Developers' Perceptions on Object-Oriented Design and Architectural Roles

Maurício Aniche, Marco Aurélio Gerosa
University of São Paulo
Department of Computer Science
São Paulo, Brazil
{aniche, gerosa}@ime.usp.br

Christoph Treude
University of Adelaide
School of Computer Science
Adelaide, Australia
christoph.treude@adelaide.edu.au

ABSTRACT

Software developers commonly rely on well-known software architecture patterns, such as MVC, to build their applications. In many of these patterns, classes play specific roles in the system, such as Controllers or Entities, which means that each of these classes has specific characteristics in terms of object-oriented class design and implementation. Indeed, as we have shown in a previous study, architectural roles are different from each other in terms of code metrics. In this paper, we present a study in a software development company in which we captured developers' perceptions on object-oriented design aspects of the architectural roles in their system and whether these perceptions match the source code metric analysis. We found that their developers do not have a common perception of how their architectural roles behave in terms of object-oriented design aspects, and that their perceptions also do not match the results of the source code metric analysis. This phenomenon also does not seem to be related to developers' experience. We find these results alarming, and thus, we suggest software development teams to invest in education and knowledge sharing about how their system's architectural roles behave.

CCS Concepts

•**Software and its engineering → Object oriented architectures; Software evolution; Maintaining software;**

Keywords

object-oriented design, software architecture, code metrics

1. INTRODUCTION

Several software architecture patterns commonly rely on specific building blocks (also known as architectural roles), each carrying a specific responsibility. As an example, a Model-View-Controller (MVC) system [22] contains classes that play the CONTROLLER architectural role (responsible

for coordinating the flow between the model and view layers) and the MODEL architectural role (responsible for representing the business concepts) roles. In practice, understanding how each architectural role behaves in terms of object-oriented design aspects is fundamental to maintenance activities.

First, understanding the behavior of each architectural role enables developers to make use of implementation practices that are specific to each of them. Indeed, we have shown in previous studies that each architectural role has its own set of specific good and bad practices [5, 4], *e.g.*, CONTROLLERS should not contain business rules and a REPOSITORY should deal with a single ENTITY only.

Second, popular code analysis tools in industry, such as PMD [37] and Sonarqube [41], are based on code metrics which are, in a nutshell, heuristics to quantitatively measure these aspects. However, these tools just calculate the metric value; it is up to the developers to interpret the results and actually decide whether a class is problematic. Suppose that a developer sees some coupling measurement from a CONTROLLER class. If s/he does not know that CONTROLLERS are usually more coupled than other classes, s/he will blame a class that is, in fact, not problematic when compared to other CONTROLLERS. Other researchers also have shown the importance of understanding the perceptions of software developers on different maintenance tasks [8, 45, 33].

In this paper, we conduct a study in a Brazilian software development company that develops a large web system. By means of a "card game", a technique that we created to capture the developers' perceptions on object-oriented design aspects in their systems, we identify a set of rules, such as *"Controllers are more coupled than Entities"*. After that, we triangulate the results by comparing their perceptions to the results of a source code metric analysis that we performed in their system (which has more than 6,000 classes and 1 million lines of code).

Our results show that 1) developers do not share a common perception of how their architectural roles compare to each other in terms of object-oriented aspects (coupling, cohesion, complexity and inheritance), and 2) their perceptions do not match the results from a source code analysis of their system, and 3) this phenomenom also does not seem to be related to developers' experience.

This paper contributes with:

1. A study on the developers' perceptions on how they expect their architectural roles to behave in terms of object-oriented design aspects, as well as a triangulation with the results of a code analysis in their system.

2. A discussion on how to mitigate the lack of a shared vision on the behavior of the architectural roles in their system architecture in terms of object-oriented design aspects.

3. An approach (namely "card game interviews") to capture developers' perceptions on the object-oriented design aspects of the architectural roles in their system.

This paper is divided as follows: in Section 2, we provide the background on object-oriented design aspects, code metrics, and architectural roles used throughtout the paper; in Section 3, we present our research questions as well as the research method; in Section 4, we present the results of our research; in Section 5, we discuss the results and how software development teams can learn from it; in Section 6, we present related work to this research; in Section 7, we discuss how we mitigate threats to the validity of the study; and, finally, in Section 8, we provide our concluding remarks.

2. BACKGROUND

In this section, we briefly introduce some definitions we use throughout this research: object-oriented design aspects (Section 2.1), code metrics (Section 2.2), and the architectural roles of the studied software (Section 2.3).

2.1 Object-Oriented Design Aspects

To design a good object-oriented system, developers must be aware of the different aspects of the class design. A good class, among other attributes, is the one that has a good balance between its coupling, cohesion, and complexity [28].

In the listing below, we provide a definition of the four object-oriented design aspects used in this research, derived from multiple authors [16, 27].

- **Coupling.** Classes can depend on other classes. This implies a dependency between the two objects. A change in one of them can impact the other. Thus, the more coupled a class is, the more problematic it may be.

