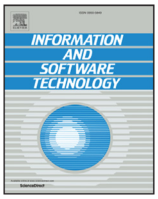
Information and Software Technology 125 (2020) 106333

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites

Amjed Tahir[a,*], Jens Dietrich[b], Steve Counsell[c], Sherlock Licorish[d], Aiko Yamashita[e]

[a] School of Fundamental Sciences, Massey University, New Zealand
[b] School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
[c] Department of Computer Science, Brunel University, United Kingdom
[d] Department of Information Sciences, University of Otago, New Zealand
[e] Department of Computer Science, Oslo Metropolitan University, Norway

ARTICLE INFO

*Keywords:*
Code smells
Anti-patterns
Mining software repositories
Stack exchange

ABSTRACT

*Context:* In this paper, we investigate how developers discuss *code smells* and *anti-patterns* across three technical Stack Exchange sites. Understanding developers perceptions of these issues is important to inform and align future research efforts and direct tools vendors to design tailored tools that best suit developers.
*Method:* we mined three Stack Exchange sites and used quantitative and qualitative methods to analyse more than 4000 posts that discuss code smells and anti-patterns.
*Results:* results showed that developers often asked their peers to smell their code, thus utilising those sites as an *informal, crowd-based* code smell/anti-pattern detector. The majority of questions (556) asked were focused on smells like Duplicated Code, Spaghetti Code, God and Data Classes. In terms of languages, most of discussions centred around popular languages such as C# (772 posts), JavaScript (720) and Java (699), however greater support is available for Java compared to other languages (especially modern languages such as Swift and Kotlin). We also found that developers often discuss the downsides of implementing specific design patterns and 'flag' them as potential anti-patterns to be avoided. Some well-defined smells and anti-patterns are discussed as potentially being acceptable practice in certain scenarios. In general, developers actively seek to consider *trade-offs* to decide whether to use a design pattern, an anti-pattern or not.
*Conclusion:* our results suggest that there is a need for: 1) more *context and domain sensitive* evaluations of code smells and anti-patterns, 2) better guidelines for making *trade-offs* when applying design patterns or eliminating smells/anti-patterns in industry, and 3) a unified, constantly updated, catalog of smells and anti-patterns. We conjecture that the *crowd-based* detection approach considers contextual factors and thus tend to be more trusted by developers than automated detection tools.

## 1. Introduction

Code smells [18] and anti-patterns [9] may reflect design and/or implementation issues in software source code that could have a negative impact on software quality. They are indicators of structures or solutions that could lead to difficulties during software evolution and maintenance. Given their potential negative impact, code smells and anti-patterns should be detected and then removed through suitable refactoring actions or the implementation of a design pattern [1,46,47,54].

A large body of research in the area of code smells has therefore focused on smell detection and removal techniques [24,30,53]. However, recent work [35,59] suggests that code smells are not always useful for identifying problematic code (i.e., code that is buggy or hard to maintain). Moreover, other studies have reported that a major challenge for those using smell detection tools is the problem of too many false positives [17]. We postulate that, currently, the software engineering community faces a gap in their understanding around the level of criticality behind the concepts of code smells and anti-patterns and, hence, software developers are not well supported. Developers thus congregate on online forums and similar portals in search of critical discourse around this issue. Stack Exchange development and programming sites offer utility for teasing out such issues and promoting awareness among developers. Stack Overflow (SO), since its launch in 2008, has moved from being a simple Q&A platform to a comprehensive repository of developer knowledge. SO covers both technical and general discussions on various topics including those related to software engineering. The Software Engineering (SE) site[1] is a more specialised portal and has a different focus to that of SO. The site is directed towards developers interested in asking general questions on the software development life cycle. Unlike SO,

* Corresponding author.
*E-mail address:* a.tahir@massey.ac.nz (A. Tahir).

[1] formally known as Programmers.SE.

https://doi.org/10.1016/j.infsof.2020.106333
Received 26 May 2019; Received in revised form 20 March 2020; Accepted 1 May 2020
Available online 7 May 2020

**Table 1**
Sites basic statistics as of May 2019.

| Site | Established | #users[2] | #questions | #answers | url |
|---|---|---|---|---|---|
| SO | 15 Sep 2008 | 10m | 18m | 27m | https://stackoverflow.com |
| SE | 1 Aug 2008 | 282k | 53k | 156k | https://softwareengineering.stackexchange.com |
| CR | 6 Sep 2008 | 172k | 59k | 94k | https://codereview.stackexchange.com |

opinion-based questions are tolerated more on this portal. Code Review (CR) allows programmers to ask direct questions about specific pieces of code (through providing code snippets or a direct link to a remote repository) and also seeks review feedback from peer developers on the quality and efficiency of their code. From now on, and for simplicity, we use the abbreviation of each site.

While SO has been widely used in other empirical software engineering studies (we discuss some of those studies in Section 2), we are not aware of any studies that have utilised information from the other two sites (i.e., SE and CR). The three sites attract large number of developers, with SO being more popular and much larger in size in terms of the number of users that contribute to the platform among all three sites. Basic statistics from all three sites are shown in Table 1. The actual users of those sites can be professional developers, enthusiasts, academics or students. We consider this mixture of users as an advantage (in terms of empirical studies) since they provide views from different perspectives.

In summary, SO and CR sites focus more on programming practices (e.g., code, languages and algorithms) whereas the SE site focuses more on process (e.g., development practices, code quality etc). Given their emergence as a popular and comprehensive (developer-based) knowledge repositories in software engineering, we assert that the three sites are a relevant source of information for investigating the state of developers' *understanding* and *praxis* regarding code smells and anti-patterns. We believe that the data collected from all three forums provide us with comprehensive and complementary views of these topics since they cover pure technical and language specific features discussions, as well as general-subjective discussions about code smells and anti-patterns. We also considered other Stack Exchange sites such as Software Quality Assurance & Testing[3] but we found that the data sample size was very small (e.g., search queries returned 38 posts in total), which makes it unsuitable for our analysis.

There are other similar popular forums that developers may use to discuss code smells and anti-patterns such as Code Project[4] and Reddit Software Engineering (and other related subreddits)[5]. However, Stack Exchange sites, and especially SO, are seen as much larger in size in terms of number of active users that contribute to the platform compared to those other forums. Reddit Programming is seen as close to SO in nature (given that it covers programming posts), while Reddit Software Engineering is the closest to Stack Exchange's SE in terms of its scope and topics. However, both sites are smaller (in terms of number of users and questions) compared to Stack Exchange Sites[6]. Therefore, we decided to include Stack Exchange sites in this study and leave Reddit and other similar forums for future studies.

The goal of this study is to examine how the topics of *code smells* and *anti-patterns* are discussed among developers in Stack Exchange sites. In particular, we seek to obtain a better understanding of *how smells and anti-patterns are understood and perceived by developers, what corrective actions* (if any) *developers take to deal with them*, and *what challenges* (technical and otherwise) *are faced by developers when dealing with smells and anti-patterns*.

The findings of this study should help to inform future research directions in the areas of code smells and anti-patterns and to devise solutions in these areas that are better fit to the realities that developers face. Findings may also help us *rethink* the notion of code smells and anti-patterns and focus not only on detection strategies, but also on finding which smells or anti-patterns would potentially be more useful for developers to detect and remove.

In this paper, we extend our earlier work on studying code smells and anti-patterns on SO [51] by: 1) using a larger dataset from other relevant Stack Exchange Sites (i.e, SE and CR) and also including more up-to-date data from SO (with 55% additional posts), 2) answering two additional research questions discussing the trends of smells in terms of programming languages used, and also possible challenges that face developers when discussing code smells and anti-patterns questions on Stack Exchange, 3) including additional observations that we made through qualitative analyses of posts from all three forums and, 4) providing a more comprehensive discussion on the implications of our findings and potential challenges that may face the software engineering community when dealing with smells and anti-patterns.

## 2. Related work

To the best of our knowledge, there are only two prior studies that have sought to understand the perceptions of developers regarding code smells. Yamashita and Moonen [59] conducted a survey with 85 software developers to understand what developers thought of code smells. Palomba et al. [35] studied perceptions of 12 code smells in Java with 19 developers and 15 postgraduate students. This latter study did not consider smells other than the 12 selected ones or languages other than Java. Also, the study was undertaken in a controlled environment (developers were asked to identify smells in given code). A recent study by Taibi et al. [52] replicated the two previous studies [35,59] and found that the majority of developers always considered them harmful. The authors found that developers perceived smells as critical in theory but not in practice.

Our study complements these works by: 1) addressing a larger set of developers (in that Stack Exchange sites offer an opportunity to study a large number of posts by many developers), and 2) studying the notions of code smells and anti-patterns across different environments, programming paradigms and languages. More interestingly, we study practitioner logs, thereby providing additional context to earlier studies.

There have been other studies that have investigated how developers follow software engineering best practices in their daily software development routine. For instance, some of the previous studies have looked into whether best practices are being following in web [11] and REST APIs development [38].

### 2.1. Studies on stack exchange and other online Q&A platforms

In recent years, studies have considered a range of issues faced by developers by mining on-line forums and platforms. Previous studies have used SO data to study user behaviour and topic trends [33,34,45,56]. Others have applied Topic Modelling techniques to analyse SO questions to categorise topics and discussions [4,6]. Several other works have analysed specific posts related to particular topics, such as web development [5], mobile development [25,43], cryptography APIs [31] and mobile app energy consumption [39].

[2] based on data retrieved from the Stack Exchange Data Explorer on May 2019
[3] https://sqa.stackexchange.com [Accessed: 23 May 2019]
[4] https://www.codeproject.com [Accessed: 23 May 2019]
[5] https://www.reddit.com/r/SoftwareEngineering [Accessed: 11 May 2019]
[6] as of 1 March 2020, there are about 20k members at Reddit Software Engineering and 3k members at Reddit Programming

Rosen and Shihab [43] conducted a study to investigate which questions and topics were related to mobile development on SO. Their study found that the most popular topics of discussion were those relevant to app distribution, APIs and data management. Reboucas et al. [42] mined SO to investigate how developers used the Swift programming language. In particular, the study focused on the problems faced by developers when using Swift. A recent study by Ahmed and Bagherzadeh [2] mined SO to study how developers discussed concurrency-related topics across different programming languages. The closest to our study is the study of Choi et al. [10], who studied how the topic of code clones was discussed on SO. The study checked all *tags* associated with "clones" in SO and found that C#, Java and C++ were the languages associated with the most questions on code clones. To the best of our knowledge, our study is the first to look into code smells and anti-patterns within Stack Exchange sites.

### 2.2. Studies on code smells and anti-patterns

The impact of code smells on a developer's daily tasks has been discussed (mostly) in controlled settings. For example, several works have studied the impact of code smells on software quality using a group of developers working on a specific project [36,46,47,57]. Most reported studies have focused on investigating the impact of smells and anti-patterns on software quality attributes (such as defect-density [21,23], maintenance [46], modularity [13] and code readability and understandability [1]); when such an impact exists, a set of refactorings are provided. Existing systematic review studies on code [55,61] and design [3] smells provide extensive discussion on the potential impact of smells. Other studies have also investigated the impact of different forms of code smells and anti-patterns on software quality, such as architectural smells [17,20,29], test smells [7,48,50] and spreadsheet smells [15,22]. In more recent work, an attempt to improve defect prediction by taking smell information into account was presented by Palomba et al. [37]. The authors found that prediction models that used smell information as an additional predictor variable had increased accuracy compared to other baseline models. A similar approach was proposed in [26] to predict change-prone files, where different smell-based metrics for effort-aware structural change-proneness prediction were examined.

## 3. Study design

### 3.1. Research aim and questions

The aim of this work is to obtain a comprehensive understanding of developer perception of code smells and anti-patterns to inform tools designers of what sort of smells/anti-patterns developers are interested in capturing and analysing. Such an understanding can also help shape future research directions, based on knowledge obtained from actual developers. By determining what developers think about the impact of smells and anti-patterns, we can also calibrate our research towards those that are of most concern to developers. To achieve the goals of this study, we address the following five research questions:

- **RQ1. Which code smells and anti-patterns are *most actively discussed*in the three Stack Exchange sites?**
- **RQ2. What are the most popular programming languages in terms of code smells and anti-patterns questions?**
- **RQ3. What are the concepts/definitions involved in the questions that developers tag with "*code smell*" or "*anti-pattern*"?**
- **RQ4. What *corrective actions*are suggested to deal with or resolve a given code smell or anti-pattern?**
- **RQ5. What are the *challenges*(technical and otherwise) that developers may face in dealing with code smells and anti-patterns?**

### 3.2. Data extraction

We extracted our data through the Stack Exchange Data Explorer web interface[7] Stack Exchange releases "data dumps" of its publicly available content. The online data explorer tool provides up-to-date access to the latest data dump by Stack Exchange. We ran SQL queries to mine relevant data for our needs from this data dump. Our data was obtained in two stages; first, data was obtained from SO on October 2016 ([51]). We then obtained the rest of data on August 2018. The data that we obtained in the second stage contained SO data from October 2016 to December 2017 and also all data from SE and CR (up to the end of 2017).

