Empir Software Eng
DOI 10.1007/s10664-017-9540-2

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · **Gabriele Bavota**[2] · **Christoph Treude**[3] · **Marco Aurélio Gerosa**[4] · **Arie van Deursen**[1]

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in the catalog by Fowler and Beck, reporting design issues that can potentially affect any kind of system, regardless of their architecture (e.g., Complex Class). However, systems adopting a specific architecture (e.g., the Model-View-Controller pattern) can be affected by other types of poor practices that only manifest themselves in the chosen architecture. We present a catalog of six smells tailored to MVC applications and defined by surveying/interviewing 53 MVC developers. We validate our catalog from different perspectives. First, we assess

Communicated by: Bram Adams and Denys Poshyvanyk

✉ Maurício Aniche
M.F.Aniche@tudelft.nl

Gabriele Bavota
gabriele.bavota@usi.ch

Christoph Treude
christoph.treude@adelaide.edu.au

Marco Aurélio Gerosa
marco.gerosa@nau.edu

Arie van Deursen
Arie.VanDeursen@tudelft.nl

[1] Delft University of Technology, Delft, Netherlands

[2] Università della Svizzera italiana (USI), Lugano, Switzerland

[3] University of Adelaide, Adelaide, Australia

[4] Northern Arizona University, Flagstaff, AZ 86011, USA

Published online: 12 September 2017

the relationship between the defined smells and the code change- and defect-proneness. Second, we investigate when these smells are introduced and how long they survive. Third, we survey 21 developers to verify their perception of the defined smells. Fourth, since our catalog has been mainly defined together with developers adopting a specific Java framework in their MVC applications (e.g., Spring), we interview four expert developers working with different technologies for the implementation of their MVC applications to check the generalizability of our catalog. The achieved results show that the defined Web MVC smells (i) more often than not, have more chances of being subject to changes and defects, (ii) are mostly introduced when the affected file (i.e., the file containing the smell) is committed for the first time in the repository and survive for long time in the system, (iii) are perceived by developers as severe problems, and (iv) generalize to other languages/frameworks.

Keywords Code smells · Code anomalies · Code anti-patterns · Software maintenance · Code quality

1 Introduction

Code smells, i.e., symptoms of poor design and implementation choices (Fowler 1997), have been the subject of several empirical studies mainly aimed at characterizing them and at assessing their impact on the maintainability of software systems. It is well known that smells tend to hinder code comprehensibility (Abbes et al. 2011) and maintainability (Sjoberg et al. 2013; Yamashita and Moonen 2013a, 2012) as well as to increase change- and defect-proneness (Khomh et al. 2012; Khomh et al. 2009). Also, previous studies showed that code smells tend to have a long survivability (Arcoverde et al. 2011; Chatzigeorgiou and Manakos 2010; Lozano et al. 2007; Tufano et al. 2017; Ratiu et al. 2004). These studies have been mostly run on the catalog of code smells defined by Martin Fowler and Kent Beck in the *Refactoring* book (Fowler 1997), and including "generic" smells that fit well in any object-oriented system (e.g., Feature Envy, Complex Class, *etc.*). These smells 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 (Krasner et al. 1988), 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 (Fowler 2002) in MVC applications are responsible for the communication with the databases. These classes, besides not containing complex and long methods (traditional smells), should also limit the complexity of SQL queries residing in them.

While "traditional" code smells (Fowler 1997) capture very general principles of good design, 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 catalog 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 recently been emer-ging as a research topic in software maintenance. In particular, researchers have studied smells specific to the usage of object-relational mapping frameworks (Chen et al. 2014), Android apps (Hecht et al. 2015), and Cascading Style Sheets (CSS) (Mazinanian et al. 2014).

In this paper, we provide a catalog 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 catalog, 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 catalog from their answers. The defined smells are: BRAIN REPOSITORY, FAT REPOSITORY, PROMISCUOUS CONTROLLER, BRAIN CONTROLLER, LABORIOUS REPOSITORY METHOD, and MEDDLING SERVICE.

We evaluate the impact of the proposed smells on change- and defect-proneness of classes in 100 Spring MVC projects. Also, we use the same set of systems to investigate when these smells are introduced (i.e., when the affected code artifact is committed for the first time in the repository) and how long they survive in the system (i.e., how long does it take to refactor/remove them from the system). In addition, we perform a survey with 21 developers to verify whether they perceived classes affected by the defined smells as problematic. Finally, since our smells catalog has been mainly defined with the help of 53 MVC developers adopting the Java Spring framework in their applications, we assessed the generalizability of our catalog by interviewing four additional experts relying on four different languages/frameworks (i.e., C# ASP.NET MVC, Ruby on Rails, Java VRaptor, and Scala Play!) for the implementation of their MVC applications.

Our **findings** show that:

1. *Impact on change and fault proneness*. Classes affected by any of the proposed smells have higher chances of being subject to changes. In addition, classes affected by the MEDDLING SERVICE smell have higher chances of being subject to bug-fixing activities over time (i.e., higher defect-proneness).
2. *Developers' perception*. Developers perceive classes affected by these smells as problematic, at least as much as classes affected by traditional smells.
3. *Smells introduction and survivability*. Confirming the findings by Tufano et al. for traditional smells (Tufano et al. 2017), we found that MVC smells (i) are introduced when the (smelly) code artifact is created in the first place, and not as the result of maintenance and evolution activities performed on such an artifact, and (ii) have a very long survivability, with 69% of the smell instances that, once introduced, are not removed by developers.
4. *Generalizability*. Our catalog of MVC smells is well suited for MVC web applications developed in different languages and with different technologies/frameworks.

Besides defining and empirically validating a whole new catalog of MVC code smells, we also defined detection strategies (Lanza and Marinescu 2007) for each smell and implemented them in an open source detection tool (Aniche 2017). Finally, all the data collected in our studies is publicly available in a comprehensive replication package (Aniche et al. 2016a).

This paper extends our ICSME 2016 paper "A Validated Set of Smells in Model-View-Controller Architectures" (Aniche et al. 2016b). In this new version, we provide new results on smells introduction, survivability, and generalizability (RQs 4, 5, and 6) as well as an extended related work section.

Paper Structure Section 2 details the procedure we used to defined our catalog and presents each of the six smells that are part of it. Section 3 describes the design of the study empirically validating our smells, while Section 4 reports the study results. We detail the threats that could affect the validity of our results in Section 5. Following the related work (Section 6), Section 7 concludes the paper outlining promising directions for future work.

2 The Catalog of Web MVC Smells

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

2.1 Background in MVC Web Development

The MVC pattern (Krasner et al. 1988) 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 at 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 that developers are required to write in both CONTROLLER and MODEL layers.

We present a schematic diagram of an MVC architecture in Fig. 1. 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 (Fowler 2002; Evans 2004), such as *Entities*, *Repositories*, and *Services*. ENTITIES represent a domain object (e.g., an *Item* or a *Product*). REPOSITORIES are responsible for encapsulating persistence logic, similar to Data Access Objects (Fowler 2002). 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. As it may or may not perform actions related to the business, e.g., UI formatting or data conversion, we represent it as both inside and outside the Model layer.

