENTS present a significant difference, but the effect size is small. As we can see in the boxplots, although the median is similar in all Android architectural roles, FRAGMENTS have a higher third quartile than other roles.

RFC. In terms of number of method calls, we see that CONTROLLERS present a medium positive effect size, while ENTITIES present a medium negative effect size. In the boxplot, we see that CONTROLLERS and SERVICES have medians above the others, while ENTITIES have a median close to zero. This indicates that CONTROLLERS and SERVICES perform more method invocations than others. Their responsibilities in the system serve as argument to justify these numbers. As part of their role in the system, CONTROLLERS need to deal with different classes from the Model and the View layers, while SERVICES can interact with many different business objects in order to provide the logic for an entire business process. In Android, ACTIVITIES and FRAGMENTS have medium effect size when compared to others. From the boxplot, we see their median is higher than others. As part of their role, they commonly have to interact and configure a reasonable number of different UI objects, which can explain their higher values.

WMC. In terms of complexity, all the effect sizes are negligible or small with statistical significance. When analyzing the boxplot, we observe that medians are similar among all roles. It indicates that classes with specific architectural roles are similar in complexity to other classes in the system.

RQ_1: The approach indeed finds architectural roles that present significantly higher (or lower) values for certain metrics when compared to other classes.

C. RQ_2: Why are architectural roles different from each other in terms of code metric values distribution?

In RQ_1, we saw that differences in code metric values distribution among architectural roles can be large. In this RQ, we provide insights on why these differences happen.

1) Method: We conducted semi-structured interviews with 3 Spring MVC (S1-S3) and 3 Android experts (A1-A3). The goal was to help us interpret, refute, or corroborate the results from RQ_1. We decided to make use of experts, as (i) they can perceive the structure of a problem or situation better than novices [32], (ii) we conjectured they are experienced and mature enough to disagree with the data and explain their reasons, reducing possible response bias. Besides many years of experience in software development (which ranges from 6 to 21 years), they give consultancy and training about the topic to different companies (S1, S2, S3, A1, A2, A3), write books (S1, A1), send patches (A2), and even participate in the core team of the framework (S3). We conjecture these skills are good indicators to classify them as experts.

As a main artefact for the interview, we created a visual chart of the effect sizes in Table I. We navigated through each data point with the experts, and asked them to reason about why that specific architectural role would present that difference in terms of that code metric. Before discussing the metric values with the Android and Spring MVC experts, we explained the details of the CK metrics used to them. To reduce response bias, we made sure to not mention any personal opinion when experts asked about it during the interviews. We also told them that not finding an explanation or to disagree with the data would not be a problem.

The full protocol is available in the online appendix [31]. In the following, we show the main part of the interview, which we repeated for each code metric:

1) We: Read aloud the effect size of each architectural role for metric X.
2) Q: Were you expecting this difference among architectural roles?

Table III: Summary of the experts' opinions.
(✓) experts endorsed the results in RQ$_1$, (number) experts diverged.

	CBO	LCOM	NOM	RFC	WMC
Spring MVC					
Controller	✓			✓	
Repository					
Service	✓				
Entity	(1)	✓	✓	✓	
Component	(2)				
Android					
Activity	✓			✓	
AsyncTask					
Fragment	(3)			✓	

3) Q: Can you explain why this happens for each architectural role?

All the interviews but one were conducted via Skype. Interviews lasted for at least 40 minutes each in Spring MVC and 30 minutes in Android, and were fully recorded. After each code metric, the first author wrote a summary of the expert's opinion, and reviewed with him/her before going to the next metric. To analyze the qualitative data, we used both the written summary and the recorded audio.

2) Findings: Experts endorsed and explained 9 out of the 12 differences in which the effect size was medium or large in RQ1. For the other three, CBO metric in ENTITIES, COMPONENTS and FRAGMENTS, developers were expecting these roles to have smaller differences when compared to other classes. In Table III, we present their opinions. In the following paragraphs, we present their main thoughts that endorsed the medium and large effect sizes in RQ1 as well as their divergences with the results.