An overview of our search process is shown in Fig. 1. The search process combines both an iterative automatic search and several manual filtration processes conducted to answer the different research questions.

In the Stack Exchange data set, both *questions* and *answers* are considered as *Posts*, and are all added to the *Posts* table. Given that we are interested primarily in the questions asked by developers, we ran our query to search for questions only. This can be identified through the *PostTypeId* field in the *Posts* table. Questions are given a *PostTypeId* value of 1, where answers are given a value of 2. Therefore, our designed SQL query searches through all posts, but returns only questions.

Note that, for the purpose of this analysis, we mined posts related to source code and software design -and specifically for OO designs - other forms of smells and anti-patterns, such as service oriented architecture and database smells were excluded from this analysis.

Searching exclusively via *Tags* can be ineffective. There are several disadvantages of using *Tags* as the means for determining whether a post is related to a topic [6,12]. A user who creates a question could be unsure about the most appropriate tag for their discussion which can lead to the use of *incorrect* or *irrelevant* tags. For example, we observed that some developers used the tag 'Clone' for questions related to three unrelated topics: Code Cloning (i.e., code duplication), Git cloning feature and 'object cloning' in OO languages (the `clone()` method that is used to duplicate objects). Another issue with user-defined tags is that many users tend to add as many tags as possible (Stack Exchange allows up to 5 tags) to increase the number of views and potentially increase the likelihood of receiving answers quickly [5]. Thus, while tags can be helpful to capture questions related to code smells and anti-patterns, using tags *exclusively* may lead to neglect of important questions on this topic. Therefore, we decided to mine the main *Posts* body to maximize our search coverage. However, we still used tags as initial indicators of the extent of discussions (RQ1) and also to check for what developers declared as a code smell or an anti-pattern question (RQ2).

During the initial runs of the search query, we found that SO's developers use different formats of the same terms. Therefore, in our search, we used several variations of terms that refer to code smells or anti-patterns, including: "code smell" (number of posts found: 1020), "bad smell" (68), "anti-pattern" (1233), "anti pattern" (168), and "antipattern" (21). These different variations have been then included in our search query. We also included "technical debt" (70) as it is seen to be a related concept (i.e., smells that are left in the code are widely treated as technical debt [60]).

Given there is a chance that a user might ask a question about a specific code smell (e.g., God Class or Spaghetti Code) without including in the question terms such as *code smell* or *anti-pattern*, we also included a list of code smells and anti-pattern names in our search. Those we included were extracted from nine relevant studies on this topic [21,23,35,36,54,57,58,61], including the seminal work on code smells by Fowler [18].

We first combined the list of smells and anti-patterns from all nine studies. We then checked how frequently each term was used across the nine studies. We decided to include in our search query all smells

[7] http://data.stackexchange.com [Accessed: 10 May 2019].

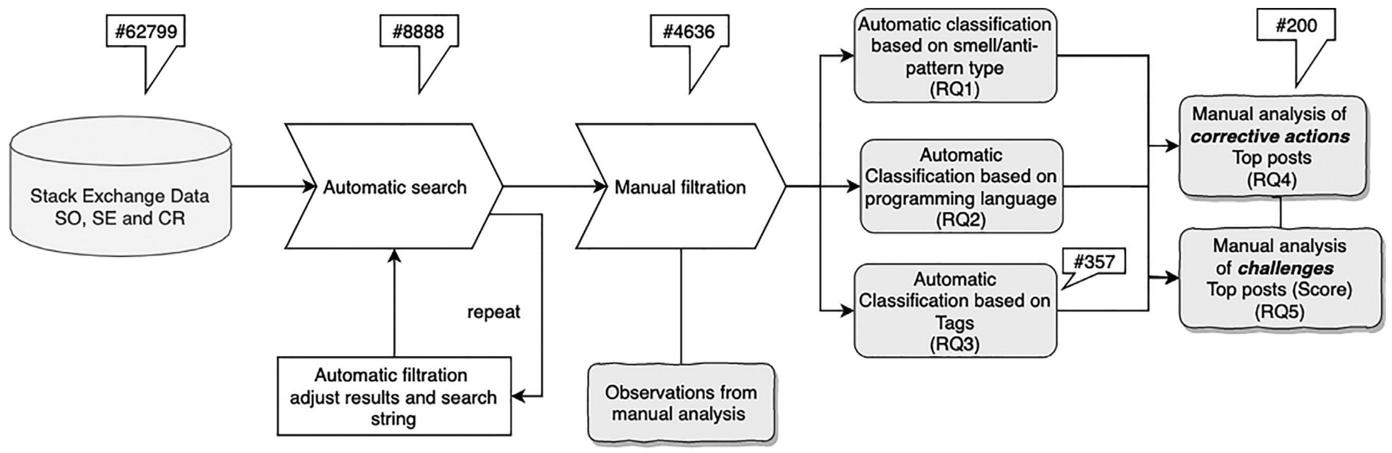

#62799
Stack Exchange Data SO, SE and CR
#8888
Automatic search
repeat
Automatic filtration adjust results and search string
#4636
Manual filtration
Observations from manual analysis
Automatic classification based on smell/anti-pattern type (RQ1)
Automatic Classification based on programming language (RQ2)
Automatic Classification based on Tags (RQ3)
#357
#200
Manual analysis of corrective actions Top posts (RQ4)
Manual analysis of challenges Top posts (Score) (RQ5)

**Fig. 1.** Our search, data extraction and filtration process (with the number of posts returned after each step).

**Table 2**
Terms that we included in our mining.

| General Terms | | | |
|---|---|---|---|
| Code Smell | Bad Smell | Technical Debt | Anti Pattern |
| Specific Smell Terms | | | |
| Feature Envy | Lazy Class | Message Chain | Speculative Generality |
| Data Clumps | Shotgun Surgery | Divergent Change | Large Class |
| Long Method | Duplicated Code | Code Duplication | Code Clone |
| Refused Bequest | Data Class | Case Statement | Switch Statement |
| Middle Man | Parallel Inheritance | Blob Class | God Class/Object |
| Brain Class | Complex Class | Spaghetti Code | Temporary Field |

that appeared in at least two studies. The resulting list thus included 34 unique code smell and anti-pattern terms. We believe this enabled us to be inclusive and reduce the chance of missing any important relevant questions on the topics of interest. When running the SQL query, however, we encountered difficulties in retrieving results when we included all of the smell and anti-pattern names, especially when including long strings (those containing more than two words). In contrast, the query would work well after the removal of any terms comprising more than two keywords. Therefore, we chose to remove the following keywords: “Long Parameter List”, “Class Data Should bePrivate” and “Alternative Classes with Different Interfaces”. The final list of terms, code smell and anti-pattern names included in our search is shown in Table 2. We provide a description of these smells and anti-patterns externally. A full replication package (including our full dataset and SQL queries) is available online [8]. In total, our search retrieved 62,799 questions posted between August 2008 and December 2017.

We conducted a follow-up investigation to verify the accuracy of our search string with regards to the possibility of missing posts due to misspellings. We used a natural language processing approach proposed by Norvig [32] (designed for spell-correction) to find a set of all possible misspellings of a given word[9]. We applied this approach on two keywords that we identified: *smell* and *antipattern*, as they are the main keywords for this study. We found a large set of possible misspelled words for the two terms (i.e., smell: 284 and antipattern: 596). We then run a search query to look for all possible posts that contain the misspelled words. We conducted this investigation on SE only, as we found that running the search query with all 596 probabilities on SO proved to be time and resource consuming due to the long list of words (a total of 880 keywords). As a result, the search returned a large number of posts (over 3000) and after manual analysis we found that all of those posts were unrelated to our study (i.e., matched common words such as “spell”, “small”, “sell” and “shell” present in many posts). By removing those words, the search query did not return any results (zero matching). This indicated that the likelihood of our search missing any relevant posts related to the two terms due to misspelling was very low. We believe the results will be of a similar nature for SO and CR, as for SE.

Next, we ran several filtering steps to clean the data and to eliminate false-positives. Our filtering processes are explained in the following section.

## 3.3. Data filtering

We performed text mining using the keywords in Table 2. We ran this process in iterations, where a search query looked for the specific keywords that we were searching for, updating the search process after each review. In the first instance, we searched for simple keywords representing the list of code smells and anti-patterns (as shown in Table 2). We then reviewed the results to identify missing posts and updated the search query accordingly. For example, we conducted a first search with simple keywords (e.g., code smell, anti-pattern) and then included variations of the same keywords and updated the search string (e.g., bad smell, technical debt, antipattern). In the second step, we conducted a more refined search using the updated search string until we derived the final search query. We performed a manual check (by the first author, in consultation with another co-author) after each result was obtained to refine the search term. Details of the automatic and manual filtration processes are explained next.

***Automated and manual filtering*** We ran a manual check on the first (earliest) 50 questions from each forum (total of 150 posts, sorted by posts *Creation Date*) to assess the form of the results and their suitability for our intended analyses. For the selected questions, we checked the *Tags*, *Title* and *Body* fields of each post. We identified 64 questions as false-positives (i.e., questions that were not directly related to code

[8] http://dx.doi.org/10.17632/fdcvtf5nx6.1
[9] we used the data corpus and scripts provided in http://norvig.com/spell-correct.html [Access: 10 March 2020].

**Table 3**
Number of posts after each filtration process.

| Platform | # retrived posts (total) | After auto filtration & removing duplicates | After manual filtration |
|---|---|---|---|
| SO | 60,283 | 7677 | 3,164 |
| SE | 1074 | 813 | 569 |
| CR | 1442 | 397 | 303 |
| Total | 62,799 | 8888 | 4036 |

smells or anti-patterns). Most of those questions were related to the term *Blob*, which could refer to the relevant term “Blob Class”, but which is also used as a type in a relational database. The identified false positives belonged to four categories: *database* questions (SQL questions that considered the use of blob as a datatype that stores a Binary Large Object as a column in a row), *Data class* posts (which is, in some languages such as Kotlin[10], is a built-in feature), *flash* posts (for questions on Adobe’s multimedia runtime used to embed animations and video into web pages) and questions related to the Google app engine (cloud computing technology for hosting web applications in Google-managed data centers). We therefore ran another SQL query to filter out these questions through the Body and Tags fields, which was able to eliminate all 64 false-positive results in the sample set. For the body of the questions, we added the term ‘*class*’ after ‘*blob*’ in order to make the query more precise. We also filtered out *Tags* related to the three categories described above. After applying our automatic filtering approach, the total number of questions fell to 8,888 (Table 3).

***Validation*** Automatic filtering was followed by a carefully conducted manual validation process, checking the *Tags, Title* and *Body* of each individual question to exclude all irrelevant questions. For the purposes of answering our research questions, we deemed it essential to complement quantitative analysis with interpretive research techniques typically used in *grounded theory* [49] to better understand the *context* of the quantitative data at hand. This is done to identify and describe different phenomena from a qualitative viewpoint and to explore the underlying mechanisms behind the identified phenomena.

This manual verification process was used for quality assurance; it did not fully replace the automatic search, but rather was used as a complementary process. Automatic search can produce large numbers of false positives and given the size of the dataset we could not be entirely sure whether we had inadvertently included irrelevant posts. The manual process helped to reduce the number of false positives and ensured that we had the right set of posts for the analysis. This approach has been widely used in similar empirical studies on mining SO posts (e.g., [25,43]).

Given the size of the dataset, more resources were needed to help with the manual coding process. We therefore recruited two external (non-author) Computer Science PhD students to help with this process. The manual process was performed by four different researchers, two of the authors and two external (non-authors). The external coders were familiar with the concepts of code smells and anti-patterns. Both were introduced to the work by one of the authors and shown examples of how questions should be categorised. Before doing the actual coding, each external coder, together with two authors, categorised 100 posts (randomly selected to avoid selection bias) together in an iterative way (i.e., where coding was conducted by one external coder, discussed with one of the authors, and then refined based on the discussion) to ensure that the coding was consistent (with a 100% agreement reached between the two). Once they had finalised the coding scheme, the work was distributed between the four coders. Each coder was given a set of posts (unequal, depending on availability, with a minimum of 500 posts) and was asked to follow specific guidelines to identify: 1) whether a post was actually relevant to code smells and anti-patterns and, 2) the main programming language that the posts are mostly associated with (to answer RQ2). When needed, we decided whether to include/exclude a post based on the definition of smells and anti-patterns as explained in previous studies (e.g., see [21,23,36,54,57,61]).