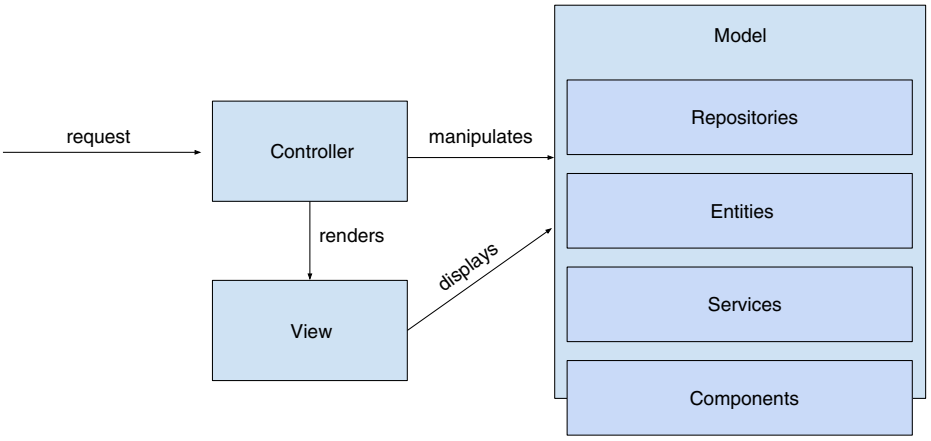


Fig. 1 A Model-View-Controller (MVC) architecture

There are different frameworks that can be used during the implementation of Web MVC applications. One of the most popular is the Spring MVC Java web framework (Turnaround 2017), that provides developers with stereotypes to annotate classes with one of the aforementioned roles. As a consequence, developers can easily understand what role that class plays in the system architecture. As discussed in detail in Section 3.2, we evaluated the impact of the cataloged smells in Spring MVC projects. Indeed, these different architectural roles can be seen in all the aforementioned frameworks.

2.2 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.

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 (Dyer et al. 2013), 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. The company develops a cloud-based ERP system that helps large retailers to manage their business. Their main software system is a 11-years old Java-based Spring MVC web application, and has more than 1 million lines of code. The focus of the interview was to make participants discuss their good and bad practices in each of the five main architectural roles in MVC Web applications. Before each interview, we informed the participants about the goals of this study and the fact that all collected data would be used for research purposes. Participants were also informed that they could end the interview at any time. 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, we asked participants about their experience in software and web development in the surveys and interviews. Complete data is shown in Fig. 2. 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 catalog. 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 (Fowler 1997), 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 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 (Aniche et al. 2016a), while in the following (Section 2.3), we detail the six smells present in our catalog as well as tool-supported detection strategies to identify each of them. These latter have been also defined in collaboration with the expert.

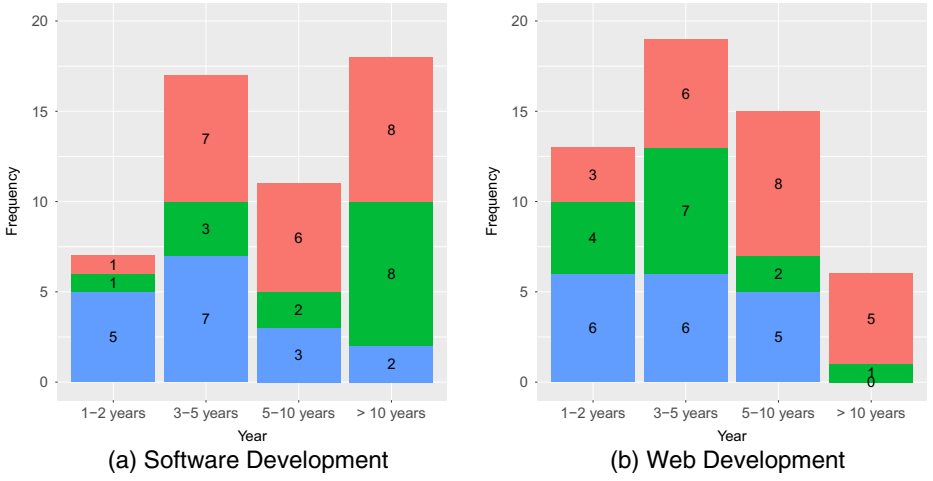


Fig. 2 Participants' experience in software and web development from participants in Step 1 (top), Step 2, and Step 3 (bottom)

Table 1 The proposed MVC smells

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

2.3 Resulting catalog of Web MVC smells

Table 1 reports the six MVC smells included in our catalog. In the following paragraphs, we discuss each of the smells, explaining why it has been considered part of our catalog (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 2.2). Together with the catalog of smells, we also provide refactoring suggestions.

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 routes, 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.

$$(NOR > \alpha) \vee (NSD > \beta) \tag{1}$$

Refactoring suggestion. Break the CONTROLLER class into two or more classes. Each new CONTROLLER class contains a cohesive set of offered actions. Repeat the operation if the newly created CONTROLLER is still promiscuous.

Brain 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.

[1] Thresholds used in the detection strategies have been tuned as described in Section 3.4.

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 Chidamber and Kemerer metric suite (Chidamber and Kemerer 1994), a well known suite of object-oriented metrics. The common RFC metric counts the number of methods that can potentially be executed in response to a message received by a class. However, it also considers in this count 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:

$$(NFRFC > \alpha) \tag{2}$$

Refactoring suggestion. Move the existing business logic to its specific ENTITY, COMPONENT or a SERVICE class.

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 two 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 claimed 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, and 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:

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

Refactoring suggestion. Move the existing SQL query to a REPOSITORY class.

Brain Repository Repositories are meant to deal with the persistence mechanism, such as databases. To that end, 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 these classes 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 emphasize: *"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 BRAIN REPOSITORY by identifying the ones in which the McCabe's Complexity Number (McCabe 1976) and the SQL complexity are higher than a threshold. McCabe's Number counts the number of different branch instructions, e.g., *if* and *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:

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

Refactoring suggestion. Move the complex logic to a method and the SQL query itself to another method. If the complex logic is used by other REPOSITORIES, move it to a COMPONENT.

Laborious Repository Method As a good practice, a method should have only one responsibility and do one thing (Martin 2009). 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 an 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()*, and *createStoredProcedure()*, *getCriteriaBuilder()*, and for JDBC, *prepareStatement()*, *createStatemente()*, and *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:

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

Refactoring suggestion. Split the multiple persistence actions that exist in a single REPOSITORY method into different methods. The newly created methods may or may not be private to that REPOSITORY, i.e., if a persistence action can be used alone, the new method may be public.

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:

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

Refactoring suggestion. Move methods that are related to other ENTITIES to the entity's specific REPOSITORY.

3 Smell Evaluation Study Design

The *goal* of the study is to investigate whether the defined catalog of MVC smells has an effect 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. Also, we aim at characterizing these smells in order to understand (i) when they are introduced, and (ii) how long they survive in the system. Finally, we assess the generalizability of our code smells to different languages and frameworks used to implement MVC Web applications. The *quality focus* is on source code quality and maintainability that might be negatively affected by the presence of the defined smells.

3.1 Research Questions

Our study aims at addressing the following six research questions:

RQ_1. *What is the relationship between the proposed code smells and the change-proneness of classes?* Previous studies have shown that the "traditional smells" (e.g., Blob Classes) (Fowler 1997) can increase class change-proneness (Khomh et al. 2012; Khomh et al. 2009). This research question aims at investigating the impact of the six Web MVC smells on change-proneness of classes.

RQ_2. *What is the relationship between the proposed code smells and the defect-proneness of classes?* This research question mirrors RQ_1. Traditional smells are also known by their impact on the defect-proneness of classes (Khomh et al. 2012, 2009). 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?* This 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.

RQ_4. *What is the survivability of code smells?* In this RQ, we investigate how long each of the proposed code smells survive after being introduced. Similar research focusing on traditional code smells showed that code smells tend to survive for long time in the system (Tufano et al. 2017). In addition, ∼80% of smell instances are never removed from the system after their introduction (Tufano et al. 2017).

RQ_5. *When are code smells introduced?* Prior research (Tufano et al. 2017) has shown that traditional code smells are introduced when the (smelly) code artifact is created in the first place, and not as the result of maintenance and evolution activities performed on

such an artifact. In this RQ, we aim to verify whether this also holds true for the proposed code smells.

RQ_6. *Can the proposed code smells be generalized to MVC-based frameworks?* Our catalog of code smells has been mainly defined with the help of Java Spring MVC developers. However, our catalog aims to generalize to MVC applications written in other frameworks/languages. This RQ aims at assessing such a generalizability.