Endorsements. For Spring MVC, experts explained that *(i)* CONTROLLERS and SERVICES are usually more coupled than other classes. S1 and S3 expected CONTROLLERS to be highly coupled to the framework itself and to third party libraries, while SERVICES are expected to be coupled to many classes from the system, *(ii)* ENTITIES typically contain many attributes representing a business concept, which implies the existence of many getters and setters, which makes LCOM increase, *(iii)* CONTROLLERS make several invocations to all their dependencies, both from the system, such as to ENTITIES, REPOSITORIES or SERVICES, as well as to the framework API, to deal with the view layer, or perform validation, which increases RFC, *(iv)* ENTITIES usually encapsulate attributes and their related behaviors, and as a consequence, there are not many method invocations to other class dependencies, which decreases their RFC values.

Regarding Android, experts *(i)* all expected ACTIVITIES and FRAGMENTS to be more coupled than other classes. According to them, both roles are responsible for dealing with all graphical user interface (GUI) components and for controlling the flow between the GUI and the logic, which makes their code highly coupled, *(ii)* were also expecting both to have a high RFC. According to them, both roles need to deal with a large number of GUI components, which requires many invocations to these dependencies,

Diverging opinions. In all occurences in which experts diverged about some of the results in RQ1, their reasoning was related to a possible lack of adherence of that architectural role to a good practice. In the following, we present experts' thoughts on the divergences.

In Spring MVC, *(1)* experts were surprised that ENTITIES have high coupling. S1 and S2 said that they should be only coupled to other ENTITIES, *(2)* experts were all surprised by how high CBO was for COMPONENTS. According to them, a COMPONENT should only do a single task and, because of that, be less coupled. S1 said COMPONENTS can be used for external API integration, which would increase coupling, but they indeed are simple in most cases.

In Android, *(3)* A3 was not expecting FRAGMENTS to have larger CBO when compared to ACTIVITIES. All of them emphasized that ACTIVITIES and FRAGMENTS have their similarities in practice, as both manage interface components and their relationships. However, as an ACTIVITY can be comprised by many FRAGMENTS, all experts expected FRAGMENTS to be smaller in terms of all code metrics when compared to ACTIVITIES.

Other concerns. In Spring MVC, S1 and S2 mentioned two different approches developers usually take: having rich models [33], which means the business rules are in ENTITIES, leaving SERVICES to encapsulate rules that do not belong to a single ENTITY, or having ENTITIES only as data holders and persistence, while storing all business logic in SERVICES. S3 (member of the Spring development team) explained that the framework lets developers decide the approach they want to take. By looking at the data, all experts agreed that the second approach seems to be the most popular one, as SERVICES tend to be the most complex and coupled role in these projects. If the first approach is taken, they suggest developers to keep ENTITIES easy to maintain; if the second one is taken, they suggest the same for SERVICES.

In Android, A3 was concerned about the how large the difference between ACTIVITIES and FRAGMENTS are when compared to other classes. According to him, that may be explained by the lack of good practices. A1 and A2 also had similar thoughts. According to A1 and A3, it is easy to write a single complex ACTIVITY; however, developers should break it into many small ACTIVITIES or FRAGMENTS. A2 said that, although the Android architecture itself enforces high coupling, developers should better separate responsibilities in their classes. Similar to Spring experts, A1 also suggested the use of rich models for mobile applications, and to avoid business rules in ACTIVITIES.

RQ$_2$: Experts considered most of the differences in metric values for architectural roles to be coherent. Their key explanation is that architectural roles have specific *responsibilities. When experts did not endorse the metric values, their explanation was related to a possible lack of adherence to a good practice.*

D. RQ_3. What impact do these differences have on the use of thresholds for quality assessments and smell detection?

1) Method: After analysing the results of the statistical tests in step 4, the SATT approach continues for the pairs *Controller/CBO, Service/CBO, Entity/CBO, Component/CBO, Entity/LCOM, Entity/NOM, Controller/RFC, Entity/RFC* in Spring MVC, and *Activity/CBO, Fragment/CBO, Activity/RFC, Fragment/RFC* in Android.

We also performed the state-of-the-art approach for both architectural systems. This approach does not take architectural roles into consideration and, thus, it produces a single threshold value for each code metric.

With both thresholds in hand, we compared the state-of-the-art threshold in cases in which the difference was significant in RQ_1.