After manually validating all 8888 posts, we followed this with a careful *cross-validation process* where two of the reviewers validated 100 of each other’s coded posts (at 95% confidence level and a 8-10% confidence interval). After each coder completed their classification, 100 posts for each set were randomly selected and then cross-validated by a different reviewer. Next, the two coders discussed each classification they performed and the process resulted in over $\sim$ 90% agreement between the two coders. This manual filtration and validation process took approximately three months to complete.

For RQ3 (Section 4), we manually analysed all 357 questions tagged as either “code smell” or “anti-pattern”. The classification for this question was done by the first author and then cross-validated 62 (11% confidence interval and 95% confidence level) of those posts and their answers and comments (17% of the posts: 24 from the code smells and 38 from the anti-pattern lists) by two other co-authors (a total of two authors for each category). The cross-validation was done iteratively (i.e., classifying past by one of the coders and then verified by another) on posts from SO to make sure that our classification of the post was reliable. As for RQs 4 and 5, we manually analysed the top 200 posts (100 posts from SO and 50 posts from SE and CR each - with a 95% confidence level and a 10–13% confidence interval), by three authors (one set each). When none of the reviewers was able to classify or was unsure about the category, the post was then referred to one of the other co-authors for verification. In total, we cross-validated 35 posts (95% confidence level and 15% confidence interval) across all three sets and reached over 95% agreement on the classification of these posts between both coders.

***Classification*** We performed automatic classification of posts based on the types of smells or anti-patterns in the dataset to answer RQ1.

We then manually classified posts based on the targeted programming language in order to answer RQ2. Many of these posts were tagged with the related programming language; however, we did not depend solely on the tags used as we noticed that some questions were not tagged based on the programming language used, but for a specific feature or framework. For example, some JavaScript questions were tagged with AngularJS or React, popular JavaScript frameworks/libraries. While analysing the data obtained via the automatic search, we noticed a relatively low recall rate (65%) and we therefore decided to conduct a manual classifications on the results obtained from the automatic search to identify posts based on the targeted programming language. In total, we analysed 4036 posts from all three forums, with the majority of these posts (78%) coming from SO (see a breakdown of the distribution of posts in Table 3). This was expected, given the size of SO compared to the other two forums.

We also employed Thematic Content Analysis [8] to identify the main themes of the corrective actions suggested by developers to resolve code smells and anti-patterns (R4). Two authors jointly discussed the reasons for developing the themes. One of the authors first read through a post and its related discussion and then classified it based on the agreed themes. Another author then read the post and either agreed with the classification or provided a counter classification for the posts. An agreement between the two coders had to be reached before each

[10] https://kotlinlang.org/docs/reference/data-classes.html [Accessed: 8 Feburary 2020].

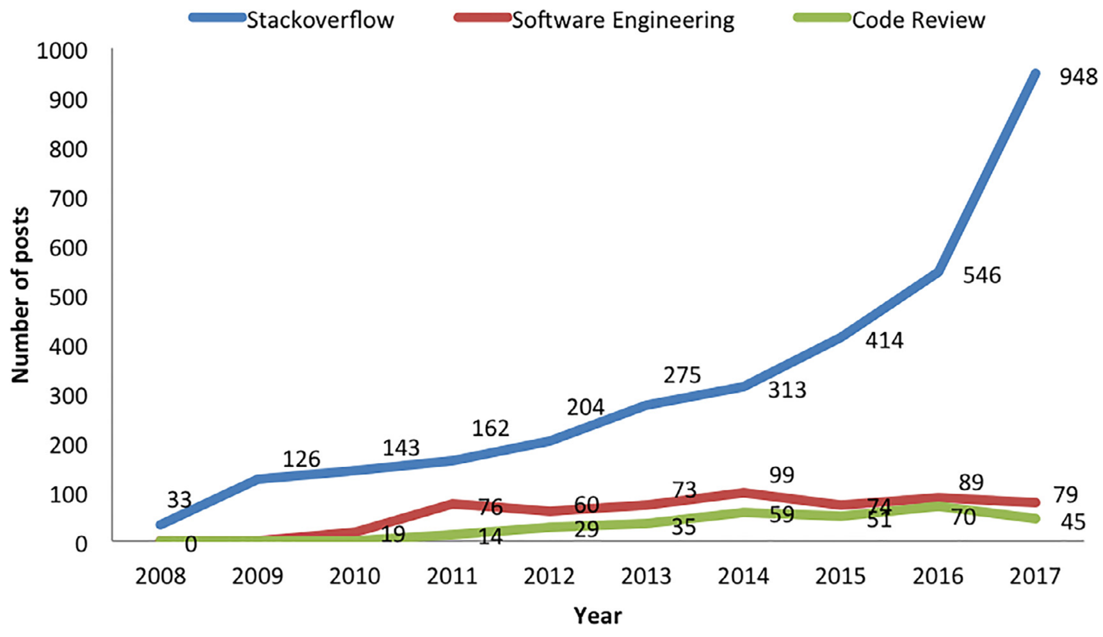


**Fig. 2.** Temporal trends in code smell and anti-pattern questions.

classification was accepted. This thematic analysis was conducted by three of the authors and each set was cross-checked with at least one co-author.

While performing the classification and reading through the posts and discussions, we noted down and then discussed different phenomena observed. This manual analysis provided contributions to the qualitative analysis that answers RQ5. We extracted a set of potential challenges that were identified by the community and discussed them. In addition, this has also led to the identification of several interesting observations from all posts - those observations are then presented in Section 5. Although it was done systematically, we acknowledge that manual classification poses a threat to the validity of our evaluation. We will discuss this in detail in Section 7.

## 4. Results

We discuss our results based on each individual research questions below. But first, as a preliminary exploration of the data, we look at the temporal trends of code smells and anti-patterns topics in the three forums to assess the level of attention paid by developers to these topics over time (Fig. 2). The results show that there has been a steady increase in the numbers of questions asked by developers over time, especially on SO. In a span of 9 years, the number of smells/anti-patterns related posts in SO has grown from 33 in 2018 to over 900 in 2019. While the trend is clearly evident, such an increase could be a result of the growth in the number of SO users over the period in question. We also note that there has been a noticeable growth in the number of questions in SE over the years, rising from 19 questions in 2010 to 99 in 2015 (the maximum point). The number of questions in SE has been fluctuating up and down over the years.

**RQ1. Which code smells and anti-patterns are *most actively discussed* in the three Stack Exchange sites?** To answer this question, we mined all questions in our dataset to check if a particular code smell or anti-pattern (from the list of smells and anti-patterns in Table 2) had been discussed in a post. For the purposes of analysis, we grouped smells of a similar nature, i.e., smells or anti-patterns that had a similar meaning and interpretation, as follows: 1) *Blob, God Class* and *Brain class*, and 2) *Duplicated Code* and *Code Clone*.

Fig. 3 shows the results for the number of questions asked for each individual (top 9) smell and anti-pattern (RQ1). We found relatively few questions naming smells or anti-patterns that we had included in our search ( $\sim$ 20% of the total number of posts we retrieved), with 444 of these posts from SO, 227 from SE and 139 from CR. Of this list, we found that smells and anti-patterns *God Class, Duplicated Code, Spaghetti Code* and *Data Class* were by far the most frequently discussed on the three forums. The same smells/anti-patterns were the most discussed in SO and SE, where *Duplicated Code* and *Complex Switch Statements* were the most discussed in CR. Size-based smells such as Large Class and Long Method were discussed to some extent across the three forums. Most of the other smells that we included in our search were rarely discussed, e.g., *Refused Bequest* and *Message Chain*. We found that no questions addressed the following smells or anti-patterns by name: *Speculative Generality, Data Clumps* and *Divergent Change*. While the number of posts that used the same terms as in our search is relatively low, we believe this is an important finding in itself. It could be that either developers do not discuss these smells and anti-patterns widely, or that they use other terms to refer to the same smells/anti-patterns. We discuss this further in Section 7.

We also investigated the popularity of questions asked about code smells and anti-patterns. To estimate the popularity of posts, we considered a number of metrics available from the dataset, including the *View Count, Score* and *Answers, Comments* and *Favourite Counts*. After investigating each metric across the three forums, we decided to use a scaling approach and combine data from both *View Count* and *Score*. We found that the *Favourite* feature is not as widely used by users as the other metrics (i.e., *UpVote*). The *Score* value provides an assessment of the level of acceptance of the questions by all users. Stack Exchanges calculates a *Score* of post by subtracting the number of down-votes from the number of up-votes. *View Count* measures the number of times a post has been accessed. We used both metrics as a better proxy of post popularity rather than using a single metric (i.e., *View Count* or *Score*). Previous studies have used a similar approach to drive a popularity metric, but included different metric variables such as Comments and Answers counts (e.g., [40]). We decided to exclude variables such as *Answers* and *Comments Counts* as we believe those metrics are biased towards older posts; newer posts in the forum are more likely to have fewer Answers and Comments.

We checked the rank correlation between *View Count* and *Score* values using Spearman's rho $\rho$. We found that *View Count* to be significantly correlated with *Score* ($\rho = 0.71$, $\alpha < 0.001$). We normalized all *View*