3.2 Context Selection

To answer RQ_1, RQ_2, RQ_4, and RQ_5 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) most of the developers involved in the definition of our catalog have experience with this framework, (ii) it uses stereotypes to explicitly mark classes playing the different roles introduced in Section 2.1 (e.g., CONTROLLERS), thus making identifying the role of each class simple, and (iii) as shown in a survey conducted with over 2,000 developers (Turnaround 2017), it is widely adopted by developers (> 40% of the respondents claimed to use it).

We use BOA (Dyer et al. 2013) 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 are stand-alone systems. We end up with 120 Spring MVC projects. The complete list is available in our online appendix (Aniche et al. 2016a), while Table 2 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 3.4. The remaining 100 are used, as detailed in Section 3.3, 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 3 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 catalog.

Finally, to answer RQ_6 we looked for experts in the development of MVC applications that have deep knowledge of MVC frameworks different from Spring. The selection was performed in our industrial partner network by making sure to involve experts meeting at least one of the following requirements: (i) has written a book on the topic, (ii) has been part of the development team of the framework, (iii) has spoken about the technology in an international conference, (iv) has more than 10 years of experience in software development with the specific framework.

[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 2 Size attributes of the 120 subject systems

Role	Total classes	Median per project	Total SLOC	Median LOC per class
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

3.3 Data Collection and Analysis

We discuss the process we adopted to collect and analyze the data needed to answer our research questions.

3.3.1 RQ_1 & RQ_2: Impact on Change- and Fault-Proneness

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 which, 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 (Fischer et al. 2003). 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

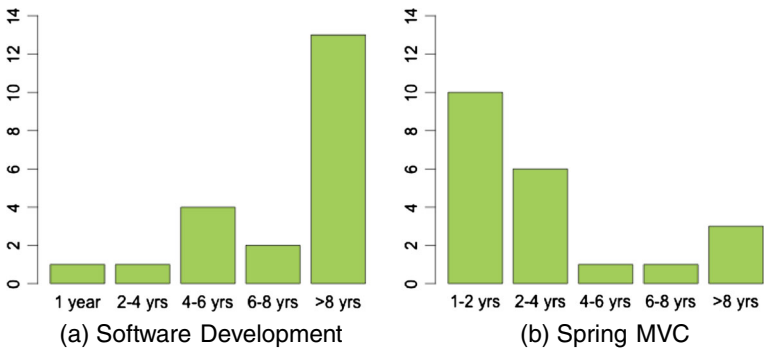


Fig. 3 Participants' experience

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 to verify whether they meet these criteria. The analysis was performed by the first author by adopting the following procedure: Projects not having commits written in English were immediately discarded by manually analyzing a few commits. For the remaining projects (i.e., the ones having the commit messages written in English) a deeper analysis was needed. In these cases, the author firstly ran an automatic search to verify the existence of commit messages reporting the keywords of interest. In case of success, these commits were manually inspected to double-check whether they were actually aimed at fixing bugs. This was the case for all the analyzed projects. If no commits reporting the relevant keywords were identified, we excluded the project from our analysis. We ended up with 16 projects meeting our requirements. These 16 projects are thus exploited in the context of RQ_2 and listed in our online appendix (Aniche et al. 2016a).

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 (Khomh et al. 2012) investigating traditional smells. Firstly, as performed by Kim et al. (2008), 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.

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 Khomh et al. (2012), 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 (Pmd 2017), 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 (Olbrich et al. 2010; Peters and Zaidman 2012; Palomba et al. 2014; Tufano et al. 2017).

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 (Sheskin 2003) 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) (Sheskin 2003) 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. On the other hand, 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 among 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 code metrics are commonly related to lines of code (El Emam et al. 2001), 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.

3.3.2 RQ_3: Developers' Perception

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 Fig. 3). 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 on 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 (Conover 1998). This test is used to analyze 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 (Grissom and Kim 2005) 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 $|d| \geq 0.474$ (Grissom and Kim 2005). Finally, we report qualitative findings derived from the participants' open answers.

3.3.3 RQ_4 & RQ_5: MVC smells Introduction and Survivability

Concerning RQ_4 and RQ_5, as previously said, we use the same 100 subject projects exploited in RQ_1 and RQ_2. In each of these systems, we detect all instances of our MVC code smells in commits occurring over their history. As re-creating the real change history of a Git repository might be tricky and error-prone (Bird et al. 2009), we only consider "merge" commits belonging to the *master* branch (which, by convention, contains all code that can go to production). Also, whenever a commit has two parents, we only visit the left node[4]. Similarly to RQ_1, also for this analysis we skip the first 500 commits.

By running our code smell detection on each of the above detailed commits, we are able to identify the exact commit in which a smell instance "appears" in the master branch (i.e., is introduced) as well as the commit (if any) in which the smell instance is removed. This allows us to:

1. Answer RQ_4 by creating a survivability model of code smells via the R's `survival` package. The model uses the number of days between the smell introduction and removal (i.e., between the commit in which the smell appears and disappears from the master branch) and considers a smell instance as *survived* if it still affects the system in the last analyzed commit.
2. Answer RQ_5 by verifying whether code smell instances are introduced when the smelly artifact is committed for the first time into the repository (as already observed for traditional smells (Tufano et al. 2017)) or as the result of continuous changes due to maintenance and evolution activities.

[4]This is achieved by using Git's `--first-parent` option.

3.3.4 RQ_6: Generalizability of the Defined MVC Smells Catalog

To answer RQ_6, we interviewed the four industrial experts introduced before. We conducted the interviews via emails. We preferred an asynchronous communication channel over face-to-face interviews in order to give time to the experts to answer our questions. We followed the same procedure of the first round of interviews, and we informed the participants about the goals of this study and the fact that all collected data would be used for research purposes. Participants were also informed that they could end the interview at any time.

We presented to the participants our catalog of code smells and the defined detection strategies. Then, we asked two open questions: *"Is this smell relevant in the MVC framework you use?"* and *"Should we adapt our detection strategies to detect the smell instances in your applications?"*. In order to keep up with the discussion, we studied the documentation of all technologies. We answer RQ_6 by analyzing the collected answers.

3.4 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 threshold 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 (2007) in order to define thresholds for their detection strategies.

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

In Table 3, we present the thresholds derived for each metric (α and β in the Formulas presented in Section 2.3).

Table 3 Thresholds used in the detection strategies

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

4 Results

Table 4 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 BRAIN 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 BRAIN 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 388 classes.

To better understand the overlap between traditional and MVC code smells, we investigated their relationship by applying Association Rule Mining (Agrawal et al. 1993). In particular, we collected from the 100 subject systems the set of 6,713 classes affected by at least one instance of code smells, considering both MVC and traditional smells. For each of these classes, we generated a transaction containing MVC and the traditional smells affecting it. We analyzed the build database of transactions using Association Rule Mining to identify patterns of co-occurrence of MVC and traditional smells. In particular, we use the statistical software R and the package *arules*. Each rule X→Y, where X and Y are of two code smell types, one belonging to our catalog and one to the traditional code smells, brings with it information about its support and confidence. The support is the proportion of transactions that contain the learned rule, while confidence indicates the fraction of transactions containing X where Y also appears.

We extracted all rules having a minimum confidence of 0.5 (i.e., at least in 50% of the rules in which X appears Y also appears) and minimum support 0.0045. The rationale for such a low support is that (i) we want to be comprehensive in studying possible relationships between traditional and MVC smells, and (ii) given the high number of transactions in our dataset (6,713), the considered support level still ensures that the rule holds in at least 30 classes.

Table 4 Quantity of smelly classes in our sample ($n = 100$)

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

Given the above described setting, we only identified three rules. The first, *Brain Controller→God Class* (support=0.02, confidence=0.59) indicates that in 59% of cases, classes affected by the *Brain Controller* MVC smell are also *God Classes*. Such a relationship can be easily explained by the fact that *Brain Controllers* are, by definition, implementing too much control flow, a characteristic that is common to *God Classes*, well known for centralizing the system behavior. Still the mined rule also shows that this association does not hold in 41% of classes affected by *Brain Controller*, thus drawing a clear distinction between the two code smells. Very similar observations can be made for the second association: *Brain Controller→Complex Class* (support=0.02, confidence=0.65). Such a relationship is also triggered by the fact that 62% of the *God Classes* in our dataset are also *Complex Classes*.

