Context-Based Code Quality Assessment

Maurício Finavaro Aniche

Thesis presented
to the
Institute of Mathematics and Statistics
of
University of São Paulo
to
the title
of
Ph.D. in Computer Science

Program: Ph.D. in Computer Science

Supervisor: Marco Aurélio Gerosa, Ph.D.

São Paulo, July 2016

Context-Based Code Quality Assessment

This thesis corresponds to the final version
argued by Maurício Finavaro Aniche.

Acknowledgments

It's true – writing acknowledgments is not easy. So many people are responsible for this PhD, and hopefully I won't forget any of them.

First of all, I thank my mother and my father. They are the main contributors to this PhD, as they supported me since day zero. Oh, thanks for the BASIC programming book you gave me when I was 9!

I thank all my colleagues at USP and, especially, Gustavo Oliva, Igor Steinmacher, and Igor Wiese. It has been seven years, guys. Thanks for all the meetings and insightful ideas.

I thank my coworkers at Caelum. In particular, Guilherme Silveira and Alberto Souza. All of our chats were very inspiring to my research. You guys are the best programmers I know.

I thank Anderson Leite and Alberto Souza for being such good friends. Without their psychological support, I am not sure if I would finish this PhD.

To everyone who wrote a paper with me. Writing is the hardest part for me, and I learned a lot from you. Thanks, Christoph Treude, Gabriele Bavota, Arie van Deursen, Andy Zaidman, Eduardo Guerra, and Fábio Petrillo.

I thank Arie and Andy for kindly accepting me as a visiting student at TU Delft. It was the best experience of my life. I now love TU Delft and The Netherlands. I will be back soon.

I thank my advisor, Marco Aurélio Gerosa. We worked together for seven years (masters and PhD), and I owe this accomplishment to him. I hope we work together again in the future.

I thank all developers that participated in my studies. In particular, developers from Bluesoft and André Faria, its team leader, who made it possible.

Finally, I thank my wife Laura, who strongly supported me in this last (and hardest) part of my PhD. She made me remember how much I enjoy being in a university. And I thank her for supporting me in my next (crazy) adventure.

Abstract

Context-Based Code Quality Assessment

Two tasks that software engineers constantly perform are writing code that is easy to evolve and maintain, and detecting poorly written pieces of code. For the former, software engineers commonly rely on well-known software architecture styles, such as Model-View-Controller (MVC). To the latter, they rely on code metrics and code smell detection approaches. However, up to now, these code metrics and code smell approaches do not take into account underlying architecture—all classes are assessed as if they were the same. In practice, software developers know that classes differ in terms of responsibilities and implementation, and thus, we expect these classes to present different levels of coupling, cohesion, and complexity. As an example, in an MVC system, Controllers are responsible for the flow between the Model and the View, and Models are responsible for representing the system's business concepts. Thus, in this thesis, we evaluate the impact of architectural roles within a system architecture on code metrics and code smells.

We performed an empirical analysis in 120 open source systems, and interviewed and surveyed more than 50 software developers. Our findings show that each architectural role has a different code metric values distribution, which is a likely consequence of their specific responsibilities. Thus, we propose SATT, an approach that provides specific thresholds for architectural roles that are significantly different from others in terms of code smells.

We also show that classes that play a specific architectural role contain specific code smells, which developers perceive as problems, and can impact class' change- and defect-proneness. Based on our findings, we suggest that developers understand the responsibilities of each architectural role in their system architecture, so that code metrics and code smells techniques can provide more accurate feedback.

Keywords: software engineering, code metrics, code smells, architectural roles, software maintenance, software architecture, code quality.

Resumo

Avaliação de Qualidade de Código Baseada em Contexto

Duas tarefas que desenvolvedores de software constantemente fazem são escrever código fácil de ser mantido e evoluído, e detectar pedaços de código problemáticos. Para a primeira tarefa, desenvolvedores comumente fazem uso de conhecidos padrões arquiteturais, como Model-View-Controller (MVC). Para a segunda tarefa, desenvolvedores fazem uso de métricas de código e estratégias de detecção de maus cheiros de código (*code smells*). No entanto, até o momento, métricas de código e estratégias de detecção de maus cheiros de código não levam em conta a arquitetura do software em análise. Isso significa que todas classes são avaliadas como se umas fossem iguais às outras. Na prática, sabemos que classes são diferentes em suas responsibilidades e implementação, e portanto, esperamos que elas variem em termos de acoplamento, coesão e complexidade. Por exemplo, em um sistema MVC, Controladores são responsáveis pelo fluxo entre a camada de Modelo e a camada de Visão, e Modelos representam a visão de negócios do sistema. Nesta tese, nós avaliamos o impacto dos papéis arquiteturais em técnicas de medição de métricas de código e de detecção de maus cheiros de código.

Nós realizamos um estudo empírico em 120 sistemas de código aberto, e entrevistamos e realizamos questionários com mais de 50 desenvolvedores. Nossos resultados mostram que cada papel arquitetural possui distribuições diferentes de valores de métrica de código, consequência das diferentes responsabilidades de cada papel. Como consequência, propomos SATT, uma abordagem que provê *thresholds* específicos para papéis arquiteturais que são significantemente diferentes de outros em termos de métricas de código.

Mostramos também que classes que cumprem um papel arquitetural específico também contêm maus cheiros de código específicos. Esses maus cheiros são percebidos por desenvolvedores como problemas reais e podem fazer com que essas classes sejam mais modificadas e apresentem mais defeitos do que classes limpas. Sugerimos então que desenvolvedores entendam a arquitetura dos seus sistemas, bem como as responsabilidades de cada papel arquitetural que as classes desempenham, para que tanto métricas de código quanto estratégias de detecção de maus cheiros de código possam prover um melhor retorno.

Palavras-chave: engenharia de software, métricas de código, maus cheiros de código, papéis arquiteturais, manutenção de software, arquitetura de software, qualidade de código.

Contents

List of Figures

List of Tables

Take time to deliberate;
but when the time for action arrives,
stop thinking and go in.

— Napoleon Bonaparte

Chapter 1

Introduction

"One man's bad code is another man's full time job" is a popular saying among software development teams. Indeed, maintaining a software system can be challenging. Software developers thus commonly rely on well-known architectural styles and design patterns [2, 101, 117] as building blocks. For instance, the popular Model-View-Controller (MVC) pattern [67] suggests separating the code into three different layers: Models, Views, and Controllers. Each layer has its own responsibility: classes that play the CONTROLLER role are responsible for coordinating the process between the VIEW and the MODEL, while classes that play the MODEL role represent business concepts. Hence, this separation guides developers towards classes that are more specific and cohesive, which facilitates maintenance.

However, although architectural styles and patterns can help developers mitigate the decay of quality, in practice it is hard to keep all classes "clean". Thus, to detect bad pieces of code, developers rely on code metrics and code smell detection tools, such as PMD [96], Sonarque [110], and JDeodorant [113]. These tools deliver a report of "good" and "bad" classes. As an example, to point to highly coupled classes, many tools make use of the CBO (Coupling Between Objects) metric, which is part of the CK suite [24]. In a nutshell, this metric counts a class' number of dependencies: the higher the metric value, the more coupled the class. In order to determine whether a class should be considered problematic, these tools make use of a threshold; if the value is higher than a pre-defined one, then the tool reports that class as problematic. The same happens when detecting code smells. For instance, to detect a God Class [68], which "does" or "know" too much [46], tools use a combination of different code metrics. If all metrics are above (or below) the pre-defined thresholds, the class is considered smelly.

As we discuss throughout this thesis, there is room for improvement in both current code metric assessments and code smell detection techniques. In practice, each class has a responsibility within a system. For instance, as already noted, in an MVC system, classes play different roles, such as CONTROLLERS, which are responsible for controlling the flow between models and views, or MODELS, which are responsible for representing the business concepts of the system; in Android applications, classes can play the ACTIVITY and FRAGMENT roles to present the user interface (UI) and respond to events, such as the user touching the screen.

This raises interesting questions related to how software developers deal with code metrics, such as: "Do developers deal with coupling in Controllers the same way as they do in Models?" or "Do developers deal with coupling in Activities the same way as they do in Fragments?". If the answer to these questions is "no", then we can improve tools and techniques that use the same threshold values to blame a class, *no matter the architectural role a class plays in the system.*

Adding context to code metrics is an emerging topic among researchers. Zhang *et al.* [128] showed that metric values can be affected by factors such as programming language, age, and lifespan. Recently, Gil and Lalouche [50] argued that metric values vary among projects, and "mean nothing when examined out of their context". Bouwers *et al.* [19] also warned the community about using a metric without a proper interpretation: the "metric in a bubble" pitfall, as they call it,.

The same conclusions can be drawn when thinking about code smells. As an example, the aforementioned God Class smell; while it fits well in any object-oriented system, does not take into account the underlying architecture of the application or the role played by a given class. Again, back to MVC's CONTROLLER classes: these classes commonly represent an endpoint for other classes, do not contain state, and manage the control flow. Besides being possibly affected by "traditional smells" (*e.g.*, God Classes), good programming practices suggest that CONTROLLERS should not contain complex business logic and should focus on a limited number of services offered to the other classes. Similarly, DATA ACCESS OBJECT (DAO) classes [47], another common architectural role in software systems, are responsible for dealing with communication to the databases. These classes, besides not containing complex or long methods (traditional smells) should also limit the complexity of their SQL queries.

Indeed, traditional code smells capture very general principles of good design. However, we suggest that specific types of code smells, such as the aforementioned ones, are necessary for capturing "bad practices" on software systems adopting a particular architecture. Other researchers have proposed smells and best practices for specific technologies or platforms, such as object-relational mapping frameworks [22], Android apps [55, 116], Cascading Style Sheets (CSS) [75], and formulas in spreadsheets [56].

In this thesis, we analyze the impact of considering the architectural role in code metrics and code smells techniques. This work is divided into two main sections: in the first one, we analyze how code metric value distribution varies among different architectural roles; we also capture developers' perceptions of architectural roles and code metrics. In the second part, we propose and analyze code smells that are specific to a particular architectural style (Web MVC systems).

Our results show that, indeed, architectural roles have an impact on code metrics and code smells analysis. In terms of code metrics, we found that each architectural role has a different metric values distribution (*e.g.*, a CONTROLLER is usually more coupled than other roles), which developers do not clearly perceive. In terms of code smells, we found that the six specific smells that we propose (*e.g.*, PROMISCUOUS CONTROLLER, a smell that only matters to CONTROLLER classes) can be as problematic as a traditional smell. We also found that developers perceive classes affected by specific smells as problematic. As a practical implication of our research, we suggest developers to understand the responsibilities of each architectural role in their system architecture, and make use of this knowledge to better assess the quality of each class in the system.

1.1 Motivational Example

To illustrate our point, we present an example. SSP, or the Student Success Plan[1], is a tool that enables students to create plans for completing 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 [95], which means it contains CONTROLLERS to handle

[1]http://www.studentsuccessplan.org/.

the requests, Entities to represent the business model, Services to implement business rules, and Repositories responsible for persisting the data in a database.

A concept that exists in SSP is “Plan”[2], wherein 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 [47]) 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 [47]. 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 8. These architectural roles are also different in terms of complexity: the third quartile of McCabe’s complexity [77] in all Entities is 17, while for Repositories the same number is 7.

Suppose we want to detect coupling issues, and to that end we use a threshold-based assessment technique. As the threshold is defined by looking at the distribution of classes in several systems [3, 42], it will probably be too low for Controllers (as they usually have a higher number) and too high for Entities (as they are usually less coupled). As a consequence, some “normal” Controllers will be reported as problematic, while some “outlier” Entities classes will not be reported at all.

As this exemplifies, different architectural roles have different metric value distributions. Why does this happen? Can the differences be explained by the fact that architectural roles vary in their responsibilities?

1.2 Research Questions

This thesis is divided into two main sections: the first one analyzes the impact of architectural roles on code metric techniques, while the second part focuses on the impact of architectural roles on code smells.

In the first part, we aim to answer the following question:

What is the impact of architectural roles on code metric analysis?

One approach to assess the quality of a class is through *code metrics*. Code metrics have been proven useful (i) to determine issues in object-oriented design [16, 34, 70, 102], and (ii) as indicators of a class’ defect- [20, 53, 79, 105] or change-proneness [36, 121]. These techniques make use of the code metric values in order to decide whether a class is problematic and needs to be inspected by

[2]Classes discussed in this section were extracted on September, 17th, 2015. The code snapshot of that date can be seen at https://github.com/Jasig/SSP/tree/b3546fc8983e9825f7395e50a02e4e54d3c384d2.

a developer. Popular industry tools, such as Sonar [110], PMD [96], and JDeodorant [113], rely on thresholds to distinguish between "good" and "bad" classes. These thresholds are usually calculated after extensive analysis of many software systems, similar to benchmarking [3, 14]. Indeed, defining thresholds is not straightforward [25, 57, 58, 126], and different approaches have been proposed [3, 35, 42, 104] over the years.

We explore the following research questions in this part:

RQ1. What differences in metric values distributions does SATT find for common architectural styles such as MVC?

As first contribution of this thesis, we present SATT, an approach to derive specific thresholds for architectural roles that are significantly different from others. In this RQ, we analyze whether SATT approach indeed finds architectural roles that are different from others, thus deserving a specific threshold.

RQ2. Can the differences in distributions thus found be explained from the architectural constraints imposed on classes fullfilling dedicated architectural roles?

For this question, we investigated whether developers were aware of how different architectural roles are from each other. With the help of 3 Spring experts, we obtained a qualitative perspective of the differences among architectural roles.

RQ3. What impact do these differences have on the use of thresholds for quality assessments and smell detection?

In this RQ, we compare the results of our approach against a state-of-the-art approach.

RQ4: Do developers share a common perception on the object-oriented design aspects in their system's architectural roles?

The developer's perception is always important when dealing with software maintenance [13, 88, 123]. Do developers experience 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 this question, we relied on "card interviews" with 17 professional developers. During the interviews, developers were asked to explicitly compare the behavior of classes in their system.

RQ5: Do developers' perceptions match the code metric analysis of their own project?

For this question, we investigated whether their perception matched with the analysis of their own system's source code. To that end, we transformed their opinions into mathematical expressions (*e.g.*, *Controller Coupling* > *Entities*) and matched these expressions with the analysis of CK code metrics [24] extracted from their own project.

In the second part of this thesis, we answer the following question:

What is the impact of architectural roles on code smell analysis?

Code metrics are used in many other techniques. When combined, code metrics can identify high-level code smells. As examples, Marinescu [73] and Lanza and Marinescu [68] combine different metrics, thresholds, and logical operators to detect different smells, such as God Classes, Feature

Envy, and Blob Classes. Researchers have also shown that smelly classes are more prone to change and defects than other classes [65, 71].

Thus, we explored the following questions:

RQ6. Are there code smells that are specific to an architectural role?

As we showed in the previous RQs, architectural roles differ from each other in terms of responsibilities. Thus, for this question, we investigated the existence of smells that are specific to some architectural roles, *e.g.,* CONTROLLERS or REPOSITORIES. To answer this question, we relied on surveys and interviews with 53 software developers. We also developed detection strategies [68, 73] for each of the proposed smells.

RQ7. What is the relationship between the proposed code smells and the classes' change-proneness?

Previous studies have shown that the "traditional smells" (*e.g.*, Blob Classes) [46] can increase class change-proneness [64, 65]. This research question aimed to investigate the impact of the code smells that we proposed in the prior RQ on change-proneness of classes.

RQ8. What is the relationship between the proposed code smells and the classes' defect-proneness?

This research question mirrors RQ6. Traditional smells are also known by their impact on the defect-proneness of classes [64, 65]. Thus, we compared the impact of the six defined smells on classes' defect-proneness.

RQ9. Do developers perceive classes affected by the proposed code smells as problematic?

We qualitatively complement the quantitative analysis performed in the context of RQ6 and RQ7. Here, we investigated whether classes affected by the defined Web MVC code smells are perceived as problematic by developers.

We made use of different research methods throughout this thesis. Therefore, each research method appears in the research question's respective chapter. All chapters require readers to know more about code metrics, code smells, and architectural roles. We present an overview of the background of this research in Chapter 2.

1.3 Contributions

The main contributions of this thesis are (in order of appearance):

1. An approach, namely SATT, that derives specific thresholds for architectural roles that present code metric distributions significantly different from others.
2. A study on how metric value distributions behave among different architectural roles. We analyzed 7 different code metrics (all in the CK suite [24]) in more than 100 Spring MVC systems.
3. A reasoning for the differences in code metrics among architectural roles. Reasons were collected after interviews with 3 experts.

4. A study of developers' perceptions of 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.
5. An approach (namely "card interviews") to capturing developers' perceptions on the object-oriented design aspects of the architectural roles in their system.
6. A validated catalogue of smells affecting Web applications relying on the MVC pattern. This catalogue has been defined by means of an open coding process after surveying and interviewing 53 software developers.
7. Detection strategies for each of the catalogued smells. We followed Lanza and Marinescu's approach [68] to propose detection strategies.
8. An empirical study on the impact of the catalogued smells on change- and defect-proneness of classes. We evaluated the impact of each catalogued smell in 100 Spring MVC projects.
9. A survey on developers' perception of the smells. We performed a survey with 21 software developers and captured their perceptions on the catalogued smells.
10. An open source tool, namely Springlint [7], described in Appendix B, that implements the findings of this thesis.

1.3.1 Other contributions

The author of this research also made other contributions during his PhD programme (2012-2016), which are listed below.

Publications of this thesis

1. Aniche, M, Bavota, G., Treude, C., van Deursen, A., Gerosa, M. "A Validated Set of Smells in Model-View-Controller Architecture". Software Maintenance and Evolution (ICSME), 2016, 32th International Conference, IEEE, 2016.
2. Aniche, M., Treude, C., Gerosa, M. "Developers' Perceptions on Object-Oriented Design and System Architecture". 30th Brazilian Symposium on Software Engineering (SBES), 2016.
3. Aniche, M., Treude, C., Zaidman, A., van Deursen, A., Gerosa, M.A., "SATT: Tailoring Code Metric Thresholds for Different Software Architectures". 16th IEEE International Working Conference on Source Code Analysis and Manipulation, 2016.
4. Aniche, Maurício F., Gustavo A. Oliva, and Marco A. Gerosa. "Are the Methods in Your Data Access Objects (DAOs) in the Right Place? A Preliminary Study.". 6th Workshop on Managing Technical Debt, in conjunction with ICSME2014, Canada.

List of other publications

1. Aniche, M., Gerosa, M. "Boas e Más Práticas no Desenvolvimento Web com MVC: Resultados de Um Questionário com Profissionais". 3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015).
2. Aniche, M., Oliva, G., Gerosa, M. "Why Statically Estimate Code Coverage Is So Hard? A Report of Lessons Learned". 29o Simpósio Brasileiro de Engenharia de Software (SBES2015).
3. Aniche, M., Gerosa, M. "Does test-driven development improve class design? A qualitative study on developers' perceptions". Journal of the Brazilian Computer Society.2015, 21:15.

4. Aniche, Maurício F., Sokol, Francisco Z. "Efeitos da Prática de Revisão de Código na Caelum: Um Estudo Preliminar em Duas Equipes". Workshop Brasileiro de Métodos Ágeis, 2014.

5. Aniche, M., Guerra, E., Gerosa, M. "A Set of Patterns to Improve Code Quality of Automated Functional Tests of Web Applications". 21th Conference on Pattern Languages of Programs. 2014.

6. Aniche, Mauricio Finavaro, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. "What Do the Asserts in a Unit Test Tell Us about Code Quality? A Study on Open Source and Industrial Projects." Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. IEEE, 2013.

Cooperation with other authors

1. Guerra, E., Aniche, M. "Achieving quality on software design through test-driven development". In: Ivan Mistrik; Richard Soley; Nour Ali; John Grundy; Bedir Tekinerdogan. (Org.). Software Quality Assurance in Large Scale and Complex Software-Intensive Systems. 1ed.Waltham, USA: Morgan Kaufmann - Elsevier, 2015, v. , p. 201-220.

2. Gerosa, M., Wiese, I., Oliva, G., Aniche, M. "Mineração de Repositórios de Software Livre. Revista Computação Brasil", ed 27, 02/2015.

3. França, D., Aniche, M., Guerra, E. "Como o Formato de Arquivos XML evolui? Um Estudo sobre sua Relação com Código-Fonte". 3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015).

4. Guerra, E. Aniche, Maurício, Gerosa, M., Yoder, J. "Preparing for a Test Driven Development Session". 21th Conference on Pattern Languages of Programs. 2014.

5. Guerra, E., Yoder, J. Aniche, M., Gerosa, M. "The Last Two Test-Driven Development Step Patterns: Bug Locator and Pause for Housekeeping." 10th Latin American Conference on Pattern Languages of Programs. 2014.

6. Guerra, E., Yoder, J. Aniche, M., Gerosa, M. "Test-Driven Development Step Patterns For Designing Objects Dependencies". 20th Conference on Pattern Languages of Programs, 2013.

7. Sokol, Francisco Zigmund, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. "Does the Act of Refactoring Really Make Code Simpler? A Preliminary Study." 4th Brazilian Workshop on Agile Methods. 2013. Best Paper Sward.

8. Sokol, Francisco Zigmund, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. "MetricMiner: Supporting researchers in mining software repositories.". Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on. IEEE, 2013.

Books

1. Test-Driven Development: Teste e Design no Mundo Real. Casa do Código, 2012. Versions: Java, C#, Ruby, PHP.

2. Orientação a Objetos e SOLID para Ninjas: Projetando Classes Flexíveis. Casa do Código, 2015. Versions: Java.

3. Testes Automatizados de Software: Um Guia Prático. Casa do Código, 2015. Versions: Java.

4. Introdução à Programação em C: Os primeiros passos de um desenvolvedor. Casa do Código, 2015.

Awards

1. Best paper award in WBMA (Brazilian Workshop on Agile Methods) 2013.
2. Mining Software Repository Challenge in PASED Canadian Summer School, 2012.