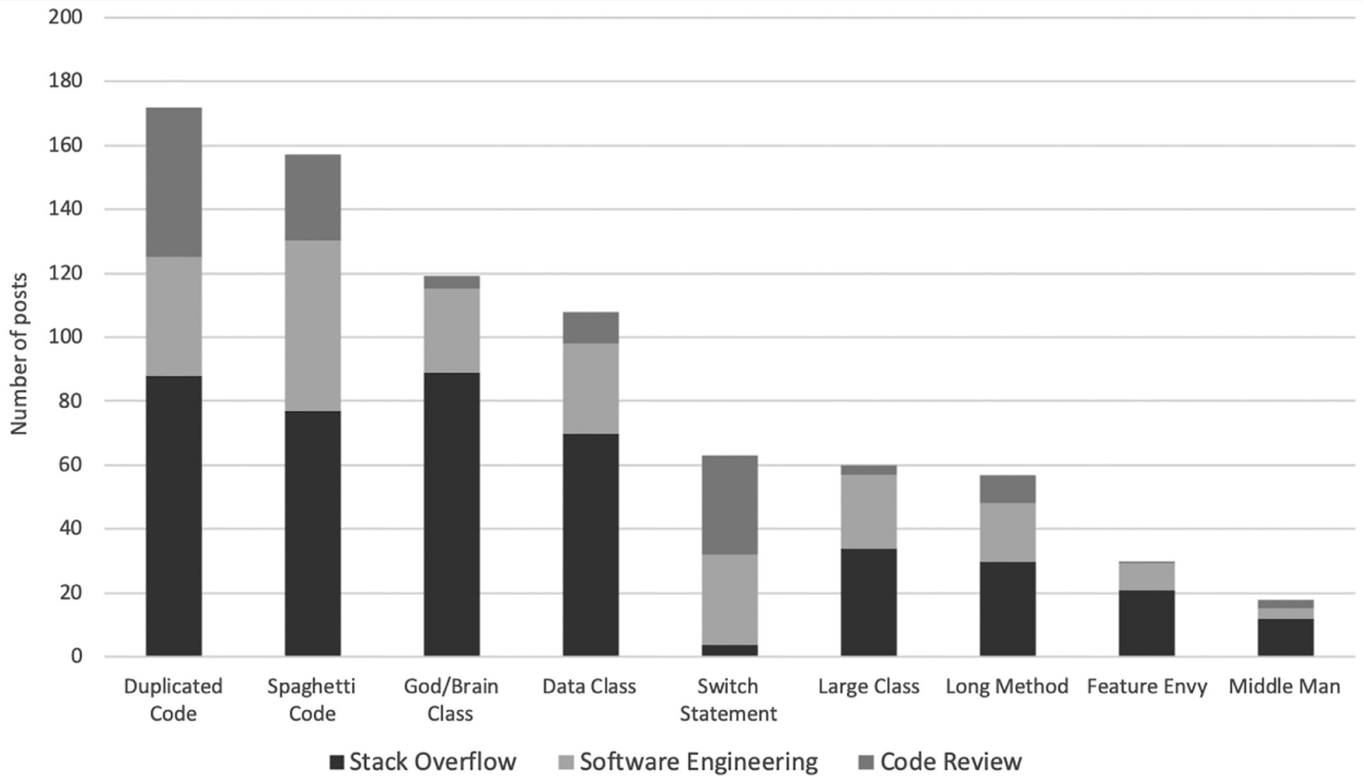


**Fig. 3.** Number of questions for each individual code smell and anti-pattern.

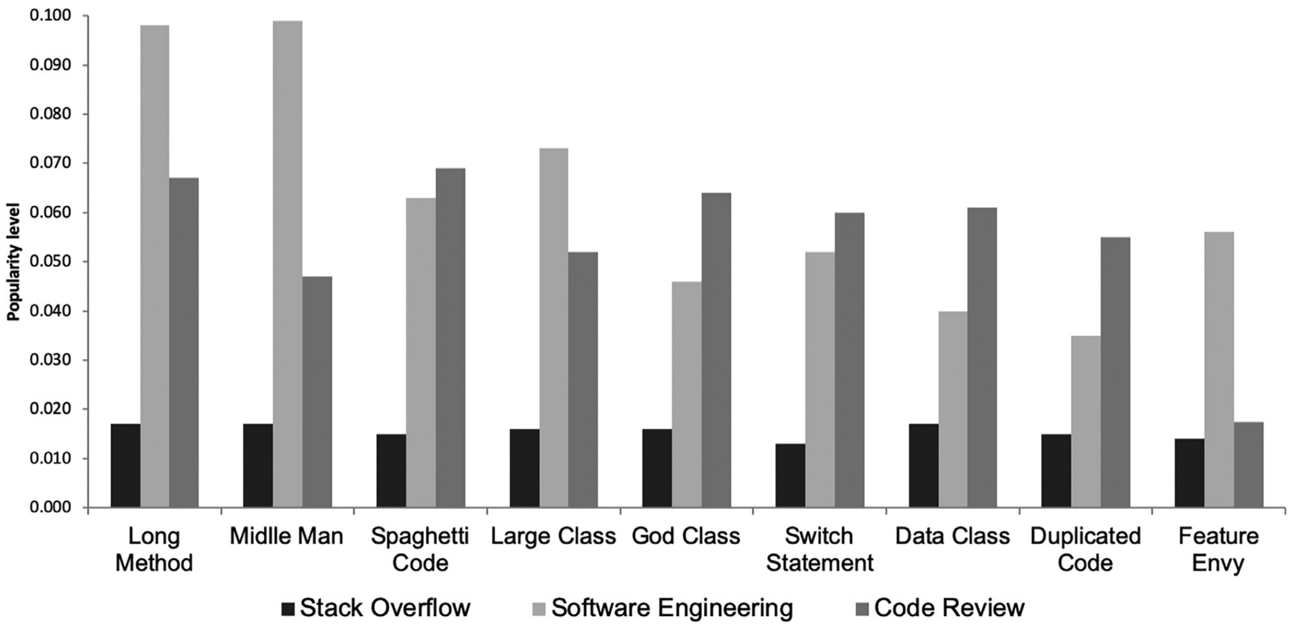


**Fig. 4.** Popularity of individual code smells and anti-patterns.

*Count* and *Score* values using Feature Scaling, according to the following formula:

$$X_{nor} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{1}$$

where $X_{nor}$ is the normalized value, X is the original value, $X_{min}$ is the minimum value in the range and $X_{max}$ is the maximum value in the range. We then created *a new question popularity metric* by taking the average of the two normalized values of *View Count* and *Score.*

The results of the popularity analysis for each smell and anti-pattern is shown in Fig. 4. We extracted the values of these metrics from all posts we obtained from the three forums. We found that posts on *Long Method, Middle Man* and *Large Class* were the most popular in SE and CR. Most questions almost rank the same for SO, compared to SE and CR. This could be due to the relatively small number of questions that use these terms compared with overall number of questions in our dataset. Questions regarding other smells/anti-patterns, such as *Shotgun Surgery* and *Large Class*, had low popularity by comparison.

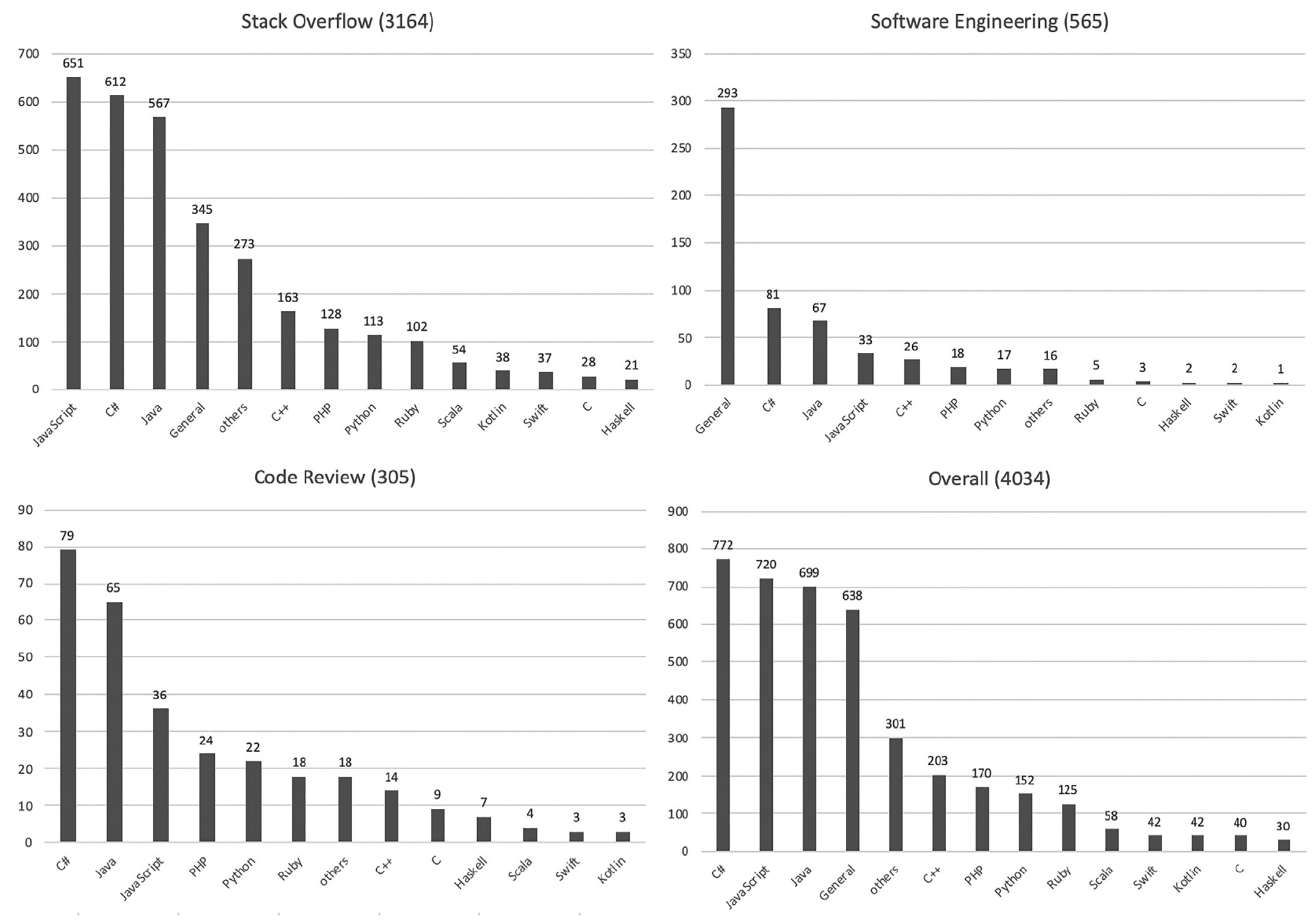


**Fig. 5.** Number of questions on code smells and anti-patterns asked on Stack Exchange sites considered by programming language (total number of posts is shown between brackets).

The results show that *God Class, Duplicated Code, Spaghetti Code* and *Data Class* are the most frequently discussed smells. We did not find any questions being asked on the following smells/anti-patterns: *Speculative Generality Data Clumps* and *Divergent Change*. In terms of popularity, the following smells/anti-patterns are shown to be the most popular in those sites: *Long Method, Middle Man* and *Spaghetti Code*.

**RQ2. What are the most popular programming languages in terms of code smells and anti-patterns questions?** In this question, we checked which programming languages were commonly referred to in questions about code smells and anti-patterns in all three forums. Fig. 5 shows that C#, JavaScript and Java were the languages with the most questions on code smells and anti-patterns across all three forums, constituting 55% of the total number of questions on these topics. The data also shows that, equally, there is a substantial proportion of questions ( ∼ 16%) which are language-agnostic (i.e., independent of any particular programming language). We also noticed that there was a relatively low incidence of questions involving functional programming languages such as Haskell and Scala compared to mainstream OO languages. We think this is due to the fact that the majority of smells studied in the literature (and also included in our search queries) are related to OO languages features and therefore questions related to OO were predominant.However, some of these smells are general and can be relevant to any programming paradigm (i.e., *code clone* and *Spaghetti Code*). When looking at the individual sites, we also noticed that there was a relatively higher number of general questions about smells and anti-patterns in SE ( ∼ 52%) compared to SO (11%). No questions that fall into this category were found in CR.

Most discussions on code smells and anti-patterns are centred around popular programming languages, with C#, JavaScript and Java dominating the landscape. Most of the code smells are OO-based, with a considerable number of general, language-agnostic smells and anti-patterns.

**RQ3. What are the concepts/definitions involved in the questions that developers tag with “ *code smell* ” or “ *anti-pattern* ”?** We examined posts tagged with either *code smell* or *anti-pattern*. This analysis was done on SO and SE data since we did not find any posts in CR tagged with either keywords. In total, we found 364 questions in this category (i.e., 141 questions with the tag *code smell* and 223 questions tagged with *anti-pattern* in both forums). After removing duplicate questions (questions that were tagged with both *code smell* and *anti-pattern*) by classifying posts depending on the first tag used, we ended up with 354 posts (82 (SO) and 56 (SE) with the tag *code smell*) and 138 (SO) and 77 (SE) with the Tag *anti-pattern*). The results for each of the two tags were then analysed separately for each dataset. We manually reviewed the body of these questions and their answers/comments and then categorised each post based on the name of code smell or anti-pattern that the question was discussing. We classified posts into three categories: **1) General:** general questions that were asked (i.e., where no specific smell or anti-pattern was mentioned) or discussed the general impact of code smells or anti-pattern, **2) Specific Code Smells/Anti-patterns:** where specific code smells or anti-patterns were being discussed and **3)**

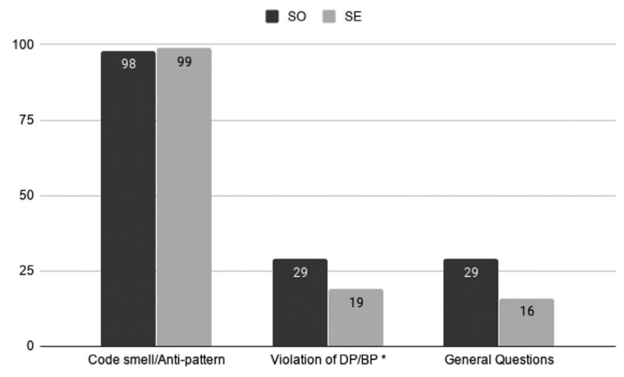


**Fig. 6.** Classification of posts tagged with *Code Smells* or *Anti-patterns*. *violations of design patterns or best practice.

**Violation of Design Principles and Code Best Practices:** where the violations of design best practices and the downsides of design patterns were considered as potential anti-patterns. We excluded questions that we considered to be irrelevant to code smells or anti-patterns (i.e., where a question was tagged as a *code smell* or *anti-pattern* but did not specifically discuss any. In total, we classified 136 posts tagged with *code smell* (80 from SO and 56 from SE) and 154 posts tagged with *anti-pattern* (76 from SO and 78 from SE). This process was undertaken by the first author of the paper and cross-validated by another co-author. We sampled 70 posts for the cross-validation - the size of the samples was set to obtain a 95% confidence level and a 10% confidence interval. A general overview of the results of this classification process is shown in Fig. 6.