Finally, the third mined rule is *Laborious Repository Method→Complex Class* (support=0.01, confidence=0.70). This was the most surprising relationship we found, since it indicates that classes having at least one *Laborious Repository Method*, tend also to be quite complex. Thus, the several database actions performed by the smelly method are likely to also increase the class complexity, probably due to the more complex application logic required to manage the output of the different actions.

Summarizing the results of the association rule mining analysis, we did not find a strong overlap between the two catalogs of smells. Indeed, only two of the six defined MVC smells are co-occurring frequently with traditional smells. Still, also in those cases, the observed values of confidence do not highlight a total overlap in the code smells definition.

4.1 Change- and Defect-Proneness (RQ_1 and RQ_2)

Table 5 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 (Aniche et al. 2016a). As explained in Section 3.3, 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

Table 5 Odds ratio in change- and defect-proneness between MVC-smelly, traditional-smelly and clean classes

		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*

(CP) Change-proneness, (DP) Defect-proneness,

(*) Fisher's exact test < 0.05

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. This indicates that classes affected by MVC-smells have a higher chance of changing (almost 3 times higher) and of being subject to bug-fixing activities (2 times higher). 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 5 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 (Khomh et al. 2012; Khomh et al. 2009).
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 6 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., BRAIN CONTROLLER for medium/large classes), it was not possible to perform the statistical test due to lack of data (i.e., very few BRAIN CONTROLLERS are medium/large classes). These cases are indicated with "-" in Table 6. The main findings drawn from the observation of Table 6 are:

1. When obtaining statistically significant difference (* cells in Table 6), classes affected by smells have always a higher chance (OR $>$ 1.00) of changing as well as 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 not statistically significant results.
2. BRAIN 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 as much chances of being subject to bug-fixing activities as compared to clean classes).

Classes affected by the web MVC smells are more prone to change (RQ_1). In terms of defect-proneness, we observed that only classes affected by the MEDDLING SERVICE *have a higher chance of being subject to bug-fixing activities (RQ_2).*

Table 6 Odds ratio in change- and defect-proneness between MVC-smelly and clean classes, per smell

		All classes	Medium/Large	Large
Promiscuous Controller	CP	2.66*	1.48*	1.51*
	DP	2.43	0.41	0.68
Brain 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
Brain 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*

(CP) Change-proneness, (DP) Defect-proneness,

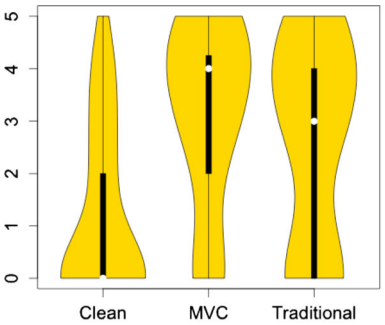
(*) Fisher's exact test < 0.05, (−) lack of data

4.2 Developers' Perception of the Web MVC Smells (RQ_3)

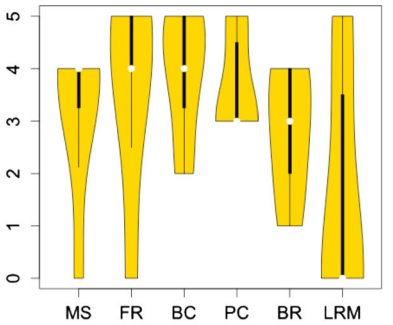
In Fig. 4a, we present violin plots of the developers' perception of MVC smells, traditional smells, and clean classes. Also, we report the developers' perception of each single MVC-smell — Fig. 4b — as well as of each considered traditional smell — Fig. 4c. On the y-axis, 0 (zero) indicates classes 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 indicate 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 Fig. 4a, 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).

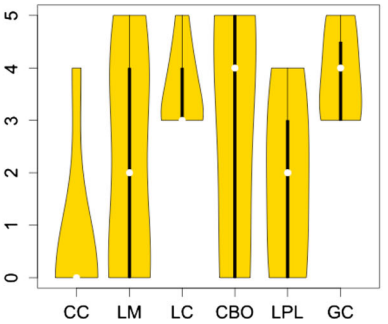
God Classes (GC) are the most perceived traditional smell (median=4). Regarding the proposed smells, MEDDLING SERVICE, FAT REPOSITORY, and BRAIN CONTROLLER achieve medians equal to 4, meaning they are perceived as really problematic by the participants.



(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, BC=Brain Controller, PC=Promiscuous Controller, BR=Brain 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

Fig. 4 Participants on the severity of each smell

Several developers, without knowing our smells' catalog, 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 BRAIN 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 BRAIN 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 (RQ_3).

4.3 On When Smells are Introduced and on Their Survival (RQ_4 and RQ_5)

Table 7 reports, for each of the defined MVC-smells, the number of instances that survived/did not survive over the observed change history. Table 7 also presents descriptive statistics (and in particular, the average and the median) for the number of days the smell instances survived in the system. Note that these statistics are computed over both survived and not survived instances, meaning that the number should be treated as an underestimation (indeed, the survived instances will clearly last longer in the system). Moreover, Fig. 5 depicts the survival models generated for the considered code smell instances, with the black line representing the survival probability of smell instances at different times (i.e., after x days from their introduction). The dotted lines depict the 95% confidence interval.

The first thing to notice from the analysis of Table 7 is that 69% of the introduced smell instances (928 out of 1,337) are never removed from the system after their introduction. In general, the smells tend to stay alive for a very long time in the codebase; for all smell types, there is an over 50% chance of surviving after 500 days from their introduction. Some of them can last even longer: a FAT REPOSITORY has around 80% chance of surviving more than 1,500 days. Although also surviving for long time, LABORIOUS REPOSITORY METHODS are the ones for which survival is reduced after 800 days. This is explained by the fact that our dataset contains a single smelly class which was refactored after 845 days. This single point makes the probability go down. If we remove this data point from the analysis, the chart becomes steady again, also with a 50% chance of a smell to survive more than 800 days. The larger confidence interval in this smell can be explained by the fact that this is indeed the one for which we have less data. Nevertheless, probabilities are higher than expected even if we analyze only the lower bound confidence interval.

From Table 8, we also see that MVC-smells are not always a consequence of code aging. Indeed, several smell instances are introduced when the smelly artifact is created in the first place (i.e., it is committed for the first time in the repository). These cases represent from 42.6% (BRAIN CONTROLLERS) up to 86.5% (LABORIOUS REPOSITORY METHOD)

Table 7 Proposed smells and descriptive statistics on their survival over time

Smell	Not survived	Survived	Median days of survival	Average days of survival
Fat Repository	81	297	1,068	1,469
Laborious Repository Method	60	72	535	1,105
Meddling Service	24	58	998	1,054
Promiscuous Controller	107	276	731	1,362
Brain Controller	99	171	839	1,505
Brain Repository	38	54	535	1,151
SUM	409	928		

