A Validated Set of Smells in Model-View-Controller Architectures

Maurício Aniche[1,4], Gabriele Bavota[2], Christoph Treude[3], Arie van Deursen[1], Marco Aurélio Gerosa[4]
{m.f.aniche,arie.vandeursen}@tudelft.nl, gabriele.bavota@gmail.com, christoph.treude@adelaide.edu.au, gerosa@ime.usp.br
[1]Delft University of Technology - The Netherlands, [2]Università della Svizzera Italiana (USI) - Switzerland
[3]University of Adelaide - Australia, [4]University of São Paulo - Brazil

***Abstract*—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and defect-proneness. A vast catalogue of smells has been defined in the literature, and it includes smells that can be found in any kind of system (*e.g.*, God Classes), regardless of their architecture. On the other hand, software systems adopting specific architectures (*e.g.*, the Model-View-Controller pattern) can be also affected by other types of poor practices. We surveyed and interviewed 53 MVC developers to collect bad practices to avoid while working on Web MVC applications. Then, we followed an open coding procedure on the collected answers to define a catalogue of six Web MVC smells, namely SMART REPOSITORY, FAT REPOSITORY, PROMISCUOUS CONTROLLER, SMART CONTROLLER, LABORIOUS REPOSITORY METHOD, and MEDDLING SERVICE. Then, we ran a study on 100 MVC projects to assess the impact of these smells on code change- and defect-proneness. In addition, we surveyed 21 developers to verify their perception of the defined smells. The achieved results show that the Web MVC smells (i) more often than not, increase change- and defect-proneness of classes, and (ii) are perceived by developers as severe problems.**

I. 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 [1], [2]. Evidence in the literature suggests that code smells can hinder code maintainability [3], [4], [5], and increase change- and defect-proneness [6], [7].

While these smells fit well in any object-oriented system, they do not take into account the underlying architecture of the application or the role played by a given class. For example, in web systems relying on the MVC pattern [8], CONTROLLERS are classes responsible to control the flow between the view and the model layers. Commonly, these classes 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 [9] in MVC applications are responsible for dealing with the communication towards the databases. These classes, besides not containing complex and long methods (traditional smells) should also limit the complexity of SQL queries residing in them.

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 needed to capture "bad practices" on software systems adopting a specific architecture. 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 alarm developers about the existence of the smell, and (iii) no empirical studies about the impact of these bad practices on code maintainability properties. For these reasons, good and bad practices that are specific to a platform, architecture or technology have been recently emerging as a research topic in software maintenance. In particular, researchers have studied smells specific to the usage of object-relational mapping frameworks [10], Android apps [11], [12], and Cascading Style Sheets (CSS) [13].

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

Specifically, the main contributions of this paper are:

1) *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.

2) *Detection strategies for each of the catalogued smells*. We followed Lanza and Marinescu's approach [14] to propose detection strategies. These strategies have been implemented in an open source detection tool [15].
3) *An empirical study on the impact of the catalogued smells on change- and defect-proneness of classes*. We evaluated the impact of each smell in 100 Spring MVC projects.
4) *A survey on developers' perception of the smells*. We performed a survey with 21 software developers and captured their perceptions on the catalogued smells.
5) *A publicly available replication package* [16], reporting all data collected in our studies.

II. The Catalogue of Web MVC Smells

This section presents the catalogue of Web MVC smells and the details of the method adopted in its definition.

A. Background in MVC Web Development

The MVC pattern [8] has been widely adopted by the web development industry. Frameworks such as Spring MVC (Java), ASP.NET MVC (.NET), Ruby on Rails (Ruby), and Django (Python) have MVC in their core. Thus, developers need to write code for each one of the three layers of the MVC. In this paper, we focus on the server-side code developers are required to write in both CONTROLLER and MODEL layers.

CONTROLLERS, as the MVC pattern states, take care of the flow between the model and the view layers. The MODEL layer represents the business model. In this layer, developers commonly make use of other patterns [9], [17], such as *Entities*, *Repositories*, and *Services*. ENTITIES represent a domain object (e.g., an *Item* or a *Product*). REPOSITORIES are responsible to encapsulate persistence logic, similar to Data Access Objects [9]. Finally, SERVICES are implemented when there is a need to offer an operation that stands alone in the model, with no encapsulated state. It is also common to write utility classes, which are commonly called COMPONENTS; practical examples of them can be UI formatting or data conversion.