Advising Students

1. Francisco Sokol. Metric Miner: Uma ferramenta web de apoio à mineração de repositórios de Software. University of São Paulo, 2012. Co-advisor.
2. Weliton Andrade Martineli. A evolução de JSP's em projetos web Java: Um estudo em projetos de código aberto. Mackenzie University, 2015.
3. Leonardo Generoso. Padrão evolutivo de arquivos CSS: um estudo em projetos de código aberto utilizando mineração de repositório de software. Mackenzie University, 2015.

Service to the Community

1. Experience Report Chair at OSS (International Conference on Open Source Systems) in 2014.
2. WBMA (Brazilian Workshop on Agile Methods) program committee in 2013, 2014, 2015.
3. PLoP (Pattern Languages of Programming) program committe in 2015, 2016.
4. Reviewer for SBES 2016, ICPC 2016, Journal of Empirical Software Engineering 2016.

Tools

1. MetricMiner2: http://www.github.com/mauricioaniche/metricminer2.
2. CK: http://www.github.com/mauricioaniche/ck.
3. SmellyCat: http://www.github.com/mauricioaniche/smellycat.
4. SpringLint: http://www.github.com/mauricioaniche/springlint.

1.4 Structure of the Dissertation

This thesis is divided as follows:

- In Chapter 2, we provide an overview of the background of this research. In particular, we present the notion of code metrics, code smells, architectural roles, and the Spring MVC architecture. We also present related work that we built on.
- In Chapter 3, we answer RQs 1, 2, and 3 by proposing the SATT approach and analyzing the metric value distributions in more than 120 open source projects and interviewing software developers and experts.
- In Chapter 4, we answer RQs 4 and 5 by performing a qualitative study with 17 software developers from a Brazilian software development company.
- In Chapter 5, we answer RQ 6 by coining six smells specific to some architectural roles after interviewing and surveying 53 software developers. We also answer RQs 7, 8, and 9, by analyzing the change- and defect-proneness of the proposed smells in 100 Spring MVC systems. We also surveyed 21 software developers and analyzed their perceptions on classes affected by the proposed smells.
- In Chapter 6, we revisit our research questions, provide some clarity on the implications of our findings to software developers, and offer suggestions for future work.

Chapter 2

Background

In this chapter, we present the three main concepts used throughout this research. In Section 2.1, we define the term "architectural role" and describe the Spring MVC system's architecture that we use as case study; in Section 2.2, we present the CK code metrics suite, as well as the implementation details of our analysis tool; in Section 2.3, we present code smells, as well as how their detection strategies work. Finally, in Section 2.4, we present work related to this thesis.

2.1 Architectural Roles

We define "architectural role" as a particular role that classes play in an architectural style. When a class plays an architectural role, its task is well-defined, and classes usually only focus on that task. As an example, CONTROLLERS in Spring MVC applications coordinate the flow between the user interface and the domain layer, and ACTIVITIES represent a user interface in an Android system.

We chose Spring MVC as the architecture to study throughout this thesis; it is a Java framework that supports developers in building web applications. As its name states, the framework requires developers to use the Model-View-Controller (MVC) pattern [67]. In addition, the framework provides several functionalities to facilitate developers' daily work, such as dependency injection and data validation helpers.

We had several reasons for selecting Spring MVC: (i) it has well-defined architectural roles, (ii) it is frequently used (in a survey with more than 2,000 respondents [115], Spring MVC was used by 40% of developers that use a web framework), and (iii) our industry contacts make use of Spring MVC to develop their systems.

A Spring MVC application must have classes playing different architectural roles, according to its official documentation [94]:

- ***Controller***: Controls the flow between the domain layer and the view layer. In a Spring MVC application, classes annotated with `@Controller` are considered CONTROLLERS by the framework. Commonly, developers create one CONTROLLER for each domain concept in the application, *e.g.*, USERCONTROLLER, which is responsible for dealing with users, or INVOICECONTROLLER, which is responsible for dealing with invoices.
- ***Entity***: Represents a lightweight persistence domain object. Examples of ENTITIES are *User* or *Invoice*. These classes usually contain several attributes and methods (of which many are getters and setters). In Spring MVC, they must be annotated with `@Entity`.
- ***Service***: Offers an operation that stands alone in the model, with no encapsulated state. SERVICES are commonly implemented when a business rule is related to more than one ENTITY. In

Spring MVC, classes need to be annotated with `@Service`.

- ***Repository***: Encapsulates persistence and retrieval, by emulating a collection of objects. These classes usually make use of a persistence framework, such as Hibernate [54], and contain several SQL/JPQL queries. REPOSITORIES are annotated with `@Repository`.
- ***Component***: Represents an isolated component in the application. Practical examples are UI formatting or data conversion. Internally, Spring MVC considers REPOSITORIES and SERVICES as special kinds of components. COMPONENTS are annotated with `@Component`.

As one can see, these roles go beyond the definition of the MVC pattern. Indeed, while MVC is defined in terms of more high-level concepts, in practice, implementations of MVC contain more details (and roles), such as we described above. To the rest of this thesis, whenever we mention MVC, we are refering to the MVC implementation of Spring MVC (which can easily be mapped to many other implementations, such as Ruby on Rails, ASP.NET MVC, etc).

Implementation details. As aforementioned, classes that play any of these specific architectural roles in Spring MVC are annotated with a specific annotation, *i.e.*, `@Controller`, `@Entity`, `@Service`, `@Repository`, and `@Component`. Thus, by means of static code analysis, we consider a class to play one of these architectural roles when it contains one of these specific annotations.

2.2 Code Metrics

A code metric measures a specific code attribute. A simple example would be the count of the number of lines of code per class, which is a good predictor for complexity issues [106]. Throughout this thesis, we rely on the Chidamber & Kemerer (CK) metrics suite [24], as: (i) it covers different aspects of object-oriented programming, such as coupling (CBO, RFC), cohesion (LCOM), inheritance (DIT, NOC), and complexity (WMC, NOM); (ii) it has already proven its usefulness in earlier studies [16, 53, 70]; and (iii) the studied Spring MVC architecture is object-oriented.

The CK suite consists of the following class level metrics. Different tools implement different versions of these metrics [3]. Thus, we also report the implementation details of the code metric tool we used.

Number of Methods (NOM). The number of methods in a class. It is considered a measure of complexity.

Implementation details. Our tool counts any method regardless of its visibility modifier (*i.e.*, `public, private, protected,` and `default`) or scope (*i.e.*, inside the current class or inside an inner class).

Weighted Methods Per Class (WMC). It is represented by the sum of McCabe's cyclomatic complexity [77] for each method in the class. The McCabe's number for a method is, in a nutshell, the count of branch instructions (*e.g.*, `if, for, while`) in the class plus one. WMC is another metric for measuring complexity.

Implementation details. Our tool calculates McCabe's number in class level. Thus, it adds one for any method declaration, `for, enhanced for, if, do/while, while, switch, catch, && (and), || (or)`, and `static constructor` found.

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

Implementation details. It recursively finds all the super classes from a class. The algorithm stops when `Object` is found. In Java, all classes inherit from `Object`. Thus, a class that inherits from `Object` has a DIT=1. A class `X` that inherits from `Y` that inherits from `Object` has DIT=2.

Number of Children (NOC). The number of direct sub-classes a class has. Suppose a class X that inherits from Y (X is a child of Y). Thus, NOC(X)=0 while NOC(Y)=1.

Implementation details. As JDT does not internally track the sub-classes of a class, we map all classes and their super classes (used to count the DIT metric), and then calculate each class' NOC.

Coupling Between Object Classes (CBO). The number of classes a class directly depends upon. It counts classes used from both external libraries as well as classes from the project.

Implementation details. Our tool counts the number of classes by storing the full name of all types that are used in the target class in a set. To that end, the tool analyzes all type declarations that can occur, such as variable or field declaration, and return types. The tool ignores types that belong to `java.*` and `javax.*` packages, as they belong to the Java language itself.

Response for a Class (RFC). It is the count of all method invocations made in a class.

Implementation details. Our tool counts all method invocations that happen in the target class. However, it ignores repetitions, *i.e.*; if a method `a()` is invoked two or more times in the target class, the tool counts it as a single invocation. Overloaded methods are counted as different methods.

Lack of Cohesion of Methods (LCOM). This cohesion metric measures the lack of method cohesion. The higher the number, the less cohesive the class. Thus, the algorithm counts the number of method pairs whose similarity in terms of used fields is zero, minus the number of method pairs whose same similarity is not zero. Using mathematical expression, we define LCOM as $|P| - |Q|$ if $|Q| > 0$, where P = set of all empty set intersections and Q = set of all nonempty set intersections.

Implementation details. Our tool performs the algorithm in the way we describe above. We take all methods and fields into account, regardless of their visibility.

2.3 Code Smells

Code smells are symptoms of poor design and implementation choices [21, 46]. Many authors have worked on definitions of different code smells. As examples of code smells, we list two: *God Classes* [46], which are classes that do or know too much, and *Feature Envy* [46], which are methods that are more interested in other classes than in their own class. Many studies show that code smells impact the change- and defect-proneness of a class [65], and can be introduced in source code during different software development tasks [114].

Throughout this thesis, we focus on how to define and detect smells that are specific to an architectural role. We rely on Marinescu's [73] and Lanza and Marinescu's [68] approaches to define a detection strategy for code smells, which we detail in this section. In their approach, metrics combine with logical operators, such as {AND, OR}. If the metric values for a given class match the detection strategy, that class is considered smelly. According to them, the goal of the approach is to provide a more abstract way to deal with code metrics, conceptually closer to the real intentions than developers have when using code metrics.

A detection strategy is composed by filter and composition mechanisms. A common example of a filter is: "a class should not have its cyclomatic complexity higher than 6". We can note that the filter is composed by a metric (cyclomatic complexity), a direction (higher than), and a threshold value (6). A detection strategy can be composed by many of these filters. Filters are connected by means of different logical operators, such as AND and OR. Thus, a piece of code is selected by the detection strategy only if it passes through all these filters.

Lanza and Marinescu [68] developed extensive work on heuristics to detect code smells in object-oriented systems. To exemplify, we provide the same *God Class* example Marinescu uses in the paper that introduces detection strategies [73] – a paper that, until 2015, was the most cited paper in the history of ICSME, the International Conference on Software Maintenance, the venue in which it was published.

God Class is the name given to a class that is known as "the one that does everything in the system" [99]. Riel [99] provides three different heuristics to detect classes with this smell:

- Top-level classes in a design should share work uniformly.
- Beware of classes with much non-communicative behavior.
- Beware of classes that directly access data from other classes.

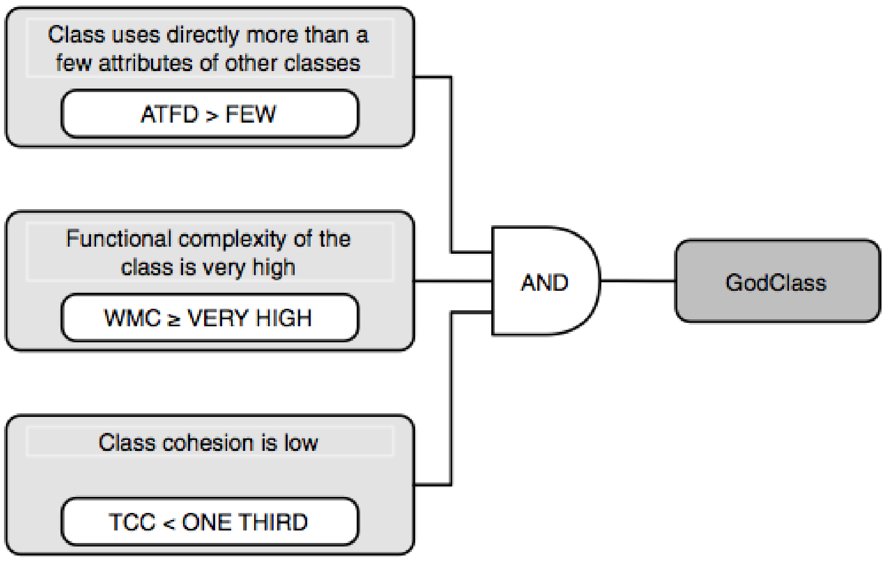

Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

Figure 2.1: *God Class Detection Strategy, extracted from Lanza and Marinescu's book [68]*

According to Marinescu [73], the first step to building a detection strategy is to transform informal heuristics in a set of symptons that can be captured by a code metric. Then, the next step is to select these metrics. To detect God Classes, he selected the following:

- *Weighted Method Count (WMC).* As we discussed in Section 2.2, it is the sum of each methods' complexity.
- *Tight Class Cohesion (TCC).* It is the relative number of methods that are directly connected [18]. This metric is similar to LCOM (discussed in Section 2.2), the goal of which is to measure a class' cohesion.
- *Access to Foreign Data (ATFD).* Represents the number of external classes from which a given class accesses attributes, directly or via accessor methods [72].

Since one of the symptoms is “the high complexity of a class”, the filter for WMC is “higher than”. For low cohesion, he chose “lower than” TCC. Finally, he used “higher than” for the ATFD metric. He chose to use “very high”, “one third”, and “few”, respectively, as thresholds. To calculate these thresholds, different approaches can be used, and we discuss them in Section 2.4.1. Authors make use of both empirical benchmarking in systems and experience to determine these thresholds. In addition, these filters are connected by the logical operator `AND`. In Figure 2.1, extracted from the author’s book, we show the full detection strategy of God Classes.

To exemplify other code smells, in the following we present a description of different existing code smells, such as listed by Palomba et al. [89]:

- **Divergent Change.** A class is changed in different ways for different reasons.
- **Shotgun Surgery.** A change to the affected class (i.e., to one of its fields/methods) triggers many little changes to several other classes.
- **Parallel Inheritance.** Every time you make a subclass of one class, you also have to make a subclass of another.
- **Blob.** A class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes.
- **Feature Envy.** A method is more interested in another class than the one it is actually in.

2.4 Related Work

This section is divided as follows: in Section 2.4.1, we present code metrics and what researchers have done in thresholds’ definition; in Section 2.4.2, we describe related work on code smells and how researchers have evaluated their impact on code quality; in Section 2.4.3, we show studies related to our definition of architectural role; in Section 2.4.4, we present studies on code quality of web systems; finally, in Section 2.4.5, we discuss how our work relates to and fits in the literature.

2.4.1 Code Metrics

Tools, such as PMD [96] and Sonar [110], rely on thresholds to determine whether a developer needs to be alarmed about a class. However, defining these thresholds is not simple. In his work, McCabe [77] not only defined the Cyclomatic Complexity metric (used in this thesis 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 [68], and others actually did it [26, 83], 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.* [35], for example, propose mean and standard deviation as a way to find thresholds. On the other hand, many studies show that distributions of code metric values are usually non-normal. Concas *et al.* [27] measured 10 different properties related to classes, methods, and the relationships among 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 [17, 25, 57, 98, 119, 126]. Regarding

CK metrics, Herraiz *et al.* [58] found that WMC, CBO, and RFC are double Pareto distributions, while NOC and LOC follow power law. DIT was the only one that could not be described by either log-normal or power law.

A common approach to define thresholds is to perform benchmarking [3, 14]. Researchers collect a large number of systems, analyze the metric values, and define the threshold that would point to the "worst" classes. Alves *et al.* [3] proposed an approach based on a weighted function. Using lines of code as weight, they select the code metric values relative to the 70%, 80%, and 90% percentiles of the accumulated weight, and uses these as thresholds. They also improved the work to include the calibration of benchmark-based thresholds from code to system level [4].

Other authors also attempted to improve the choice of the thresholds. Fontana *et al.* [45] 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 [85], 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.

In addition, several 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 [20, 53, 61, 62, 79, 105], while WMC, CBO, and RFC have been to change-proneness [15, 36, 121]. D'Ambros *et al.* [32] show that different learning algorithms can have a good performance predicting defect-proneness when using CK metrics.

On the other hand, none of aforementioned techniques take the architectural role of the class into account. Thresholds are used for all classes in the system regardless of their responsibility. Indeed, the evaluation of the context has been an important discussion in the code metrics field, as other authors such as Gil and Lalouche [50] and Zhang *et al.* [128] have shown that metric values are sensitive to contextual variables, such as programming language, age, and lifespan of a project.

2.4.2 Code Smells

Webster's book [118] may be the first one in which the term *code smells* was used to refer to bad practices. Long methods and excessive complexity are examples. Since then, many other researchers and practitioners have defined catalogues of code smells. As example, Riel [99] has defined more than 60 different characteristics of good object-oriented code, and Fowler [46] suggests refactorings in more than 20 different code issues. Smells such as *God Classes*, *Feature Envy*, and *Blob Classes* are popular among practitioners and popular tools in industry, such as PMD and Sonar, attempt to detect them.

Interestingly, Tufano *et al.* [114], after an empirical study in more than 200 open source projects, showed that most code smells are introduced during the creation of the file (and not introduced during maintenance phase, as one could suppose), and that code smells can be introduced even during refactoring tasks.

Researchers evaluated the impact of these smells in terms of code quality. As a first step to identify smelly classes, Marinescu [73] proposed detection strategies that rely on the combination of metric-based rules, as explained in Section 2.3. Then, Lanza and Marinescu [68] defined a set of thresholds based on benchmarking metrics in real software systems. In their approach, authors relied on quartile analysis. There exist other approaches in literature, such as HIST [87], which makes use of the evolution history to detect the smells, and DECOR [80], a DSL for specifying smells using high-level abstractions.

Based on these techniques to identify code smells, Khomh *et al.* [65] showed that smelly classes are more prone to change and to defects than other classes. Li and Shatnawi [71] also empirically evaluated the effects of code smells and showed a high correlation between defect-prone and some bad smells. Padilha *et al.* [86] showed that concern metrics [33] can also be useful to detect code smells. Yamashita and Moonen [124] showed that the existence of more than a single smell in a class can negatively impact the maintenance of that piece of code. This was also confirmed by Abbes *et al.* [1], who conducted controlled experiments investigating the impact of some code smells on program comprehension. They showed that the existence of a single smell in a class does not significantly decrease the performance of a developer during maintenance tasks. However, when a class presented more than one smell, their performance was significantly reduced.

Indeed, the perception of a developer may be not precise. A study from Palomba *et al.* [88] 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. Yamashita and Moonen [123] 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.* [13] 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 [93] 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 a smell, they do not refactor.

2.4.3 Architectural Roles

There exist different architectural styles and design patterns [101]. These patterns are considered "best practices" among practitioners [103]. As an example, the Core J2EE patterns [2, 29], a catalogue of patterns of enterprise applications in Java, contains patterns such as INTERCEPTING FILTER, BUSINESS DELEGATE, and TRANSFER OBJECT. Some platforms, such as Android and iOS, require developers to implement specific architectural roles in their systems, *i.e.*, any Android application should contain ACTIVITIES, classes that represent a user interface in the mobile phone.

As described in Section 2.1, classes that play any of these roles have a specific responsibility. Design patterns are a similar concept to the idea of architectural roles. Design patterns are elegant solutions to recurrent problems [117]. Examples of well-known design patterns from the Gang of Four [117] are STRATEGY, STATE, or DECORATOR. However, we differentiate design patterns from architectural roles as architectural roles have a fundamental role in the system architecture, *i.e.*, CONTROLLERS are fundamental (and required) in Web MVC systems, while a STRATEGY is not.

Detecting the existence of patterns in a system has been studied by researchers. According to Ghéhéneuc *et al.* [52], system architectures contain several micro-architectures similar to design motifs. To detect roles, they designed a propositional rule learner algorithm based on a set of code metrics computed on classes that play these roles. They were able to identify 20 different roles with a recall higher than 75%. Tsantalis and Stephanides [112] proposed a graph similarity algorithm to the same problem. They evaluated their approach in 3 open source systems, and the recall was 100% in most cases. Fabri and Mens [39] proposed a language-independent design pattern detection approach. They use a meta-language to reason about different object-oriented languages. Heuzeroth *et al.* [60] propose an approach that mixes static and dynamic analysis. According to them, many patterns only appear during runtime. De Lucia *et al.* [30] also use dynamic analysis to identify behavioral design patterns.

2.4.4 Best Practices in Web MVC systems

In this thesis, we study the architectural roles in Spring MVC, a Java web development framework. Regarding web development, most studies focus on related technologies, such as CSS, Javascript, and AJAX. Gharachorlu [49] evaluated the existence of bad practices in the CSS of open source projects. These bad practices were extracted from popular technical sites and blogs. According to the author, the most frequent bad practices were hard-coded values, styles that undo what previous styles do, and the use of IDs in selectors.

Mesbah *et al.* [78] proposed a tool called *Cilla* that is able to detect CSS flaws. Mazinanian and Tsantalis [76] analyzed how developers use CSS preprocessors. They found that developers have a clear preference for global variables, widely use the nesting feature, tend to reuse mixins, and prefer to use parameterless mixins.

Silva *et al.* [108] performed an evaluation in open source systems, and showed that Javascript code tends to use object-oriented concepts in 36% of the analyzed systems. Fard and Mesbah [41] developed a tool to detect bad practices in Javascript. According to them, the most common bad practices are lazy objects (objects that do only a few things), long methods, bad use of clojure, excessive use of global variables, and coupling between Javascript, CSS and HTML.

Nederlof *et al.* [82] investigated the quality of HTML in more than 4,000 web applications. They found several issues, such as W3C errors and warnings, non-unique IDs in the page, broken layout, and accessibility issues.