In total, we marked 6 questions from the *code smells* list and 20 questions from the *anti-pattern* list as irrelevant or unrelated. We found that some questions discussed more than one code smell or anti-pattern. In such cases, we included all the names of these smells or anti-patterns. As a result, we found a total of 156 posts with the tag *anti-pattern* (79 from SO and 77 from SE). For the questions assigned the Tag *code smell*, we found that 22 of these were general (16 from SO and 6 from SE) and not specific to a particular code smell or anti-pattern. The nature of these questions ranged from general knowledge about code smells, discussion of specific coding standards, implication of implementing specific design principles to tool-specific questions. The majority (89 posts) of the other questions revolved around various forms of code smells, including production-code and test smells. While many of the smell types we found were known (such as *God Class/object, Type Checking, Spaghetti Code*, etc.), we also found new, so-called ‘code smells’ being discussed between developers. Such smells are not widely discussed in the community or studied in the literature. This includes the following: *Nested Try-Catch block, Exploit Polymorphism* and *Swallowed Exceptions*. There were 25 posts across both forums (with the code smell tag) that represented violations of design patterns or coding “best practice”. For example, several smells suggested by developers represent violations of design principles such as *Don’t Repeat Yourself (DRY), Law of Demeter (LoD), Liskov Substitution Principle (LSP)* and other *SOLID* principles [28].

For the posts with the ‘anti-pattern’ *Tag* (154 posts in total), the majority of questions involved a wide range of anti-patterns (108) with posts discussing the implications of certain anti-patterns (such as *God Object*, and *Big Ball of Mud*). There were also some general anti-pattern questions (23 posts in total) that discussed the potential significance of having anti-patterns in a program. Also, there were 23 posts across the two platforms that discussed the implications of violating basic design principles (such as DRY and Open-Closed Principles) and suggested these violations as anti-patterns. Other questions discussed the downfalls of certain design patterns (i.e., how some design patterns could potentially constitute anti-patterns). The *Service Locator* and *Singleton* design patterns were particularly widely discussed in the context of anti-patterns. Such questions discussed the downsides of these patterns and suggested that certain implementations of these patterns could result in anti-patterns.

One additional finding was that developers do not always distinguish between the terms *code smell* and *anti-pattern* - rather, they use the two terms interchangeably to refer to the same or similar design or implementation issues. Several questions that were tagged with *code smell* in fact discussed *anti-patterns* and *vice versa*. In SO for example, there were 47 posts where the terms “smell” and “anti-pattern” were used interchangeably within the same post to discuss the same smell/anti-pattern.

There is a lack of consistency in the definitions used in posts tagged as Code Smell and Anti-Pattern. Violations of certain design principles such as LSP and DRY are widely described as anti-patterns. The downsides of some well-known design patterns (such as Service Locator and Dependency Injunction) are suggested as potential causes of anti-patterns.

**RQ4. What *corrective actions* are suggested to deal with or resolve a given code smell or anti-pattern?** We manually analysed the most popular posts from each forum (based on the popularity metric described in formula (1)) to check the actions taken (or recommended) by developers to deal with smells and anti-patterns. We chose only the top posts (100 posts from SO and 50 posts from SE and CR each) so that we could carefully analyse these questions and all their answers with the available resources. Our analysis comprised three main aspects: 1) the name or type of the smell or anti-pattern, 2) the action recommended in the accepted answer and, 3), if there were some suggested actions taken to remove the smells or anti-pattern (i.e, refactoring operations), the name of the refactoring operation suggested in the answers. For the actions recommended, we focused mainly on accepted answers (i.e., an answer that was accepted by the person who asked the original question). For the purpose of the analysis of this question, we categorised *Actions* into one of the following four categories: (1) **Fix**: recommendations made to fix the code, e.g., remove the smell or anti-pattern by refactoring, (2) **Capture**: detect the smell/anti-pattern but no direct refactoring recommendations are given, (3) **Ignore**: recommend to ignore taking any action, e.g., assumes that the smell or anti-pattern has no bad side effect, and (4) **Explain**: if the question asks for information and the accepted answer provides only an explanation of the smell/anti-pattern, e.g., if the answer provides only an example of why something is considered a smell or anti-pattern.

A summary of results from this analysis is shown in Table 4. Note that, while analysing the data from CR, we were not able to extract the same information as we did with SO and SE. The nature of this forum and the structure of the questions being asked on it (being code-based (snippet) oriented) made it hard to analyse the data across multiple programming languages (as it would have required deep understanding of multiple programming languages syntax and semantics). Several of those questions appear to be more generic (not smells or anti-pattern focused) and the mention of the smells or anti-patterns was not the focus of the discussion in the answers/comments of the posts. We therefore decided to exclude CR data from the analysis conducted in RQ4 and RQ5.

Of the 100 questions from SO, we found five that were not relevant to the topics of smells or anti-patterns (i.e., it used terms like ‘smells’ or ‘anti-patterns’ but did not necessarily discuss anything related to these topics), and therefore we decided to exclude these. In total, we include 95 posts that we analysed manually.There were also a small set of posts (less than 5%) that we were not able to categorise. Such posts were classified as *General*.

In general, our results show that most questions asking for general opinions on smells or anti-patterns sought only an explanation rather than a solution (Fig. 7). We found that 59 of these questions fell into the *Explain* category. There were no clear actions recommended to deal with smells or anti-patterns, but rather an explanation of why something was considered a smell or anti-pattern and what the potential side effects were. There are at least 24 posts (25%) that provided fixing strategies to

**Table 4**
Summary results of the manual analysis of top posts in all forums.

| Site | #posts analysed | #posts with refactoring recommendation | Recommendation in the accept answer | | | |
|---|---|---|---|---|---|---|
| | | | Fix | Capture | Explain | Ignore |
| SO | 100 | 42 | 24 | 5 | 59 | 7 |
| SE | 50 | 14 | 11 | 0 | 10 | 18 |

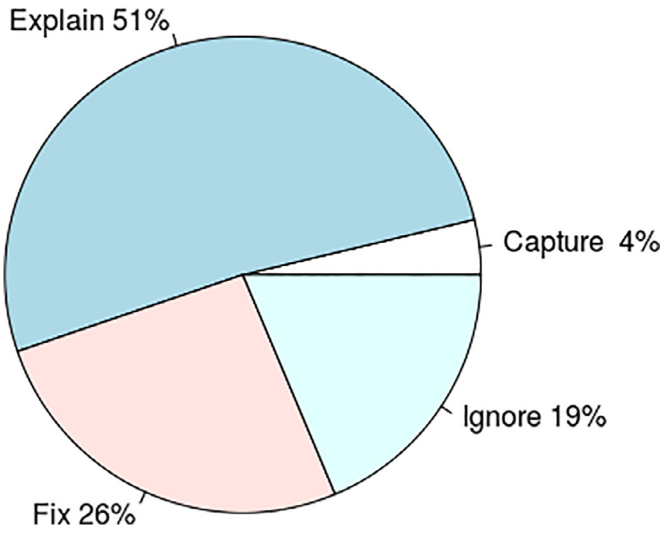


**Fig. 7.** Recommendation in the accepted answers in SO and SE.

deal with smells/anti-patterns. The majority of questions (56%) did not provide any refactoring recommendations. For those questions that were categorised as *Fix* or *Capture*, a number of refactoring operations were typically suggested in the answers. However, we observed that some of the refactoring recommendations (17%) were code-based (i.e., provide a coding example via a code snippet). In most cases, no specific refactorings were identified by name. Some answers (in 7 different questions) recommended the implementation of certain design patterns, such as *Dependency Injection* as a potential solution for certain smells/anti-patterns questions. In comparison, we found only a few posts that fell into the *Capture* (5 questions) and *Ignore* (7 questions) categories. For the questions from SE, we identified 11 posts as suggesting *Fix* operations for the identified smells or anti-patterns (with only two recommendation provided through code snippets). The majority of the posts fell into the *Ignore* (18) and *Explain* (10) categories.

Most answers provide explanations of a particular smell/anti-pattern, without providing specific refactoring recommendations. Only a small subset of posts have some **fixing recommendations** in their accepted answers. Even when such recommendations are provided, no specific names are given to such operations, as some of these recommendations are shown as code snippets.

**RQ5. What are the *challenges* (technical and otherwise) that developers may face in dealing with code smells and anti-patterns?** We manually analysed the top posts from each forum (similar set of posts as the ones the we analysed in RQ4). Of those 200 posts, we read through all questions, answers and comments and made notes about what was perceived as challenging in terms of those discussions on code smells and anti-patterns. As explained in the answer of RQ4, after analysing CR posts, we decided to exclude it from this analysis due to it's limitations. In total, we analysed 150 posts in this question (100 from SO and 50 from SE).

In general, we found that there was a lack of agreement between developers of what a code smell (anti-pattern) was and was not and also how harmful a smell (anti-pattern) was. In most discussions, developers provided different views on specific smells or anti-patterns. A general pattern found in the majority of the posts (in over 90% of the posts) was that most developers do not agree on the "authoritative definition" of a smell or an anti-pattern. When a user asks a question about 'smelly' code, the confirmation of whether the code actually contains a specific smell varied from one developer to another and arguments were usually presented to show the severity of this smell or anti-pattern. As a result, developers tended not to agree on the proposed refactoring recommendations in their answers. A mix of views are usually presented when the topics of code smells and anti-patterns are discussed. When we looked at the general discussions, the long term impact of code smells and anti-patterns was widely discussed while taking into consideration the language and the specific domain of the program. We noted the following challenges in the posts the we manually analysed in order to answer RQ5: 1) a consistent definition of what is a smell or anti-pattern and what is not, 2) consistent evidence of the negative impact of smells or anti-patterns, 3) the time/context/domain factors that indicate the best scenarios for implementing certain design patterns without having to deal with negative consequences (i.e., introducing anti-patterns).

We discuss the implications of those challenges on research and practice in Section 6.2.

The main challenge is in defining what is considered a smell or anti-pattern and what is an acceptable flaw in the code. A smell or anti-pattern can have many different meanings. Also, the real impact of smells and anti-patterns is highly debatable between developers. Factors such as the context and the domain of the program context are considered very important.

## 5. Observations from manual analysis

While we categorised and read through the posts, we made a number of observations based on our analyses of the questions examined in this study; these we present alongside some sample illustrative quotes (for reference we also include Post IDs with an abbreviation of the site)[11] A brief discussion of those observations are also included below.

**Observation 1**: *Developers use the terms "code smell" or "anti-pattern" rather liberally (i.e., everything that is not to their liking)*. There are questions that use the term "code smell", but do not specifically refer to an actual code smell. They rather use the term to express their disgust or repugnance about their code. Here is an example of such usage (SO #8921639):

"*...**my code smell is that they are not good enough.**Is there any better and faster way?"*

Similarly, the term (and also the tag) anti-pattern is used in many ways. It does not always refer to design issues, but rather to any 'pattern' issues in the source code, in the configuration files or even in the tools. Some developers discuss the usefulness of some of the common design patterns. In contrast, some of the design patterns (or their implementations) have been suggested as the possible cause of anti-patterns. For example (SO #31516310):

[11] To access the question on the web, add the given PostId number to the question URL as follows: [site].com/questions/[Post Id number]. Site names are *stackoverflow, softwareengineering.stackexchange* and *codereview.stackexchange*.

"*...before i implemented this, i went out (sic) a did a bit of research about whether it was a good idea or not, and all the information i could find (sic) we statements calling it an anti-pattern but without explaining why. Why is a generic repository considered an anti-pattern?*"

**Observation 2**: *Developers ask peers to review their code and name the potential smell or anti-pattern that they suspect that their code contain.* In those questions, developers would usually provide an example of what they thought was a smell or an anti-pattern and ask other users to confirm whether it was indeed present, and then ask for the specific name of it if known. Such posts would usually start with a question like "can you name this smell or anti-pattern". For example, the following questions discuss a case where a user asks for the name of a specific anti-pattern that they have in their code (SO #41774383):

"*Does the following (anti?)pattern have a name? every function returns an instance of a result class (or structure, or associative array) that has at least these two public variables: success.result....if success is false an error variable may hold information (exception/stack trace/debug) about the error a message variable holding an user friendly message might be defined if error is not empty only the result of the operation is encapsulated... I've seen this used both in.Net and PHP (on server and in a script invoked by JavaScript/Ajax requests). Is it best practice? Is this an anti-pattern?*

**Observation 3**: *Developers more often ask whether something is a smell or an anti-pattern or not, rather than referring to a particular one.* Much of the code designated as "smelly" or a design that is called an "anti-pattern" are presented generally, without mentioning any specific names for the type of code smell or anti-pattern. For example, in one of the posts (SO #7190450) a user wrote the following to declare that his/her code might contain a smell:

*"...I spent already too much time on this problem and my current approach uses a convenience method to add the: username parameter. Nevertheless, I think using this method all over the place just to add an entry stinks (bad code smell)....".*

**Observation 4**: *Developers ask about the trade-offs of the usage of known design patterns.* The *overuse* of certain design patterns has been questioned. We observed that such posts discuss how the 'overuse' of certain design patterns can potentially turn them into anti-patterns. For example, in post (CR #47865) a user asks the following question about the *Generic repository* pattern:

"*...**I am having a service for each of my entity object, I see no reason why I shouldn't use a generic repository as creating a repository for each entity object as-well... However I have heard many people say the generic repository is an anti-pattern so I'm not sure if this is the right way***"

Here is also another example of how the implementation of one design principle, i.e., Single Responsibility Principle (SRP), can lead to an anti-pattern, i.e., Anemic Domain Model (SO #1399027):

"*I'm trying to improve my understanding of how to apply SRP properly. It seems to me that SRP is in opposition to adding business modelling behaviour that shares the same context to one object, because the object inevitably ends up either doing more than one related thing, or doing one thing but knowing multiple business rules that change the shape of its outputs. If that is so, then **it feels like the end result is an Anemic Domain Model ... Yet the Anemic Domain Model is an anti-pattern.**Can these two ideas coexist?*"

The *Dependency Injection, Singleton* and *Service Locator* patterns appear as the most discussed topics in the *anti-pattern* context, including their use in different platforms and scenarios. Most of the discussions are centered on whether they are design patterns or anti-patterns.In many cases, developers provide some scenarios (mostly *via* code snippets) where they discuss the negative impact of using and implementing these patterns in their application. Here is an example of a *Service Locator* question (SO #22795459):

"*Recently I've read Mark Seemann's article about Service Locator anti-pattern...two main reasons why ServiceLocator is anti-pattern: API usage issue ... (and) Maintenance issue... I'm not trying to defend Service Locator approach. But this misunderstanding make me think that I'm losing something very important. Could somebody dispel my doubts?*"

In other questions, some developers wonder whether anti-patterns or code smells truly have an impact on their code. In particular, we found questions on why developers should not concern themselves with the presence of code smells or anti-patterns at all in certain scenarios.We also noticed questions which doubt the practicality of certain known design patterns such as *Singleton* and *Service Locator*. An example of such a question is shown here (SO #568365):

"*Am I the only one that sometimes take the seemingly easy, but wrong, way out of certain design situations? I'll admit I've made my share of questionable Singleton objects. Besides that, I've been known to make a God object or two to make things seem easier. **Do you ever use an anti-pattern even though you know you shouldn't?***"

On the other hand, developers also ask about when using a particular anti-pattern or smell is actually not bad, or even useful in some scenarios. For example, here is a user who asked when Primitive Obsession is actually not a bad smell (SE #365017)

"*...I have read plenty of articles recently that describe Primitive Obsession as a code smell.... There are plenty of articles out there that describe when it is a code smell.... However, what about the following objects (code snippet) is this a case of over engineering as there is not a lot of validation. Is there a rule that describes when and when not to remove primitive obsession or should you always do it if possible.*".

In fact, some developers argue that having code smells or anti-patterns is not always bad practice.Smells and anti-patterns are, in some cases, seen as an acceptable trade-off. For example, the following is a comment to a question asked whether implementing a *factory pattern* will help in eliminating a *nested if-else statement* (SE #357932):

"*I always interpret "code smell" as snippets of implementation that might point to bigger problem. I know some people think all code smells are bad. No, sometimes it is a delicious cheese!*".

**Observation 5**: *Developers ask general questions about code smells and anti-patterns.* Some developers ask general or open questions about the *notion* of smells and anti-patterns. These questions could be classified as *knowledge enquiry* as they are not discussing a specific scenario or programming language. An example is evident in the following two posts (SO #851412), (SE #209497) respectively:

"*I'm looking to see how other programmers find anti-patterns, <ddq>'code-smells'</ddq>... what things start setting you off when you're looking at code that tells you, something has gone wrong here?*"

"*...What I am looking for are anti-patterns that appear to be good solutions at first glance, but tend to lead to issues further down the line. Do you know any good source of such anti-patterns?....*

**Observation 6**: *Developers ask about general tooling support for code smells and anti-patterns detection and removal.* Several posts discuss code smells or anti-patterns in the context of tools or IDEs. Developers may ask questions to look for help with tools to detect code smells or anti-patterns, or look for specific configurations for those tools.

An example is shown here (SO #2710503):

"***Does anyone know of an updated list of refactoring support for different IDEs? How many of Fowler's refactorings have tool support in popular IDEs?****And does any IDE use code smells to any greater extent? I guess one would have to use addons for some IDEs, so even if I* *did find an updated list of refactoring support for say Eclipse, that would probably not be representative*".

We also observed that developers frequently provided answers that contained refactoring solutions to the code smells and anti-patterns that were submitted in the questions. However, a large proportion of such refactoring solutions ( ∼ 45%) were suggested through code examples (snippets). When available, developers *mostly* did not provide *specific names* for the refactoring implementations that they provided as solutions.

**Observation 7**: Many posts discuss smells or anti-patterns that are language- (or even library-)specific. Usually, users provide a scenario or an example from a specific programming language about a smell or an anti-pattern, and then ask for confirmation of their initial assessment. The following is an example from this category which discusses why the use of `Thread.Sleep()` is a bad design in Java (SO #7813662):

"*...the point came up that use of the `Thread.Sleep()`method is a code smell. I have a hard time believing there is no use of this method that doesn't indicate that you're doing something wrong....why would Thread.Sleep be such a terrible line of code to use in any circumstance that it would smell so?*".

**Observation 8**: *Developers ask code or design-specific questions about particular problems in their software which they believe may be code smells or anti-patterns*. In such questions, users may use real-world examples supported with visual design and code snippets to explain their questions. The goal is to get a community assessment of the code/design and whether it contains smells or anti-patterns. In many cases, an explanation of the potential smell is given in the answer with the common name, impact and possible solutions (if any). An example is shown in the following post (SE #152563):

"*I occasionally write code like this when I want to replace small parts of an existing implementation: (code snippet).... Is this a pattern or an anti pattern? If so, does it have a name and are there any well known pros and cons to this approach?*".

The accepted answer for this question (SE #/152568) suggested the name of this pattern as a 'decorator pattern'.

**Observation 9**: *The potential side-effects of removing some smells or anti-patterns is also widely discussed between developers.* In some cases, refactoring code to remove specific smells can potentially lead to the introduction of other unwanted smells. In such posts, the trade-offs and side-effects of applying refactoring to remove certain smells and anti-patterns from the code are considered. For example, the following questions discuss a case where refactoring a *God Class* smell led to the introduction of another code smell i.e., *Feature Envy*, in the code (SE #359642):

"*I'm trying to refactor a 2.5KLOC god-class (with about 68 data members and 62 member functions) that performs a wide variety of text formatting and layout operations. After a careful analysis....I've partitioned the data members and members functions into 9 classes. The problem now is that most of these smaller classes are repeatedly reaching for data held by other classes (classic feature envy). What sort of refactoring would be advisable in this situation?*".

**Observation 10**: *Many smells and anti-patterns posts are tool-specific, and users would ask some specific configuration or customisation questions about those tools*. There were more than 80 posts discussing the use of tools (configuration, optimisation, etc) in the context of smells/anti-patterns. However, none of those posts discussed the quality of those tools, the way they measured smells or anti-patterns or the reliability of the reported results. Among known tools, there were many questions on SonarQube[12] (60 posts). Other known tools that are widely used in academic studies such as PMD[13] and SpotBugs[14] were rarely discussed. Those tools are known among developers to analyse source code and are easily integrated in modern IDEs and automation tools. Here is an example of question on the general rules of SonarQuabe (SE #349427):

"*SonarQube is a software product which runs various coding style rules and other metrics similar to FxCop or Re-sharper. It defines breaking the style rules as: "MAINTAINABILITY ISSUE"....Issues associated with maintainability are named "code smells" in our products." The code might well meet all the style rules, have no duplication etc and be fine on its own. Its only considered technical debt because it doesn't fit the pattern of the other code in the project. I would think that this kind of difference would be hard to pick up with code analysis rule sets and it seems disingenuous to call style rule violations "technical debt". Am I wrong or right? Do some objective rules or some class of rules make a good definition of technical debt or at least a type of technical debt?*"

In the following section we discuss our results and the implications of our findings for researchers and practitioners.

## 6. Discussion and implications

### 6.1. Dissuasion

As shown in Section 4, the results for RQ1 shows that the smells of *God Class, Duplicated Code, Spaghetti Code* were by far the most frequently discussed on the three forums. Those three smells are either size-related or complexity-related and we expected this group of smells to be more popular than other form of smells. Those smells might be easier to identify and detect in code, which thus leads to greater tools availability, precision and support to capture those smells. However, such smells and anti-patterns can be inherently parameterised by a certain threshold. This makes them ambiguous for developers to use, in particular when applied to a different language (different to the one used when they were first introduced). They are "statistical" as opposed to structural anti-patterns like LSP violations, cyclic dependencies etc, that are inherently less ambiguous.

We note that the number of posts that used the same terms in our search query is relatively low compared to the size of the dataset. It could be that either developers do not discuss these smells and anti-patterns as widely as we, in academic research, anticipated or that they use other terms to refer to those smells/anti-patterns. Recent surveys on code smells [41,55] revealed that there is academic bias towards a certain, small number of smells. We will discuss this issue further in Section 7. However, those results provide an overview of the popularity of the selected smells and anti-patterns terms that we included in our search and do not provide a complete popularity assessment of all smells and anti-patterns. Again, the results could be highly influenced by the choice of smells and anti-patterns that we included in our search (although we used a list of smells and anti-patterns that were extracted from a popular list of empirical studies and represent what have been widely studied in this domain).

The results of the smells posts in respect of the programming language used (RQ2) are not surprising as this features the most popular programming languages that are currently used. The results are in line with the Popularity of Programming Language Index (PYPL)[15], in that 8 of the programming languages that we found in our analysis are represented in the top 10 languages in the PYPL list, although in a slightly different order (i.e., top 5 languages in the PYPL list are Java, Python, PHP, C# and JavaScript). This is also in line with the language popularity statistics provided by SO[16], which postulates that JavaScript, C#,

[12] https://www.sonarqube.org [Accessed: 23 May 2019].

[13] https://pmd.github.io [Accessed: 11 May 2019].

[14] formally FindBugshttps://spotbugs.github.io [Accessed: 7 Feburary 2020].

[15] http://pypl.github.io/PYPL.html [Accessed: 19 May 2019].

[16] https://insights.stackoverflow.com/survey [Accessed: 19 May 2019].

Java, C++, PHP and Python are among the top 10 languages most commonly used by developers. It is also interesting to find that 16% of the questions are language-agnostic (i.e., general questions on code smells or anti-pattern that do not associate with a specific programming language). We expected the majority of smells and anti-patterns studied here to be relevant to wider programming paradigms (rather than a language) and can be easily extended from one language to another. We noticed that most questions asked on SE site follow that pattern (especially for OO smells and anti-patterns). While there are discussion on multi-languages, generally, there is a lack of smell-detection tool support for languages other than Java and Javascript. Most of the known smells tools are Java-focused, with only a few tools providing multi-language support [16]. Greater support to cover languages such as C# and Python is needed.

While there are a large number of posts that discuss language-agnostic smells and anti-patterns, there are also language specific posts. Interestingly, those questions discuss specific language features or specific practices perceived as smells or anti-patterns, but do not follow the classical definition of one (there are simply pattern/styles/implementation that are considered bad). Some developers would consider specific language features as a potential anti-pattern. An example of such posts is a question discussing the purpose of `exit(-1)` in C (and Java) and why it may be considered as an anti-pattern[17] This adds to the question of what can be considered a code smell or an anti-pattern and what is not. It is evident that the terms are used rather liberally (see Observation 1 in Section 5).

With regards to their underlying meanings, both terms can be seen as used interchangeably (i.e., in the same text to refer to code smells as anti-patterns, or *vice versa*). There is also a general lack of consistency in the definitions used in posts tagged as “code smell” and “anti-pattern’. Although the two terms are different [9,18], they frequently get mixed-up. This can also be observed in academic studies, as the two terms are widely used interchangeably to refer to the same problem. Indeed, some code smells and anti-patterns can refer to the same thing (at both design and implementation levels). Also, some anti-patterns may lead to the introduction of particular code smells. However, in most cases the two refer to two different (but sometimes related) issues and should not be mixed-up.

While not particularly smell-like, the violations of certain design principles such as LSP and DRY are widely described as anti-patterns; even the downsides of some well-known design patterns (such as Service Locator and Dependency Injunction) have been suggested as potential causes of anti-patterns. Results indicated that developers assumed that some of these design patterns can have a negative impact, which is considered bad practice. We think this is more of a reference to *flawed* or *incorrect* implementations of certain design patterns, which might lead developers to believe that a particular pattern is bad in general. This is a typical context issue, where the case of patterns and anti-patterns usage or implementation depends largely on many factors such as the context of the program, the development environment and developers” experience. Gamma et al. [19] discussed *consequences* (i.e., trade-offs) as one of four *essential elements* that should be considered when using a pattern. Those consequences are critical for evaluating design alternatives and for understanding the cost-benefits of applying a design patterns. As per the findings of RQ4 reported in Fig. 7, the refactoring recommendations provided are rather vague, as they heavily depend on the context of the question thus making them difficult to generalise.