As discussed in detail in Section III-B, we evaluated the impact of the catalogued smells in Spring MVC projects, a popular Java framework for web development. Indeed, these different architectural roles can be seen in all the aforementioned frameworks.

B. 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.

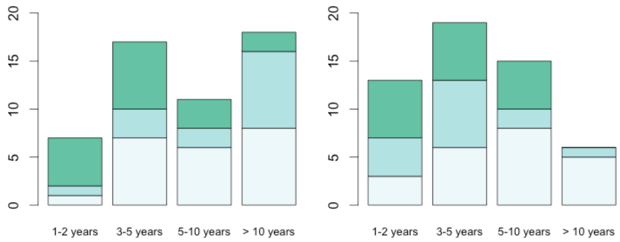
Step 2: Role-focused survey (S2). We designed a survey aimed at investigating good and bad practices related to code components playing a specific role in the MVC architecture in web applications.

The questionnaire contained five open questions, one for each of the roles mentioned in Section II-A: 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 [18], a dataset with structured information about projects in GitHub. We received 14 answers to this survey.

Step 3: Unstructured interviews with industrial developers (S3). We interviewed 17 professional developers from one of our industry partners. 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 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.

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 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 [1], 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.,



(a) Software Development (b) Web Development

Figure 1: Participants' experience in software and web development in S1 (bottom), S2, S3 (top).

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 [16], while in the following (Section II-C), 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.

C. Resulting Catalogue of Web MVC Smells

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[1]. 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 II-B).

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 1, we present the detection strategy, where α and β are thresholds.

$$(NOR > \alpha) \vee (NSD > \beta) \qquad (1)$$

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.

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 [19], 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 2, we present the detection strategy, where α represents the threshold:

$$(NFRFC > \alpha) \qquad (2)$$

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

[1]Thresholds used in the detection strategies have been tuned as described in Section III-D.

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 3, we present its detection strategy for a class C:

$$\exists persistenceDependency(C) \tag{3}$$

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 [20] 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 4, we present the detection strategy, where α and β are thresholds:

$$(McCabe > \alpha \wedge SQLComplexity > \beta) \tag{4}$$

Laborious Repository Method. As a good practice, a method should have only one responsibility and do one thing [21]. 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, we present the smell's detection strategy for class C:

$$\forall m \in C \ \exists \ qtyPersistenceActions(m) > 1 \tag{5}$$

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 6, we present the detection strategy, where α is the threshold:

$$(CTE > \alpha) \tag{6}$$

In the following sections, 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.

III. SMELL EVALUATION STUDY DESIGN

The *goal* of the study is to investigate whether the defined catalogue of MVC smells has impact on 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.

A. Research Questions

Our study aims at addressing the following three research questions:

RQ$_1$. *What is the impact of the proposed code smells on change-proneness of classes?* Previous studies have shown that the "traditional smells" (*e.g.*, Blob Classes) [1] can increase class change-proneness [6], [7]. This research question aims at investigating the impact of the six Web MVC smells on change-proneness of classes.

RQ$_2$. *What is the impact of the proposed code smells on defect-proneness of classes?* This research question mirrors RQ$_1$. Traditional smells are also known by their impact on the defect-proneness of classes [6], [7]. Thus, we compare the impact of the six defined smells on defect-proneness of classes.

RQ$_3$. *Do developers perceive classes affected by the proposed code smells as problematic?* Our last research question qualitatively complements the quantitative analysis performed in the context of RQ$_1$ and RQ$_2$. Here we investigate whether classes affected by the defined Web MVC code smells are perceived as problematic by developers.

B. 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 II-A (*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 [22], it is widely adopted by developers ($>$ 40% of the respondents claimed to use it).

We use BOA [18] to select our sample. BOA allows users to query its data using its own domain specific language. We define a query[2] to select Spring MVC projects: (i) having more than 500 commits in their history, and (ii) containing at least 10 CONTROLLERS. Although the constants 500 and 10 are chosen by convenience, we conjecture that they filter out pet projects and small experiments developers host on GitHub. We also manually inspect the sample to make sure they were stand-alone systems. We end up with 120 Spring MVC projects. The complete list is available in our online appendix [16], while Table I reports size attributes of the subject systems.