To the best of our knowledge, no research was focused on code smells for server-side MVC web applications. The smells we propose in this research (specifically in Chapter 5) are currently not captured by "traditional smells", as the former aim to more general good practices, *i.e.*, they do not focus on SQL complexity (as SMART REPOSITORIES, one of the proposed smells in this research, do) or number of dependencies to entity classes (as FAT REPOSITORIES do).

2.4.5 Where does this thesis fit?

Researchers have worked on 1) definition of code metrics, 2) how metric values distribution behave, and 3) how to define thresholds for each code metric. Researchers have also worked on the 1) definition of code smells, 2) detection strategies, 3) their impact on defect- and change-proneness of classes, 4) developers' perceptions on them.

However, up to now, none of the studies took the architectural role of a class into consideration. In the first part, we analyze the impact of architectural roles on code metrics. Thus, we evaluate how metric values distribution in each architectural role behave and discuss how we can define thresholds for each code metric and architectural role. We also perform a study on developers' perceptions on why each metric behaves differently in each architectural role. In the second part, we provide a catalogue of smells that are specific to some architectural roles, evaluate their impact on the change- and defect-proneness of classes, and captured developers' perceptions on the new set of smells.

Therefore, we contribute to both code metrics and code smells fields with a new dimension: *architectural roles*. In Figure 2.2, we present how this thesis complements the current research field.

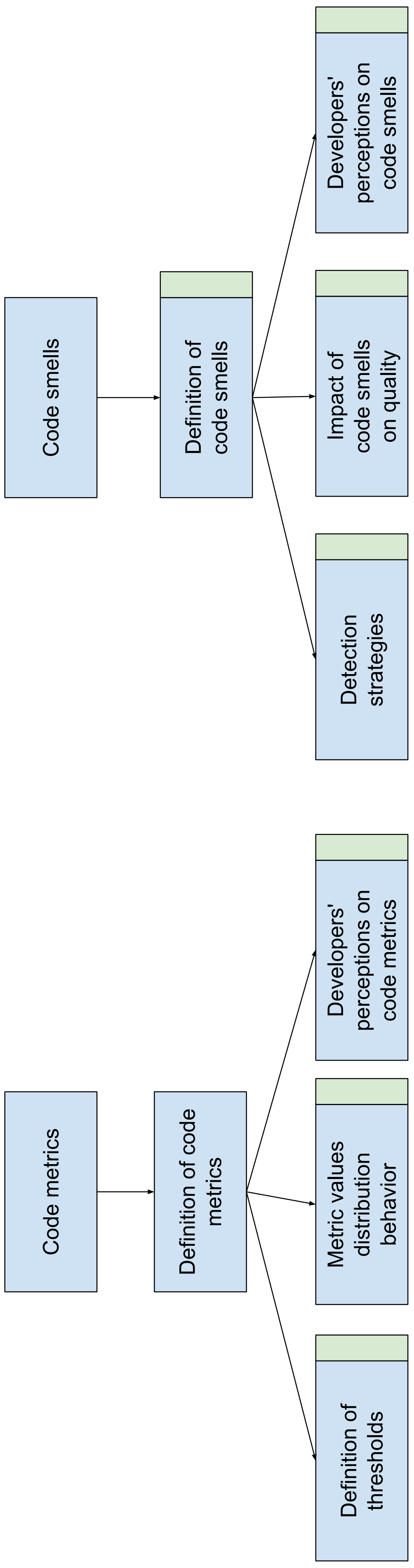

Code metrics
Definition of code metrics
Definition of thresholds
Metric values distribution behavior
Developers' perceptions on code metrics
Code smells
Definition of code smells
Detection strategies
Impact of code smells on quality
Developers' perceptions on code smells

Figure 2.2: *How this thesis complements the current research field. Blue large boxes represent existing research, while green small boxes represent our addition to the field.*

Chapter 3

Code Metrics and Architectural Roles

The content of this chapter was originally published in Aniche, M., Treude, C., Zaidman, A., van Deursen, A., Gerosa, M.A., "SATT: Tailoring Code Metric Thresholds for Different Software Architectures". 16th IEEE International Working Conference on Source Code Analysis and Manipulation, 2016.

3.1 Introduction

When dealing with code metrics and code quality assessment, should software developers treat each class differently? After all, researchers have already shown that context is important to code metrics [50, 128]. Suppose a class that plays the CONTROLLER role in the system: are CONTROLLERS more coupled than other classes? Or are CONTROLLERS similar to any other class regarding to their coupling values?

Throughout this chapter, we investigate the impact of the architectural role of a class in terms of code metrics. We also propose SATT, an approach that tailors specific thresholds for architectural roles that are indeed significantly different from others in terms of code metrics.

We answer the following questions:

RQ1. What differences in metric values distributions does SATT find for common architectural styles such as MVC?

RQ2. Can the differences in distributions thus found be explained from the architectural constraints imposed on classes fullfilling dedicated architectural roles?

RQ3. What impact do these differences have on the use of thresholds for quality assessments and smell detection?

Toward this goal, we studied how metric values behave in different architectural roles from 120 Spring MVC projects. Also, to fully comprehend these differences, we interviewed 3 experts and 17 professional developers about perceptions on architectural roles and their differences in both code metrics and maintenance practices.

Our findings show that architectural roles present specific characteristics with regard to to metric values. Most of them consistently present differences for certain metrics, such as CBO and RFC. The architectural role's specific responsibilities (*e.g.*, CONTROLLERS are responsible for controlling the flow of the system) or implementation characteristics (*e.g.*, an ENTITY usually contains many getters and setters) are the key explanation for their different metric values distribution. In the end, we show that SATT provides better thresholds, in particular to cases which architectural roles are indeed different from others in terms of code metrics.

3.2 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.* [3], 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 2.4.1), 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 [100] 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).

Table 3.1: *Cliff's Delta effect size of the comparison between architectural roles and population in Spring MVC.*
$*$ *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	*

3.3 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? 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 MVC systems. To that end, we collected 120 Spring MVC systems and both Alves *et al.*'s [3] and our approach. We also relied on a qualitative analysis of interviews with 3 different experts in both architectures.

3.3.1 Data Collection

To select Spring MVC, we made use of BOA [31], 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[1] 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 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.

In the architecture, 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".

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

Table 3.2: *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

We obtained 120 Spring MVC. In Table 3.2, 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 [12].

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

Method

We performed the SATT approach in Spring MVC 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 [12].

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). Thus, we adjusted the p-values to 0.002.

Findings

In step 3, our approach checks whether code metric values distributions are different among architectural roles. In Table 3.1, 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. In Figures 3.1 and 3.2, we present quantile plots and boxplots.

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.

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.

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 fields.

NOM. In Spring MVC, Entities present a large effect size when compared to other classes. In

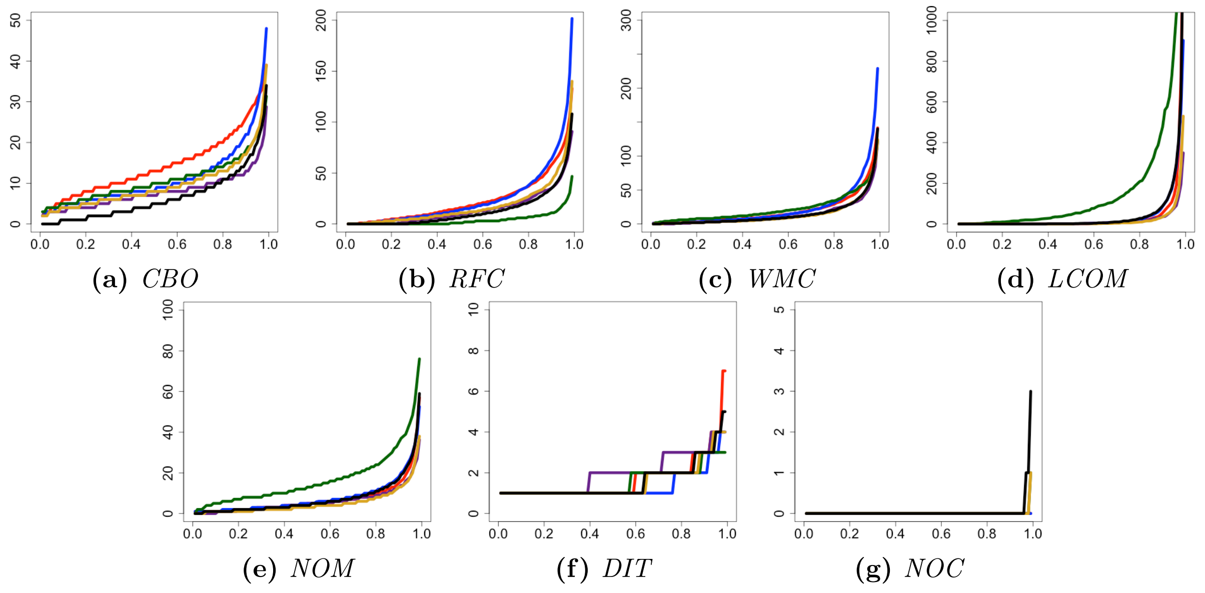


Figure 3.1: *Quantile plot for metrics and architectural roles in Spring MVC.* CONTROLLER=*Red,* REPOSITORY=*Purple,* SERVICE=*Blue,* ENTITY=*Green,* COMPONENT=*Yellow, Entire distribution=Black.*

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.

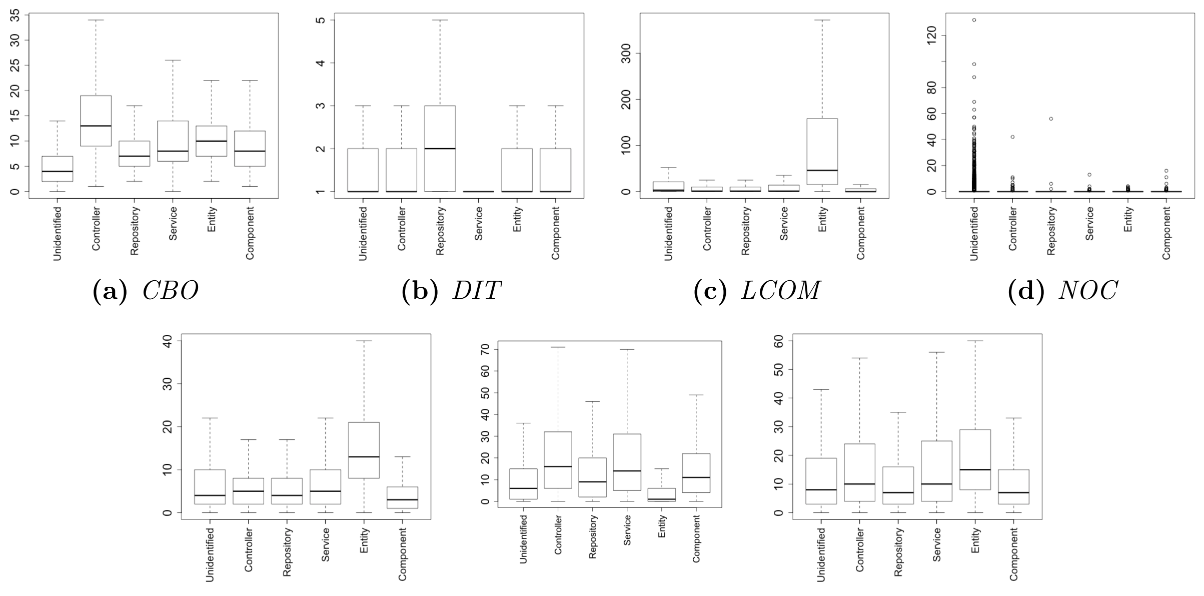
RFC. In terms of number of method calls, we see that CONTROLLERS present a medium positive effect size, while ENTITIES present a medium negative effect size. In the boxplot, we see that CONTROLLERS and SERVICES have medians above the others, while ENTITIES have a median close to zero. This indicates 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.

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.

3.3.3 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.



(a) *CBO* (b) *DIT* (c) *LCOM* (d) *NOC*

(e) *NOM* (f) *RFC* (g) *WMC*

Figure 3.2: *Boxplots of the distribution of code metrics and architectural roles.*

Table 3.3: *Experts and their experience*

#	**Years of experience**	**Degree**	**Note**
Spring MVC			
S1	10 years	BSc	Book author
S2	12 years	BSc	Course author
S3	21 years	MSc	Spring committer

Method

We conducted semi-structured interviews with 3 Spring MVC (S1-S3) experts. 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 [23], (ii) we conjectured they are experienced and mature enough to disagree with the data and explain their reasons, reducing possible response bias. In Table 3.3, we show a brief summary of their expertise. Besides many years of experience in software development, they give consultancy and training about the topic to different companies (S1, S2, S3), write books (S1), 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 3.1. 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 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 [12]. 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.

Table 3.4: *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)				

2. Q: Were you expecting this difference among architectural roles?
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, 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.

Findings

Experts endorsed and explained 6 out of the 8 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 3.4, 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. 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.

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.

(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.

Other concerns. In Spring MVC, S1 and S2 mentioned two different approches developers usually take: having rich models [38], 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.

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.

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

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.*

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.

Findings

In Table 3.5, 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 [12]. 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".
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 [12]). 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.

Table 3.5: *Results of the SATT approach in Spring MVC 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	-	-	-	-

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.

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.

3.4 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.

3.4.1 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 [97], 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 [90].

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.* [44] proposed a different way of reducing the number of false positives in code smells detection strategies [68, 73]. 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.* [10] 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 chapter 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.

3.4.2 Research implications

Researchers have shown that metric value distributions are sensitive to context information, such as the project, the application domain, and age [50, 128]. 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 [3]. Other authors also attempted to improve the choice of the thresholds. Fontana *et al.* [45] 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 [85], 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.

3.4.3 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 [3], 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 (3 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). Still, we do not claim the findings to be generalizable to industrial software [63]. 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 Spring MVC (web) systems. 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.

3.5 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 chapter, 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 MVC systems in more than 120 projects; in addition, we interviewed 3 experts in order to understand why some architectural roles are different from others.

The main contributions of this chapter 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 architectural style, 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.

Chapter 4

Developers' Perceptions on Architectural Roles

The content of this chapter was originally published in Aniche, M., Treude, C., Gerosa, M. "Developers' Perceptions on Object-Oriented Design and System Architecture". 30th Brazilian Symposium on Software Engineering (SBES), 2016.

4.1 Introduction

In practice, understanding how each architectural role behaves in terms of object-oriented design aspects is fundamental for 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 [8], *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 [96] and Sonarqube [110], are based on code metrics which are, in a nutshell, heuristics to quantitatively measure quality aspects of a source code. 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.

Thus, knowing whether developers have a common perception (or understanding) of how each class behaves (or should behave) in their systems is fundamental. Therefore, we ask:

RQ4. Do developers share a common perception on the object-oriented design aspects in their system's architectural roles?

RQ5: Do developers' perceptions match the code metric analysis of their own project?

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 with the results of a code metric analysis. We conducted a study in a Brazilian software development company that develops a large web system. By means of a "card interview", 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, we triangulated 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 findings show that developers do not share a common perception of how their system's architectural roles are characterized in terms of object-oriented design aspects. Their perceptions also do not match with the results of a code metric analysis in their system source code. Experience seems not to be a factor of influence.

4.2 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 4.2.1, we present our research questions, and in sub-section 4.2.2, we discuss both the qualitative and the quantitative method performed in this study.

4.2.1 Research Questions

RQ4: 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 interviews" with 17 professional developers. During the interviews, developers are asked to explicitly compare the behavior of classes in their system.

RQ5: 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 [24] extracted from their own project.

4.2.2 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 4.2), 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

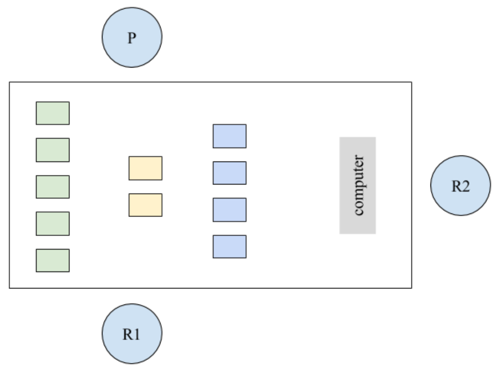


Figure 4.1: *Card sketch. P=Interviewee, R1 and R2=Researchers, Green cards=Architectural roles, Yellow Cards=Comparison operators, Blue cards=OOP concepts.*

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 RQ4, 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 interview", which we detail in the following paragraphs.

On a table, we put 3 sets of cards as shown in Figure 4.1:

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 [8].
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 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:

<role> <concept> <comparison> <role>

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 interview 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 interview. The role of the first researcher was to conduct and support the interview 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 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 4.1, 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 [120] to compare the quantity of rules provided by participants in each group.

To answer RQ5, 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 [6], which is open source and freely available for inspection.

Table 4.1: *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 4.2: *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 4.2, 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 [120]. 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 [81, 92]. 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 interview 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 [100], > 0.147) (if the rule 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

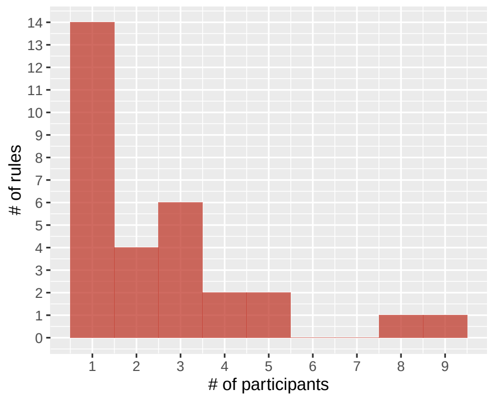


Figure 4.2: *Histogram of rules shared among participants.*

Table 4.3: *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**

used the following map to convert concepts to metrics: 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 [107] to check whether the proportions of correct and incorrect answers by both the experienced and non-experienced groups were statistically different.

4.3 Findings

In this section, we report our results for both RQ4 (Section 4.3.1) and RQ5 (Section 4.3.2).

4.3.1 RQ4: Developers' Perceptions

In Table 4.3, we present the number of rules created during our card 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 remainding 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 4.2, 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

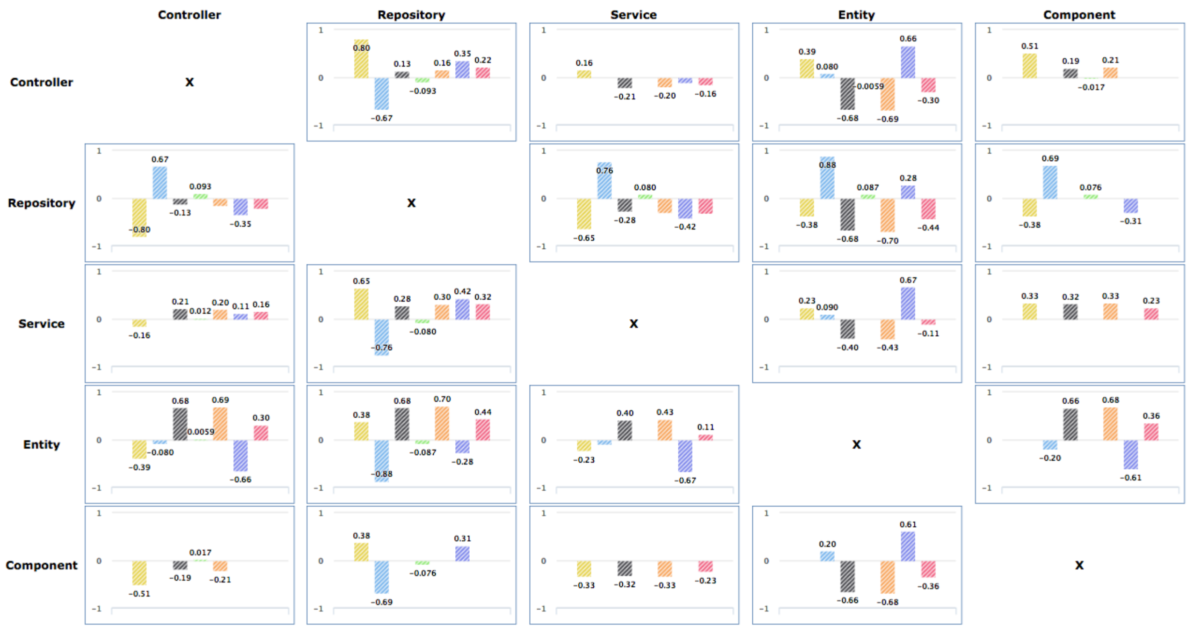


Figure 4.3: *Code metrics distribution in the system (bars, from left to right: Yellow=CBO, Blue=DIT, Black=LCOM, Green=NOC, Orange=NOM, Purple=RFC, Pink=WMC). Striped bars = significant difference (p-value < 0.05), height = Cliff's Delta effect size.*

Table 4.4: *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

(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 4.4, 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.

Mauricio:: one more about complexity here

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 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 4.5, 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 4.5, nine participants think the other way around. P9 believes Controllers are less coupled

Table 4.5: *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

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 4.5, 5 other participants believe in the opposite.

Thus, we answer RQ4: 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.
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.

4.3.2 RQ5: Code Metric Analysis

In Figure 4.3, 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-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 [9].

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 4.6. We see that their opinions matched in 14

Table 4.6: *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%

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 4.4) 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 RQ5: 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.

4.4 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 [123], 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 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 [90], 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 chapter, 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 [5, 37, 40]. 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.

4.5 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 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 [3], 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.

4.6 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 chapter, 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 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.

Chapter 5

Architectural Roles' Specific Code Smells

The content of this chapter was originally published in Aniche, M, Bavota, G., Treude, C., van Deursen, A., Gerosa, M. "A Validated Set of Smells in Model-View-Controller Architecture". Software Maintenance and Evolution (ICSME), 2016, 32th International Conference, IEEE, 2016.