There is also a lack of agreement between developers of what a code smell (or anti-pattern) was or was not and how harmful either were. In most discussions, developers provide different views on specific smells or anti-patterns. They generally do not agree on the “common definitions” of smells or anti-patterns. Again, we think this is another context issue. As discussed by Taibi et al., [52], developers may agree that a smell *X* is bad when they are provided with the definition of that smell, but their view may shift when given a piece of code that contains smell *X* and are asked to identify the smell from the code. Extending those definitions to other domains and applications is another challenge for research and practice. Even when developers agree on the severity of a specific smell or anti-pattern, there seems to be disagreement on which particular solution should be implemented to remove either. Developers tend not to agree on the proposed refactoring recommendations in their answers, as well as the impact of code smells and anti-patterns. There are no clear acceptable (universal) refactoring recommendations that developers agree upon to deal with different smells. We believe that definitions of smells and anti-patterns should also consider the language and the specific domain of the program. This is an important factor in deciding when developers should be worried about the specific smells and anti-patterns. With better definitions of those smells and anti-patterns we, as a community, might get closer to providing relevant solutions applicable for developers and tool vendors.

In summary, we found that smells/anti-patterns maybe perceived differently between developers and their impact depends largely on context. Even design-patterns and best practices can be considered harmful in some cases and we should not therefore assume that design patterns are always positive for a program. We noted examples when specific design-patterns were considered as anti-patterns or could lead to an anti-pattern (and *vice versa*). While previous studies have indicated that developers generally perceive code smells negatively [35,52,59], our results show that this is not always the case for all smells.

RQ1 revealed that God Class, Duplicated Code and Spaghetti Code were the most common three smells. The value of any empirical study is that it adds to a body of knowledge in a specific area; in this case, it is code smells and anti-patterns. Most studies in the past however, have used the code base as the empirical means of drawing conclusions about the regularity and popularity of these smells. They could only speculate on some of the background motivation of developers. Our work is novel in that it focuses on the views of developers as a basis of the analysis. Future studies could therefore use our results to explore new avenues in the human insights of code smells/anti-patterns which a study of the code base alone would not permit. The work thus recognised the gap in our knowledge from previous studies and informs new research avenues that complement past studies.

One benefit of the work described in this paper is that it highlights, on a wide scale, the views and thoughts about smells/ anti-patterns by developers. Too often, tools are developed which fail to consider these developer views; the net result is often a product of limited use and impact. At the very least, the results in this paper contribute to a greater understanding of developer concerns and, from that, how we might address those concerns through tools. A further benefit of the research is through the set of ten observations we make in Section 5; these could easily be used to inform the human design elements of tools and, in particular, the development of recommendation tools to cover the inadequacies in current tools. Finally, personalising tools tailored to a specific user can only be achieved if we understand the behaviour of developers on a broad basis.

Our findings can guide tools builders as follows: smell and anti-patterns detection is basically static analysis, and its value as with any static analysis heavily depends on its precision [14,44], i.e., controlling noise (false positives). This noise consists of (1) the false positives detected for a particular anti-patterns or smell, but also (2) of all results (false positives and true positives) of smells and anti-patterns not considered relevant by developers. Our study provides insights about what the important smells and anti-patterns are, and, therefore, which ones could be safely omitted (from default analysis settings).

The fact that most smells and anti-patterns are discussed in the context of C#, Java and JavaScript is of interest as those languages just

[17] https://softwareengineering.stackexchange.com/questions/314563 [Accessed: 3 May 2019].

**Table 5**
Tools support for smells and languages.

| Tool | Smells/Anti-patterns | Language |
|---|---|---|
| PMD | Data Class, God Class and Duplicated Code (C&P code) | Java (C&P support for other languages) |
| JDeodorent | God Class, Duplicated Code, Feature Envy Switch Statements and Long Method | Java |
| DECOR | Data Class, God Class, Duplicated Code, Spaghetti Code Switch Statement, Large Class and Long Method | Java |
| SonarQube | Spaghetti code (nested structures), Switch Statement Large Class and Long Method | Multiple main-stream languages |
| inFusion | Data Class, God Class, Duplicated Code and Feature Envy | Java, C, and C+ |
| NDepend | Size-based smells (Large Class and Long Method) | C# |
| SpotBugs | None | Java |
| Designite/DesigniteJava | Size-based smells, God Class, Duplicated Code and Feature Envy | C# and Java |

started to emerge[18] when many smells and anti-patterns were defined [18,19]. So they were defined in the context of other languages and other languages uses - especially in the late 1990s – programming consisted mainly of desktop applications, often more monolithic, with (also monolithic) web applications just emerging. It is therefore not surprising that developers find it difficult to adapt those concepts to modern environments; new languages have evolved into multi-paradigm languages (e.g., Scala, Swift and Kotlin), with finer-grained modularity supported by automated dependency resolution and package ecosystems, simultaneous use of multiple frameworks and dynamic language features; this has lead to smaller applications which are loosely coupled through services. The results of this study reflect attempts to adapt the potentially "outdated" definitions of smells and anti-patterns to these contexts, inevitably leading to inconsistencies. Therefore, tool designers can potentially use these results to build tools that specifically target languages that lack support for anti-pattern and smell detection. In addition, this result provides the opportunity for specific *features* of those languages to be targeted by tools, depending on which elements of those languages share most concerns of developers and the types of smell and anti-pattern that are specific to those languages.

To further investigate the gap between the most actively discussed smells in these forums and the available smells-detection tools, we compared the state of seven well-known tools and their ability in capturing the list of smells that we identified in RQ1. The tools are PMD[19], JDeodorent[20], DECOR[21], SonarQube[22], inFusion\iPlasma[23], NDepend[24], Designite[25] and SpotBugs[26] We found that the tools (Table 5) cover most of the common smell patterns that are shown to be the most popular between developers. Other popular smells such as *Feature Envy* are not widely supported by these tools (with only three tools able to detect this smell). Most of these tools are Java-focused with three tools being able to detect smells in other popular languages (besides Java) as identified in RQ2. As there are a noticeably higher number of smell detection tools for Java, the same should be extended to cover other popular languages such as C#, JavaScript, C++ and Python. We did not find many tools, other than SonarQube, able to detect smells in other popular languages such as Python, Ruby and PHP. Modern multi-paradigm languages that are growing in popularity (such as Swift and Kotlin) are also lacking support in terms of smell detection tools. Code designers can potentially help by providing *smells detection* and *removal support* for such languages.

[18] https://www.tiobe.com/tiobe-index [Accessed: 7 November 2019].
[19] https://pmd.github.io [Accessed: 5 November 2019].
[20] https://github.com/tsantalis/JDeodorant [Accessed: 5 November 2019].
[21] http://www.ptidej.net/research/designsmells [Accessed: 5 November 2019].
[22] https://www.sonarqube.org [Accessed: 5 November 2019].
[23] http://loose.upt.ro/iplasma [Accessed: 5 November 2019].
[24] https://www.ndepend.com [Accessed: 5 November 2019].
[25] https://www.designite-tools.com/ [Accessed: 8 February 2020].
[26] https://spotbugs.github.io/ [Accessed: 5 November 2019].

### 6.2. Implications of the results

The research in this paper has a number of implications for both research and practice.

- *Evidence of how harmful developers think smells and anti-patterns are is not clear enough*; the standard texts on both make no judgment on which are more likely to cause problems in a system and, as far as we know, there is no other evidence to support actual developer opinion on smells and anti-patterns (we could not find strong evidence in Stack Exchange sites either). Perhaps the academic community perceive a problem caused by smells or anti-patterns that simply does not exist in industry, or at least is not considered a pressing problem. If that is the case, then more research is needed to assess *the true impact* of code smells and anti-patterns and more collaboration with industry is urgently needed to achieve this goal.
- *The use of academic tools for detecting code smells in academic empirical studies is mostly based on open-source programs*, but perhaps as well as reporting these results, academic studies should focus on the usefulness and viability of these tools in industrial contexts. In those respects, our *Observations #10* hints that Stack Exchange sites can be used as an arena to expose, test and tailor methodologies/tools coming from academia. At the very least, we should be exploring ways in which tools/methods can be tailored to what industry need specifically, rather than a "one size fits all" approach which is often not an effective use of developers' time.
- *The "badness" of a smell is very much in the eye of the beholder*. A consistent problem with previous studies of code smells and anti-patterns is that the threshold of *what is* and *what is not* a smell or anti-pattern is largely subjective in nature. Only a developer can really judge the extent of "smelliness" in a piece of code and decide to do something about it. This judgment may also be *application domain* specific. To attempt any generalisations about the impact of a code smell is clearly flawed if that is the case. For example, telecoms systems might create different smells to those in the banking domain, which in turn might be different to those of the computer games domain. On the other hand, most industrial systems recognise the need for key or hub classes; they are central (God) and large classes because they need to be large and nothing to do with code smells.

### 6.3. Challenges for research and practice

There are a number of challenges that may face the community in regards to code smells and anti-patterns. There appears to be considerable confusion about what an anti-pattern is, *vis-a-vis* a code smell. Our results reveal that the meaning associated with concepts like smell and anti-pattern differs significantly between users and their discussions. This is not surprising if one considers the meaning of those concepts defined by how they are used in context. These contexts (time, experience of participants, programming language, etc.) vary widely between discussions. While this might be a valid observation, it is not satisfying from an engineering point of view. Engineering requires normative definitions – for teams to communicate effectively and vendors to create tools that solve well-understood problems. Such normative definitions exist, in seminal work on the topic by Fowler and others [18,27]. They seem to fail, however, in the sense that there is little shared understanding (and thus, practicality) of those terms. There are two possible reasons for this: (1) *these normative definitions are not suitable* and (2) *users do not have sufficient knowledge about them*. We think that there might be some truth in both views. Many of the original smell and anti-pattern definitions might be *outdated*. For instance, users complain about the low precision of smell detection tools which might indicate that the original smell definitions are too coarse and need to be refined in order to reduce false positives [17]. The number of such issues is likely to have increased, since many smells were defined before new technologies were adopted (e.g., domain specific languages, code generation). This might have changed the impact those smells have (for instance, by creating false positives or by considering smells and anti-patterns as acceptable trade-offs).

On the other hand, the meaning of code smells and anti-patterns are not static. Many definitions refer to specific languages and tools and those change over time. When smells and anti-patterns were first defined, they were introduced "bottom up" as an abstraction from what experts had observed as problems in many projects (think about it as an equivalence class in the mathematical sense). Some developers may try to shortcut the process by approaching this "top down" – try to understand the smell or anti-pattern first without being aware of the intricacies of actual instances/occurrences. This could be a result of the tools support that are available for developers, which detect smells and anti-patterns and then developers pick those up without any further investigations. This can result in a lack of depth in understanding the cause of the smell or anti-pattern, as we observed in some of the discussions in Stack Exchange sites. Equally, developers discussing smell and anti-pattern topics have also changed over time. While smells and (design and anti-) patterns were discussed in eclectic circles in the late 1990s and early 2000s, platforms such as SO and SE have commoditised this knowledge and brought people into the discussion who do not have the 'same' level of experience participants in those discussions would have had 15 years earlier. Our results highlight the need for updated, canonical definitions which service this purpose. This could be done by revisiting and refining some smell and anti-pattern definitions in the context of current best practice, by creating updates like *"Smells 2.0"*. However, it is unclear what an acceptable process to update those definitions would look like and whether the community would accept this. Several community-driven efforts to catalog smells and anti-patterns already exist[27][28], but such catalogs should be open for discussion and frequently updated.

The confusion between smells, patterns and anti-patterns opens up the discussion on what can be really considered as a code smell or an anti-pattern, and whether we need to consider any violation of certain design principles or best-practices as anti-patterns. While the three terms (code smells, anti-patterns and design patterns) are fundamentally distinct, the confusion between those terms probably comes from the fact that developers may attribute any problem in the code as a "code smell". The underlying problem is that we use those concepts for three different purposes: 1) to improve quality, 2) to power tools, and 3) to facilitate communication between developers. Those purposes might not always align well. Therefore, it is important for both industry and academia to establish the challenges of determining a proper definition of code smells and anti-patterns, and then tackle those challenges. Our results suggest that developers need more context-sensitive research results that can help them make "trade offs" between introducing/removing anti-patterns and code smells. A deeper investigation into these issues is an interesting topic for future research.