From the 120 subject projects, 20 are randomly selected[3], to tune the thresholds of our detection strategies, as described in Section III-D. The remaining 100 are used, as detailed in Section III-C, to answer our research questions.

To answer RQ$_3$, 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 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.

[2]Job ID in BOA: 11947.

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

Table I: Size attributes of the 120 subject systems.

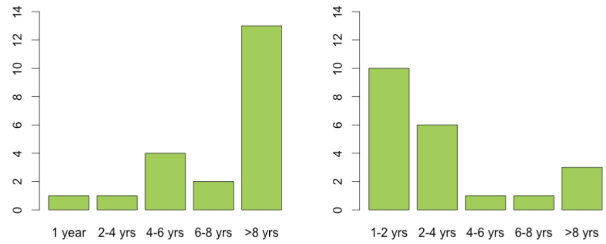
Role	Total classes	Median per proj	Total SLOC	Median class size
Controller	3,126	20	365,274	79
Repository	1,325	14	105,842	46
Service	2,845	16	326,778	59
Entity	1,666	20	169,838	78
Component	2,167	12	158,975	43
Others	52,397	269	3,654,035	39

C. Data Collection and Analysis

To answer RQ$_1$ and RQ$_2$, we need to assess the impact on change- and defect-proneness, respectively, of the defined Web MVC smells. Firstly, it is important to clarify that, while we answer RQ$_1$ 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 (RQ$_2$). 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 [23]. 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 RQ$_2$ and listed in our online appendix [16].

To assess the impact on change- and defect-proneness of the Web MVC smells, we follow an approach similar to what is done in a previous study [6] investigating traditional smells. Firstly, as performed by Kim *et al.* [24], we split the change history of the subject systems (100 for RQ$_1$ and 16 for RQ$_2$) 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.



(a) Software Development (b) Spring MVC

Figure 2: Participants' experience

We obtain 291 chunks for systems used in RQ_1 and 77 for those used in RQ_2. 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.* [6], 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 5.4.2 [25], 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 [26], [27], [28].

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 [29] to test whether the proportions of $C_{MVC-smelly}$/$NC_{MVC-smelly}$ and C_{Clean}/NC_{Clean} significantly differ. As a baseline, we also compare the differences between $C_{T-smelly}$/$NC_{T-smelly}$ and C_{Clean}/NC_{Clean}. In addition, we use the Odds Ratio (OR) [29] 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 (RQ_2). 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 $\mathbf{ND}_{Clean}$, $\mathbf{D}_{Clean}$, $\mathbf{ND}_{MVC-smelly}$, $\mathbf{D}_{MVC-smelly}$, $\mathbf{ND}_{T-smelly}$, $\mathbf{D}_{T-smelly}$, 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. We also made sure to remove classes that were affected by both smells (MVC- and T-smell). In addition, 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 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 selected classes are randomly selected for each participant from a set of 90 classes randomly sampled from the 100 subject systems. This set contains 30 classes affected by one of the proposed MVC-smells (five classes per smell type), 30 classes affected by the six traditional smells (five classes per smell type), and 30 non-smelly classes. Note that also in this case we reduce possible bias 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 evaluated six randomly selected classes, two from each of these three groups, *i.e.*, two MVC-smelly, two T-smelly, two clean classes.

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 that the survey studied 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 [30]. The latter is used 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 [31] 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 $|d| \geq 0.474$ [31]. Finally, we report qualitative findings derived from the participants' open answers.

D. 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 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 [14] in order to define thresholds for their detection strategies.

$$TS = 3Q + 1.5 \times IQR \tag{7}$$

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

Table II: Thresholds 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

IV. RESULTS

Table III 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%).

Table III: Quantity of smelly classes in our 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%

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.

Table IV: 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*

A. Impact on Change- and Defect-Proneness

Table IV 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 [16]. As explained in Section III-C, 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 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 IV 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 [6], [7].
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 V reports the impact on change- and defect-proneness of each of the six MVC-smells presented in this paper. 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 V. The main findings drawn from the observation of Table V are:

1) When obtaining statistically significant difference (* cells in Table V), 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 V: 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.