5.1 Introduction

God Classes, Feature Envy, Blob Classes, and Spaghetti Code are examples of well-known code smells, *i.e.*, symptoms of poor design and implementation choices [21, 46]. Indeed, traditional code smells capture very general principles of good design. However, we suggest that specific types of code smells, such as the ones we just mentioned, are needed to capture "bad practices" on software systems adopting a specific architecture[1]. Hence, the non-existence of a rigorous smells catalogue specific to an architecture (*e.g.*, Web MVC) implies (i) a lack of explicit knowledge to be shared with practitioners about good and bad practices in that architecture, (ii) no available detection tools to alert developers about the existence of the smell, and (iii) no empirical studies about the impact of these bad practices on code maintainability properties.

Therefore, we ask:

RQ5. Are there code smells that are specific to an architectural role?

Evidence in the literature suggests that code smells can hinder code maintainability [109, 124, 125], and increase change- and defect-proneness [64, 65], and are perceived by developers [88]. Thus, we conjecture that classes affected by the smells we proposed in Chapter 5 will also have a negative impact in the class' change- and defect-proneness, and will also be perceived by developers.

Thus, we also ask:

RQ6. What is the relationship between the proposed code smells and the classes' change-proneness?

RQ7. What is the relationship between the proposed code smells and the classes' defect-proneness?

RQ8. Do developers perceive classes affected by the proposed code smells as problematic?

In this chapter, we provide a catalogue of six smells that are specific to web systems that rely

[1]In Appendix A, we present a study that we conducted in a Brazilian software development company. There, we coined a smell that was specific to one of the architectural roles in their system. That study also motivated us to research on specific code smells.

on the MVC pattern. The use of MVC for web development is widely spread and applied by many of the most popular frameworks in the market, such as Ruby on Rails, Spring MVC, and ASP.NET MVC. To produce the catalogue, we surveyed and interviewed 53 different software developers about good and bad practices they follow while developing MVC web applications. Then, we applied an open coding procedure to derive the smell catalogue from their answers. The defined smells are: Smart Repository, Fat Repository, Promiscuous Controller, Smart Controller, Laborious Repository Method, and Meddling Service.

We evaluated the impact of the proposed smells on change- and defect-proneness of classes in 100 Spring MVC projects. In addition, we performed a survey with 21 developers to verify whether they perceived classes affected by the defined smells as problematic. Our findings show that all the proposed smells have a negative impact on class change-proneness. Also, Meddling Services increase class defect-proneness. Finally, developers perceive classes affected by these smells as problematic, at least as much as classes affected by traditional smells.

5.2 Smell Discovery Approach

We collected good and bad practices followed by developers while working on Web MVC applications. The data collection included three different steps detailed in the following.

Step 1: Layer-focused survey (S1). We designed a simple survey comprising three sections: Model, View, and Controller. In each section, we asked two questions to the participants:

1. *Do you have any good practices to deal with X?*
2. *Do you have anything you consider a bad practice when dealing with X?*

where X was one of the three investigated layers (*i.e.*, Model, View, or Controller).

The goal of this first survey was to shed some light on good and bad practices followed by developers when dealing with code belonging to the three different MVC layers.

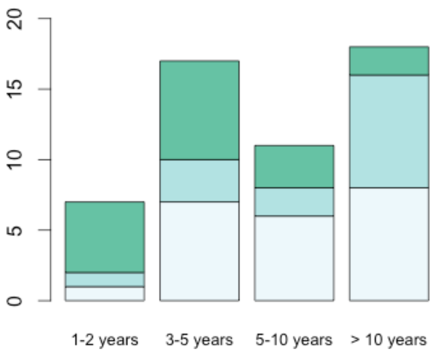
We shared the survey in software development discussion lists as well as in personal and industry partners' Twitter accounts. We collected 22 complete answers. The survey can be found in Appendix D.2.

Step 2: Role-focused survey (S2). We designed a survey aimed at investigating good and bad practices related to code components playing a specific role in the MVC architecture in web applications.

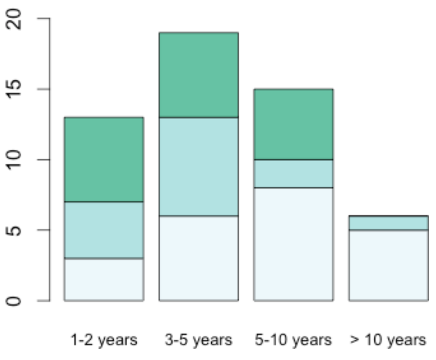
The questionnaire contained five open questions, one for each of the roles mentioned in Section 2.1: Controller, Entity, Service, Component, and Repository. We asked participants about good and bad practices they perceive for classes playing each of these roles. In order to recruit participants, we sent invitations to 711 developers who did at least one commit in the previous six months (July-December, 2014) in one of the 120 Spring MVC projects hosted on GitHub. Such a list of projects has been collected using BOA [31], a dataset with structured information about projects in GitHub. We received 14 answers to this survey. The survey can be found in Appendix D.3.

Step 3: Unstructured interviews with industrial developers (S3). We interviewed 17 professional developers from one of our industry partners[2]. All participants worked at the time of the interview on a Java-based Spring MVC web application that has been developed for 11 years, and has more than 1 million lines of code in its main module. The focus of the interview was to

[2]The same industry partner that we used in Chapters 3 and 4.



(a) *Software Development*



(b) *Web Development*

Figure 5.1: *Participants' experience in software and web development in S1 (bottom), S2, S3 (top).*

make participants discuss about their good and bad practices in each of the five main architectural roles in MVC Web applications. All interviewees were developers or technical leaders. Interviews were conducted by two of the authors, and took 4:30 hours in total. They were fully transcribed. The survey can be found in Appendix D.4.

Overall, we collected information about good and bad practices followed in MVC Web applications from 53 participants. To report some demographic data, our surveys as well as our interviews asked participants about their experience in software and web development. Complete data is shown in Figure 5.1. Participants were mostly experienced in both software and web development. 46 (83%) had more than 3 years of experience in software development, and 18 (33%) had more than 10 years.

We used the answers provided by participants to our surveys and interviews as the starting point to define our smells catalogue. In particular, two of the authors performed an open coding process on the reported good and bad practices in order to group them into categories. They focused on identifying smells that can be considered as specific of the Web MVC architecture. For example, answers like "*large classes should be avoided*" were not taken into consideration, since large classes should be avoided in any type of system [46], independently from its architecture. Instead, answers like "*a repository method should not have multiple queries*" were considered indicative of MVC-specific smells, and thus categorized into a high-level concept, which afterwards became a smell (e.g., LABORIOUS REPOSITORY METHOD). Note that the two authors independently created classifications for the participants' good and bad practices. Then, they met to discuss, refine, and merge the identified categories, reaching an agreement when needed. They ended up with a list of nine possible smells.

To further validate the defined list of smells and reduce the subjectivity bias, we presented the nine smells to Arjen Poutsma, one of the core Spring MVC developers since its creation, and currently Spring Technical Advisor at Pivotal, the company that maintains the framework. After listening to his opinions, we removed three of the defined smells (two related to SERVICE classes and one to REPOSITORY classes). The main reason for the removal was that these three smells were not really generalizable to arbitrary MVC web applications. The complete list of the nine smells we defined as result of the open coding procedure is reported as part of our replication package [11], while in the following (Section 5.3), we detail the six smells present in our catalogue as well as tool-supported detection strategies to identify each of them. These latter have been also defined in collaboration with the expert.

Table 5.1: *MVC smells*

Name	Description
Promiscuous Controller	Offer too many actions
Smart Controller	Too much flow control
Meddling Service	The service directly query the database
Smart repositories	Complex logic in the repository
Laborious Repository Method	A method having multiple actions
Fat Repository	A repository managing too many entities

5.3 RQ5: Resulting Catalogue of Web MVC Smells

Table 5.1 reports the six MVC smells included in our catalogue. In the following paragraphs, we discuss each of smells, explaining why it has been considered part of our catalogue (*i.e.*, which answers provided by participants indicated the existence of this smell) and which detection strategy we adopted to spot its instances[3]. We use the notation SX-PY to refer to answers provided by participant Y in the context of the X data collection step (S1, S2, or S3 presented in Section 5.2).

Promiscuous Controller. CONTROLLERS should be lean and provide cohesive operations and endpoints to clients. As CONTROLLERS are the start point of any request in Web MVC applications, participants ($n = 6$) argued that the ones that offer many different services to the outside are harder to maintain as they deal with many different functionalities of the system. As S3-P13 stated: *"With many services you end up having a Controller with a thousand lines, a thousand methods, and I think this is bad."*. According to S1-P1, *"Something happens in a Controller with more than 5 methods (routes)"*. S1-P3 even had a name for that: *"Jack-of-all-trades controllers, controllers that do a lot of things in the application."*.

We define the smell as *"Controllers offering too many actions"*. To detect them, we rely on the number of routes implemented in the CONTROLLER and the number of SERVICES the CONTROLLER depends on. The reasoning is that a CONTROLLER offers many actions when it provides many different endpoints and/or deals with many different SERVICE classes. Therefore, to detect the smell, we propose the metrics *NOR (Number of Routes)*, which counts the number of different routes a CONTROLLER offers, and *NSD (Number of Services as Dependencies)*, which counts the number of dependencies that are SERVICES. In Formula 5.1, we present the detection strategy, where α and β are thresholds.

$$(NOR > \alpha) \vee (NSD > \beta) \tag{5.1}$$

Smart Controller. The most mentioned smell by our participants ($n = 25$) is the existence of complex flow control in CONTROLLERS. In Web MVC applications, ENTITIES and SERVICES should contain all business rules of the system and manage complex control flow. Even if a CONTROLLER contains just a few routes (*i.e.*, is not a PROMISCUOUS CONTROLLER), it can be overly smart. According to S1-P19, this is a common mistake among beginners: *"Many beginners in the fever to meet demands quickly, begin to do everything in the controller and virtually kill the Model and the Domain, leaving the system just like VC."* S3-P7 also states that his team does not unit test CONTROLLERS, and thus, complex logic and control flow in them should be avoided.

[3]Thresholds used in the detection strategies have been tuned as described in Section 5.4.3.

When discussing the smell with the expert, he agreed that the flow control in CONTROLLERS should be very simple. Thus, we come up with the following definition for the smell: *"Controllers with too much flow control"*.

As a proxy to measure the amount of flow control in a CONTROLLER, we derived the NFRFC (Non-Framework RFC) from the RFC (Response for a Class) metric that is part of the CK metric suite [24], an ofted used suite of object-oriented metrics. The common RFC metric counts the number of all method invocations that happen in a class. However, it also counts invocations to the underlying framework. As confirmed by our expert, CONTROLLERS perform several operations on the underlying framework, and these should happen there. Thus, NFRFC ignores invocations to the framework API, which makes the metric value represent the number of invocations that happen to other classes that belong to the system. In Formula 5.2, we present the detection strategy, where α represents the threshold:

$$(NFRFC > \alpha) \tag{5.2}$$

Meddling Service. Services are meant to contain business rules and/or to control complicated business logic among different domain classes. However, they should not contain SQL queries. While 2 participants mentioned that this is a bad practice, all participants in the interview were clear about where the SQLs should be (good practice): in REPOSITORIES. In addition, two of the participants affirmed that queries in SERVICES may be problematic. S3-P15 stated: *"Never get data [from the database] directly in the Service; Services should always make use of a Repository."*. Our expert also confirmed the smell with no further thoughts.

We define this smell as *"Services that directly query the database"*. If a SERVICE contains a dependency to any persistence API provided (*e.g.*, JDBC, Hibernate, JPA, iBatis) and makes use (one or more times) of this dependency, then we consider this class to be smelly. In Formula 5.3, we present its detection strategy for a class C:

$$\exists persistenceDependency(C) \tag{5.3}$$

Smart Repository. Repositories are meant to deal with the persistence mechanism, such as databases. To that, they commonly make use of querying languages, such as SQL or JPQL (Java's JPA Query Language). However, when REPOSITORIES contain complicated (business) logic or even complex queries, participants ($n = 24$) consider that class smelly. S3-P10 states that *"When it is too big [the query], ..., if we break it a little, it will be easier to understand."*. S3-P14 strongly states: *"No business rules in Repositories. It can search and filter data. But no rules."* Therefore, we define this smell as *"Complex logic in the repository"*.

When discussing the smell with the expert, he mentioned that two situations are common in real world REPOSITORIES, and sometimes can happen in the same class: (1) very complex SQL queries, *i.e.*, a single query that joins different tables, contains complex filters, etc, and (2) complex logic to build dynamic queries or assembly objects that result from the execution of the query. According to him, if both these two types of complexity are in a class, then the class has a symptom of bad code. Thus, we detect a SMART REPOSITORY by identifying the ones in which the McCabe's Complexity Number [77] and the SQL complexity are higher than a threshold. McCabe's Number counts the number of different branch instructions, *e.g.*, *if, for*, inside of a class. Similarly, to define the SQL complexity, we counted the occurrence of the following SQL commands in a query: *WHERE, AND, OR, JOIN, EXISTS, NOT, FROM, XOR, IF, ELSE, CASE, IN*. In Formula 5.4, we present the detection strategy, where α and β are thresholds:

$$(McCabe > \alpha \land SQLComplexity > \beta) \tag{5.4}$$

Laborious Repository Method. As a good practice, a method should have only one responsibility and do one thing [74]. Analogously, if a single method contains more than one query (or does more than one action with the database), it may be considered too complex or non-cohesive. Although just one participant (S1-P1) raised this point, both authors selected the smell during the analysis, and our expert confirmed that it is indeed a bad practice, as it reduces the understandability of that method.

Thus, we define the smell as *"a Repository method having multiple database actions"*. The detection strategy relies on the number of methods that "execute" a command in the underlying persistence mechanism. We argue this is a good proxy for the number of actions or executed queries. In practice, developers need to invoke many different methods of the API to build the query, pass the parameters, execute, and deal with its return. Using Java as example, we present a list of methods (actions) for many different persistence APIs which should happen only once in each method: For Spring Data, *query()*, for Hibernate, *createQuery(), createSqlQuery(), createFilter(), createNamedQuery(), createCriteria()*, for JPA, *createNamedQuery(), createNativeQuery(), createQuery(), createStoredProcedure(), getCriteriaBuilder()*, and for JDBC, *prepareStatement(), createStatemente(), prepareCall()*. If a method contains two invocations to any of the methods above, we consider the class as smelly. In Formula 5.5, we present the smell's detection strategy for class C:

$$\forall m \in C \ \exists \ qtyPersistenceActions(m) > 1 \tag{5.5}$$

Fat Repository. Commonly, there is a one-to-one relation between an Entity and a Repository, *e.g.*, the entity *Item* is persisted by ItemRepository. If a Repository deals with many entities at once, this may imply low cohesion and make maintenance harder. Participants ($n = 6$) mentioned that repositories should deal with only a single entity. S3-P12 stated: *"[A problem is to] use more than one Entity in a Repository. The repository starts to loose its cohesion."*.

Our expert agreed with this smell with no further comments. Therefore, we define it as *"a Repository managing too many entities"*. We count the number of dependencies a Repository has directly to classes that are *Entities*. We call this metric *CTE*. If this number is higher than the threshold, the class is considered smelly. In Formula 5.6, we present the detection strategy, where α is the threshold:

$$(CTE > \alpha) \tag{5.6}$$

In the following chapter, we evaluate the impact of our catalogue of smells from a quantitative (change- and defect-proneness of classes) and a qualitative (developers' perception) point-of-view.

5.4 Smell Evaluation Study Design

The *goal* of the study is to investigate whether the defined catalogue of MVC smells can affect different maintainability aspects of a class, such as its change- and defect-proneness, and whether developers perceive classes affected by our six smells as problematic. The *quality focus* is on source code quality and maintainability that might be negatively affected by the presence of the defined smells.

5.4.1 Context Selection

To answer our research questions, we need to identify instances of the defined code smells in MVC software projects. We select Spring MVC projects from Github as subject systems. We focus our attention on the Spring MVC framework since: (i) it uses stereotypes to explicitly mark classes playing the different roles introduced in Section 2.1 (*e.g.*, CONTROLLERS), thus making simple identifying the role of each class, and (ii) as shown in a survey conducted with over 2,000 developers [115], it is widely adopted by developers ($>$ 40% of the respondents claimed to use it).

We made use of the same 120 projects of Chapter 3. The complete list is available in our appendix C.2. From the 120 subject projects, 20 are randomly selected[4], to tune the thresholds of our detection strategies, as described in Section 5.4.3. The remaining 100 are used, as detailed in Section 5.4.2, to answer our research questions.

To answer RQ8, we recruit 21 Spring MVC developers among our industry contacts, asking them to take part in an online survey aimed at assessing their perception of the defined smells. Figure 5.2 depicts participants' experience in software development as well as in the development of Spring MVC applications. Participants are generally quite experienced in software development. In particular, 13 of them have more than 8 years of experience. Their level of experience with the Spring MVC framework is spread, varying from 1 to 2 years of experience (10 participants) to more than 8 years (3 participants). None of the developers surveyed in RQ_3 had been contacted or involved in the steps performed for the definition of the code smells catalogue.

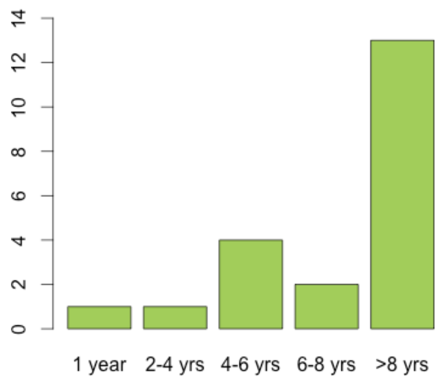
5.4.2 Data Collection and Analysis

To answer RQ6 and RQ7, we assessed the impact on change- and defect-proneness, respectively, of the defined Web MVC smells. Firstly, it is important to clarify that, while we answer RQ6 by analyzing the complete change history of all 100 subject systems, we only consider a subset of 16 manually selected projects to assess the impact of the MVC smells on defect-proneness (RQ7). These systems are the ones having enough information to compute the classes' defect-proneness.

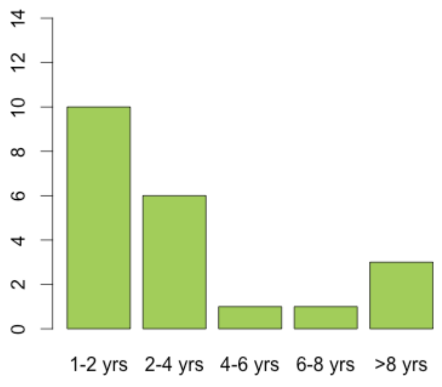
Indeed, while to measure the change-proneness of a class C in a time period T it is sufficient to count the number of commits in which C has been modified during T, to assess C's defect-proneness we need to count the number of bugs found in C during T. This information is typically stored in the issue tracker that, however, was not available for most of the subject systems. Thus, to measure the defect-proneness of C over T, we rely on Fischer *et al.*'s approach [43]. The approach uses regular expressions to identify fixing-commits as the ones having commit messages containing keywords indicating bug fixing activities, such as *bug* or *fix* (*i.e.*, the defect-proneness of C over T is the number of fixing-commits in which C was involved during T). However, to succeed in this measurement, we need software projects having (i) commit messages written in English, and (ii) using words such as "bug" or "fix" in commit messages. We manually analyze the commits of the 100 projects ending up with 16 of them meeting our requirements. These 16 projects are thus exploited in the context of RQ7 and listed in Appendix C.

To assess the impact on change- and defect-proneness of the Web MVC smells, we follow an

[4]For the random selection, we performed a single execution of R's *sample()* function with seed set to 123.



(a) *Software Development*



(b) *Spring MVC*

Figure 5.2: *Participants' experience*

approach similar to what is done in a previous study [65] investigating traditional smells. Firstly, as performed by Kim *et al.* [66], we split the change history of the subject systems (100 for RQ6 and 16 for RQ7) in chunks of 500 commits, excluding the first chunk likely representing the project's startup. We indicate the two commits delimiting each chunk as C_{start} (*i.e.,* the 1^{st} commit) and C_{end} (*i.e.,* the 500^{th} commit). We only analyze commits that were merged into the main development branch, *i.e.*, in Git, the *master* branch.

We obtain 291 chunks for systems used in RQ6 and 77 for those used in RQ7. We run our detection strategies on the C_{start} of each chunk, obtaining a list of smelly and of clean classes. Then, we compute the change proneness of each class (both smelly and clean classes) as the number of commits impacting it in the 500 commits between C_{start} and C_{end}. As done by Khohm *et al.* [65], we mark a class as change-prone if it has been changed at least once in the considered time period. Finally, to have a term of comparison, we also detect six traditional smells in the C_{start} commit of each chunk. We identify traditional smells by executing PMD [96], a popular smell detector. We use it to detect instances of six smells, namely God Class, Complex Class, Long Method, Long Parameter List, Coupling Between Objects, and Long Class. Our choice of the traditional smells to consider is not random, but based on the will to consider smells capturing poor practices in different aspects of object-oriented programming, such as complexity and coupling, and previously studied by other researchers [84, 88, 93].

To compare the change-proneness of MVC-smelly, traditional-smelly, and clean classes, we compute the following six groups:

- $\mathbf{NC}_{Clean}$, the number of clean classes (not affected by any MVC or traditional smell) that are not change-prone;
- $\mathbf{C}_{Clean}$, the number of clean classes that are change-prone;
- $\mathbf{NC}_{MVC-smelly}$, the number of MVC-smelly classes that are not change-prone;
- $\mathbf{C}_{MVC-smelly}$, the number of MVC-smelly classes that are change-prone;
- $\mathbf{NC}_{T-smelly}$, the number of traditional-smelly classes that are not change-prone;
- $\mathbf{C}_{T-smelly}$, the number of traditional-smelly classes that are change-prone.