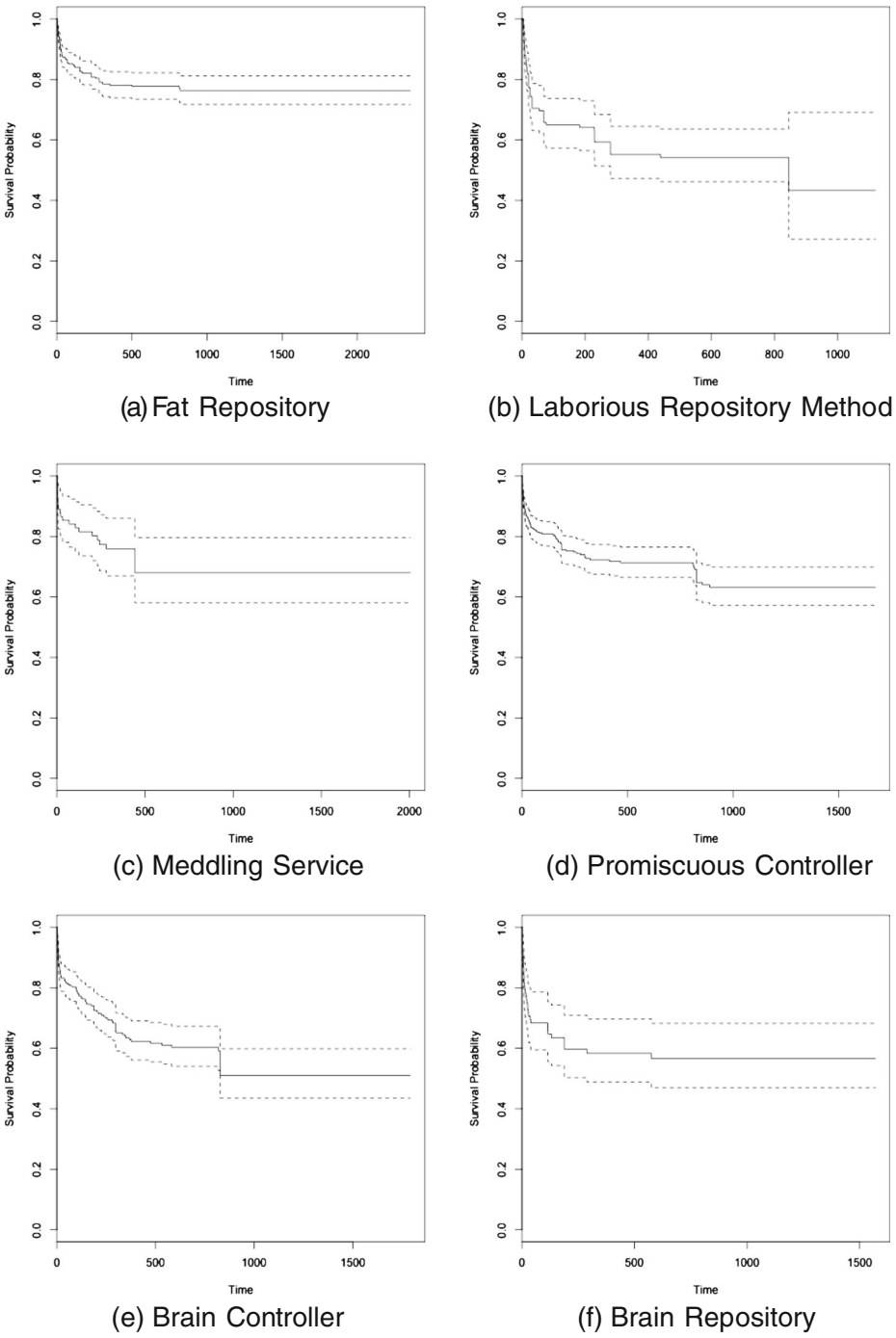


Fig. 5 Survival analysis of the six proposed smells (95% CI). Time on the x axis is represented in "days"

Table 8 Number of times the code smell was introduced in the first version of the class

Smell	Distinct classes	Smelly since its first appearance
Promiscuous Controller	327	215 (65.7%)
Brain Controller	211	90 (42.6%)
Fat Repository	346	250 (72.2%)
Meddling Service	75	61 (81.3%)
Brain Repository	64	28 (43.7%)
Laborious Repository Method	104	90 (86.5%)
SUM	1,127	734

of all considered instances. Interestingly, these results are inline with what has already been observed for traditional (Tufano et al. 2017) and for test smells (Tufano et al. 2016).

In many cases (from 42% to 86%), MVC-smells are introduced during the creation of the smelly artifact (RQ_5). Once introduced, 69% of instances are not removed from the system and, in general, they have a long survivability (RQ_4).

4.4 Generalizability of the Proposed Smells to MVC Frameworks (RQ_6)

Before discussing how the four experts commented on the smells present in our catalog, Table 9 summarizes the results of our interviews. In particular, for each of the four considered MVC frameworks/languages and for each smell we report "Y" if, based on the expert opinion, the smell is relevant in the MVC application in which he developers, "N" otherwise. We use "Y*" to indicate cases in which the smell can be considered relevant with some adaptation.

As we can see, most of the proposed smells can be generalized to the different MVC frameworks considered in our study. However, there are exceptions to discuss. In the following, we present experts' opinions collected in our interviews.

Table 9 Experts' opinions on the generalizability of the code smells

	VRaptor	Ruby on Rails	Asp.Net MVC	Play!
Promiscuous Controller	Y	Y	Y	Y
Brain Controller	Y	Y	Y	Y
Brain Repository	Y	Y*	Y	Y*
Fat Repository	Y	Y*	Y	Y*
Laborious Repository Method	Y	Y*	Y	Y*
Meddling Service	Y	N	Y	N

(*) means the smell makes sense, but it needs some adaptation

4.4.1 VRaptor

The expert affirmed that all the smells in our catalog generalize to VRaptor applications. Indeed, both VRaptor and Spring MVC are similar frameworks built on top of the same programming language (i.e., Java).

Detection Strategies According to the expert, the challenge in the detection strategies would be the automatic identification of Service and Repository classes. Indeed, while Spring MVC has annotations for both, VRaptor does not provide them. Thus, our detection strategy should be adapted to work on VRaptor applications, and we are currently looking into possible solutions to such a problem.

Fat Repository The expert agreed with the relevance of this smell and said that the threshold can even be very restrictive: *"If a repository depends on 2+ entities, there's something wrong"*.

4.4.2 Ruby on Rails

In order to better understand our smells, the expert involved in this interview opened some of his company's projects and manually explored the existence of these smells. Once he identified instances of the MVC-smells in our catalog, he commented as follows.

Brain Controller According to him, all Controller-related smells can be generalized to Rails applications. He also confirmed that, although the *Fat Model, Skinny Controller* philosophy is quite popular in Rails, it is common to find business rules in Controllers. He also suggested that a possible detection strategy for BRAIN CONTROLLER could be to analyze private/protected methods in Controllers: "[when] *developers need some logic, they end up adding a private method in the Controller. However, that method should be in a class that deals with business rules"*.

Meddling Service The smell cannot be directly applied in Rails applications, as Rails natively implements the *Active Record* pattern (Fowler 2002). This means that there are no Repositories and all database access happens directly from the model classes.

Brain Repository and Laborious Repository Method According to the expert, both smells occur quite often in Rails applications (the expert used the sentence *"these are the champions"*). Although there are no explicit Repository classes, SQL queries are encapsulated in the Models: *"Imagine that the developer tries to put the business rule in the model, but there are no Repositories! What happens is that you see lots of business rules mixed with database code* [in the Model]*... A lot, really"*.

Fat Repository As each Model contains its own database access logic, the smell cannot be directly transferred to Rails. According to the expert, a derivative of this smell could be the *Fat Model*, which is a model that contains several relationships with other models. Indeed, we conjecture that this could be a potential smell in any MVC application. We will consider it when revising our catalog in the future.

4.4.3 Play Framework

The expert considered as generalizable five out of the six smells in our catalog (even though three of them would require some adaptations).

Meddling Service and Repository Smells Similar to Ruby on Rails, Play framework makes use of the *Active Record* pattern (Fowler 2002). This means there are no explicit Repositories and each model is responsible for its own database access logic.

4.4.4 Asp.NET MVC

Similar to VRaptor, the ASP.NET MVC expert also affirmed that all smells are valid to ASP.NET MVC applications as well.