- **Cohesion.** Classes should have few responsibilities. After all, the more responsibilities in a single class, the more problems that class can have. The more cohesive a class is, the higher its reuse can be.

- **Complexity.** The more complex a class is, the harder it is to understand and maintain it.

- **Inheritance.** Classes can inherit both data and behavior from other classes as a way to reuse code or specialize the behavior of the base class.

2.2 Code Metrics

There exist many different source code metrics. We rely on the Chidamber & Kemerer (CK) metrics suite [11], as (i) it covers different aspects of object-oriented programming, such as coupling (CBO, RFC), cohesion (LCOM), inheritance (DIT, NOC), and complexity (WMC, NOM), and (ii) it has already proven its usefulness in earlier studies [24, 10, 18]. The CK suite consists of the following class level metrics, which are all related to the object-oriented design aspects we mention in the previous sub-section:

- **Number of Methods (NOM) - Complexity.** The count of number of methods in a class.

- **Weighted Methods Per Class (WMC) - Complexity.** Sum of McCabe's cyclomatic complexity [29] for each method in the class.

- **Depth of Inheritance Tree (DIT) - Inheritance.** The length of the path from a class to its highest superclass.

- **Number of Children (NOC) - Inheritance.** The number of direct sub-classes a class has.

- **Coupling Between Object Classes (CBO) - Coupling.** 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) - Coupling.** It is the count of all method invocations that happen in a class.

- **Lack of Cohesion of Methods (LCOM) - Cohesion.** The count of the 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.

2.3 Spring MVC Architectural Roles

We conducted this research in a software development company that uses Spring MVC, a Java web development framework, as the basis of their software architecture. As its name states, the framework makes use of the Model-View-Controller pattern [22]. A common MVC application contains different "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. 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 [42] can be optionally applied in the system.

In the following, we present the five main architectural roles in Spring MVC:

- **Controllers.** Take care of the flow between the model and the view layers.

- **Entities.** Represent a domain object (*e.g.*, an *Item* or a *Product*).

- **Repositories.** Responsible to encapsulate persistence logic, similar to Data Access Objects [17].

- **Services.** Implemented when there is a need to offer an operation that stands alone in the model, with no encapsulated state.

- **Components.** Represent small components, such as utility classes. Practical examples can be UI formatting or data conversion classes.

3. STUDY DESIGN

The main *goal* of this study is to understand the developers' perceptions on their system's architectural roles' object-oriented design aspects and whether these perceptions match the results of a code metric analysis. We conducted the study in a Brazilian software development company located in São Paulo. We interviewed 17 of their developers and analyzed their main Java system, which is composed of more than 1 million lines of code.

In sub-section 3.1, we present our research questions, and in sub-section 3.2, we discuss both the qualitative and the quantitative method performed in this study.

3.1 Research Questions

RQ$_1$: Do developers share a common perception on the object-oriented design aspects in their system's architectural roles?

The feeling of a developer is always important when dealing with software maintenance. Do developers feel any difference in terms of object-oriented design aspects in classes during their daily development? As an example, do they feel that some role A in the system is more coupled than some other role B? To answer the question, we rely on "card game" interviews with 17 professional developers. During the interviews, developers are asked to explicitly compare the behavior of classes in their system.

RQ$_2$: Do developers' perceptions match the code metric analysis of their own project?

Do the results of a code metric analysis match the developers' perceptions? To answer the question, we transform their opinions into mathematical expressions (*e.g.*, *Controller Coupling > Entities*) and match these expressions with the analysis of CK code metrics [11] extracted from their own project.

3.2 Method

As a first step, we invited one of our industry partners to be part of the study. The company works on a Java-based Spring MVC web application. The product supports supermarket stores in all their needs. The software has been developed for 11 years, and has more than 1 million lines of code in all its modules. We chose the company because (i) they have a team composed by both experienced and beginner developers which gives us the possibility of measuring the impact of experience (ii) they develop in Java, which is the language supported by our tools, (iii) their software contains many classes that implement the studied architectural roles (see Table 2), and (iv) they are based in São Paulo, which enables us to personally interview the participants.

We invited participants to talk about how they perceive the architectural roles of their system in terms of the aforementioned object-oriented design aspects. We also made them compare their perceptions among architectural roles, *i.e.*, instead of saying that Controllers are highly coupled, and Entities are highly complex, which are highly subjective, they should tell us that Controllers are more coupled than Services, or that Entities are more complex than Repositories. This way, we are able to explicitly compare their answers to the results of the code metric analysis. Note that participants did not discuss specific classes in their systems (with which they might not have had any contact). Instead, participants discussed existing architectural roles in their system architecture. Thus, we avoid their possible lack of knowledge on specific classes in the system, *e.g.*, one may not know the Controller A, but one understands what a Controller is.

To answer RQ$_1$, we developed an approach in which developers make use of cards to better express their perceptions on their system's design. From now on, we call this approach "card game interview", which we detail in the following paragraphs.