Then, we use Fisher's exact test [107] to test whether the proportions of $\text{C}_{MVC-smelly}/\text{NC}_{MVC-smelly}$ and $\text{C}_{Clean}/\text{NC}_{Clean}$ significantly differ. As a baseline, we also compare the differences between $\text{C}_{T-smelly}/\text{NC}_{T-smelly}$ and $\text{C}_{Clean}/\text{NC}_{Clean}$. In addition, we use the Odds Ratio (OR) [107] of the three proportions as effect size measure. An OR of 1 indicates that the condition or event under study (*i.e.*, the chances of inducing change-proneness) is equally likely in two compared groups (*e.g.*, clean *vs* MVC-smelly). An OR greater than 1 indicates that the condition or event is more likely in the first group. Vice versa, an OR lower than 1 indicates that the condition or event is more likely in the second group.

We mirror the same analysis for defect-proneness (RQ7). Again, a class is considered to be defect-prone in a chunk if it is involved in at least one fixing-commit during the 500 commits composing the chunk. In this case, the six groups of classes considered to compute the Fisher's exact test and the OR are $\textbf{ND}_{Clean}$, $\textbf{D}_{Clean}$, $\textbf{ND}_{MVC-smelly}$, $\textbf{D}_{MVC-smelly}$, $\textbf{ND}_{T-smelly}$, $\textbf{D}_{T-smelly}$, where D and ND indicate classes in the different sets being (D) and not being (ND) defect-prone.

Note that, to reduce bias in our analysis, we only consider CONTROLLERS, SERVICES, and REPOSITORIES in the sets of clean, MVC-smelly, and T-smelly, since our smells focus on these classes. Also, since classes affected by traditional smells or by our defined MVC-smells are expected to be large classes (*e.g.*, a PROMISCUOUS CONTROLLER is likely to be a large class), and it is well known that large classes have a higher change- and defect- proneness, we control for the size confounding factor. To this aim, we report the results of our analysis when considering all classes (no control for size) as well as when grouping classes into four groups, on the basis of their LOC: Small=$[1, 1Q[$, Medium-Small=$[1Q, 2Q[$, Medium-Large=$[2Q, 3Q[$, and Large=$[3Q, \infty]$, where 1Q, 2Q, and 3Q represent the first, the second (median), and third quartile, respectively, of the size distribution of all classes considered in our study. In this way, we compare the change- and defect-proneness of clean and smelly classes having comparable size.

Finally, concerning RQ_3, all 21 participants took part in an online survey composed of two main sections. The first one aimed at collecting basic information on the participants' background, and in particular on their experience (data previously presented in Figure 5.2). In the second section, participants were asked to look into the source code of six classes and, for each of them, answer the following questions:

Q1 *In your opinion, does this class exhibit any design and/or implementation problem?* Possible answers: YES/NO.

Q2 *If YES, please explain what are, in your opinion, the problems affecting the class.* Open answer.

Q3 *If YES, please rate the severity of the design and/or implementation problem by assigning a score.* Possible answers on a 5-point Likert scale going from 1 (very low) to 5 (very high).

Q4 *In your opinion, does this class need to be refactored?* Possible answers: YES/NO.

Q5 *If YES, how would you refactor this class?* Open answer.

The six selected classes are randomly selected for each participant from a set of 90 classes containing: 30 classes affected by one of the proposed MVC-smells (five classes per smell type), 30 classes affected by the six traditional smells also exploited in the context of RQ6 and RQ7 (five classes per smell type), and 30 non-smelly classes. Note that also in this case we reduce possible bias

Table 5.2: *Thresholds for the metrics used in the detection strategies*

Metric	**Threshold**
Promiscuous Controller	
Number of Routes (NOR)	10
Number of Services as Dependencies (NSD)	3
Smart Controller	
Non-Framework RFC (NFRFC)	55
Smart Repository	
McCabe's Complexity	24
SQL Complexity	29
Fat Repository	
Coupling to Entities (CTE)	1

by only considering in all three sets classes being CONTROLLERS, SERVICES, or REPOSITORIES, since these are the specific architectural roles on which our smells focus. Each participant evaluates two classes randomly selected from each of these three groups. The 90 classes were randomly sampled from the 100 subject systems.

To reduce learning and tiring effects, each participant received the six randomly selected classes in a random order. Also, participants were not aware of which classes belong to which group (*i.e.*, MVC-smelly, traditional-smelly, and clean). They were simply told we were studying code quality in MVC web applications. No time limit was imposed for them to complete the task.

To compare the distributions of the severity indicated by participants for the three groups of classes, we use the unpaired Mann-Whitney test [28] to analyse statistical significance of the differences between the severity assigned by participants to problems they spot in MVC-smelly, traditional-smelly, and clean classes. The results are considered statistically significant at $\alpha = 0.05$. We also estimated the magnitude of the measured differences by using Cliff's Delta (or d), a non-parametric effect size measure [51] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.14$, small for $0.14 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [51]. Finally, we report qualitative findings derived from the participants' open answers.

5.4.3 Thresholds Tuning

The detection strategies are based on the combination of different measurements (*e.g.*, code metrics) and use a set of thresholds to spot smelly classes.

In Formula 5.7, we present the formula used to define the thresholds (TS) for each metric. Basically, the formula aims at defining thresholds spotting classes that, for a specific metric, represent outliers. It makes use of the third quartile (3Q) and the interquartile range ($3Q - 1Q$) that was extracted from projects that were selected for this tuning. As each smell corresponds to a single specific role, and some metrics are specific to them, *i.e.,* number of routes can only be calculated in CONTROLLERS, only classes of that role were taken into account during the analysis of the distribution. The use of quantile analysis is similar to what has been proposed by Lanza and Marinescu [68] in order to define thresholds for their detection strategies.

$$TS = 3Q + 1.5 \times IQR \tag{5.7}$$

In Table 5.2, we present the thresholds derived for each metric (α and β in the Formulas presented in Section 5.3).

Table 5.3: *Quantity of smelly classes in our test sample (n = 100)*

Role/Smells	**# of Classes**	**%**
Controllers	**2,742**	**100%**
Promiscuous Controller	336	12.2%
Smart Controller	205	7.4%
Repositories	**1,185**	**100%**
Smart Repository	85	7.1%
Fat Repository	243	20.5%
Laborious Repository Method	79	6.6%
Services	**2,509**	**100%**
Meddling Service	99	3.9%

5.5 Results

Table 5.3 reports the number of smells identified in the last snapshot of the 100 subject systems. In particular, we report for each of the three architectural roles taken into account by our smells (*i.e.*, REPOSITORIES, CONTROLLERS, and SERVICES) (i) the total number of classes playing this role in the 100 systems (*e.g.*, 1,185 REPOSITORIES), (ii) the number and percentage of these classes affected by each smell (*e.g.*, 85 REPOSITORIES are SMART REPOSITORY — 7.1%).

Overall, we identified 1,047 smells in 851 classes out of the 6,436 classes playing one of the three roles described above (16%). The most common smell in terms of percentage of affected classes is the FAT REPOSITORY (20.5%) followed by the PROMISCUOUS CONTROLLER (12.2%) and the SMART CONTROLLER (7.4%). The least diffused smell is instead the MEDDLING SERVICE with only 3.9% of affected SERVICES.

We also detected 4,619 traditional smells in 1,580 classes of the same sample (24%). The intersection between the 851 MVC-smelly classes and the 1,580 traditional-smelly classes contains only 388 classes. Also, all the proposed smells identified classes that were not identified by the traditional ones. This indicates that the proposed smells select classes which are not currently identified by any of the traditional smells used in this study.

5.5.1 RQs 6 and 7: Relationship between the smells and the change- and defect-proneness

Table 5.4 reports the results of Fisher's exact test (significant *p-value* represented by the star symbol) and the OR obtained when comparing the change- and defect-proneness of (i) MVC-smelly classes *vs* clean classes, (ii) traditional-smelly classes *vs* clean classes, and (iii) MVC-smelly classes *vs* traditional-smelly classes. We also report the confidence intervals (at 95% confidence level) in our online appendix [11]. As explained in Section 5.4.2, we report both results when considering in the comparison all classes (no control for size) as well as when grouping classes into groups, on the basis of their size. Note that we do not report the results for small and medium/small classes due to lack of data: classes affected by MVC and traditional smells are for the vast majority at least medium/large classes.

When comparing the change- and defect-proneness of MVC-smelly classes and of clean classes not controlling for size, Fisher's exact test reports a significant difference, with an OR of 2.97 for change- and 2.05 for defect-proneness. When controlling for size, differences are also significant, but less marked. For change-proneness, we observe an OR of 1.42 in medium/large classes (*i.e.*, 42% higher chance of changing with respect to clean classes), and 1.60 in large classes. In terms of

Table 5.4: *Odds ratio in change- and defect-proneness between MVC-smelly, traditional-smelly and clean classes.*
(CP) Change-proneness, (DP) Defect-proneness,
(*) Fisher's exact test < 0.05.

		All classes	**Medium/Large**	**Large**
MVC-smelly *vs* clean	CP	2.97*	1.42*	1.60*
	DP	2.05*	0.72	1.06
Traditional-smelly *vs* clean	CP	3.87*	1.18	1.75*
	DP	5.69*	1.16	2.31
MVC-smelly *vs* Traditional-smelly	CP	0.77*	1.19	0.81*
	DP	0.36*	0.55	0.42*

defect-proneness, we do not observe any significant difference when controlling for size.

As a term of comparison, it is interesting to have a look to the results obtained when comparing the change- and defect-proneness of classes affected by traditional smells with clean classes and with classes affected by MVC-smells. Results in Table 5.4 show that:

1. Traditional smells have a strong negative impact on change-proneness. However, as also observed for MVC-smells, they have no impact on defect-proneness when controlling for size. Thus, this only partially confirms previous findings about traditional smells in the literature [64, 65].
2. Traditional smells have a stronger negative impact on change- and defect-proneness as compared to MVC-smells. This also holds for large classes when controlling for size.

To have a closer look into the data, Table 5.5 reports the impact on change- and defect-proneness of each of the six MVC-smells presented in this chapter. It is important to note that in some cases (e.g., Smart Controller for medium/large classes), it was not possible to perform the statistical test due to lack of data (*i.e.*, very few Smart Controllers are medium/large classes). These cases are indicated with "-" in Table 5.5. The main findings drawn from the observation of Table 5.5 are:

1. When obtaining statistically significant difference (* cells in Table 5.5), classes affected by smells have always a higher chance (OR > 1.00) of changing as well as of being subject to bug-fixing activities. This holds both when controlling for size as well as when considering all classes. We cannot claim anything for non-statistically significant results.
2. Smart Repository and Meddling Service are the smells having the strongest impact on change-proneness with an OR close to 3 in large classes (*i.e.*, classes affected by these smells have almost three times more chances to change as compared to clean classes).
3. The Meddling Service smell is the only one having a significant impact on defect-proneness when controlling for size (OR=2.53 in large classes, *i.e.*, classes affected by this smell have over twice more chances of being subject to bug-fixing activities as compared to clean classes).

Table 5.5: *Odds ratio in change- and defect-proneness between MVC-smelly and clean classes, per smell. (CP) Change-proneness, (DP) Defect-proneness,
(*) Fisher's exact test < 0.05, (–) lack of data.*

		All classes	**Medium/Large**	**Large**
Promiscuous Controller	CP	2.66*	1.48*	1.51*
	DP	2.43	0.41	0.68
Smart Controller	CP	3.72*	-	1.81*
	DP	3.42*	-	1.34
Fat Repository	CP	2.04*	0.80	1.75*
	DP	1.79*	0.90	0.99
Laborious Repository Method	CP	2.03*	2.38	1.06
	DP	2.36	-	0.48
Smart Repository	CP	5.08*	-	2.79*
	DP	5.02*	-	2.03
Meddling Service	CP	3.74*	2.41*	2.89*
	DP	3.39*	1.15	2.53*

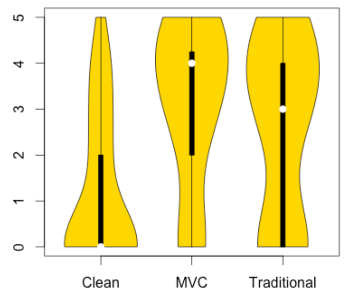
The defined web MVC smells have a negative impact on class change-proneness (RQ6). In terms of defect-proneness, we claim a negative impact for only the MEDDLING SERVICE *smell (RQ7).*

5.5.2 RQ8: Developers Perception of the Web MVC Smells

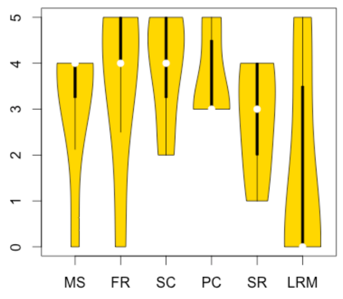
In Figure 5.3a, we present violin plots of the developers' perception of MVC smells, traditional smells, and clean classes. Also, we report the developers' perception of each single MVC-smell — Figure 5.3b — as well as of each considered traditional smell — Figure 5.3c. On the y-axis, 0 (zero) indicates classes not perceived by the developers as problematic (i.e., answer "no" to the question: *Does this class exhibit any design and/or implementation problem?*), while values from 1 to 5 indicates the severity level for the problem perceived by the developer.

Clean classes have a median of severity equal to 0 (Q3=2). This indicates that, as expected, developers do not consider these classes as problematic. As a comparison, classes affected by MVC-smells have median=4 (Q3=4.25) and thus, are perceived as serious problems by developers. The difference in developers' perception between MVC-smelly and clean classes is statistically significant (p-value<0.001) with a large effect size (d = 0.56). Concerning the traditional smells, the severity median is 3 (Q3=4). It shows that classes affected by these smells are perceived by developers as problematic, even if less than MVC-smells. However, while this difference in perception is clear by looking at the violin plots in Figure 5.3a, such a difference is not statistically significant (p-value=0.21). We conjecture that this might be due to the limited number of data points (21 participants).

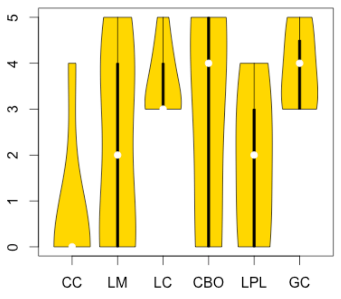
God Classes (GC) are the most perceived traditional smell (median=4). Regarding the proposed



(a) *Clean=Non smelly classes, MVC=Classes affected by our defined smells, Traditional=Classes affected by traditional smells*



(b) *MS=Meddling Service, FR=Fat Repository, SC=Smart Controller, PC=Promiscuous Controller, SR=Smart Repository, LRM=Laborious Repository Method*



(c) *CC=Complex Class, LM=Long Method, LC=Long Class, CBO=Coupling Between Objects, LPL=Long Parameter List, GC=God Class*

Figure 5.3: *Participants on the severity of each smell.*

smells, Meddling Service, Fat Repository, and Smart Controller achieve medians equal to 4, meaning they are perceived as really problematic by the participants. Several developers, without knowing our smells' catalogue, were able to correctly identify the smell, providing a description very close to the definition of our smells. For instance, one of them when facing a Smart Controller stated: *"Property validation and entity construction are really responsibilities that should be encapsulated within the service layer; a lot of domain model knowledge is needlessly leaked into the Controller."*. Another participant simply claimed: *"it does too much for a Controller"*. Also when facing a Promiscuous Controller, developers were able to catch the problem (*e.g.*, *"I count 12 @RequestMapping!"*). The annotation *@RequestMapping* is indeed used to define a route in a Spring MVC Controller. This maps directly to the concept of our smell. Participants also noticed that Smart Repositories are complex: *"programmer(s) should worry just about querying instead of handling and logging hibernate errors"*.

The least perceived smells by developers are Laborious Repository Method (MVC) and Complex Classes (traditional), as both medians are zero, *i.e.*, over half of the participants did not perceive classes affected by this smell as problematic.

Classes affected by the proposed MVC smells are perceived as problematic by developers when compared to non-smelly classes (RQ8).

5.6 Threats to 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. Since the subject systems did not have an issue tracker, we relied on the heuristic proposed by Fischer *et al.* [43] to identify bug fixing commits. We are aware that such a heuristic can introduce imprecisions in the computation of the classes' defect-proneness. To diminish the issue, we made sure via manual analysis that the systems used in this study use meaningful commit messages written in English.

Detection strategies for the defined smells were derived from the participants' answers and the expert analysis. There might be other better strategies for their detection. Further research needs to be conducted in order to optimize them. However, our current strategies were able to detect classes perceived as problematic by developers and possibly increasing change-proneness. Also, possible imprecisions might be introduced due to errors in the implementation of the tool we wrote to detect the smells. We made sure to write automated tests to ensure the correct behavior of our tool that, as stated before, is open source and publicly available.

To determine the thresholds we used in our detection strategies, we applied quartile analysis on a set of projects that were not used to answer our research questions. While other strategies can be used (*e.g.*, Alves *et al.*'s [3] and Oliveira *et al.*'s [85]), up to now there is no empirical evaluation of which strategy works best.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. We did not consider possible tangled changes [59] and thus we cannot exclude that some bug fixing commits grouped together tangled code changes, of which just a subset aimed at fixing the bug.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalisation of results: (1) Although we derived the thresholds to identify the smells on 20 systems, we do not claim these thresholds are the optimal ones. A larger set of systems is needed to increase the thresholds' generalizability; (2) We evaluated our smells in Spring MVC applications. Although there might be similarities between most of the MVC frameworks, we do not claim that our results are generalizable to all of them; (3) Finally, our catalogue only includes six smells for MVC web applications. We do not claim this is a comprehensive catalogue. Further research is needed to investigate other possible bad practices in MVC applications.

5.7 Conclusions

Good practices and code smells are a great asset for developers to increase the quality (and the maintainability) of their software systems. However, most code smell catalogues are focused on general good practices, *i.e.* practices that can be applied to any system, regardless of its architecture.

We provided a catalogue of 6 smells that are specific to Web MVC systems, namely SMART REPOSITORY, FAT REPOSITORY, PROMISCUOUS CONTROLLER, SMART CONTROLLER, LABORIOUS REPOSITORY METHOD, and MEDDLING SERVICE. This catalogue was coined after interviewing and surveying 53 software developers. We also analyzed 100 Spring MVC projects in order to quantify the impact of the proposed smells. We showed that these smells can have a negative impact on class change-proneness and, in the case of MEDDLING SERVICE, also on defect-proneness. We also performed a survey with 21 developers about their perceptions on classes affected by these smells, and results show that they perceived smelly classes as highly problematic.

We learned that besides traditional smells, architecture-specific smells can also be problematic for the maintenance of software systems. Web developers can already start to benefit from our catalogue and from the publicly available detection tool [7]. Indeed, a deeper investigation is needed to carefully assess the comprehensiveness of our catalogue as well as the impact of these smells on maintainability (*e.g.*, by running a controlled experiment). This is part of our future work agenda, together with the definition and empirical analysis of other catalogue of smells, specific for other architectures.

Chapter 6

Conclusion

In this chapter, we revisit the research questions and provide guidance on how practitioners can make use of our findings. We also give our last remarks and suggest future work.

6.1 The Research Questions Revisited

Throughout this thesis, we analyzed the importance of architectural roles in code metrics and code smells. To that end, we analyzed 120 open source projects, and interviewed and surveyed 53 software developers and 3 experts. We summarize the entire research in one sentence: *When assessing the quality of a class, by means of code metrics or code smells, the architectural role of a class matters.*

In the following, we revisit the research questions and their answers.

RQ1. What differences in metric values distributions does SATT find for common architectural styles such as MVC?

The SATT approach indeed finds architectural roles that present significantly higher (or lower) values for certain metrics when compared to other classes. In the studied architectural roles (Controller, Service, Entity, Components, and Repositories), we see that they are commonly more coupled (higher CBO and RFC).

RQ2: Why are architectural roles different from each other in terms of code metric values distribution?

Because of the different responsibilities of architectural roles, developers deal with the classes involved in the architectural role in a different way. We find these practices to be consistent with the outcomes of RQ1. Also, 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 differences, their explanation related to a possible lack of adherence to a good practice.

RQ3. What impact do these differences have on the use of thresholds for quality assessments and smell detection?

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.

RQ4: Do developers share a common perception on the object-oriented design aspects in their system's architectural roles?

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. 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. Also, we observe that both experienced and non-experienced groups are similar in terms of behavior. Thus, experience does not seem to influence the results.

RQ5: Do developers' perceptions match with the code metric analysis of their own project?

Developers' perceptions match with the code metric analysis in only 50% of the cases. We consider this to be a low and worrisome number. Even the perceptions that are shared among many participants are wrong in many cases. From the top 5 most popular perceptions, only 2 were correct. We are unable 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.

RQ6. What are smells that are specific to architectural roles that belong to the studied architecture?

We coined six smells: Complex logic in the repository (SMART REPOSITORY), Controllers that offer too many actions (PROMISCUOUS CONTROLLER), Too much flow control in Controllers (SMART CONTROLLER), Service that directly queries the database (MEDDLING SERVICE) A method in Repository that has multiple actions (LABORIOUS REPOSITORY METHOD), and a repository that manages too many entities (FAT REPOSITORY).

RQ7. What is the impact of the proposed code smells on classes' change-proneness?