2) Findings: In Table IV, we present "moderate risk" thresholds calculated by both the state-of-the-art (Alves *et al.*'s) and our approach. Due to space restrictions, we provide high and very high thresholds in our appendix [31]. Also, for each pair of architectural role and code metric, we present the percentile in which the state-of-the-art threshold relies in that architectural role's metric values distribution.

We provide a few insights from these results:

1) The state-of-the-art CBO threshold is 16. However, as we saw, CONTROLLERS present higher CBO values when compared to other classes. As expected, we see that the state-of-the-art threshold lies in the 0.35 percentile of the CONTROLLERS' distribution. It means that 65% of all CONTROLLERS in our benchmark would be classified as "moderate risk". Similar effect happens with SERVICES, ACTIVITIES, and FRAGMENTS.
2) ENTITIES' threshold for LCOM is much higher than the state-of-the-art one (147 vs 440). As a consequence, it would consider more than half of ENTITIES as having moderated risk.
3) ENTITIES' threshold for CBO is similar to the state-of-the-art one. However, this role is different from the others in the "right side of the tail". While the state-of-art very high threshold for CBO is 32, for ENTITIES it is 25 (data in the appendix [31]). Thus, although the state-of-the-art moderate threshold would assess classes in a similar way than our threshold, the very high threshold lies in the 96% percentile of the distribution, which is higher than expected.
4) State-of-the-art RFC threshold is much higher than ENTITIES' specific threshold. As we can see, the number 48 lies in the 96% percentile of the role's specific distribution. Thus, an ENTITY only appears in the RFC assessment if it compares to the 4% worst classes of the benchmark.

Table IV: Results of the SATT approach in Spring MVC and Android systems compared to Alves et al.'s approach. We present moderate thresholds. High and very high results can be found in our online appendix.

	CBO	**LCOM**	**NOM**	**RFC**	**WMC**
Spring MVC					
Alves et al.'s threshold	16	147	23	48	65
Controller					
Percentile	0.35	-	-	0.60	-
SATT threshold	26	-	-	62	-
Service					
Percentile	0.43	-	-	-	-
SATT threshold	27	-	-	-	-
Entity					
Percentile	0.67	0.48	0.52	0.96	-
SATT threshold	16	440	33	8	-
Component					
Percentile	0.60	-	-	-	-
SATT threshold	20	-	-	-	-
	CBO	**LCOM**	**NOM**	**RFC**	**WMC**
Android					
Alves et al.'s threshold	23	414	36	75	141
Activity					
Percentile	0.41	-	-	0.56	-
SATT threshold	40	-	-	107	-
Fragment					
Percentile	0.32	-	-	0.58	-
SATT threshold	41	-	-	98	-

RQ_3: The state-of-the-art approach tends to return doubtful results for architectural roles that have metric values distribution significantly different from other classes. Our approach improves it by using the architectural role's metric values distribution to define thresholds.

VI. DISCUSSION

The key findings of our study are: 1) some architectural roles present significantly different values for certain metrics when compared to other classes, 2) these differences in code metrics can be explained by each architectural role's specific responsibilities, 3) the state-of-the-art approach tends to return doubtful results for architectural roles that have metric values distributions significantly different from other classes; instead, our approach improves it by using the architectural role's metric values distribution to define thresholds.

These findings have important implications for both research and practice, which we discuss in the following sections.

A. Using metrics in practice

Code assessment tools use a single threshold for a code metric, regardless of the architectural role of the class in the system. However, as we saw, some architectural roles present metric values distributions that are different from others. Thus, these tools may perform doubtful assessments. PMD, as an example, relies on the CBO metric to point developers to highly coupled classes. In its documentation [34], we see that the threshold used by the tool to assess the coupling of any class in the system is 20. However, CONTROLLERS are usually more coupled than other classes. It means that some of them will be blamed by the tool when, in fact, they are not problematic if compared to their peers. The number of false positives is indeed a common problem in these kind of tools [35].

Benchmarking techniques currently focus on (i) better identifying thresholds that would point to classes that are outliers within the benchmark, and (ii) producing different benchmarks for different application domains. Our SATT approach tries to prevent that from happening by providing a different threshold when an architectural role presents a code metric values distribution considerably different from other classes. Thus, our approach provides a more fair comparison, as classes are compared only to their peers.