Controller Smells The expert affirmed that both smells are common in ASP.NET MVC applications. According to him, developers create Promiscuous Controllers because "it is easier to do so". According to him, developers think: *"Why should I create another Controller if it's simpler to just add another action in an existing Controller?"*. In addition, the expert said he is always concerned about Brain Controllers. He believes it is indeed a bad practice that makes code more prone to defects. He affirmed that, every time he finds this smell during a code review, he asks the author of the code to fix it.

Repository Smells The expert affirmed that detecting Brain Repositories can be useful for his team. According to him, the development team should review the repository code every time such smell happens. He had similar opinions to other Repository smells. For Laborious Repository Method in particular, he also suggested that dynamic analysis (by means of a .NET profiler) could be conducted in order to detect methods that do too many database operations.

Meddling Service The expert says he also finds instances of this smell in his daily job. In order to avoid it, he suggests that the Service layer should only make use of abstractions (e.g., Data Access Object) that encapsulate database-related code.

> *Most of the proposed code smells generalize in the context of several frameworks/programming languages. Repository smells should be adapted for frameworks that make use of Active Records rather than Repositories. Meddling Service seems to be specific to the Spring MVC implementation (RQ6).*

5 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 most of the subject systems did not have an issue tracker, we relied on the heuristic proposed by Fischer et al. (2003) to identify bug fixing commits. We are aware that this 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 is needed in order to optimize them. However, our current strategies were able to detect classes perceived as problematic by developers and with increasing change-proneness. Also, possible imprecisions might be introduced due to errors in the implementation of the tool we wrote to detect the smells. To mitigate such problem, the tool is available for inspection together with an automated test suite.

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. (2010) and Oliveira et al. (2014)), up to now there is no empirical evaluation of which strategy works best.

Some of the proposed detection strategies rely on well-known code metrics, e.g., McCabe's number as a proxy for code complexity. Some authors have shown that McCabe's number may not be a good proxy for complexity or that it can even be outperformed by lines of code (Jay et al. 2009; Shepperd 1988). On the one hand, we acknowledge the fact that different strategies for calculating complexity may lead to different classes being pointed as smelly. On the other hand, in the particular `Brain Repository` detection strategy, we observed during the interviews and surveys that participants were mostly concerned about Repositories that build dynamic queries; such code is implemented by means of several branch instructions (e.g., `if`, `for`), which are captured by McCabe's number.

During the survival analysis, we noticed some smell instances showing a curious pattern: they were introduced in a commit c_i and immediately removed in the following c_{i+1} commit. From a manual inspection, we concluded that this was due to commits that were *rebased* in the Git repository. Still, the findings of our RQ_4 were not significantly impacted by the inclusion/exclusion of these specific cases (the survival models obtained when excluding these cases are available in our replication package (Aniche et al. 2016a)).

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 (Herzig and Zeller 2013) 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 generalization 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) In RQ_1-RQ_5 we evaluated our smells in Spring MVC applications, since our detection tool is implemented to work with systems exploiting Spring; However, we verified the generalizability of our smells by interviewing experts working with other MVC frameworks;
(3) The response rate of our role-focused survey (RQ_3) was low. Still, we do not use the answers to generalize the provided catalog of smells;
(4) As part of our procedure to derive the smells catalog, we also interviewed developers. Such interviews were conducted in a single company, which may reflect specific smells faced only by that particular company. On the other hand, our procedure also included layer- and role-focused surveys that were sent to a broader audience. The set

of different methods to collect the smells catalog gives us, to some extent, more generalizable results. Nevertheless, our catalog only includes six smells for MVC web applications. We do not claim this is a comprehensive catalog. Further research is needed to investigate other possible bad practices in MVC applications;

(5) Finally, while we verified the generalizability of our smells by interviewing experts working with other MVC frameworks (RQ_6), the reported data only represents (possibly biased) subjective opinions.

6 Related Work

Code smells have been discussed in the software engineering community for a while. Smells such as *God Classes*, *Feature Envy*, and *Blob Classes* are popular among practitioners and popular tools in industry, such as PMD and Sonar, are designed to detect them.

The idea of code smells dates back from the 90's. Riel (1996) and Webster (1995) books presented common mistakes and pitfalls that developers fall into when designing an object-oriented system. Riel (1996), in particular, has defined more than 60 different characteristics of bad object-oriented code, such as long methods and excessive complexity.

Later on, Fowler (1997) coined the term "code smells" to refer to code that may suffer from bad design or implementation. In their book, both authors suggested more than 20 different code smells together with suggestions on how they could be refactored. A similar idea to code smells can also be found in literature by the name "antipattern". In their book, Brown et al. (1998) describe antipatterns as poor solutions to common design problems. The authors also describe 40 antipatterns in object-oriented design, such as Blob Classes and Spaghetti Code.

The idea of code smells has been also spread over specific contexts, technologies and paradigms. Some of them even focused on web technologies, such as bad practices in CSS (Mesbah and Mirshokraie 2012), Javascript (Silva et al. 2015; Fard and Mesbah 2013), and HTML (Nederlof et al. 2014). However, to the best of our knowledge, no research has focused on code smells for server-side MVC Web applications. The smells we propose in this study are currently not captured by "traditional smells", which aim at more general good practices, i.e., they do not focus on SQL complexity (as BRAIN REPOSITORIES do) or number of dependencies to entity classes (as FAT REPOSITORIES do).

In the following sub-sections, we present literature on different topics related to code smells, such as detection strategies, their impact on software maintenance, and how they are perceived by developers.

6.1 Detection Strategies

As a first step to identify smelly classes, Marinescu (2004) proposed detection strategies that rely on the combination of metric-based rules and logical operators, such as AND or OR. Then, Lanza and Marinescu (2007) defined a set of thresholds based on benchmarking metrics in real software systems. In their approach, the authors relied on quartile analysis.

Defining the threshold, as done by the aforementioned techniques, is not a straightforward task. McCabe (1976), for example, suggested the value 10 as a threshold for the Cyclomatic Complexity metric. However, this number was derived from his experience. Deriving thresholds from experience has also been done by other authors (Coleman et al. 1995; Nejmeh 1988). Benchmarks are also a common strategy for deriving thresholds. Alves et al. (2010) proposed an approach based on weighted functions. 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 (Alves et al. 2011). Fontana et al. (2015) worked on an algorithm to automatically identify these 3 percentiles. The approach analyzes 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 (Oliveira et al. 2014), 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.

Other approaches exist in literature, such as HIST (Palomba et al. 2013), which makes use of the evolution history to detect the smells, and DECOR (Moha et al. 2010), a DSL for specifying smells using high-level abstractions. In HIST, the authors propose an approach to detect Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy smells. To do that, the approach makes use of change history information mined from versioning systems. After comparing their approach with a state-of-the-art approach, the results show that its precision ranges between 61% and 80%, and its recall ranges between 61% and 100%. The approach is also able to detect code smells that could not be identified by only applying code analysis. In DECOR, the authors define a method that contains all the steps that are necessary to define a code smell detection strategy. Then, by means of a DSL, users can specify the strategy using high-level abstractions. The authors show that the implemented tool, DETEX, also scales for large systems.

6.2 The Impact of Code Smells

The impact of code smells is also commonly targeted by researchers. Commonly, researchers explore the relationship of smelly classes and their proneness to change or to have defects. As an example, Khomh et al. (2012) studied different releases of three large open source software systems and observed that classes affected by code smells (any of the 29 evaluated ones) were two to five times more likely to change than clean classes. In addition, they also showed that the proneness increases when a class suffered from more than one code smell.

The presence of multiple smells is indeed an issue that was explored by other researchers, e.g., Yamashita and Moonen (2013a) 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. (2011), 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.

Li and Shatnawi (2007) empirically evaluated the effects of code smells and showed a high correlation between defect-proneness and some bad smells. They analyzed classes affected by six different smells (Data Class, God Class, God Method, Refused Bequest, Feature Envy, and Shotgun Surgery) in three versions of Eclipse. Their findings show that classes affected by God Class, God Method and Shotgun Surgery are more prone to defects than clean classes.