The defined smells have a negative impact on class change-proneness, *i.e.*, they increase the change-proneness of classes that are affected by the smells. SMART REPOSITORY and MEDDLING SERVICE are the smells with the strongest impact on change-proneness with an OR close to 3 in large classes (*i.e.*, classes affected by these smells have almost three times more chances to change as compared to clean classes).

RQ8. What is the impact of the proposed code smells on classes' defect-proneness?

The MEDDLING SERVICE smell is the only one with a significant impact on defect-proneness when controlling for size (OR=2.53 in large classes, *i.e.*, classes affected by this smell have over twice more chances of being subjected to bug-fixing activities as compared to clean classes). We need more data to affirm whether the other smells are also more prone to defects.

RQ9. Do developers perceive classes affected by the proposed code smells as problematic?

Classes affected by the proposed smells are perceived as problematic by developers when compared to non-smelly classes. Participants considered MEDDLING SERVICE, FAT REPOSITORY, and SMART CONTROLLER the most problematic ones. Also, we claim that developers perceive the new set of smells at least as problematic as traditional smells.

6.2 Discussion

As we have noted before, other authors have shown how context can impact code metrics [19, 50, 128]. They noticed that variables such as the domain of the system, the programming language, and even the age of the software can affect the behavior of code metric values. This thesis adds another item in the list of contextual information that can affect how metrics behave: *the architectural role of a class.*

Understanding that architectural roles differ from each other in terms of specific responsibilities can benefit software developers during maintenance activities. Up to now, both code metric and code smell assessment techniques are based on 1) "generic" benchmarking and 2) "generic" object-oriented smells. By "generic", we mean "no specific context". Clearly, we acknowledge the importance of the current techniques. Object-oriented smells, such as God Classes or Feature Envy, still apply to any system, regardless of the architecture. And a class with a very-high coupling metric value is still an important class for developers to assess. However, we claim that when a developer fully understands the architectural role of the class (*i.e.,* its responsibilities, how it behaves in terms of code metrics), s/he can better assess the quality of that class.

As as example, we saw that CONTROLLERS have higher CBO than other classes, mainly because they are responsible for the control of the flow. If a software development team decides to use a "traditional benchmarking approach", around 65% of all CONTROLLERS will be perceived as problematic. We also show that CONTROLLERS suffer from specific smells such as PROMISCUOUS CONTROLLER and SMART CONTROLLER. Thus, this CONTROLLER class may not suffer from a traditional smell, but may suffer from one of these specific smells, which are as problematic as the traditional ones. Thus, when assessing CONTROLLERS, developers should be aware of how CONTROLLERS behave.

Thus, based on everything we learned, we suggest that practitioners:

- Understand how the system architecture and its specific architectural roles behave in their systems in terms of object-oriented design aspects, *i.e.*, which roles are more coupled and why, which roles are more complex and why, etc.
- Build their own role-based benchmarking (using SATT) in order to assess classes by means of code metrics, which can be done using a similar approach to ours (collecting data from open source projects with the same architecture).
- Interview their developers and understand what the specific smells are for each of the existing architectural roles; then, develop metric-based detection strategies, so that a tool can automatically detect the smelly classes.
- Constantly share the knowledge among developers, so that all of them are aligned in terms of what should be expected in each architectural role.

Our findings can also be important for researchers in the field, since:

- When proposing benchmarking techniques [3, 4, 85], researchers can now take the architectural role into account.
- When predicting classes that are prone to defects, researchers make use of code metrics as a feature for the prediction algorithm [32]. As we now know that the architectural role is a factor of influence, research is needed in order to evaluate whether this information can improve such algorithms.

- Although “traditional” code smells are also important, we showed that specific smells are at least as important as the traditional ones. Thus, researchers can work on specific smells for specific architectures.

6.3 Future Work

We evaluated the impact of architectural roles in one architecture. We conjecture that these findings can be applied to other architectures, such as Android and iOS applications. Thus, research needs to be conducted in order to identify the better thresholds for code metric analysis, as well as for a set of specific smells to these architectures.

In Chapter 5, we used a heuristic to detect classes that were prone to defects. Clearly, a better way would be to extract the results from issue tracking systems. We then suggest a new study that extracts this information from issue tracking. We conjecture this would result in more precise information about the relationship between the defined smells and the defect-proneness of classes affected by them.

Also, the majority of this research was conducted in open source systems with different domains and sizes. Another future work would be to conduct the same study in large industrial systems and check whether the results are similar. In addition, validating our findings in other languages and frameworks might be useful for the community.

We also wonder about other smells that could be important for developers using the studied architecture. More interviews need to be conducted in order to collect different smells. Furthermore, while we catalogued 6 smells, there might be other important code smells to be explored.

Appendix A

The Intruder Method: A Code Smell for Repositories

The content of this chapter was originally published in Aniche, Maurício F., Gustavo A. Oliva, and Marco A. Gerosa. “Are the Methods in Your Data Access Objects (DAOs) in the Right Place? A Preliminary Study.”. 6th Workshop on Managing Technical Debt, in conjunction with ICSME, Canada, 2014.

A.1 Introduction

Object-oriented software development taught us about the separation of concerns [48, 69, 91, 122]. Today, it is common sense among developers to separate code that deals with system infrastructure from code that deals with business rules and domain objects. Database access, for example, may require large and complex code and the use of external libraries, so developers usually isolate it in Data Access Object (DAO) classes. Moreover, as the number of database queries in a system can be huge, the data access code is split into many different DAO classes, usually one per table/entity. For example, the *InvoiceDAO* is responsible for persisting and retrieving data from invoices, while the ProductDAO is responsible for dealing with products.

However, a query can be complex and make use of many different tables. When that happens, a question that often comes to developers' minds is **in which DAO should I put this query?** In practice, what happens is that developers sometimes write the method in the wrong -— or in an ambiguous —- DAO.[1] A possible explanation for this would be that, at the time the developer was writing the query, s/he put the method on the DAO related to the first table in the query.

When the location of a method is ambiguous, developers spend much time searching for it. If they do not find it, they will duplicate code and write the same query in a new method. Writing the method in the right DAO avoids code duplication and improves separation of concerns, leading to higher maintainability. Zazworka et al. [127] studied technical debts and their relationship to software quality, and found that a modularity violation such this one, in which the method is located in the wrong module, may be highly related to bugs and a higher change likelihood.

Therefore, we ask:

RQ9. How can one automatically identify methods that may have been placed in the wrong or in ambiguous DAOs?

Our findings show that we can quickly and automatically identify methods that may be in an ambiguous location. Calculations showed that our heuristic selected from 13% to 18% of all methods

[1] "Wrong" may be not be the best word to describe this action. It is hard to affirm that a method is in the wrong place, as this depends on the context. By wrong, we mean a method that could possibly be in another DAO, in which it would be easier to find– we use the word ambiguous to express this throughout the chapter.

in DAOs, and was correct on 50% to 75% of these selections.

A.2 Data Access Objects (DAOs)

A Data Access Object (DAO) is an abstraction to a data source, such as a database [111]. The goal of this pattern is to separate the business logic from the database persistence, so that both parts can evolve and be changed independently [91].

This pattern is very common in many applications. As each entity may be queried in many different ways, developers tend to write one DAO class per table/entity, to facilitate maintenance. This means that a system with three entities, such as *Invoice, Product, and Order,* will have three tables to store each of them, and three different DAOs (usually named after the entity plus the suffix "DAO").

This naming convention also facilitates maintenance. If a database query that deals with the *Product* table needs to be changed, then it needs to be done in the *ProductDAO* class. If, for any reason, the query is not there, the developer will have to search for it in the other DAOs, or write a new one.

However, deciding in which DAO a specific query should be is not an easy task. A single query may touch different tables (using the JOIN instruction, for example). In Listing A.1, we show an example of that. This query joins three tables: *Projects, Commits, and Artifacts.*

Listing A.1: *An example of a SQL query that deals with three tables*

```sql
SELECT
  p.name as projectName,
  c.id as commitId,
  a.name as artifactName,
  a.path as artifactPath
FROM
  Projects p
JOIN
  Commits c on c.project_id = p.id
JOIN
  Artifacts a on a.commit_id = c.id
WHERE
  p.repository = 'Apache';
```

If one analyzes the query, s/he may notice that it probably belongs to *ArtifactDAO*. The SQL returns many artifacts and their relative commit and project. This method probably returns a list of *Artifact*. On the other hand, there are two other tables in the query. It would not be an exception to find this query in *ProjectDAO*, or even in *CommitDAO*. This is an example of what we call an ambiguous method.

This technical debt may cost the team. Code duplication, for example, is a possible consequence. When developers open a DAO class and do not find the method they want, they may write a new one, resulting in two different methods that do the same thing in the system.

Based on this, it is clear that developers must find a way to detect methods and queries that were written in ambiguous DAOs. However, we do not aim for a perfect algorithm to find them – as with any metric –, we are looking for good indicators. In practice, if a project contains 1000 methods in its DAOs, it is impossible for a human to check each one of them manually.

Table A.1: *Decision table*

Suggested method	Developer's opinion	Result
X()	Right place	Heuristic Failed
Y()	Wrong place	Heuristic OK
Z()	Not sure	Heuristic OK

If, however, the algorithm filters 50 methods, a human can verify those at an acceptable cost. With that being said, our goal is to write an algorithm that selects a reasonably small set of classes that need to be manually checked.

A.3 Experiment Design

Based on the discussion and motivations given above, this chapter aims to answer the following research question:

RQ9. *How can one automatically identify methods that may have been placed in the wrong or in ambiguous DAOs?*

To find a solution to that question, we conceived a heuristic and implemented it in a tool that outputs the methods that seem to be in the wrong place. As a case study, we conveniently selected three projects from a Brazilian software development company. Maurício, one of the authors of this chapter, works for this company. All three are web projects, developed in Java. They all use the same technology and frameworks, namely VRaptor (MVC Framework), Hibernate (Object-Relational Mapping Framework), JSP, and JSTL (view layer). Each deals with a specific domain: *Caelumweb* is an ERP, *Gnarus* is an e-learning system, and *Codesheriff* is a code metric visualization application. Although the projects were developed by different teams, teams typically share ideas and experiences.

After running the tool on the three projects, we invited a developer from each project to manually inspect all methods that were output. The developer from *Caelumweb* had worked on the project for the previous two years; the developers from *Gnarus* and *Codesheriff* had worked on the projects from their beginnings (three and one years, respectively).

The developers were instructed to state whether each listed method was in the right place. There was no specific technique – they all had the list of all methods selected by the algorithm and the full source code of the project. They then navigated through each selected method (and sometimes through other related classes) and made decisions. As discussed before, it is hard to precisely define whether a method is in the right place, thus the decisions were purely based on the feelings and knowledge of the project of the developers.

If a developer thought the method was in a wrong or ambiguous place, we considered that the heuristic had correctly selected that method. If s/he stated otherwise, we considered that the heuristic was wrong. If, for some reason, the developer was in doubt about the method, we considered the heuristic as correct for that method as well (if even the human did not know if the method was in the right place, such a method definitely merits human analysis). In Table A.1, we summarize the aforementioned decision algorithm.

In Table A.2, we describe the size of each project. The number of classes and number of commits represent the size and age of the project. The total number of DAOs and methods reflects the number

Table A.2: *Analyzed Projects*

Project	# of Classes	# of Commits	# of DAOs	# of Methods
Gnarus	924	10451	39	233
Caelumweb	1321	12077	81	590
Codesheriff	56	339	10	70

Table A.3: *The numbers of the heuristic execution on the projects*

Project	# of Methods	# of Right Methods	# of Wrong Methods	% of Wrong Methods
Caelumweb	590	511	79	13.38%
Codesheriff	70	57	13	18.57%
Gnarus	233	200	33	14.16%

of queries and methods that needed to be analyzed. This number varied from 50 to 900 Java classes, and from 10 to 80 DAO classes. *Caelumweb*, for example, contained almost 600 methods in DAOs. If a developer were to go method-by-method to check if the methods were in the right place, s/he would expend a lot of effort.

In the following subsections, we detail the designed heuristic to identify potentially misplaced methods. All the gathered data (including the SQL queries used to manipulate it), as well as the specific build of the tool we employed, are available on a website[2].

A.3.1 The Heuristic

The heuristic is based on the assumption that *all queries that deal with the X table/entity should be on XDao*. By dealing with the entity, we mean that the method should return the entity or receive it as an input parameter. Also, in practice, we noticed that many methods return a primitive type. That happens when the query returns a single item, such as an integer or a double (by executing a *COUNT* or a *SUM* in the database).

To be considered "correct," a method should follow at least one of the following rules. To simplify the rule, we will assume that *X* is the type that the *XDao* is associated with.

1. The return type of the method is X.
2. The return type of the method is a primitive.
3. The return type of the method is an enum.
4. The return type of the method is a sub-type of X.
5. The return type is a generic type with more than one type.
6. The return type is a generic type of X, such as List <X >.
7. The return type name contains a substring of X.
8. One parameter of the method is an instance of X.
9. One parameter of the method is a generic type of X, such as List <X >.
10. All parameters of the method are primitives.

[2]http://www.github.com/mauricioaniche/icsm2014-daos. Last access on June 27, 2014.

Table A.4: *Developers agreement on the heuristic*

Project	# of Inspected Methods	# of Agreements	% of Agreement
Caelumweb	79	59	74.68%
Codesheriff	13	8	61.53%
Gnarus	33	16	48.48%

If a method, for any reason, does not match any criteria, then it is determined that the method may be in the wrong place, and needs to be validated by a human.

A.3.2 Implementation

The heuristic is simple in terms of implementation. Essentially, it deals with methods' return types and parameters. We developed a Java parser that navigates through all the classes in a system and returns all methods that do not match the criteria.

The parser was developed in Java and uses of ANTLR to parse the language. Currently, it is a simple command-line tool that prints a list of methods and their respective classes. The source code is freely available[3].

A.4 RQ9: Results and Discussion

In Table A.3, we show the results of the heuristic for the three projects. The heuristic pointed out that 13% to 18% of the methods were potentially misplaced.

As noted before, to validate the effectiveness of our approach, we invited one developer per project to manually inspect the whole output given by our tool. In Table A.4, we show the number of methods on which the developers agreed with the heuristic.

One may note that the agreement ranges from 48% (CodeSheriff) to 75% (Caelumweb). This means that, in the worst case, 1 out of 2 selected methods were considered to be in an ambiguous place. When analyzing the cases in which the heuristic selected a method considered to be in the right place, we found out a few things:

- There are many DAOs whose name does not match the entity's name. Sometimes the DAO's name is a shortcut to the name of the entity. The DAO that deals with the entity *ParcelaDeBoleto* (which, in portuguese, means "bankslip parcel") was named *ParcelaDao*.
- There are two DAOs assigned to the same entity, each one with its own perspective. As an example, the *Course* entity in *Gnarus* contains both the *CourseDAO*, which contains all queries needed by the front-end application, and the *AdminCourseDAO*, which contains all queries needed by the administration part of the application.
- Queries that were grouped by features. In *Gnarus*, we found a DAO that represented all queries needed by a given feature. This feature, in particular, deals with many entities.
- Data Transfer Objects (DTOs) are used frequently. Report queries are good examples of this. The heuristic tries to guess if a class is a DTO, by matching the name of the entity with the name of the DTO. However, this has only worked for a few cases.

[3]http://www.github.com/mauricioaniche/calculadora-de-daos. Last accessed on June 26, 2014.

In all such cases, although the developer stated that those methods were in the right place, they may still need attention. New developers, or even developers who are not familiar with specific parts of the system, may struggle to interpret these subtle design decisions behind some DAOs.

We conclude that:

It seems to be possible to automatically identify methods that may be located in an ambiguous DAO. Our approach, in particular, filters around 13% to 18% of the all methods in DAOs, and it is correct 50% to 75% of the time. As the number of selected methods is small and the assertion rate is high, it may be worthwhile to allocate a developer to manually inspect the methods and move them to a better location, if necessary.

A.5 Threats to Validity

As with any initial research, this study contains a few threats to the validity of the results. However, we believe that these can all be treated in future versions of this work.

- We evaluated only three projects, and they were all from the same company. As companies usually have a standard method of developing applications, the heuristic may not be valid for projects developed by other companies. More projects need to be analyzed.
- We relied on the point of view of a single developer per project. Although these developers were knowledgeable about the projects, a discussion with the whole teams or with the developers that actually implemented the methods could provide greater insight and a more accurate view of the reasons behind the designs. However, as the developers are potential users of those pieces of code, if they considered the methods misplaced, the heuristics actually identified points needing attention.
- Although we validated the methods that were selected by the heuristic (false positives), we did not evaluate those that were in the wrong place but were not identified by the heuristic (false negatives). A more complete manual analysis should be done.

A.6 Conclusion and Future Work

Data Access Objects (DAOs) are abstractions to a data source. This is a well-known pattern used in many different applications. However, as argued in this study, developers sometimes misplace their methods. This makes maintenance difficult, as the team cannot easily locate the queries. In the worst cases, they even write duplicate methods.

In this chapter, we propose an approach to quickly and automatically identify methods that may be in an ambiguous location. Calculations showed that our heuristic selected from 13% to 18% of all methods in DAOs, and was correct on 50% to 75% of these selections. It seems feasible to allocate a developer to manually investigate each of the filtered methods, and to move them to a better location, if necessary.

As a next step, we want to ask an outside expert to assess the methods and identify which are incorrect. That would allow us to use standard precision/recall metrics. Understanding the costs of this technical debt is an important step, too; if a method is in the wrong place, but the refactoring is more expensive than the cost of the developer searching for the method, keeping the method in the wrong place may be the best option. In addition, we need to further evaluate this model by analyzing other projects belonging to different companies, as well as open-source projects.

Appendix B

Springlint

Springlint is a linter for Spring MVC systems. This linter is different from other popular tools, such as PMD and Sonarqube, as it takes the Spring MVC context into account. Instead of presenting generic advices, the tool performs different analysis for each architectural role in a Spring MVC system.

In this appendix, we present Springlint, the tool we developed based on the findings of this thesis. The tool performs code metrics and code smells analysis, taking the architectural roles of the system into account. We believe our tool can be useful to practitioners.

B.1 How to use the tool

Springlint is a Java command-line tool. It means that developers should invoke it using commands directly in the terminal. A practical example of the execution of tool is:

Listing B.1: *Example of a Springlint's execution command*

```
java -jar springlint
 -p /Users/mauricioaniche/SSP
 -o /Users/mauricioaniche/results/
 -otype html
```

Also, as many Java systems make use of Maven, we also provide a plugin that works out-of-box. To execute it, developers must perform the following Maven task, where 0.4 is the version of the tool:

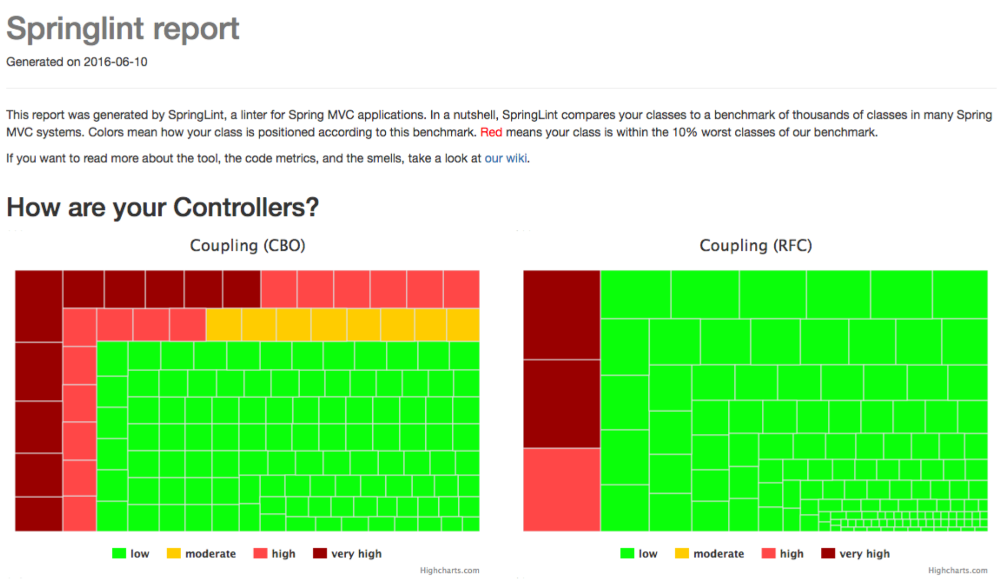
```
mvn com.github.mauricioaniche:springlint-maven-plugin:0.4:springlint
```

B.2 Features

The HTML output generates a single HTML page that contains all the information about the code metrics and code smells analysis. In Figure B.1, we show an screenshot of the report.

To the code metrics, the tool provides users with a heat map. The tool separates classes in three different risk levels: moderate, high, and very high. Green classes are the ones with no risk. Yellow blocks represent classes with moderate risk, while light and dark red represent high and very high risks, respectively. The size of the box means the metric value itself; the higher the value the larger the box when compared to the others.

To the code smells, the tool shows a list of classes and the smell (or smells) that affect those classes. It detects the six catalogued smells in this thesis: Smart Repository, Fat Repository, Promiscuous Controller, Smart Controller, Laborious Repository Method, and Meddling Service.


Springlint report

Generated on 2016-06-10

This report was generated by SpringLint, a linter for Spring MVC applications. In a nutshell, SpringLint compares your classes to a benchmark of thousands of classes in many Spring MVC systems. Colors mean how your class is positioned according to this benchmark. Red means your class is within the 10% worst classes of our benchmark.

If you want to read more about the tool, the code metrics, and the smells, take a look at our wiki.

How are your Controllers?