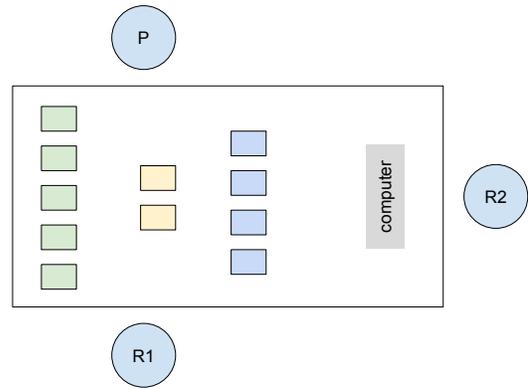On a table, we put 3 sets of cards as shown in Figure 1:

Figure 1: Card game sketch. P=Interviewee, R1 and R2=Researchers, Green cards=Architectural roles, Yellow Cards=Comparison operators, Blue cards=OOP concepts.

1. **Architectural roles deck.** This set contains all the common architectural roles in their software system: Controllers, Services, Components, Entities, and Repositories. The goal of these cards is to make developers compare two roles. We decided to make them compare architectural roles instead of single classes, as architectural roles are (i) generalizable during a discussion, *i.e.*, you can state that Controllers are commonly more coupled than Repositories, without talking about some exceptional cases, and (ii) we have shown in the past that developers have different best practices for each of the MVC layers [5].

2. **OOP concepts deck.** This set contains four object-oriented principles, which we want developers to use when comparing the architectural roles: "coupling", "cohesion", "complexity", and "inheritance".

3. **Comparison operators deck.** We provide developers with a set of cards with two comparison operators: "higher than" and "lower than".

An advantage of using cards to refer to architectural roles and concepts is that cards remind participants about all the roles and concepts being considered. In that way, instead of thinking about a single role at a time, they are continuously thinking about all of them before making an assumption.

Before beginning an interview, we explain to them how the card game interview works: in each round, participants select two cards from the architectural roles deck (*e.g.*, Controller and Repository), and put each one on one side of the table (left and right). Then, they get one card from the OOP concept deck and add a comparison operator to it. The goal is to make them create rules in the following format:

<center><role> <concept> <comparison> <role></center>

Examples of created rules are *[Controller Coupling > Entity]*, and *[Component Complexity < Service]*. We told participants to create any number of rules they want. To facilitate, they could use more than one role or concept at the same time, creating rules like "Component Controller Complexity < Entity". For all the rules, they had to explain the reasoning behind it. We asked the following question for

each rule: *"Why do you think A is more [or less] B than C?"*, where A and C are the architectural roles, and B is an OOP concept. The question for the first example above is: *"Why do you think Controllers are more coupled than Entities?"*.

As we want to capture participants' perceptions on each architectural role as they were in the wild, we do not give them any explanation about the OOP concepts or the architectural roles. We clearly state that participants should use their own experience to create the rules. If a participant was not familiar with a specific role, s/he had the choice of removing it from the deck. In practice, the only architectural role a few participants were not familiar with was COMPONENT. According to them, "they do not use it very much in their system". We did not discuss about COMPONENTS with these participants.

We also designed the card game to give us the freedom to ask them any other questions that seemed interesting, according to their answers. During the interviews, as researchers, we supported the process, but did not influence the participants. Sometimes, we asked if they could create a rule with some architectural role or concept they had not used yet. For each participant, we randomized the order of the cards in each deck, in order to reduce a possible bias of one role (*e.g.*, the first card in the deck) being used more than others.

Two researchers were present during the card game. The role of the first researcher was to conduct and support the game with the participant, while the second researcher was focused on taking notes of all the rules created. We decided to take notes in a CSV (comma-separated values) with the format: "participantId,roleA,concept,comparison,roleB".

We interviewed 17 software developers. Interviews took 4:30 hours in total, and they were fully transcribed. All of the participants were developers or technical leaders. No managers or product owners were interviewed. After a manual analysis on their answers, we decided to remove participants P1, P2 and P13, due to their apparent lack of understanding of the OOP concepts used in this study. Specifically, participants P2 and P13 provided a higher number of rules without clear explanations about them, while P1 changed his mind about the rules many times during the interview. In Table 1, we present their experience in software development. Participants have different levels of experience in software development. 5 of them have more than 6 years of experience, while 4 of them have been working as software developers for the last 1 or 2 years.

To analyze the influence of experience, we divided participants in two groups: experienced (more than 4 years of experience) and non-experienced (less than 4 years of experience). We chose "4 years" as it divides the participants in two sets with similar size (7 non-experienced and 7 experienced), which allows us to better compare the data from both groups. We used the unpaired Wilcoxon signed rank test [43] to compare the quantity of rules provided by participants in each group.

To answer RQ_2, we first got access to their source code, and executed a code metrics tool. As we had access only to their source code (and not to external libraries), we were not able to compile the code. Thus, we made use of static analysis to calculate the code metrics. As we did not find a tool that measures all the CK metrics, we implemented our tool [3], which is open source and freely available for inspection.