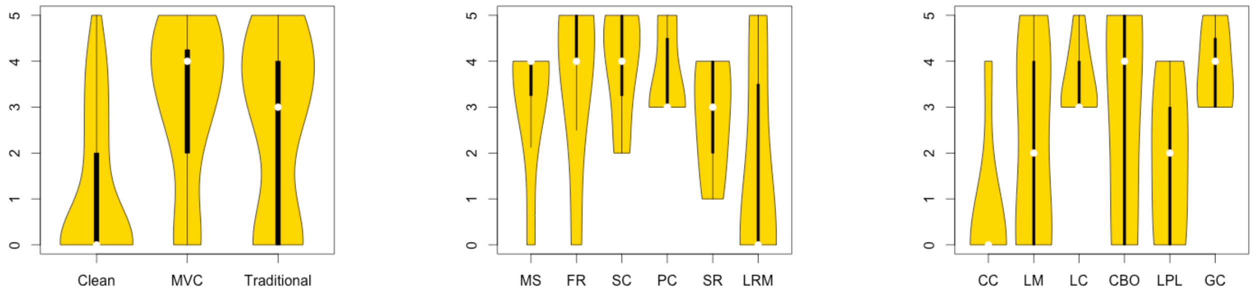
		All classes	**Medium/Large**	**Large**
Promiscuous Controller	CP	2.66*	1.48*	1.51*
	DP	2.43	0.41	0.68
Smart Controller	CP	3.72*	-	1.81*
	DP	3.42*	-	1.34
Fat Repository	CP	2.04*	0.80	1.75*
	DP	1.79*	0.90	0.99
Laborious Repository Method	CP	2.03*	2.38	1.06
	DP	2.36	-	0.48
Smart Repository	CP	5.08*	-	2.79*
	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 (RQ1). In terms of defect-proneness, we claim a negative impact for only for the MEDDLING SERVICE *smell (RQ2).*

B. Developers Perception of the Web MVC Smells

In Figure 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 3b — as well as of each considered traditional smell — Figure 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



(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 3: Participants on the severity of each smell.

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 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 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 (RQ3).*

V. RELATED WORK

Code smells have been discussed for a while in the software engineering community. Webster's book [32] 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 [33] has defined more than 60 different characteristics of good object-oriented code, and Fowler [1] 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.

Researchers evaluated the impact of these smells in terms of code quality. As a first step to identify smelly classes, Marinescu [34] proposed detection strategies that rely on the combination of metric-based rules and logical operators, such as AND or OR. Then, Lanza and Marinescu [14] 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 [35], which makes use of the evolution history to detect the smells, and DECOR [36], a DSL for specifying smells using high-level abstractions.

After, Khohm *et al.* [6] showed that smelly classes are more prone to change and to defects than other classes. Li and Shatnawi [37] also empirically evaluated the effects of code smells and showed a high correlation between defect-prone and some bad smells. Yamashita and Moonen [4] 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.* [38], 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. However, when a class presented more than one smell, the performance of developers during maintenance tasks was significantly reduced.

Indeed, the perception of a developer may be not precise.

A study from Palomba *et al.* [28] 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 [39] 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.* [40] 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 [27] analyzed the behavior of developers regarding the life cycle of code smells and results show that, even when developers are aware of the presence of the smell, they do not refactor.

Regarding web development, most studies focus on related client-side technologies, such as bad practices in CSS [41], [42], Javascript [43], [44], and HTML [45]. 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 study 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 do) or number of dependencies to entity classes (as FAT REPOSITORIES do). To perform this research, we relied on current findings and approaches of the field, such as the definition of metric-based code smells detection strategies [14], analysis of the impact on class change- and defect-proneness [6], [7], as well as on developers' perceptions [28]. As a derivated product of this research, we developed an open source tool [15] that reports any class that contains a smell from our catalogue.

VI. THREATS TO VALIDITY

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly 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.* [23] 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.

Detection strategies for the defined smells were derived from the participants' answers and the expert analysis. There might be 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 wrote automated tests to ensure the correct behavior of our tool that is open source.

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 [46] and Oliveira *et al.*'s [47]), 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 [48] 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. Although this is mainly an observational study, 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) The response rate of our role-focused survey was low. Still, we do not use the answers to generalize the provided catalogue of smells; (4) 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.

VII. CONCLUSIONS AND FUTURE WORK

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.

In this study, we provide 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 [15]. 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.

References

[1] M. Fowler, “Refactoring: Improving the design of existing code,” in *11th European Conference. Jyväskylä, Finland*, 1997.