One may argue that our approach may lead to an increase in the number of false negatives, *i.e.*, some CONTROLLER has coupling issues, but this is not detected by our approach as the specific CBO threshold for CONTROLLERS is too high. We claim this may not be the case, mostly because of the nature of a benchmarking technique. Suppose that we performed a traditional benchmark in a large number of systems, and derived the thresholds. When assessing a CONTROLLER class using this threshold, we are basically "comparing the Controller class with all other classes in the benchmark". If a class has a metric value larger than the moderate threshold, it means that this class belongs to the 30% worst classes when compared to the benchmark. Instead, our SATT approach, improves this by "comparing the Controller class with the other Controller classes in the benchmark". Thus, if a CONTROLLER has a metric value larger than the moderate threshold, it means that this CONTROLLER belongs to the 30% worst CONTROLLER classes when compared to the benchmark.

Interestingly, Fontana *et al.* [36] proposed a different way of reducing the number of false positives in code smells detection strategies [37], [5]. In their work, the authors propose a catalogue of common false positives for different code smells, *e.g.* a class that builds GUIs (graphical user interfaces) is a common false positive in God Class detection. Aniche *et al.* [38] also proposed code smells that are specific to a certain architectural role. These smells were a consequence of the architectural role's specific responsibilities. Thus, our findings reinforce the importance of analyzing the class' responsibilities when assessing its quality. Indeed, taking the architectural role of a class into consideration can be a step towards reducing false positives.

Both of the architectures we studied in this paper happen to have easily detectable architectural roles (annotations or inheritance). We strongly encourage users of SATT to identify a way to automatically detect the role of classes in their systems. Other strategies might be the use of package names, directories. Detection strategies are out of the scope of this research.

B. Research implications

Researchers have shown that metric value distributions are sensitive to context information, such as the project, the application domain, and age [20], [21]. As an additional result of our study, we now know that metric value distributions are influenced by the system architecture, and there are clear reasons for these differences. In other words, this means that some classes are usually more coupled or complex than others (*e.g.*, CONTROLLERS and SERVICES), or have more methods (*e.g.*, ENTITIES), and that happens not just as a result of a bad practice — it is just the natural consequence of their specific responsibilities.

Identifying the most important architectural roles in other system architectures is another topic that deserves attention. Desktop applications and plugin development are two examples of other common system architectures in the market. In addition, the same system architecture can be implemented in different ways by different technologies. In this study, we made use of Spring MVC, which is a particular implementation of the MVC architecture. Other popular frameworks in industry, such as Asp.Net MVC and Ruby on Rails, also implement it. However, each implementation has its own particularities. Understanding the extent of these differences and how our findings can be generalized also deserves future studies.

In this work, we relied on Alves *et al.*'s approach [8]. Other authors also attempted to improve the choice of the thresholds. Fontana *et al.* [39] worked on an algorithm to automatically identify these 3 thresholds. The approach analyses each metric distribution, and use the table of frequencies of each value to determine the percentile in which the rest of the data will be "discarded". Then, the authors use the 25th, 50th, and 75th percentiles to define moderate, high, and very high risk. In Oliveira *et al.*'s work [40], instead of using the threshold as a hard filter, they proposed a minimal percentage of classes that should be above this limit. They derived and calibrated the thresholds so that they were not based on lenient upper limits.

Still, none of these approaches derive specific thresholds according to the architectural roles of the software architecture. Research needs to be conducted in order to understand whether they can be adapted and how their results would compare to SATT.

C. Threats to Validity

Construct Validity. To compare the differences among different architectural roles, we relied on code metrics. To that end, we selected the CK suite. Although there might be different metrics, we believe CK was a good choice as it covers many aspects of object-oriented programming. Also, as most tools rely on compiled code, we developed our own tool that uses static analysis. Because of that, metrics may present small variations when compared to other tools. It also happens with other tools [8], and we do not think the small variation that might happen in each metric/tool would affect the results because: (1) the difference is probably small, as the original algorithm of the metric is well-defined, (2) both statistical tests used (Wilcoxon and Cliff's Delta) are strong against small variations.