Figure B.1: *Example of a Springlint report.*

B.3 Implementation Details

Springlint is a command-line tool developed in Java. As all analysis rely on source code parsing, the tool makes use of JDT, the Eclipse internal compiler. In this section, we highlight the most important part of its architecture: how to support new architectures and code smells.

As depicted in Figure B.2, an important concept inside the tool is `Smell`. Each of them is composed by a list of metrics and a "conciliate" operation, which combines the result of the executed metrics and decide whether that class presents the smell.

As an example, the PROMISCUOUS CONTROLLER smell contains two metrics: number of route methods (NOR) and number of SERVICES as dependencies (NSD). Then, if NOR returns a value higher than 10 or NSD returns a value higher than 3, then, the smell detector returns "true", indicating that the class is smelly.

Metrics are implemented as JDT's AST visitors. JDT requires all visitors to implement the `ASTVisitor` abstract class. This class contains a `visit()` method for each node in the abstract syntax tree, such as `MethodDeclaration`, `MethodInvocation`, etc. In the following listing, we present part of the code of the "SQLComplexityVisitor" used by the SMART REPOSITORY smell. The visitor checks for all string literals in the class and counts the SQL operations (such as JOIN, WHERE, and EXISTS) in that string.

Listing B.2: *Piece of code of a AST visitor*

```java
public boolean visit(StringLiteral node) {
  String sql = node.getLiteralValue();
  if(isSql(sql)) {
    calculateComplexity(sql);
    update();
  }

  return super.visit(node);
}
```

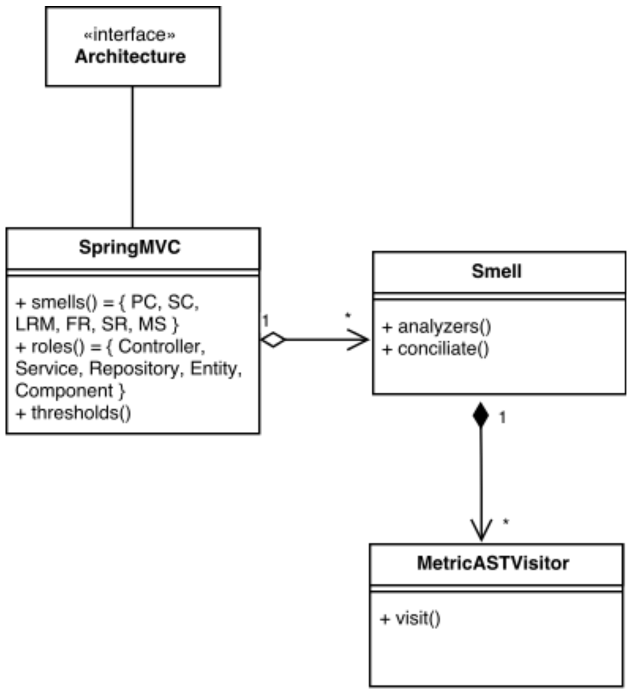


Figure B.2: *Springlint architecture.*

Internally, the tool already contains the implementation of all the CK metrics. The tool is executed in two phases: in the first phase, the tool calculates all the CK metrics. In the second, it executes all the smell detection strategies. Then, the tool generates the desired output to the end user.

Our tool inherits JDT's capabilities of running on machines with different limitations, such as low memory. In these cases, the tool takes more time to run. As an example, in a computer with 4GB of RAM and a 2.8GHz Intel Core i7, the tool took 5 minutes to analyze a system with 1,740 Java files and 133,000 lines of code.

The tool also contains an automated test suite. The suite is mainly composed by integration tests which assert the results of code metrics and code smells in real Java classes.

Appendix C

Projects Sample

C.1 BOA query

The online query can be found here: http://boa.cs.iastate.edu/boa/?q=boa/job/11947.

Listing C.1: *BOA query to select Spring MVC projects*

```
p: Project = input;
projects: output set of string;

qty := 0;

visit(p, visitor {

  before node : CodeRepository -> {
    # ignoring pet projects
    if(len(node.revisions) < 500) stop;

    # latest snapshot only
    snapshot := getsnapshot(node);
    foreach (j: int; def(snapshot[j])) {
      visit(snapshot[j]);
    }
    stop;
  }

  before node : ASTRoot -> {
    foreach(i: int; node.imports[i]) {
      if(node.imports[i] == "org.springframework.stereotype.Controller") {
        qty = qty + 1;
      }
    }
  }

});

if(qty>10)
  projects << p.code_repositories[0].url;
```

C.2 Spring MVC projects

Project	Initial Commit	Final Commit
https://github.com/Tasktop/code2cloud.server.git *	5830e70	dcdcccc
https://github.com/jtalks-org/poulpe.git *	f399fa5	a8646a36
https://github.com/ewcmsfree/ewcms.git *	edf94685	5e8a7377
https://github.com/nulpteam/school.git *	8387878e	477f56ec
https://github.com/yangjiandong/sshapp.git *	d436c72a8	20a573490
https://github.com/BroadleafCommerce/DemoSite.git *	9d2beadd	d6f7ced
https://github.com/infoScoop/infoscoop.git *	051facac	f5e2eb1948
https://github.com/resthub/resthub-spring-stack.git *	19392fc	f4a44f450
https://github.com/dcoraboeuf/iteach.git *	45ed0c54	9f4a61273
https://github.com/blacksnow666/falcon.git *	00c23c813	513a58d72
https://github.com/apache/shiro.git *	89ffe41fd	f61c0eb56
https://github.com/ralscha/extdirectspring.git *	af88c62134	18e664d887
https://github.com/bruceMacLeod/motech-server-pillreminder-0.18.git *	87563b67e	ef4ce65e4
https://github.com/ualerts-org/ualerts-server.git *	88a0623c	58e921d9
https://github.com/OpenConext/OpenConext-selfservice.git *	6ddac6917	74216b83
https://github.com/europeana/portal.git *	a48a582cb	2485859ab
https://github.com/akvo/akvo-flow.git *	d50cb84130	bd2a4c28
https://github.com/Fernandolima/Condominio.git *	a3a17a1d	66a563f3
https://github.com/dreambt/cms4j.git *	6cb73688	26ac73d0
https://github.com/skyscreamer/yoga.git *	1167c85d	d98f6b0b
https://github.com/atlasapi/atlas.git +	03fb8826	564e1650
https://github.com/BandwidthOnDemand/bandwidth-on-demand.git +	5e075eb3	c01a6259
https://github.com/open-infinity/cmt.git +	6aeeebdf	a49622d1
https://github.com/encuestame/encuestame.git +	522d23cc	331be1e7
https://github.com/laffer1/justjournal.git +	4d02a139	c9bd0a67
https://github.com/vorburger/mifos-head.git +	f2efef88	b77be9aa
https://github.com/php-coder/mystamps.git +	f957a3e3	55b0f347
https://github.com/OpenLMIS/open-lmis.git +	00d916f7	f1cb4850
https://github.com/ldangelo/orangeleap-tangerine.git +	165aa803	11dad15f
https://github.com/uq-eresearch/oztrack.git +	015e6681	bfc8ded1
https://github.com/qcadoo/qcadoo.git +	aece6b35	10c5d8b2
https://github.com/sukrupa/school-admin.git +	e75369a4	d3976352
https://github.com/Jasig/SSP.git +	3d2379a6	b3546fc8
https://github.com/ippontech/tatami-team1.git +	795569c1	5d1c8b2a
https://github.com/cloudfoundry/uaa.git +	ace57779	cab137b8
https://github.com/kai8406/cmop.git	17f8043c	8c76932a
https://github.com/mscheper/onebusaway-ms-application-modules.git	1ee89c86	06901a21
https://github.com/antme/pms.git	49320fb6	abf69018
https://github.com/blizzy78/documentr.git	7a367b73	ba04df4c
https://github.com/source3g/hermes.git	a3445bcf	49d3d6d7
https://github.com/spring-projects/greenhouse.git	88556a95	4250e20f
https://github.com/0x006EA1E5/oo6.git	97ffd7b0	44943bbd
https://github.com/bdult/farmstory_backoffice.git	ac83ba0d	c844bdb1
https://github.com/delving/delving.git	585c2404	c3fcdbef
https://github.com/tonytw1/wellynews.git	849cb223	5a55326d
https://github.com/fits/try_samples.git	b4de6e75	09f0d359
https://github.com/czakian/openmrs_gsoc.git	22349dc4	e74afa54
https://github.com/cc14514/pet-service.git	27a67afc	4161ac50
https://github.com/alexbud/vkontakte.git	a3dec3eb	e5a5163f
https://github.com/shazichuanhuo/heat_final_version.git	6b2df392	7a3203cc
https://github.com/springside/springside4.git	71ee99d5	df81d06c
https://github.com/societies/SOCIETIES-SCE-Services.git	94101bcf	ea2ed1f6
https://github.com/cwoolner/flex-poker.git	64a4649b	a41358bd
https://github.com/motech/MOTECH-Ghana.git	e1574af0	624e1f86
https://github.com/qcadoo/mes.git	e8bf7eb1	5546613f
https://github.com/badqiu/fpcms.git	b7a69ecd	f57f4928
https://github.com/openmrs/openmrs-module-reporting.git	2227df44	cb919321
https://github.com/ktisdell/2gx_demo.git	9d2beadd	0e94b866
https://github.com/microlcl/scheme_repo.git	3ebb1699	dd97df8b
https://github.com/dfci-cccb/mev.git	10e3ca37	b2162b38
https://github.com/nesi/projectdb-webapp.git	ac77dc49	31143308
https://github.com/cws-khuntly/cws-esolutions.git	2ca17973	d7c43bd8
https://github.com/apache/oltu.git	efb14cdc	89186dc4
https://github.com/energyos/OpenESPI-DataCustodian-java.git	26971057	b39856c1
https://github.com/glo-ulaval/Architecture.git	c1dba219	deeda218
https://github.com/openforis/collect.git	781d9b0b	db3a7589
https://github.com/C4G/V2V.git	b49ab994	77bc0b74
https://github.com/motech/ananya-care-reports.git	67a5f008	8d23d005
https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server.git	f3ca8a27	9fe98e01
https://github.com/openmrs/openmrs-module-webservices.rest.git	5d3cf179	535c8928
https://github.com/dana-i2cat/opennaas-routing-nfv.git	33d426f2	542fd405
https://github.com/thm-projects/arsnova-war.git	ba09f439	11d38ec0
https://github.com/apache/archiva.git	94fe3a7f	57cc17cd
https://github.com/bazhenov/logwatcher.git	7767209b	24288fa4
https://github.com/zhjt19800720/dms-project.git	27346b2d	7a3024c3
https://github.com/vuamitom/openmrs.git	20679480	63336f2b
https://github.com/endosnipe/ENdoSnipe.git	7c7d8df0	09f47325
https://github.com/kabassociates/SpringFaces.git	d70fbf3e	cad4e679
https://github.com/Macarse/TDA.git	85d37c3d	d70404be

https://github.com/motech/motech-server-pillreminder.git	87563b67	25e7e839
https://github.com/Digt/TrustedNet.git	4e1329e0	5ed47e36
https://github.com/nhnopensource/ngrinder.git	42bee554	133f6781
https://github.com/abdulazeezsk/netvogue.git	bd9b93a9	9518a1eb
https://github.com/mstepan/incubator.git	0d1f1894	f9cfdcdf
https://github.com/motech/GHANA-National.git	1ef59dbf	d2fdbcf5
https://github.com/ORCID/ORCID-Source.git	d3d8f917	fae8c25d
https://github.com/svn2github/icemobile.git	17c2feb8	23e4c3d4
https://github.com/svn2github/shopizer-portlet.git	8f1bf67c	f25fd769
https://github.com/svn2github/rave.git	2da933e9	caf825db
https://github.com/INCF/eeg2.git	a8d6f4b8	de633071
https://github.com/sjzheng-arcie/RUCLab.git	f8b4c858	2c7161fe
https://github.com/sdywcd/jshoperv2.git	b778f9b1	911534a1
https://github.com/jhpoelen/eol-globi-data.git	d3fa3ede	ed155563
https://github.com/koa/RAoA.git	577d78c5	285b1a3e
https://github.com/icehockeystar/militaryshop.git	33f6db5b	8610f8b3
https://github.com/rongjihuang/bc-framework.git	940694ca	d169c34c
https://github.com/haha1903/document.git	ad99cc3c	53112f16
https://github.com/jboss-developer/jboss-eap-quickstarts.git	c13c0616	7908a64f
https://github.com/ESGF/esgf-web-fe.git	64054a90	7f3f5228
https://github.com/xdylxdyl/ppteng.git	3a21ee18	55c27adb
https://github.com/xmx0632/deliciousfruit.git	2d1e4a58	562a36b4
https://github.com/jblaya/openmrs-core.git	20679480	641978fb
https://github.com/qalingo/qalingo-b2c-engine.git	2805cf60	b9560e91
https://github.com/maodouzi/PY.git	24fed6d5	ac5a40dc
https://github.com/itru/timesheet.git	61cf78a8	d5ae85c2
https://github.com/YAMJ/yamj-v3.git	e0db7414	bb99f517
https://github.com/datacite/mds.git	d669ac28	0745e85d
https://github.com/alexandergavrikov1989/jcommune-alex.git	f399fa51	9cb7e933
https://github.com/kiivihal/open-europeana.git	585c2404	018bd4cc
https://github.com/niths/niths.git	b2bc7733	7e1fdb88
https://github.com/openforis/calc.git	a088f8e9	9a4aa417
https://github.com/IndiCrowd/was.indicrowd.com.git	e3b38a71	af8c4282
https://github.com/ImmobilienScout24/yum-repo-server.git	920052ab	25a3d69d
https://github.com/UNC-Libraries/Carolina-Digital-Repository.git	24a08ae9	8e3d8825
https://github.com/cnnic/restfulwhois.git	619d44cd	abbb2d18
https://github.com/sanger-pathogens/crawl2.git	25b56d68	d510fcfa
https://github.com/ansell/openrdf-sesame.git	92fd590d	e5bd1ed3
https://github.com/testinfected/simple-petstore.git	25d8ed62	3b402752
https://github.com/RBC1B/ROMS.git	cc0f187c	d1609e61
https://github.com/jdmr/mateo.git	93061468	47bf8770
https://github.com/callista-software/phk-patientdata-pump.git	fa5ff9b0	fb97df5c

Table C.1: *120 Spring MVC projects used in this thesis. * means the project was used in the initial 20 projects sample (Chapter 5), + means the project was used for defect-proneness detection in RQ7.*

Appendix D

Qualitative Data Collection Protocols

D.1 Interview with Experts (RQ2)

D.1.1 Part 1: Profile

1. What is your current position? (Professional software developer, Open source developer, Faculty, PhD student, Graduate student, Undergraduate student)
2. What is your highest degree? (High school, Undergraduate, Master, PhD, Postdoc)
3. How many years of experience in software development?
4. How many years of experience in Spring MVC?
5. How many projects have you worked with Spring MVC?
6. Are there anything that 'shows' your expertise with Spring?

D.1.2 Part 2: Metrics and Architectural roles

Me: I will show you some code metrics we extracted from a set of Spring MVC projects in GitHub. We separated metrics by different architectural roles in a Spring MVC application.

Me: Are you familiar with these Spring MVC stereotypes: Controllers, Components, Repositories, Entities, and Services? If not, remove the ones from the rest of the interview.

Me: As code metrics, we chose WMC, RFC, CBO, NOM, and LCOM. I will explain all of them to you.

- WMC=It is a measure of complexity. It basically counts the number of ifs, fors, whiles, etc in the class.
- RFC=It is a measure of coupling. It counts the number of method invocations that happens in a class.
- CBO=Another coupling measure. It counts the number of classes a class depends upon.
- NOM=Complexity. Counts the number of methods in a class.
- LCOM=It is a cohesion metric. Imagine attributes A, B, C, D, and methods X and Y. If X uses A and B, and Y uses C and D, then it is possible to separate this class in two classes. The higher the number the less cohesive is a class.

For each metric M in the chart:

1. Instruction: Explain the metric (again, if needed). Guide him to check whether the metric is above or below 0. Then, compare to the same metric in other roles.
2. Were you expecting this difference? Does it make sense to you?
3. Can you explain me why this happens?
4. Is there any good practice that could explain it?

D.1.3 Part 3: Conclusions

1. Do you see something that goes against any good practice?

D.2 Layer-focused survey

Good and Bad Practices in Web Application Development

This research is focused on good and bad practices in web application development. Our aim is to improve the quality of software development. Help us out to find what the best and bad practices are.

You will take around 10 15 minutes to fill up this survey. It may look like a long time to spend in a single survey, but remember: your 10 minutes may be really helpful for you in the future.

We divided the survey into many pages in this survey. So, please click on "Continue". We do appreciate your time.

A little bit about yourself

Q: Years of experience with software development, in general.

Q: Years of experience with web application development

Q: How many web applications have you worked that went to production?

Q: Language that you have used more to develop your web applications?

Q: How do you see yourself in terms of good practices (and code quality) of web application development? 1-10

Q: How often do you study/read about good practices in web application development? 1-5

Q: What is the framework you have used more?

The "M" - Model

In MVC, "models" are responsible to all business logic, as well as any other logic, such as data access, webservices, and so on.

Q: Is there any characteristic that would make a infrastructure code in domain classes *problematic*? Is there any characteristic that would make it *not problematic*?

Q: Is there any characteristic that would make a long method *problematic*? Is there any characteristic that would make it *not problematic*?

Q: Do you have any good practices to deal with domain classes?

Q: Do you have anything you consider a bad practice when dealing with domain classes?

Data Access Objects (DAOs)

A web app usually consumes a database. It is common to isolate that code in DAOs.

Q: Is there any business logic that you *would* consider a problem in a DAO? Is there any business logic what you *would not* consider a problem in a DAO?

Q: Do you have any good practices to deal with DAO classes?

Q: Do you have anything you consider a bad practice when dealing with DAO classes?

The "C" - Controller

Controllers are responsible to connect the web world (with all that HTTP requests and responses) and the domain model (with all the business logic).

Q: Is there any business logic that *would be* a problem in a controller? Is there any businness logic that *would not be* a problem?

Q: Do you have any good practices to deal with Controller classes?

Q: Do you have anything you consider a bad practice when dealing with Controller classes?

The "V" - View

Views are responsible to present the data and interact with the user. Developers usually use a mixture of HTML, CSS, LESS, SASS, Javascript, etc.

Q: Is there any characteristic that *would* make a scriptlet problematic? Is there any characteristic that *would not* make it problematic?

Q: Do you have any good practices to deal with View files?

Q: Do you have anything you consider a bad practice when dealing with View files?

Last thoughts

Q: Are there any other *GOOD* practices we did not asked you or you did not said yet?

Q: Are there any other *BAD* practices we did not asked you or you did not said yet?

D.3 Role-focused survey

My name is Maurício, I am a PhD student researching code quality. I found your Github profile, and I believe your experience in Spring MVC can be really helpful to us.

We are studying code smells in Spring MVC. By smell, we mean symptoms of poor design and implementation choices. Smells contribute to technical debt, and possibly affect the maintainability of a software system.

We want you to tell us about your experience. For each architectural role in Spring, we want you to describe smells you have faced and why you think they are smells. The survey has around 10 questions and requires less than 15 minutes to complete.

If you have any questions, just contact us at mauricioaniche@gmail.com.

Background

Q: What is your current position?

Q: How many years of experience in software development do you have?

Q: How many years of experience in Spring MVC do you have?

Q: How many Spring MVC applications have you put in production?

Smells in Spring MVC

We want you to tell us about your experience. For each architectural role in Spring, we want you to describe smells (all of them!) you have faced and why you think they are smells. By smell, we mean symptoms of poor design and implementation choices. Smells contribute to technical debt, and possibly affect the maintainability of a software system. If you perceive the same smell in more than one architectural role, please copy and paste or refer to it.

Q: What are the smells you face in Controllers?

Q: What are the smells you face in Entities?

Q: What are the smells you face in Services?

Q: What are the smells you face in Components?

Q: What are the smells you face in Repositories?

Final thoughts

Q: Do you have any other thoughts on smells in Spring MVC which you haven't mention yet?

Q: Can we contact you for an interview if we need to?

D.4 Interviews with developers

Q: What is your name?

Q: What is your current position?

Q: How many years of experience in software development do you have?

Q: How many years of experience in Web development do you have?

Q: What are the smells you face in Controllers?

Q: What are the smells you face in Entities?

Q: What are the smells you face in Services?

Q: What are the smells you face in Components?

Q: What are the smells you face in Repositories?