Table 1: Participant's experience in software development

Experience (in years)	Participants (n = 14)
1-2 years	P3, P6, P9, P10
2-4 years	P5, P16, P17
4-6 years	P7, P12, P15
6-8 years	P14
8-10 years	P4, P11
10+ years	P8

Table 2: Our industry partner's project's numbers

Architectural Role	# of classes
Controllers	681
Repositories	765
Services	1,139
Entities	854
Components	59
Other classes	6,308

As a first step in analyzing their source code, we identified each class that belongs to one of the roles. If we were not able to identify the role of a class, then we considered it as "another role", and discarded it from the rest of the study. In Spring MVC, all classes that play one of the architectural roles need to be annotated with one of the stereotypes provided by the framework. CONTROLLER classes, for example, need to be annotated with the *@Controller* annotation. Other annotations are *@Repository, @Component, @Entity,* and *@Service*. In Table 2, we show the number of classes identified in each architectural role, as well as the number of classes discarded.

We then measured all the CK metrics in these classes: CBO, RFC, WMC, DIT, NOC, LCOM, and NOM. Next step is to compare the code metric values for each architectural role. We combined the architectural roles in pairs to see whether their metrics distribution is significantly different from each other. As Spring MVC systems have 5 different architectural roles (thus, 10 pairwise comparisons, *e.g.*, Controllers vs Services, Services vs Entities, etc.) and the CK suite contains 7 different metrics, we performed 70 comparisons.

We used the unpaired Wilcoxon signed rank test [43]. The ones that present a significant *p-value* are significantly different from each other. We do not correct the *p-value* with strategies such as Bonferroni, because of its contradictory use [35, 32]. We also calculate Cliff's Delta effect size for each comparison to identify the direction of the effect, *i.e.*, $A > B$ or $A < B$. A positive effect size indicates that the first architectural role (A) presents higher values in the distribution than the second architectural role (B) in the comparison. We also measured Cliff's Delta effect size of the comparison. The higher the effect size, the larger the difference among these two groups.

We matched each of the rules created by the participants during the card game to the measured difference. For a rule $A > B$ in terms of X, where A and B are architectural roles and X is an OOP aspect, we considered this rule to be true (match) if the difference between the measurements in A and B is significantly different (Wilcoxon < 0.05) and the effect size is positively higher than negligible (according to Romano *et al.*'s classification [38], > 0.147) (if the rule
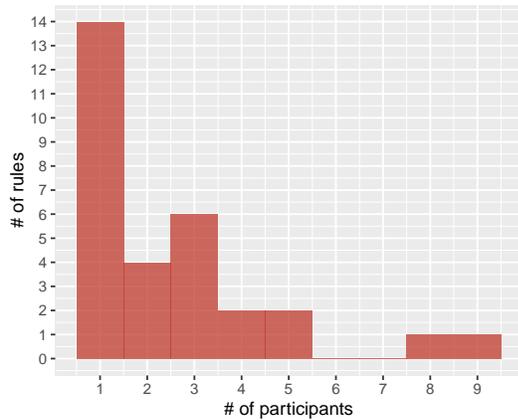
Figure 2: Histogram of rules shared among participants.

Table 3: Number of rules created by the participants

	Experienced participants	Non-experienced participants
Cohesion	8	5
Complexity	15	10
Coupling	9	10
Inheritance	7	11
Total	**39**	**36**

Table 4: Top 5 most popular rules created by participants

Role A	Metric	Sign	Role B	Participants (n=14)
Service	Complexity	>	Controller	9
Service	Coupling	>	Controller	8
Entity	Inheritance	>	Controller	5
Entity	Inheritance	>	Service	5
Entity	Complexity	>	Controller	4

is $A > B$) or negatively smaller than negligible (< 0.147) (if the rule is $A < B$) in at least one of the metrics that are related to X. Each aspect has its own set of metrics in the CK suite. We used the following map to convert concepts to metrics (see Section 2.2): Complexity $\rightarrow$ {WMC, NOM}, Coupling $\rightarrow$ {CBO, RFC}, Inheritance $\rightarrow$ {DIT, NOC}, Cohesion $\rightarrow$ {LCOM}.

During the analysis, we inverted the positions of the architectural roles when the rule was related to cohesion, as the LCOM metric relates to the "lack of cohesion of a class", and the rules created by participants relate to "cohesion" (the opposite).

To measure the influence of the developers' experience in both RQs, we calculated Fisher's exact test [40] to check whether the proportions of correct and incorrect answers by both the experienced and non-experienced groups were statistically different.

4. FINDINGS

In this section, we report our results for both RQ_1 (Section 4.1) and RQ_2 (Section 4.2).

4.1 RQ1: Developers' Perceptions

In Table 3, we present the number of rules created during our card game interviews. Participants create 30 distinct rules for a total of 75 rules. Experienced programmers are responsible for 39 of them, while non-experienced programmers are responsible for the remaining 36 rules. We do not find a statistically significant difference in the quantity of answers by both experienced and non-experienced developers ($p = 0.60$).