Internal Validity. The studied architectural roles are easy to detect (as they were based on annotations or inheritance). The chance of wrongly annotating a class is low, as the architecture enforces these rules, and a single wrongly annotated class could make the software sometimes not even to execute. However, a developer can make use of different implementation strategies within the same architectural role, *i.e.*, a REPOSITORY can be implemented using a object-relational mapping framework, such as Hibernate, or using JDBC, the Java Database Connectivity API. Indeed, we did not isolate the class' implementation decisions confound factor. However, we conjecture that the main findings would still apply, and the possible REPOSITORY-HIBERNATE or REPOSITORY-JDBC roles would also present their own specific metric value distributions.

External Validity. (1) The number of participants in our qualitative studies is small (6 experts). Still, we made sure all of them were very experienced in software architecture. Also, experts' opinions matched on most of the questions. Hence, we do not believe a different set of participants would have completely different opinions; (2) The number of selected projects for the quantitative analysis was high (more than 100 in Spring, and more than 300 in Android). Still, we do not claim the findings to be generalizable to industrial software [41]. Still, we performed a small evaluation in a single software from our industry partner (with more than 1 million lines of code), and differences were significant within their project. Future work is to evaluate these differences in other industrial software; (3) We presented data for two different concrete architectural styles: Spring MVC (web) and Android (mobile). Indeed, there are many other popular architectures in which these ideas would potentially apply, but this should be subject of further study. Still, our approach is generalizable enough for that to happen.

VII. RELATED WORK

Different authors have studied the distribution of code metric values. Concas *et al.* [42] measured 10 different properties related to classes, methods, and the relationships between them in a Smalltalk system, and they found that distributions are usually Pareto or log-normal distributions. Because of that, the standard evaluation based on means and standard deviations is misleading. According to the authors, these distributions have a fat tail, which means the existence of classes with extreme values. Many other authors corroborate and show that the distribution of code metrics is rarely normal [43], [44], [9], [10], [45], [11]. Regarding CK metrics, Herraiz *et al.* [12] found that WMC, CBO, and RFC are double Pareto distributions, while NOC and LOC follow power law. DIT was the only one which could not be described by either log-normal or power law.

The context also has been an important discussion in the field. Gil and Lalouche [21], by means of visual inspection, showed that metric values are sensitive to the context, and because of it, the measurements in one project are not good predictors for other projects. According to them, one way to neutralize the problem is by using log normal standardization. Zhang *et al.* [20] showed that metric values can be affected by factors, such as programming language, age and lifespan.

We find our work similar to the ones above in the sense that we are also evaluating the effects of context on software metrics. To the best of our knowledge, this is the first study that evaluates the influence of the system architecture on code metric values distributions.

As we said before, the understanding of code metric values distributions is fundamental. Thus, research has already been devoted to finding the best threshold for a code metric. In his work, McCabe [24] not only defined the Cyclomatic Complexity metric (used in this paper as the WMC metric), but also defined a threshold, namely 10. However, this number was derived from experience, and not from empirical studies.

Although some authors mention that using experience is a valid approach [5], and others actually did it [46], [47], researchers also studied the distribution of these code metrics over time in order to find a threshold that would indicate a symptom of bad code. Erni *et al.* [13], for example, propose mean and standard deviation as a way to find thresholds. However, as we said in Sections III-B and VI-B, we made use of benchmarking techniques are they are currently more robust than the past ones.

VIII. CONCLUSIONS

Software developers have been relying on code metrics to assess the quality of their software systems. However, to the best of our knowledge, assessment techniques have not taken the architectural role of a class as an important concern when performing their analyses up to now.

In this paper, we propose SATT (Software Architecture Tailored Thresholds), a technique that detects whether an architectural role is considerably different from others in the system, and provides a specific threshold for that role. To evaluate whether our approach can be applied in real settings, we analyzed it on 2 different architectures (MVC and Android) in more than 400 projects; in addition, we interviewed 6 experts in order to understand why some architectural roles are different from others.

The main contributions of this paper are:

1) The so-called SATT approach which provides specific thresholds for architectural roles that are considerably different from others.
2) Application of SATT to MVC and Android architectural styles, demonstrating that our approach can overcome issues that currently exist in traditional approaches, especially when some architectural role presents very different metric values than others.

Our results call for architecture-specific treatment of class level metrics in tools used for code quality assessments, such as SonarQube and PMD.

References

[1] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, 1993.

[2] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, 1996.

[3] M. O. Elish and D. Rine, "Investigation of metrics for object-oriented design logical stability," in *Software Maintenance and Reengineering. Proceedings. Seventh European Conf. on*. IEEE, 2003.