Appendix E

Participants' rules in card interviews

Table E.1: *Rules that were created by participants*

#	Role	Comparison sign	OOP Concept	Role	Qty of Rules
1	Service	>	complexity	Controller	9
2	Service	>	coupling	Controller	8
3	Entity	>	inheritance	Controller	5
4	Entity	>	inheritance	Service	5
5	Service	>	complexity	Repository	4
6	Service	>	coupling	Entity	4
7	Entity	>	cohesion	Service	3
8	Entity	>	complexity	Controller	3
9	Entity	>	inheritance	component	3
10	Entity	>	inheritance	Repository	3
11	Service	>	complexity	Entity	3
12	Service	>	coupling	Repository	3
13	component	>	cohesion	Service	2
14	Entity	>	cohesion	Controller	2
15	Entity	>	cohesion	Repository	2
16	Service	>	coupling	component	2
17	component	>	complexity	Repository	1
18	Controller	>	cohesion	Service	1
19	Controller	>	complexity	Repository	1
20	Controller	>	complexity	Service	1
21	Controller	>	coupling	Service	1
22	Entity	>	cohesion	component	1
23	Entity	>	complexity	Service	1
24	Repository	>	cohesion	Controller	1
25	Repository	>	cohesion	Service	1
26	Repository	>	complexity	Controller	1
27	Repository	>	coupling	Controller	1
28	Repository	>	inheritance	Service	1
29	Service	>	complexity	component	1
30	Service	>	inheritance	Entity	1

Bibliography

[1] **Abbes** *et al.***(2011)** Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. Em *Software Maintenance and Reengineering (CSMR), 15th European Conf. on*, páginas 181–190. IEEE. Cited in page 17

[2] **Alur** *et al.***(2003)** Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies.* Sun Microsystems, Inc. Cited in page 2, 17

[3] **Alves** *et al.***(2010)** Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. Em *Software Maintenance (ICSM), IEEE Intl. Conf. on.* IEEE. Cited in page 4, 5, 12, 16, 21, 22, 30, 43, 59, 62

[4] **Alves** *et al.***(2011)** Tiago L Alves, José Pedro Correia, and Joost Visser. Benchmark-based aggregation of metrics to ratings. Em *Software Measurement, Joint Conf. of the 21st Int'l Workshop on and 6th Int'l Conf. on Software Process and Product Measurement (IWSM-MENSURA).* IEEE. Cited in page 16, 62

[5] **Aniche and de Azevedo Silveira(2011)** Mauricio Finavaro Aniche and Guilherme de Azevedo Silveira. Increasing learning in an agile environment: Lessons learned in an agile team. Em *Agile Conference (AGILE), 2011*, páginas 289–295. IEEE. Cited in page 42

[6] **Aniche(2016)** Maurício Aniche. Ck calculator. http://www.github.com/mauricioaniche/ck., 2016. Cited in page 35

[7] **Aniche(2016)** Maurício Aniche. Springlint. http://www.github.com/mauricioaniche/springlint, 2016. Cited in page 7, 59

[8] **Aniche and Gerosa(2015)** Maurício Aniche and Marco Gerosa. Boas e más práticas em desenvolvimento web com mvc: Resultados de um questionário com profissionais. *3rd Workshop of Software Visualization, Evolution and Maintenance.* Cited in page 32, 34

[9] **Aniche** *et al.***()** Maurício Aniche, Christoph Treude, and Marco Aurélio Gerosa. Appendix: Developers' perceptions on object-oriented design and system architecture. http://mauricioaniche.github.io/sbes2016. Cited in page 40

[10] **Aniche** *et al.***(2016)** Maurício Aniche, Gabriele Bavota, Christoph Treude, Arie van Deursen, and Marco Aurélio Gerosa. A validated set of smells in model-view-controller architecture. Em *Software Maintenance and Evolution (ICSME), 2016 IEEE 31th International Conference on.* IEEE. Cited in page 29

[11] **Aniche** *et al.***(2016)** Maurício Aniche, Gabriele Bavota, Christoph Treude, Arie van Deursen, and Marco Aurélio Gerosa. A validated set of smells in model-view-controller architectures? replication package., 2016. URL http://mauricioaniche.github.io/icsme2016. Cited in page 47, 55

[12] **Aniche** *et al.***(2016)** Maurício Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. Satt: Tailoring code metric thresholds for different software architectures: Replication package., 2016. URL http://mauricioaniche.github.io/scam2016. Cited in page 23, 25, 27

[13] **Arcoverde** *et al.***(2011)** Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. Em *Proceedings of the 4th Workshop on Refactoring Tools*, páginas 33–36. ACM. Cited in page 5, 17

[14] **Athanasiou** *et al.***(2014)** Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering (TSE)*, 40(11):1100–1125. Cited in page 5, 16

[15] **Ayalew and Mguni(2013)** Yirsaw Ayalew and Kagiso Mguni. An assessment of changeability of open source software. *Computer and Information Science*, 6(3). Cited in page 16

[16] **Basili** *et al.***(1996)** Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10). Cited in page 4, 12

[17] **Baxter** *et al.***(2006)** Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of java software. Em *ACM Sigplan Notices*, volume 41. ACM. Cited in page 15

[18] **Bieman and Kang(1995)** James M Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. Em *ACM SIGSOFT Software Engineering Notes*, volume 20, páginas 259–262. ACM. Cited in page 14

[19] **Bouwers** *et al.***(2012)** Eric Bouwers, Joost Visser, and Arie Van Deursen. Getting what you measure. *Communications of the ACM*, 55(7):54–59. Cited in page 3, 61

[20] **Briand** *et al.***(1998)** Lionel C Briand, John Daly, Victor Porter, and Jürgen Wüst. Predicting fault-prone classes with design measures in object-oriented systems. Em *Software Reliability Engineering. Proceedings. The Ninth Intl. Symposium on.* IEEE. Cited in page 4, 16

[21] **Brown** *et al.***(1998)** William H Brown, Raphael C Malveau, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* Wiley. Cited in page 13, 45

[22] **Chen** *et al.***(2014)** Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. Em *Proceedings of the 36th Intl. Conference on Software Engineering*, páginas 1001–1012. ACM. Cited in page 3

[23] **Chi** *et al.***(1981)** Michelene TH Chi, Paul J Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive science*, 5(2):121–152. Cited in page 25

[24] **Chidamber and Kemerer(1994)** Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6). Cited in page 2, 5, 6, 12, 33, 49

[25] **Clauset** *et al.***(2009)** Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4). Cited in page 5, 15

[26] **Coleman** *et al.***(1995)** Don Coleman, Bruce Lowther, and Paul Oman. The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1). Cited in page 15

[27] **Concas** *et al.***(2007)** Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *Software Engineering, IEEE Transactions on*, 33(10). Cited in page 15

[28] **Conover(1998)** W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edição. Cited in page 54

[29] **Crupi** *et al.***(2001)** John Crupi, Dan Malks, and DEEPAK ALUR. *Core J2EE Patterns*. Gulf Professional Publishing. Cited in page 17

[30] **De Lucia** *et al.***(2015)** Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Towards automating dynamic analysis for behavioral design pattern detection. Em *Software Maintenance and Evolution (ICSME), IEEE Intl. Conf. on*. IEEE. Cited in page 18

[31] **Dyer** *et al.***(2013)** Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. Em *Proceedings of the Intl. Conf. on Software Engineering*. IEEE Press. Cited in page 22, 46

[32] **D'Ambros** *et al.***(2012)** Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577. Cited in page 16, 62

[33] **Eaddy** *et al.***(2008)** Marc Eaddy, Thomas Zimmermann, Kaitlin D Sherwood, Vibhav Garg, Gail C Murphy, Nachiappan Nagappan, and Alfred V Aho. Do crosscutting concerns cause defects? *Software Engineering, IEEE Transactions on*, 34(4):497–515. Cited in page 17

[34] **Elish and Rine(2003)** Mahmoud O Elish and David Rine. Investigation of metrics for object-oriented design logical stability. Em *Software Maintenance and Reengineering. Proceedings. Seventh European Conf. on*. IEEE. Cited in page 4

[35] **Erni and Lewerentz(1996)** Karin Erni and Claus Lewerentz. Applying design-metrics to object-oriented frameworks. Em *Software Metrics Symposium, Proceedings of the 3rd Intl.* IEEE. Cited in page 5, 15

[36] **Eski and Buzluca(2011)** Sinan Eski and Feza Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. Em *Software Testing, Verification and Validation Workshops (ICSTW), IEEE Fourth Intl. Conf. on*. IEEE. Cited in page 4, 16

[37] **Espinosa** *et al.***(2007)** J Alberto Espinosa, Sandra A Slaughter, Robert E Kraut, and James D Herbsleb. Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, 24(1):135–169. Cited in page 42

[38] **Evans(2004)** Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional. Cited in page 26

[39] **Fabry and Mens(2004)** Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. *Computer Languages, Systems & Structures*, 30(1). Cited in page 17

[40] **Faraj and Sproull(2000)** Samer Faraj and Lee Sproull. Coordinating expertise in software development teams. *Management science*, 46(12):1554–1568. Cited in page 42

[41] **Fard and Mesbah(2013)** Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. Em *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th Intl. Working Conference on*, páginas 116–125. IEEE. Cited in page 18

[42] **Ferreira** *et al.***(2012)** Kecia AM Ferreira, Mariza AS Bigonha, Roberto S Bigonha, Luiz FO Mendes, and Heitor C Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2). Cited in page 4, 5

[43] **Fischer** *et al.***(2003)** Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. Em *Software Maintenance. ICSM. Proceedings. Intl. Conf. on.* IEEE. Cited in page 51, 58

[44] **Fontana** *et al.***(2015)** F Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Preliminary catalogue of anti-pattern and code smell false positives. *Poznan University of Technology, Tech. Rep. RA-5/15.* Cited in page 29

[45] **Fontana** *et al.***(2015)** Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. Em *Proceedings of the Sixth Intl. Workshop on Emerging Trends in Software Metrics.* IEEE Press. Cited in page 16, 30

[46] **Fowler(1997)** Martin Fowler. Refactoring: Improving the design of existing code. Em *11th European Conf.. Jyväskylä, Finland.* Cited in page 2, 6, 13, 16, 45, 47

[47] **Fowler(2002)** Martin Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc. Cited in page 3, 4

[48] **Gamma** *et al.***(1994)** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education. Cited in page 64

[49] **Gharachorlu(2014)** Golnaz Gharachorlu. *Code smells in cascading style sheets: an empirical study and a predictive model.* Tese de Doutorado, University of British Columbia. Cited in page 18

[50] **Gil and Lalouche(2016)** Joseph Yossi Gil and Gal Lalouche. When do software complexity metrics mean nothing?–when examined out of context. *Journal of Object Technology*, 15(1). Cited in page 3, 16, 20, 29, 61

[51] **Grissom and Kim(2005)** Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach.* Lawrence Earlbaum Associates, 2nd edition edição. Cited in page 54

[52] **Gueheneuc** *et al.***(2004)** Yann-Gael Gueheneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. Em *Reverse Engineering, Proceedings. 11th Working Conf. on.* IEEE. Cited in page 17

[53] **Gyimothy** *et al.***(2005)** Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10). Cited in page 4, 12, 16

[54] **Hat(2016)** Red Hat. Hibernate. http://hibernate.org/, 2016. Cited in page 12

[55] **Hecht** *et al.***(2015)** Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in android apps. Em *Proceedings of the Second ACM Intl. Conference on Mobile Software Engineering and Systems*, páginas 148–149. IEEE Press. Cited in page 3

[56] **Hermans** *et al.***(2012)** Frederik Hermans, Martin Pinzger, and Arie van Deursen. Detecting code smells in spreadsheet formulas. Em *Software Maintenance (ICSM), 2012 28th IEEE Intl. Conference on*, páginas 409–418. IEEE. Cited in page 3

[57] **Herraiz** *et al.***(2011)** Israel Herraiz, Daniel M German, and Ahmed E Hassan. On the distribution of source code file sizes. Em *ICSOFT (2)*, páginas 5–14. Cited in page 5, 15

[58] **Herraiz** *et al.***(2012)** Israel Herraiz, Daniel Rodriguez, and Rachel Harrison. On the statistical distribution of object-oriented system properties. Em *Emerging Trends in Software Metrics (WETSoM), 3rd Intl. Workshop on.* IEEE. Cited in page 5, 16

[59] **Herzig and Zeller(2013)** Kim Herzig and Andreas Zeller. The impact of tangled code changes. Em *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, páginas 121–130. Cited in page 59

[60] **Heuzeroth** *et al.***(2003)** Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. Em *Program Comprehension,. 11th IEEE Intl. Workshop on.* IEEE. Cited in page 18

[61] **Janes** *et al.***(2006)** Andrea Janes, Marco Scotto, Witold Pedrycz, Barbara Russo, Milorad Stefanovic, and Giancarlo Succi. Identification of defect-prone classes in telecommunication software systems using design metrics. *Information sciences*, 176(24). Cited in page 16

[62] **Jureczko and Spinellis(2010)** Marian Jureczko and Diomidis Spinellis. Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej.* Cited in page 16

[63] **Kalliamvakou** *et al.***(2014)** Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. Em *Proceedings of the 11th working conference on mining software repositories*, páginas 92–101. ACM. Cited in page 30

[64] **Khomh** *et al.***(2009)** Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. Em *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, páginas 75–84. IEEE. Cited in page 6, 45, 56

[65] **Khomh** *et al.***(2012)** Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3). Cited in page 6, 13, 17, 45, 52, 56

[66] **Kim** *et al.***(2008)** Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196. Cited in page 52

[67] **Krasner** *et al.***(1988)** Glenn E Krasner, Stephen 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). Cited in page 2, 11

[68] **Lanza and Marinescu(2007)** Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media. Cited in page vii, 2, 5, 6, 7, 13, 14, 15, 16, 29, 54

[69] **Larman(2004)** Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, third edição. Cited in page 64

[70] **Li and Henry(1993)** Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2). Cited in page 4, 12

[71] **Li and Shatnawi(2007)** Wei Li and Raed 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. Cited in page 6, 17

[72] **Marinescu(2001)** Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. Em *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, páginas 173–182. IEEE. Cited in page 14

[73] **Marinescu(2004)** Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. Em *Software Maintenance, 20th IEEE Intl. Conf. on.* IEEE. Cited in page 5, 6, 13, 14, 16, 29

[74] **Martin(2009)** Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education. Cited in page 50

[75] **Mazinanian and Tsantalis(2016)** Davood Mazinanian and Nikolaos Tsantalis. An empirical study on the use of css preprocessors. Em *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd Intl. Conference on.* Cited in page 3

[76] **Mazinanian** *et al.***(2014)** Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. Em *Proceedings of the 22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, páginas 496–506. ACM. Cited in page 18

[77] **McCabe(1976)** Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, 4. Cited in page 4, 12, 15, 49

[78] **Mesbah and Mirshokraie(2012)** Ali Mesbah and Shabnam Mirshokraie. Automated analysis of css rules to support style maintenance. Em *Software Engineering (ICSE), 2012 34th Intl. Conference on*, páginas 408–418. IEEE. Cited in page 18

[79] **Misra and Bhavsar(2003)** Subhas C Misra and Virendra C Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. Em *Computational Science and Its Applications—ICCSA 2003.* Springer. Cited in page 4, 16

[80] **Moha** *et al.***(2010)** Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1). Cited in page 17

[81] **Nakagawa(2004)** Shinichi Nakagawa. A farewell to bonferroni: the problems of low statistical power and publication bias. *Behavioral Ecology*, 15(6). Cited in page 36

[82] **Nederlof** *et al.***(2014)** Alex Nederlof, Ali Mesbah, and Arie van Deursen. Software engineering for the web: the state of the practice. Em *Companion Proceedings of the 36th Intl. Conference on Software Engineering*, páginas 4–13. ACM. Cited in page 18

[83] **Nejmeh(1988)** Brian A Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2). Cited in page 15

[84] **Olbrich** *et al.***(2010)** Steffen M Olbrich, Daniela S Cruze, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. Em *Software Maintenance (ICSM), IEEE Intl. Conf. on.* IEEE. Cited in page 52

[85] **Oliveira** *et al.***(2014)** Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. Extracting relative thresholds for source code metrics. Em *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Software Evolution Week-IEEE Conf. on.* IEEE. Cited in page 16, 30, 59, 62

[86] **Padilha** *et al.***(2014)** Juliana Padilha, Juliana Pereira, Eduardo Figueiredo, Jussara Almeida, Alessandro Garcia, and Cláudio Sant'Anna. On the effectiveness of concern metrics to detect code smells: An empirical study. Em *Advanced Information Systems Engineering*, páginas 656–671. Springer. Cited in page 17

[87] **Palomba** *et al.***(2013)** Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. Em *Automated Software Engineering (ASE), IEEE/ACM 28th Intl. Conf. on*. IEEE. Cited in page 17

[88] **Palomba** *et al.***(2014)** Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. Em *Software Maintenance and Evolution (ICSME), IEEE Intl. Conf. on*. IEEE. Cited in page 5, 17, 45, 52

[89] **Palomba** *et al.***(2015)** Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489. Cited in page 15

[90] **Panichella** *et al.***(2015)** Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? Em *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, páginas 161–170. IEEE. Cited in page 28, 42

[91] **Parnas(1972)** D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361598.361623. URL http://doi.acm.org/10.1145/361598.361623. Cited in page 64, 65

[92] **Perneger(1998)** Thomas V Perneger. What's wrong with bonferroni adjustments. *BMJ: British Medical Journal*, 316(7139). Cited in page 36

[93] **Peters and Zaidman(2012)** Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. Em *Software Maintenance and Reengineering (CSMR), 16th European Conf. on*. IEEE. Cited in page 17, 52

[94] **Pivotal(2016)** Pivotal. Spring documentation. http://spring.io/docs., 2016. Cited in page 11

[95] **Pivotal(2004)** Pivotal. Spring mvc. https://projects.spring.io/spring-framework/., 2004. Cited in page 3

[96] **PMD(2016)** PMD. Pmd. https://pmd.github.io/., 2016. Cited in page 2, 5, 15, 32, 52

[97] **PMD(2016)** PMD. Pmd cbo documentation. https://pmd.github.io/pmd-5.4.1/pmd-java/rules/java/coupling.html#CouplingBetweenObjects., 2016. Cited in page 28

[98] **Potanin** *et al.***(2005)** Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48(5). Cited in page 15

[99] **Riel(1996)** Arthur J Riel. *Object-oriented design heuristics*, volume 335. Addison-Wesley Reading. Cited in page 14, 16

[100] **Romano** *et al.***(2006)** Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff 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. Em *annual meeting of the Florida Association of Institutional Research*. Cited in page 21, 36

[101] **Rozanski and Woods(2012)** Nick Rozanski and Eóin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley. Cited in page 2, 17

[102] **Sahraoui** *et al.***(2000)** Houari Sahraoui, Robert Godin, Thieny Miceli, *et al.* Can metrics help to bridge the gap between the improvement of oo design quality and its automation? Em *Software Maintenance, Intl. Conf. on*. IEEE. Cited in page 4

[103] **Schmidt(1995)** Douglas C Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74. Cited in page 17

[104] **Shatnawi(2010)** Raed Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *Software Engineering, IEEE Transactions on*, 36(2). Cited in page 5

[105] **Shatnawi and Li(2008)** Raed Shatnawi and Wei Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of systems and software*, 81(11). Cited in page 4, 16

[106] **Shepperd(1988)** Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2). Cited in page 12

[107] **Sheskin(2003)** David J Sheskin. *Handbook of parametric and nonparametric statistical procedures.* crc Press. Cited in page 37, 53

[108] **Silva** *et al.***(2015)** Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. Does javascript software embrace classes? Em *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd Intl. Conference on*, páginas 73–82. IEEE. Cited in page 18

[109] **Sjoberg** *et al.***(2013)** Dag IK Sjoberg, Atsushi Yamashita, Bente Cecilie Dahlum Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156. Cited in page 45

[110] **Sonarqube(2016)** Sonarqube. Sonarqube. http://www.sonarqube.org/., 2016. Cited in page 2, 5, 15, 32

[111] **Sun Microsystems(2007)** Inc. Sun Microsystems. Core j2ee patterns - data access object. http://www.oracle.com/technetwork/java/dataaccessobject-138824.html, 02 2007. Cited in page 65

[112] **Tsantalis** *et al.***(2006)** Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11). Cited in page 17

[113] **Tsantalis** *et al.***(2008)** Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. Em *Software Maintenance and Reengineering, 12th European Conf. on.* IEEE. Cited in page 2, 5

[114] **Tufano** *et al.***(2015)** Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. Em *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, páginas 403–414. IEEE Press. Cited in page 13, 16

[115] **Turnaround(2015)** Zero Turnaround. Top 4 java web frameworks revealed: Real life usage data of spring mvc, vaadin, gwt and jsf. http://bit.ly/1smVDf9, 2015. Cited in page 11, 51

[116] **Verloop(2013)** D Verloop. *Code Smells in the Mobile Applications Domain.* Tese de Doutorado, TU Delft, Delft University of Technology. Cited in page 3

[117] **Vlissides** *et al.***(1995)** John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49 (120):11. Cited in page 2, 17

[118] **Webster(1995)** Bruce Webster. *Pitfalls of Object-Oriented Development.* M T Books. Cited in page 16

[119] **Wheeldon and Counsell(2003)** Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. Em *Source Code Analysis and Manipulation, Proceedings. Third IEEE Intl. Workshop on.* IEEE. Cited in page 15

[120] **Wilcoxon(1945)** Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin.* Cited in page 35, 36

[121] **Wilkie and Kitchenham(2000)** F George Wilkie and Barbara A Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2). Cited in page 4, 16

[122] **Wirfs-Brock and McKean(2002)** Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities, and Collaborations.* Pearson Education. ISBN 0201379430. Cited in page 64

[123] **Yamashita and Moonen(2013)** Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. Em *2013 20th Working Conference on Reverse Engineering (WCRE)*, páginas 242–251. IEEE. Cited in page 5, 17, 42

[124] **Yamashita and Moonen(2013)** Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. Em *Proceedings of the 2013 Intl. Conference on Software Engineering*, páginas 682–691. IEEE Press. Cited in page 17, 45

[125] **Yamashita and Moonen(2012)** Atsushi Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? Em *Software Maintenance (ICSM), 2012 28th IEEE Intl. Conference on*, páginas 306–315. IEEE. Cited in page 45

[126] **Yi** *et al.***(2009)** Yao Yi, Huang Song, Ren Zheng-ping, and Liu Xiao-ming. Scale-free property in large scale object-oriented software and its significance on software engineering. Em *Information and Computing Science, Second Intl. Conf. on*, volume 3. IEEE. Cited in page 5, 15

[127] **Zazworka** *et al.***(2013)** Nico Zazworka, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull, *et al.* Comparing four approaches for technical debt identification. *Software Quality Journal*, páginas 1–24. Cited in page 64

[128] **Zhang** *et al.***(2013)** Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. How does context affect the distribution of software maintainability metrics? Em *IEEE International Conference on Software Maintenance*, páginas 350–359. IEEE. Cited in page 3, 16, 20, 29, 61