Interestingly, as depicted in Figure 2, which we show the histogram of rules created by the same participants, 14 rules (out of the 75, thus, 18% of the distinct rules) are mentioned by only a single participant. The same happens when we filter by experience: we find many rules mentioned by only one developer. If we analyze the rules created by only experienced developers (39 in total), we see that 13 rules (33%) are mentioned by just one participant. When analyzing only the non-experienced participants, 12 rules (33% out of 36 rules) are mentioned by a single participant.

Complexity was the most common concept used in the rules (25 out of 77), followed by Coupling (19), Inheritance

(18), and Cohesion (13). In addition, SERVICES are the most popular architectural role among the rules, as it appears 51 times in rules (both in the left and the right side of the rule). ENTITIES (36), CONTROLLERS (34) are the next ones, followed by REPOSITORIES (19) and COMPONENTS (10). In Table 4, we show the top 5 most mentioned rules, and below we present the participants' explanation for each of them. We can see that only 2 rules are perceived by more than 50% of the participants.

According to them (and we cite P10 and P12), SERVICES are more complex than CONTROLLERS because the last one just controls flow, and that should be simpler in terms of complexity than SERVICES, which hold the system's complexity. Also, P7, P14 (cited), P15, and P17 gave us strong opinions on why a SERVICE is more complex than all other roles.

P10: Because I think all the complexity should be here [in Services]. Controllers is just "it goes here, that goes there". And when it goes, than the complexity is there.

P12: Because a Controller should be a class that only gets and manages, from where things come and go. So, it is more a lean class. There must be less complexity in it. It is like that phrase: thin controllers and fat models. It is something like that.

P14: Because the [business] rules and even the design patterns that I will use, I will be doing it all in the Service. (...)

To explain why ENTITIES are more complex, P6 affirmed that they have many business and validation rules, while other classes do not.

P6: Services are less complex than Entities. Because Services do just one thing. Entities do a lot of validation, have a lot of [business] rules related to itself.

Regarding coupling, P15 believes SERVICES are more coupled than the rest, because they deal with integration with other services. On the other hand, P9 affirms that CONTROLLERS are more coupled than SERVICES, because they

Figure 3: Code metrics distribution in the system (bars, from left to right: Yellow=CBO, Blue=DIT, Black=LCOM, Green=NOC, Orange=NOM, Purple=RFC, Pink=WMC). Striped bars = significant difference ($p\text{-}value < 0.05$), height = Cliff's Delta effect size.

contain more rules that depend upon other classes.

> P9: Controllers are more coupled than Services. I've seen some Controllers containing rules that depend upon other guys, and if you are going to change these guys, you end up breaking an infinity number of things.

> P15: I think Services are the most coupled, mainly because it deals with integrations and things like that. Everything has to be sort of coupled (...), and depending on the case, if you change it, you break the entire system, as one Service can depend upon another Service.

Regarding inheritance, participants state that ENTITIES make use of it more than other roles, as ENTITIES represent business concepts, and the use of inheritance allows developers to derive other related business concepts (we quote P15 and P16). Also, according to them, the use of inheritance is not normal in other roles. As an example, P2 clearly stated that he does not see a reason for making use of inheritance in CONTROLLERS. Interestingly, P1 said that, although it is normal to use inheritance in ENTITIES, they try to avoid it.

> P15: Inheritance makes more sense in Entities, as you can build "an object inside of the other".

> P16: Because you can add just basic data in an Entity, and then other Entities [by means of inheritance] can reuse them.

Interestingly, 5 rules have an exact opposite version mentioned by another participant. The Service Complexity > Controller rule was created by 9 participants, while the rule Controller Complexity > Service was created by a single

participant. In Table 5, we show the rules that had an opposite version and the number of participants that created them. The phenomenon also happens when we separate per experience; experienced developers only diverged in one rule (Entity has higher inheritance than Service, 2 participants against 1), while non-experienced developers diverged in 2 rules (Service has higher complexity than Controllers, 4 participants vs 1, and Service has higher coupling than Controllers, also 4 participants vs 1). However, although it happens, we see that it is usually a single participant that disagrees with the others. In the following paragraph, we illustrate the participants' opinions on the contradictory rules.

P5 said that, although SERVICES should contain business rules, in practice, these rules are often written in CONTROLLERS, making CONTROLLERS more complex than SERVICES. As we see in Table 5, nine participants think the other way around. P9 believes CONTROLLERS are less coupled than SERVICES. According to him, it is more common to "break" the behavior of a CONTROLLER when changing a business rules than to break a SERVICE. P14 affirmed that, although inheritance is common in ENTITIES, he thinks this is more common in SERVICES. As we see in Table 5, 5 other participants believe in the opposite.

Thus, we answer RQ_1: Do developers share a common perception on the object-oriented design aspects in their system's architectural roles?

1. Developers do not have a common perception of how their architectural roles work in terms of object-oriented design aspects. Many perceptions are felt by a single participant, while just a few are shared by many of them.