[4] H. Sahraoui, R. Godin, T. Miceli *et al.*, "Can metrics help to bridge the gap between the improvement of oo design quality and its automation?" in *Software Maintenance, Intl. Conf. on*. IEEE, 2000.

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

[6] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, 2010.

[7] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *Software Engineering, IEEE Transactions on*, vol. 40, no. 11, pp. 1100–1125, 2014.

[8] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Software Maintenance (ICSM), IEEE Intl. Conf. on*. IEEE, 2010.

[9] Y. Yi, H. Song, R. Zheng-ping, and L. Xiao-ming, "Scale-free property in large scale object-oriented software and its significance on software engineering," in *Information and Computing Science, Second Intl. Conf. on*, vol. 3. IEEE, 2009.

[10] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, 2009.

[11] I. Herraiz, D. M. German, and A. E. Hassan, "On the distribution of source code file sizes," 2011.

[12] I. Herraiz, D. Rodriguez, and R. Harrison, "On the statistical distribution of object-oriented system properties," in *Emerging Trends in Software Metrics (WETSoM), 3rd Intl. Workshop on*. IEEE, 2012.

[13] K. Erni and C. Lewerentz, "Applying design-metrics to object-oriented frameworks," in *Software Metrics Symposium, Proceedings of the 3rd Intl.* IEEE, 1996.

[14] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, 2010.

[15] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, 2012.

[16] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[17] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler, *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., 2003.

[18] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[19] G. E. Krasner, S. T. Pope *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, 1988.

[20] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How does context affect the distribution of software maintainability metrics?" in *IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 350–359.

[21] J. Y. Gil and G. Lalouche, "When do software complexity metrics mean nothing?–when examined out of context," *Journal of Object Technology*, vol. 15, no. 1, 2016.

[22] E. Bouwers, J. Visser, and A. Van Deursen, "Getting what you measure," *Communications of the ACM*, vol. 55, no. 7, pp. 54–59, 2012.

[23] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[24] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, 1976.

[25] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, 1994.

[26] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, 2005.

[27] Z. Turnaround, "Top 4 java web frameworks revealed: Real life usage data of spring mvc, vaadin, gwt and jsf," http://bit.ly/1smVDf9.

[28] Pivotal, "Spring documentation," http://spring.io/docs.

[29] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006.

[30] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the Intl. Conf. on Software Engineering*. IEEE Press, 2013.

[31] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "Appendix: On the distribution of code metrics and system architecture." [Online]. Available: http://mauricioaniche.github.io/scam2016.

[32] M. T. Chi, P. J. Feltovich, and R. Glaser, "Categorization and representation of physics problems by experts and novices," *Cognitive science*, vol. 5, no. 2, pp. 121–152, 1981.

[33] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[34] "Pmd cbo documentation." [Online]. Available: https://pmd.github.io/pmd-5.4.1/pmd-java/rules/java/coupling.html#CouplingBetweenObjects.

[35] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 161–170.

[36] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Preliminary catalogue of anti-pattern and code smell false positives," *Poznan University of Technology, Tech. Rep. RA-5/15*, 2015.

[37] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 20th IEEE Intl. Conf. on*. IEEE, 2004.

[38] M. Aniche, G. Bavota, C. Treude, A. van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architecture," in *Software Maintenance and Evolution (ICSME), 2016 IEEE 31th International Conference on*. IEEE, 2016.

[39] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the Sixth Intl. Workshop on Emerging Trends in Software Metrics*. IEEE Press, 2015.

[40] P. Oliveira, M. T. Valente, and F. Paim Lima, "Extracting relative thresholds for source code metrics," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Software Evolution Week-IEEE Conf. on*. IEEE, 2014.

[41] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[42] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, 2007.

[43] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Source Code Analysis and Manipulation, Proceedings. Third IEEE Intl. Workshop on*. IEEE, 2003.

[44] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in oo programs," *Communications of the ACM*, vol. 48, no. 5, 2005.

[45] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006.

[46] D. Coleman, B. Lowther, and P. Oman, "The application of software maintainability models in industrial software systems," *Journal of Systems and Software*, vol. 29, no. 1, 1995.

[47] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, 1988.