[2] W. H. Brown, R. C. Malveau, and T. J. Mowbray, “Antipatterns: refactoring software, architectures, and projects in crisis,” 1998.

[3] D. I. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, “Quantifying the effect of code smells on maintenance effort,” *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, 2013.

[4] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in *Proceedings of the 2013 Intl. Conference on Software Engineering*. IEEE Press, 2013, pp. 682–691.

[5] ——, “Do code smells reflect important maintainability aspects?” in *Software Maintenance (ICSM), 2012 28th IEEE Intl. Conference on*. IEEE, 2012, pp. 306–315.

[6] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change-and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, 2012.

[7] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in *Reverse Engineering, 2009. WCRE’09. 16th Working Conference on*. IEEE, 2009, pp. 75–84.

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

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

[10] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, “Detecting performance anti-patterns for applications developed using object-relational mapping,” in *Proceedings of the 36th Intl. Conference on Software Engineering*. ACM, 2014, pp. 1001–1012.

[11] D. Verloop, “Code smells in the mobile applications domain,” Ph.D. dissertation, TU Delft, Delft University of Technology, 2013.

[12] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, “Detecting antipatterns in android apps,” in *Proceedings of the Second ACM Intl. Conference on Mobile Software Engineering and Systems*. IEEE Press, 2015, pp. 148–149.

[13] D. Mazinanian, N. Tsantalis, and A. Mesbah, “Discovering refactoring opportunities in cascading style sheets,” in *Proceedings of the 22nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 496–506.

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

[15] M. A. et al, “Springlint.” [Online]. Available: http://www.github.com/mauricioaniche/springlint.

[16] M. Aniche, G. Bavota, C. Treude, A. van Deursen, and M. A. Gerosa, “A validated set of smells in model-view-controller architectures: Replication package.” [Online]. Available: http://mauricioaniche.github.io/icsme2016.

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

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

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

[20] T. J. McCabe, “A complexity measure,” *Software Engineering, IEEE Transactions on*, no. 4, 1976.

[21] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

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

[23] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *Software Maintenance. ICSM. Proceedings. Intl. Conference on*. IEEE, 2003.

[24] S. Kim, E. J. Whitehead Jr, and Y. Zhang, “Classifying software changes: Clean or buggy?” *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, 2008.

[25] “Pmd.” [Online]. Available: https://pmd.github.io/.

[26] S. M. Olbrich, D. S. Cruze, and D. I. Sjøberg, “Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems,” in *Software Maintenance (ICSM), IEEE Intl. Conference on*. IEEE, 2010.

[27] R. Peters and A. Zaidman, “Evaluating the lifespan of code smells using software repository mining,” in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.

[28] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, “Do they really smell bad? a study on developers’ perception of bad code smells,” in *2014 IEEE Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.

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

[30] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[31] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[32] B. Webster, *Pitfalls of Object-Oriented Development*. M T Books, 1995.

[33] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Reading, 1996, vol. 335.

[34] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *Software Maintenance, 2004. Proceedings. 20th IEEE Intl. Conference on*. IEEE, 2004, pp. 350–359.

[35] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “Detecting bad smells in source code using change history information,” in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th Intl. Conference on*. IEEE, 2013, pp. 268–278.

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

[37] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[38] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension,” in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 181–190.

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

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

[41] G. Gharachorlu, “Code smells in cascading style sheets: an empirical study and a predictive model,” 2014.

[42] A. Mesbah and S. Mirshokraie, “Automated analysis of css rules to support style maintenance,” in *Software Engineering (ICSE), 2012 34th Intl. Conference on*. IEEE, 2012, pp. 408–418.

[43] L. H. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil, “Does javascript software embrace classes?” in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd Intl. Conference on*. IEEE, 2015, pp. 73–82.

[44] A. M. Fard and A. Mesbah, “Jsnose: Detecting javascript code smells,” in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th Intl. Working Conference on*. IEEE, 2013, pp. 116–125.

[45] A. Nederlof, A. Mesbah, and A. v. Deursen, “Software engineering for the web: the state of the practice,” in *Companion Proceedings of the 36th Intl. Conference on Software Engineering*. ACM, 2014, pp. 4–13.

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

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

[48] K. Herzig and A. Zeller, “The impact of tangled code changes,” in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.