Code smells can also affect other aspects of maintainability, such as program comprehension. In Abbes et al.'s study (Abbes et al. 2011), the authors examined the relations between

God Classes and Spaghetti Code, and program comprehension by analyzing the behavior of 24 graduate students that were subject to the study. Their findings showed that participants' understanding was not affected by classes that suffered from a single antipattern. However, participants' comprehension was negatively impacted by classes that suffered from more than one antipattern.

Sjoberg et al. (2013) measured the effects of code smells on maintenance tasks. Their results show that developers spent more time in maintaining classes that were affected by God Class and Feature Envy smells. Yamashita and Moonen (2012) also evaluated the effects of code smells in other activities, such as reading, editing, searching and navigating. They found that different code smells impact different maintenance activities. As examples, God Classes increase editing effort while Feature Envy increases searching effort.

Saboury et al. (2017) investigate the impact on defect-proneness of 12 types of code smells affecting JavaScript applications. Their results show that files affected by code smells have hazard rates 65% higher than clean files. They also studied the developers' perception of JavaScript code smells by surveying 1,484 developers. Interestingly, one of the code smells that developers perceived as particularly problematic (i.e., Variable Re-assign), is also the one having the strongest negative impact on file defect-proneness.

Van Deursen et al. (2001) share their experience on 11 test smells (Mystery Guest, Resource Optimism, Test Run War, General Fixture, Eager Test, Lazy Test, Assertion Roulette, Indirect Testing, For Testers Only, Sensitive Equality, and Test Code Duplication) as well as refactoring suggestions to each of them. Bavota et al. (2012) analyzed the distribution of these test smells in 18 software systems (of which 2 were industrial systems). Their findings show that test smells are often found in these systems. In addition, after a controlled experiment with 20 master's students, they show that most of them can also negatively impact code comprehensibility. Tufano et al. (2016), after collecting the perceptions of 19 developers, show that developers usually do not recognize these smells, and that this fact increases the need of such test smell detectors. After studying two large open source systems, the authors also show that test smells are usually introduced during the test creation and that smells tend to live for a long time.

6.3 Developers' Perceptions of Code Smells

The perception and knowledge of developers about code smells is also a relevant topic in the literature. Palomba et al. (2014) studied the developers' perception of code smells. Results show that smells related to complex or long source code are perceived as harmful by developers, while other types of smells are only perceived when their intensity is high.

Yamashita and Moonen (2013b) conducted a survey with 73 professionals, and results indicate that 32% of developers do not know or have limited knowledge about code smells. Regarding knowledge, 57% of the respondents had never heard of code smells or antipatterns or had heard of them but were not sure what they are. Only 18% affirmed to have a good understanding of code smells and to actually apply the knowledge in practice.

Peters and Zaidman (2012) analyzed the behavior of developers regarding the life cycle of code smells. More specifically, they examined God Class, Feature Envy, Data Class, Message Chain Class and Long Parameter List smells. On average, smells survived for around 50% of revisions. Results also show that, even when developers are aware of the presence of the smell, they do not refactor it.

Bavota et al. (2015) analyzed 13,000 refactorings in three software systems, and noticed that 40% of the refactorings were applied on smelly classes. On the other hand, in only 7% of the cases the smells were removed. The most common refactored smells were Blob

Class, Long Method, Spaghetti Code and Feature Envy. Arcoverde et al. (2011) performed a survey to understand how developers react to the presence of code smells. According to the 33 participants of their survey, Duplicated Code and Long Methods are the two most common ones. Their results also show that developers postpone the removal of a code smell in order to avoid API modifications.

7 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 catalogs are focused on general good practices, i.e., practices that can be applied to any system, regardless of its architecture.

In this paper, we defined and empirically evaluated a catalog of six smells that are specific to Web MVC systems, namely BRAIN REPOSITORY, FAT REPOSITORY, PROMISCUOUS CONTROLLER, BRAIN CONTROLLER, LABORIOUS REPOSITORY METHOD, and MEDDLING SERVICE.

Our work stressed the importance of focusing on contextual smells. Indeed, while the general catalog of code smells available in the literature (Fowler 1997) represents a precious guideline for building high-quality object-oriented applications, specific systems exploiting specific architectures might be affected by smells that only make sense in that specific context. Our work represents a step ahead in the definition of contextual smells, capturing poor implementation choices in the context of specific architectures. We showed that contextual smells (MVC-smells in our case) have a negative impact on maintainability proxies (i.e., change- and defect-proneness), are perceived by developers as design issues and generalize to several different frameworks. We believe that the methodology used in this work to define (interviews + surveys + experts) and evaluate (quantitatively and qualitatively) our catalog of smells represents a contribution by itself, and can be used as a guidance for other researchers interested in defining their own catalogs of code smells in different contexts/architectures.

We also invested effort in the implementation of a production-ready code smell detection tool (Aniche 2017). The tool can be used via the command-line and generates a HTML report with all the detected smells. To facilitate adoption, we also created a Maven plugin which enables users to execute the tool directly from their build scripts[5]. Given the results of our study and, in particular, the finding reporting the possible relationship between the proposed MVC code smells and the code change- and fault-proneness, we invite developers to run such a tool on their MVC application and consider the refactoring of the identified code smell (if any). The need for such a recommendation is highlighted by the results of our survival analysis, showing that most smell instances are (dangerously) ignored and not refactored, thus affecting the system for long time periods. To foster the adoption of our tool, we also made sure to share it in the community, which appreciated our effort; the official Spring MVC channel on Twitter also shared our link[6]. Nevertheless, as showed, the proposed smells can be also applied to other MVC-based frameworks; we invite the community to extend our tool to support other frameworks.

[5]mvn com.github.mauricioaniche:springlint-maven-plugin:0.5:springlint. Change "0.5" to the most recent version of the tool.

[6]https://twitter.com/springcentral/status/781894510292963328.

Our future work agenda includes:

1. Expanding our catalog, to include new code smells specific of the MVC architecture (e.g., the Fat Model suggested by the Ruby on Rails expert);
2. Stemming from the results of our RQ4 (i.e., a minority of smell instances are removed by developers), we plan to investigate what the reasons pushing developers to remove smell instances are;
3. The definition and empirical analysis of other catalogs of smells, specific for other architectures.

Acknowledgments Bavota thanks the Swiss National Science foundation for the financial support through SNF Project JITRA, No. 172479.

References

Abbes M, Khomh F, Gueheneuc Y-G, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 15th European conference on software maintenance and reengineering (CSMR). IEEE, Oldenburg, pp 181–190

Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD international conference on management of data. Washington, USA, pp 207–216

Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: IEEE international conference on software maintenance (ICSM). IEEE, Timisoara

Alves TL, Correia JP, Visser J (2011) Benchmark-based aggregation of metrics to ratings. In: joint conference of the 21st int'l workshop on and 6th int'l conference on software process and product measurement (IWSM-MENSURA). IEEE, Nara

Aniche M (2017) Springlint. http://www.github.com/mauricioaniche/springlint (accessed June 20, 2017)

Aniche M, Bavota G, Treude C, van Deursen A, Gerosa MA (2016a) Code smells for Model-View-Controller architectures: Online appendix. https://doi.org/10.5281/zenodo.253681 (accessed June 20, 2017)

Aniche M, Bavota G, Treude C, van Deursen A, Gerosa MA (2016b) A validated set of smells in Model-View-Controller architecture. In: 32nd international conference on software maintenance and evolution (ICSME). Raleigh, USA

Arcoverde R, Garcia A, Figueiredo E (2011) Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the 4th workshop on refactoring tools. ACM, Waikiki, pp 33–36

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: 28th IEEE international conference on software maintenance (ICSM). IEEE, Riva del Garda, pp 56–65

Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. J Syst Softw 107:1–14

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: 2009 6th IEEE international working conference on mining software repositories. IEEE, Vancouver, pp 1–10

Brown WH, Malveau RC, Mowbray TJ (1998) Antipatterns: refactoring software, architectures, and projects in crisis. Wiley

Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: International conference on the quality of information and communications technology (QUATIC). IEEE, Oporto, pp 106–115

Chen T-H, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the 36th international conference on software engineering. ACM, Hyderabad, pp 1001–1012

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Coleman D, Lowther B, Oman P (1995) The application of software maintainability models in industrial software systems. J Syst Softwa 29(1):3–16

Conover WJ (1998) Practical nonparametric statistics, 3rd edn. Wiley, New York

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 35th international conference on software engineering. IEEE Press, San Francisco

Evans E (2004) Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional

El Emam K, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650

Fard AM, Mesbah A (2013) Jsnose: detecting javascript code smells. In: IEEE 13th international working conference on source code analysis and manipulation (SCAM). IEEE, Eindhoven, pp 116–125

Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: 11th international conference on software maintenance (ICSM). IEEE, Amsterdam

Fontana FA, Ferme V, Zanoni M, Yamashita A (2015) Automatic metric thresholds derivation for code smell detection. In: Proceedings of the sixth international workshop on emerging trends in software metrics. IEEE Press, Florence

Fowler M (1997) Refactoring: improving the design of existing code. Addison-Wesley Professional

Fowler M (2002) Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co. Inc.

Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates

Hecht G, Rouvoy R, Moha N, Duchien L (2015) Detecting antipatterns in android apps. In: Proceedings of the second ACM international conference on mobile software engineering and systems. IEEE Press, Florence, pp 148–149

Herzig K, Zeller A (2013) The impact of tangled code changes. In: Proceedings of the 10th working conference on mining software repositories, MSR '13. San Francisco, CA, USA, pp 121–130

Jay G, Hale JE, Smith RK, Hale DP, Kraft NA, Ward C (2009) Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. J Softw Eng Appl 2(3):137–143

Krasner GE, Pope ST et al (1988) A description of the model-view-controller user interface paradigm in the smalltalk-80 system. J Object Oriented Programming 1(3):26–49

Khomh F, Di Penta M, Guéhéneuc Y-G, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empir Softw Eng 17(3):243–275

Khomh F, Penta MD, Gueheneuc Y-G (2009) An exploratory study of the impact of code smells on software change-proneness. In: 16th working conference on reverse engineering (WCRE). IEEE, Lille, pp 75–84

Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196

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

Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128

Lozano A, Wermelinger M, Nuseibeh B (2007) Assessing the impact of bad smells using historical information. In: Ninth international workshop on principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, IWPSE '07. ACM, New York, pp 31–34

Martin RC (2009) Clean code: a handbook of agile software craftsmanship. Pearson Education

Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: 20th IEEE international conference on software maintenance. IEEE, Chicago, pp 350–359

Mazinanian D, Tsantalis N, Mesbah A (2014) Discovering refactoring opportunities in cascading style sheets. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, Hong Kong, pp 496–506

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng (4):308–320

Mesbah A, Mirshokraie S (2012) Automated analysis of css rules to support style maintenance. In: 34th international conference on software engineering (ICSE). IEEE, Zurich, pp 408–418

Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F (2010) Decor: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Nederlof A, Mesbah A, Deursen Av (2014) Software engineering for the web: the state of the practice. In: Companion proceedings of the 36th international conference on software engineering. ACM, Hyderabad, pp 4–13

Nejmeh BA (1988) Npath: a measure of execution path complexity and its applications. Commun the ACM 31(2):188–200

Oliveira P, Valente MT, Paim Lima F (2014) Extracting relative thresholds for source code metrics. In: IEEE conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE). IEEE, Antwerp

Olbrich SM, Cruze DS, Sjøberg DI (2010) Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: IEEE international conference on software maintenance (ICSM). IEEE, Timisoara

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: IEEE/ACM 28th international conference on automated software engineering (ASE). IEEE, Palo Alto, pp 268–278

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? a study on developers' perception of bad code smells. In: IEEE international conference on software maintenance and evolution (ICSME). IEEE, Victoria, pp 101–110

Peters R, Zaidman A (2012) Evaluating the lifespan of code smells using software repository mining. In: 16th European conference on software maintenance and reengineering (CSMR). IEEE, Szeged, pp 411–416

Pmd (2017) https://pmd.github.io/ (accessed June 20, 2017)

Ratiu D, Ducasse S, Gîrba T, Marinescu R (2004) Using history information to improve design flaws detection. In: 8th European conference on software maintenance and reengineering (CSMR 2004), 24-26 March 2004. IEEE Computer Society, Tampere, pp 223–232

Riel AJ (1996) Object-oriented design heuristics, vol 335. Addison-Wesley Reading

Sjoberg DI, Yamashita A, Anda BCD, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156

Saboury A, Musavi P, Khomh F, Antoniol G (2017) An empirical study of code smells in javascript projects. In: IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE, Klagenfurt, pp 294–305

Shepperd M (1988) A critique of cyclomatic complexity as a software metric. Softw Eng J 3(2):30–36

Sheskin DJ (2003) Handbook of parametric and nonparametric statistical procedures. CRC Press

Silva LH, Ramos M, Valente MT, Bergel A, Anquetil N (2015) Does javascript software embrace classes? In: IEEE 22nd international conference on software analysis, evolution and reengineering (SANER). IEEE, Montreal, pp 73–82

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM, Singapore, pp 4–15

Turnaround Z (2017) Top 4 java web frameworks revealed. http://bit.ly/1smVDf9 (Accessed 20 June 2017)

Van Deursen A, Moonen L, van den Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pp 92–95

Webster B (1995) Pitfalls of object-oriented development. M T books

Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects? In: 28th IEEE international conference on software maintenance (ICSM). IEEE, Riva del Garda, pp 306–315

Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, San Francisco, pp 682–691

Yamashita A, Moonen L (2013) Do developers care about code smells? An exploratory survey. In: 20th working conference on reverse engineering (WCRE). IEEE, Koblenz, pp 242–251

Maurício Aniche is a postdoc researcher at Delft University of Technology. Maurício holds a PhD from University of São Paulo, where he researched about the important of contextual measurements in software maintenance. Maurício likes to have one foot in research and one foot in industry. Throughout his PhD, Maurício co-founded Alura, the most popular e-learning platform for software engineers in Brazil. Maurício has authored two technical books (in portuguese) about pragmatic testing (TDD in Real World) and object-oriented design (OOP for Ninjas) which have sold more than 7k copies. His research interests are empirical software engineering, more specifically in software testing, maintenance, and architecture.

Gabriele Bavota is an Assistant Professor at the Università della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is author of over 90 papers appeared in international journals, conferences and workshops. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, ICSME, MSR, ICPC, SANER, SCAM, and others.

Christoph Treude is a Senior Lecturer in the School of Computer Science at the University of Adelaide, Australia. He completed his PhD in Computer Science at the University of Victoria, Canada, in 2012 and received his Diplom degree in Computer Science / Management Information Systems from the University of Siegen, Germany, in 2007. The goal of his research is to advance collaborative software engineering through empirical studies and the innovation of processes and tools that explicitly take the wide variety of artifacts available in a software repository into account.

Marco Aurélio Gerosa is an associate professor at the Northern Arizona University and a member of the graduate program in the Computer Science Department at the University of São Paulo, Brazil. His research lies in the intersection between Software Engineering and Social Computing, focusing on empirical software engineering, mining software repositories, software evolution, and social dimensions of software development. For more information, visit http://www.marcoagerosa.com.

Arie van Deursen is professor in software engineering at Delft University of Technology, The Netherlands, where he heads the Software Engineering Research Group (SERG) and chairs the Department of Software Technology. His research interests include empirical software engineering, software testing, and software architecture. He aims at conducting research that will impact software engineering practice, and has co-founded two spin-off companies from earlier research. He serves on the editorial boards of Empirical Software Engineering, the ACM Transactions on Software Engineering, and the open access PeerJ/CS, and is program co-chair of ESEC/FSE 2017, the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.