Table 5: Top 5 rules with a contradictory version. Qty=number of participants that created the rule, Opp=number of participants that created the opposite version.

Role A	Metric	Sign	Role B	Qty	Opp
Service	Complexity	>	Controller	9	1
Service	Coupling	>	Controller	8	1
Entity	Inheritance	>	Service	5	1
Service	Complexity	>	Entity	3	1
Controller	Complexity	>	Repository	1	1

2. Still, most of the perceptions that are shared by many participants (in a few cases, shared by more than 50% of participants) were contradicted by another participant, indicating that not all of them share the same vision.

3. By visual inspection, we observe that both experienced and non-experienced groups are similar in terms of behavior. Thus, we were not able to affirm whether experience positively influences these results, as our statistical tests did not present significant results.

4.2 RQ2: Code Metric Analysis

In Figure 3, we present the pairwise comparison of each metric and architectural role in the system. Each cell contains a chart with 7 bars, one for each metric: CBO, DIT, LCOM, NOC, NOM, RFC, and WMC, respectively. The height of the bar represents Cliff's Delta effect size. If the bar is positive, then the architectural role in the row is significantly greater than the architectural role in the column, regarding that code metric. If the bar is negative, then the architectual role in the row is significantly smaller than the architectural role in the column. We do not show bars in which the result is non-significant ($p\text{-value} > 0.05$). As examples, (i) the yellow bar (CBO) in the CONTROLLER-REPOSITORY bar chart has a value of 0.80 (large effect size), meaning that CONTROLLERS have much higher values of CBO than REPOSITORIES, and (ii) the purple bar (RFC) in the CONTROLLER-ENTITY bar chart means that CONTROLLERS have much higher values of RFC than ENTITIES (0.66 is considered a large effect size). Note that the purple bar in the ENTITY-CONTROLLER bar chart has a value of -0.66. We exhibit both directions of the relationship to facilitate the interpretation of the chart. We observe that 57 out of the 70 comparisons were significantly different. The significant p-values mean that we are able to confidently compare the effect size between roles. The complete list of values can be found in our online appendix [6].

With these numbers in hand, we show the number of times participants created a rule that matches with the result of the code analysis, in Table 6. We see that their opinions matched in 14 out of their 30 distinct rules (46%). Experienced and non-experienced developers also present a low assertiveness rate. Fisher's exact test does not show a difference in the proportions of assertiveness between both groups of developers ($p = 0.22$).

Interestingly, as we discussed in the method, we consider a rule to be true only and if only the effect size of the difference is significant and the effect size is higher than negligible. When analysing the data, we noticed that many rules were considered "wrong" because of this. The most affected

rules are the ones that deal with "inheritance", as in most cases, the difference of the usage of inheritance among the architectural roles is negligible.

In addition, even the "popular rules" (rules which many participants mentioned, where we show the top 5 in Table 4) do not entirely match. The rule *Services are more complex than Controllers* (the most popular one, $n = 9$) and the rule *Services are more complex than Repositories* ($n = 4$) are true. However, the differences are negligible in the other three rules among the top 5.

Thus, we answer RQ$_2$: Do developers' perceptions match the code metric analysis in their own project?

1. Developers' perceptions match the code metric analysis in only 50% of the cases. We consider this to be a low and worrisome number.

2. Even the perceptions that are shared among many participants are wrong in many cases. From the top 5 most popular perceptions, only 2 matched.

3. We are not able to affirm whether experience can positively influence the assertiveness of the perception. By means of visual inspection, we observe that both experienced and non-experienced groups present similar low performance.

5. DISCUSSION

From our results, we learn one important thing: *developers do not have a common perception on how their architectural roles compare in terms of object-oriented aspects*. We conjecture two main problems that this issue can lead to: lack of a common sense about specific best practices for each architectural role, and in the interpretation of code metric analysis. We discuss them in the following paragraphs.

The lack of perception about how each architectural role behaves may lead developers to make use of different code implementation patterns, *i.e.*, some developers may not care about coupling inside CONTROLLERS, while others may care. During the interviews, we noticed that even the definition of the object-oriented design aspects, such as coupling and cohesion, varied among developers. Yet, the lack of knowledge in many other important concepts in software development has been reported by other researchers. Yamashita and Moonen [45], as an example, conducted a survey with 85 professionals, and results indicate that 32% of developers do not know or have limited knowledge about code smells. Curiously, more than 40% of the participants in their survey affirm that they are extremely familiar with object-oriented design. Although this was not the focus of their research, we wonder how much their participants actually know about it in reality.

We also conjecture that this lack of common knowledge can be harmful when developers are interpreting the results of a code metric analysis. Indeed, if they do not share a common vision of how their system should behave, how should they evaluate the results of such analysis? At the end of each interview, we asked participants about their opinions on code metrics. To our surprise, most of them said that, although the team has a Sonarqube (a tool that calculates code metrics and warns developers about problematic classes) plugged into their continuous integration software, they do not care about the reports. P6 stated that *"we say*

Table 6: Comparing the perceptions of developers and the code metrics analysis in their system

	All participants (unique rules=30)		Experienced (unique rules=22)		Non-experienced (unique rules=21)	
	Correct	Wrong	Correct	Wrong	Correct	Wrong
Cohesion	3	5	2	5	1	3
Complexity	6	4	3	3	6	1
Coupling	4	2	3	2	4	1
Inheritance	1	5	0	4	1	4
Total	14	16	8	14	12	9
	46%	54%	36%	64%	57%	43%

a lot [about code metrics], but we don't use them in practice". P13 also said that the other problem is to know how to fix the possible problems that are pointed out by code metrics. The number of false positives in tools is a current problem [34], and we argue that this problem can be intensified when developers do not share the same vision on how each architectural role behaves in terms of object-oriented design aspects.

Experience also appears to not be a factor of influence. In this paper, we separated participants in less than 4 years and more than 4 years. However, we also tried different combinations, but none of them presented significance.

We find this result very alarming. Therefore, our suggestion for software development teams is to invest in education and knowledge sharing about how their architectural roles compare to each other in terms of object-oriented design aspects. Knowledge sharing and expertise coordination are indeed related to team performance [15, 14, 7]. Therefore, we suggest software development teams to discuss:

1. A shared definition of object-oriented concepts, such as coupling, cohesion, and complexity.

2. How each architectural role should behave in terms of these aspects, *e.g.,* SERVICES should be always cohesive, or CONTROLLERS can present high coupling, but should not contain business rules.

3. How they should expect each architectural role to behave during code metric analysis.

This study was conducted in a single software company. We have no evidence that our findings generalize to other companies or architectural patterns. Still, we argue our findings are important for software companies to know that developers may have different perceptions about their system architecture. Future work needs to be conducted in order to validate whether these suggestions can improve their daily work.

6. RELATED WORK

Code metrics are used in many other techniques. When combined, code metrics can identify high-level code smells. Marinescu [26] and Lanza and Marinescu [23] combine different metrics, thresholds, and logical operators to detect different smells, such as God Classes, Feature Envy, and Blob Classes. Other techniques follow the same idea. Moha *et al.* [30] proposed a text-based description called DECOR, in which smells are described by code metrics and thresholds. Munro [31] and Alikacem and Sahraoui [1] proposed similar approaches. Indeed, Khohm *et al.* [21] showed that smelly classes are more prone to change and to defects than other classes. Li and Shatnawi [25] also empirically evaluated the effects of code smells and showed a high correlation between defect-prone and some bad smells.

Other maintainability issues, such as change- or defect-proneness of a class, can also be detected by means of code metrics. Some studies show a relationship between a higher code metric value in a CK metric and the defect- or change-proneness of that class. WMC, CBO, LCOM, and RFC have been related to defect-proneness [19, 39, 20], while WMC, CBO, and RFC have been related to change-proneness [13, 9, 44]. D'Ambros *et al.* [12] show that different learning algorithms can have a good performance predicting defect-proneness when using CK metrics.

However, although code metrics can point to problematic pieces of code, the perception of a developer about the problem may be not precise. A study from Palomba *et al.* [33] showed that smells related to complex or long source code are perceived as harmful by developers; other types of smells are only perceived when their intensity is high. As said before, Yamashita and Moonen [45] conducted a survey with 85 professionals, and results indicate that 32% of developers do not know or have limited knowledge about code smells. Arcoverde *et al.* [8] performed a survey to understand how developers react to the presence of code smells. The results show that developers postpone the removal to avoid API modifications. Peters and Zaidman [36] analyzed the behavior of developers regarding the life cycle of code smells and results show that, even when developers are aware of the presence of the smell, they do not refactor.

To the best of our knowledge, this is the first study that captures the developers' perceptions on object-oriented aspects in their system's architectural roles and compares these perceptions to the results of a code metric analysis in their own software system.

7. THREATS TO VALIDITY

Construct validity. Threats to construct validity concern the relation between the theory and the observation, and in this work are mainly at risk due to the measurements we performed. To collect the perceptions from developers, we proposed the "card game interviews". However, we can not assure that participants mentioned everything they perceive on the architectural roles. To mitigate the problem, during the interviews, we supported them by making them comfortable enough to create any rule they want, and by reminding them about cards they have not used.

We also calculated CK metrics from their source code. As said before, we made use of our internal tool. Thus, metrics may present small variations when compared to other tools. It also happens with other tools [2], 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, and (2) both statistical tests used (Wilcoxon and Cliff's Delta) are strong against small variations.

Participants had different visions of coupling and cohesion. We decided not to give them a common definition before starting the interview, as we wanted to see how they would behave in the wild. Although providing them with a definition could change, or even improve the results in their favor, we affirm that this would bias the results. Another related threat might be that we chose the CK metric suite to measure the OOP concepts. A different code metric suite may imply different results in RQ2. However, we conjecture it would not change the final message of this study.

Internal validity. Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. We did not take into account classes with no defined architectural role in the system. In this case study, around 6,000 classes were ignored. However, before starting the interviews, we reminded them that we were going to discuss only classes with those architectural roles. Thus, we do not believe they were influenced by these other classes.

Also, we only asked participants to create rules using "greater than" or "smaller than". Further study is required to understand whether participants would agree more in rules with "equals to".

External validity. Threats to external validity concern the generalisation of results. As we performed a single case study in one company, we do not affirm these results are generalizable to other software development teams. Still, our findings should be shared with other software development companies, so that they can avoid these misconceptions among their developers.

8. CONCLUSION

Designing a high quality object-oriented system is challenging. Understanding how architectural roles behave in terms of object-oriented design aspects also is. In this paper, we performed a study in a Brazilian software development company. We studied the developers' perceptions on the object-oriented design aspects of their system's architectural roles and compared them to the results of a code metrics analysis. Our study is divided in two phases: in the first part, we performed a "card game interview", a technique that we developed to collect their perceptions on object-oriented aspects of their systems, with 17 developers. Then, we collected code metrics from their software system, and matched the results of the analysis with their perceptions.

We sum up our main findings:

1. Developers do not share a common perception of how their system's architectural roles are characterized in terms of object-oriented design aspects.

2. Developers' perceptions do not match the results of a code metric analysis in their system source code.

3. Experience seems not to be a factor of influence. Thus,

even experienced developers do not have a common or a more accurate perception of how architectural roles behave.

Although we can not argue that our findings are generalizable, we suggest software teams to invest in internal knowledge sharing and coordination expertise so that all developers can be aware of how their system's architectural roles should work.

9. REFERENCES

[1] E. Alikacem and H. Sahraoui. Generic metric extraction framework. In *Proceedings of the 16th Intl. Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)*, 2006.

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

[3] M. Aniche. Ck calculator. http://www.github.com/mauricioaniche/ck.

[4] 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 COnverence on.* IEEE, 2016.

[5] M. Aniche and M. Gerosa. Boas e más práticas em desenvolvimento web com mvc: Resultados de um questionário com profissionais. *Workshop of Software Visualization, Evolution and Maintenance*, 2015.

[6] M. Aniche, C. Treude, and M. A. Gerosa. Appendix: Developers' perceptions on object-oriented design and system architecture. http://mauricioaniche.github.io/sbes2016.

[7] M. F. Aniche and G. de Azevedo Silveira. Increasing learning in an agile environment: Lessons learned in an agile team. In *Agile Conference (AGILE), 2011*, pages 289–295. IEEE, 2011.

[8] R. Arcoverde, A. Garcia, and E. Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 33–36. ACM, 2011.

[9] Y. Ayalew and K. Mguni. An assessment of changeability of open source software. *Computer and Information Science*, 6(3), 2013.

[10] 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*, 22(10), 1996.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6), 1994.

[12] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[13] S. Eski and F. Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In *Software Testing, Verification and Validation Workshops (ICSTW), IEEE Fourth Intl. Conf. on.* IEEE, 2011.

[14] J. A. Espinosa, S. A. Slaughter, R. E. Kraut, and J. D. Herbsleb. Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, 24(1):135–169, 2007.

[15] S. Faraj and L. Sproull. Coordinating expertise in software development teams. *Management science*, 46(12):1554–1568, 2000.

[16] M. Fowler. Refactoring: Improving the design of existing code. In *11th European Conf.. Jyväskylä, Finland*, 1997.

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

[18] 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*, 31(10), 2005.

[19] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi. Identification of defect-prone classes in telecommunication software systems using design metrics. *Information sciences*, 176(24), 2006.

[20] M. Jureczko and D. Spinellis. Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, 2010.

[21] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3), 2012.

[22] 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*, 1(3), 1988.

[23] 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.

[24] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2), 1993.

[25] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[26] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE Intl. Conference on*, pages 350–359. IEEE, 2004.

[27] M. Martin and R. C. Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, 2006.

[28] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[29] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4), 1976.

[30] 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*, 36(1), 2010.

[31] M. J. Munro. Product metrics for automatic identification of" bad smell" design problems in java source-code. In *Software Metrics, 11th IEEE Intl. Symposium*. IEEE, 2005.

[32] S. Nakagawa. A farewell to bonferroni: the problems of low statistical power and publication bias. *Behavioral Ecology*, 15(6), 2004.

[33] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Software Maintenance and Evolution (ICSME), IEEE Intl. Conf. on*. IEEE, 2014.

[34] 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*, pages 161–170. IEEE, 2015.

[35] T. V. Perneger. What's wrong with bonferroni adjustments. *BMJ: British Medical Journal*, 316(7139), 1998.

[36] R. Peters and A. Zaidman. Evaluating the lifespan of code smells using software repository mining. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012.

[37] PMD. Pmd. http://pmd.github.io.

[38] 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.

[39] R. Shatnawi and W. Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of systems and software*, 81(11), 2008.

[40] D. J. Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[41] Sonarqube. Sonarqube. http://www.sonarqube.org/.

[42] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[43] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1945.

[44] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2), 2000.

[45] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE, 2013.