In summary, we believe that future research needs to address the following:

(1) the need for more context-sensitive results (and potentially tools) that can support developers in making trade offs between removing and keeping smells/patterns/anti-patterns,
(2) academic studies should focus on the usefulness and viability of smells detection tools in industry (with less false-positive rate), and
(3) more research is needed to understand the real (longitudinal) impact of smells and anti-patterns in real life projects.

## 7. Threats to validity

***Search process:*** Determining whether a post is related to code smell or anti-pattern topics was based on selected keywords used either in the question title or body. However, string search approaches might suffer from misspelling, or there is a possibility that developers may use keywords other than those that we used in our mining query, which would introduce a validity threat. To reduce this threat, we first combined a list of code smell and anti-pattern terms that developers and researchers use, as reported in several previous empirical studies. These studies are widely known and highly cited and include key works in this area such as Fowler's text on code smells and refactoring. When possible, we also included alternative terms that are used to refer to code smells and anti-patterns. For example, for the *Duplicated Code* smell we used other alternative terms such as *Code Clone* and *Code Duplication*.

We also checked the potential impact of misspelling in Section 3.2. We reported results from a study on SE which showed that the possibility of missing any relevant posts related to the two key terms ('smell' and 'antipattern') due to misspelling was very low.

As explained in Section 3, searching exclusively through *Tags* can be ineffective, as Tags can be less informative. Therefore, we decided to mine the body of the questions. In this way, we sought to minimize the risk of missing questions that used incorrect or irrelevant *Tags*.

***Popularity metric:*** We designed a new metric for measuring the popularity of a question that takes into account posts' *Score* and *View Count* values. However, this metric could be biased in the sense that it does not take into account the time-frame of the questions. As our analysis does not distinguish between questions based on time, new questions might not have high *View Count* or *Score* and therefore may have low popularity values; they will automatically fail to qualify for the manual analysis set. We have not yet arrived at a suitable treatment for this threat but some form of time-based normalization might be feasible.

***Bias introduced by manual analysis:*** Manually performed analysis could introduce bias due to multiple interpretations and/or oversight. We are aware that human interpretation introduces bias, but we attempt to account for it via cross-validation involving multiple evaluators and by cross-checking the results from the classification stage, by involving three of the authors and two external reviewers (Section 3).

***Dealing with closed posts:*** Given the nature of Stack Exchange sites, there is a possibility that some of the questions asked on the forum might be closed by the moderators for one reason or another. The nature of questions about code smells and anti-pattern can be very broad and opinion-based and those sort of questions are not favoured by SO or CR moderators[29] Therefore, some interesting questions might be the subject of closure by the moderators. However, we argue that some interesting programming related discussions can be established because they are directly or indirectly related to code smells or anti-patterns and such discussions should be investigated to see if users generally share the same view about the impact of smells and anti-patterns. In general, code smells and anti-patterns should not be seen as a separate topic from other common programming topics. Consequently, we

[27] https://sourcemaking.com/antipatterns [Accessed: 23 May 2019].

[28] http://wiki.c2.com/?CodeSmell [Accessed: 23 May 2019].

[29] there is some tolerance for such questions on SE.

checked how many closed questions were contained in our dataset to ensure that we are not dealing with a large set of irrelevant posts. Of the final list of questions included in the analysis, only 115 questions were marked as closed for various reasons, representing less than 4%, 18% and 2.6% of the total of number of questions in SO, SE and CR, respectively. For those closed questions, the average time between the time the question was posted and the time it was closed was 11, 10 and 15 months in SO, SE and CR, respectively. Even though those questions were closed, they still had a relatively high *Score* (as high as 156 in one case) and *view count* (as high as 90304). We thus acknowledge that there is only a small set of questions in our dataset marked as closed and even though those questions are closed, they have stayed active for a while and are actively viewed by users indicating their relevance to the issues under consideration.

## 8. Conclusions and future work

This study provides insights into what developers think of (and how they feel about) code smells and anti-patterns. The findings of this study should help to guide future research directions in the area of code smells and anti-patterns, by learning about practitioners' perceptions, experiences, decisions and actions around code smells and anti-patterns. The results indicate that it is not entirely clear how developers define code smells or anti-patterns and exactly what are classified as smells or anti-patterns is highly subjective. The findings also show that most discussions on code smells and anti-patterns are centered around popular programming languages, with C#, JavaScript and Java dominating the landscape. We also found that *God Class, Duplicated Code* and *Spaghetti Code* are discussed more frequently than other smells and anti-patterns. In terms of popularity, *Middle Man* and *Long Method* are seen to be relatively more popular in discussions than other smells.Also, we observed that developers widely discuss the trade-offs of the usage of specific design patterns, and, in some cases, some well-known design patterns have been suggested as potential anti-patterns that should be avoided; the *Service Locater, Singleton* and *Dependency Injection* patterns are the most discussed of these topics. Similarly, the violations of certain design principles (especially the SOLID principles) such as the LSP, SRP and DRY principles are widely described as potential anti-patterns that should be avoided.We also observed that there are discussions on why well-known anti-patterns are *not* considered very harmful and should not in fact be flagged as *anti-patterns*.

In general, our results show that there is a gap between what researchers and developers discuss in terms of code smells and anti-patterns. Developers generally have negative feelings toward code smells and anti-patterns, but the actual impact of these smells or anti-patterns is open to question. However, there are cases where users explain that some smells and anti-patterns are not always *bad* - some posts suggest that they are not necessarily always worrying, and their potential (negative) impact is seen as somewhat limited.

We see a number of avenues for future work in this area. As suggested by a number of observations from this study, we plan to develop more precise, context-sensitive smells and anti-pattern detection tools (leading to fewer false-positive results). Machine learning techniques may be useful for identifying context for such tools. Another avenue of research is to work towards providing a unified catalog of smells or anti-patterns (similar to the IEEE Standard Glossary of Software Engineering and SWEBOK), which can help to bridge the gap between tool vendors and developers. While this study provides a view on what developers think of the impact of code smells and anti-patterns in practice, more investigation into *why* some design patterns are seen as potential anti-patterns is strongly needed.We also aim to study wider aspects of developer knowledge and opinion on code smells and anti-pattern. Our next target is to study publicly available projects repositories, code reviews and issue tracking systems.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## CRediT authorship contribution statement

**Amjed Tahir:** Conceptualization, Methodology, Data curation, Formal analysis, Writing - original draft, Visualization, Validation, Writing - review & editing, Project administration. **Jens Dietrich:** Conceptualization, Validation, Writing - review & editing. **Steve Counsell:** Conceptualization, Validation, Writing - review & editing. **Sherlock Licorish:** Conceptualization, Methodology, Validation. **Aiko Yamashita:** Conceptualization, Methodology, Validation.

## Acknowledgements

We would like to thank Shawn Rasheed and Li Sui for their help with the data filtration and manual classification of posts. We would like also to thank the anonymous reviewers for the constructive feedback on an earlier version of this manuscript, which were helpful in improving the quality of this paper.

## References

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, 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, 2011.

[2] S. Ahmed, M. Bagherzadeh, What do concurrency developers ask about?: A large-scale study using stack overflow, in: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2018, doi:10.1145/3239235.3239524.

[3] K. Alkharabsheh, Y. Crespo, E. Manso, J. Taboada, Software design smell detection: a systematic mapping study, Software Quality Journal (2019), doi:10.1007/s11219-018-9424-8.

[4] M. Allamanis, C. Sutton, Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code, in: 2013 10th Working Conference on Mining Software Repositories, 2013, doi:10.1109/MSR.2013.6624004.

[5] K. Bajaj, K. Pattabiraman, A. Mesbah, Mining Questions Asked by Web Developers, in: 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, 2014, doi:10.1145/2597073.2597083.

[6] A. Barua, S.W. Thomas, A.E. Hassan, What are developers talking about? an analysis of topics and trends in stack overflow, Empirical Software Engineering 19 (3) (2014), doi:10.1007/s10664-012-9231-y.

[7] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, D. Binkley, Are test smells really harmful? an empirical study 20 (4) (2014), doi:10.1007/s10664-014-9313-0.

[8] V. Braun, V. Clarke, Using thematic analysis in psychology, Qual Res Psychol 3 (2) (2006) 77–101.

[9] W.J. Brown, R.C. Malveau, H. McCormick III, T.J. Mowbray, Antipatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc, 1998.

[10] E. Choi, N. Yoshida, R.G. Kula, K. Inoue, What do practitioners ask about code clone? a preliminary investigation of stack overflow, 2015 IEEE 9th International Workshop on Software Clones, 2015, doi:10.1109/IWSC.2015.7069890.

[11] M. del Pilar Salas-Zárate, G. Alor-Hernández, F. Valencia-García, L. Rodríguez-Mazahua, A. Rodríguez-González, J.L.L. Cuadrado, Analyzing best practices on web development frameworks: the lift approach, Sci Comput Program 102 (2015).

[12] J. Dietrich, M. Luczak-Roesch, E. Dalefield, Man vs machine: a study into language identification of stack overflow code snippets, in: Proceedings of the 16th International Conference on Mining Software Repositories, IEEE, 2019.

[13] J. Dietrich, C. McCartin, E. Tempero, S. Shah, On the existence of high-impact refactoring opportunities in programs, in: Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122, 2012.

[14] D. Distefano, M. Fähndrich, D. Logozzo, P. O'Hearn, Scaling static analyses at facebook, Commun. ACM 62 (8) (2019), doi:10.1145/3338112.

[15] W. Dou, S.-C. Cheung, J. Wei, Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 848–858.

[16] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: an experimental assessment., Journal of Object Technology 11 (2) (2012).

[17] F.A. Fontana, J. Dietrich, B. Walter, A. Yamashita, M. Zanoni, Anti-pattern and code smell false positives: Preliminary conceptualisation and classification, in: 23rd International Conference on Software Analysis, Evolution, and Reengineering, 2016-Janua, 2016, doi:10.1109/SANER.2016.84.

[18] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Elements of reusable object-oriented software, Pearson, 1995.

[20] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009.

[21] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Trans. Software Eng. Method. 23 (4) (2014), doi:10.1145/2629648.

[22] F. Hermans, M. Pinzger, A. van Deursen, Detecting and refactoring code smells in spreadsheet formulas, Empirical Software Engineering (2014), doi:10.1007/s10664-013-9296-2.

[23] F. Khomh, M. Di. Penta, Y.-G. Gueheneuc, An Exploratory Study of the Impact of Code Smells on Software Change-proneness, in: 16th Working Conference on Reverse Engineering, IEEE, 2009.

[24] M. Lanza, R. Marinescu, Object-oriented metrics in practice, Springer, 2006, doi:10.1007/3-540-39538-5.

[25] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK, in: 22nd International Conference on Program Comprehension, ACM, New York, NY, USA, 2014, doi:10.1145/2597008.2597155.

[26] H. Liu, Y. Yu, B. Li, Y. Yang, R. Jia, Are smell-based metrics actually useful in effort-aware structural change-proneness prediction? an empirical study, in: Proceedings of the 25th Asia-Pacific Software Engineering Conference, IEEE, 2018.

[27] MV. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: an empirical study, Empirical Software Engineering 11 (3) (2006) 395–431, doi:10.1007/s10664-006-9002-8.

[28] R.C. Martin, M. Micah, Agile principles, patterns, and practices in c#, Prentice-Hall, 2006.

[29] A. Martini, F.A. Fontana, A. Biaggi, R. Roveda, Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company, in: European Conference on Software Architecture, Springer, 2018, pp. 320–335.

[30] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, DECOR: A Method For the specification and detection of code and design smells, IEEE Trans. Software Eng. 36 (1) (2010).

[31] S. Nadi, S. Krüger, M. Mezini, E. Bodden, Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? in: 38th International Conference on Software Engineering, ACM, 2016, doi:10.1145/2884781.2884790.

[32] P. Norvig, Natural Language Corpus Data, in: Beautiful Data, O'Reilly Media, 2009, pp. 219–242.

[33] N. Novielli, F. Calefato, F. Lanubile, The Challenges of Sentiment Detection in the Social Programmer Ecosystem, 7th International Workshop on Social Software Engineering, ACM, 2015, doi:10.1145/2804381.2804387.

[34] A. Pal, S. Chang, J. Konstan, Evolution of Experts in Question Answering Communities, in: 6th International AAAI Conference on Weblogs and Social Media, 2012.

[35] F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells, in: 30th International Conference on Software Maintenance and Evolution, 2014, doi:10.1109/ICSME.2014.32.

[36] F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, D. Poshyvanyk, A.D. Lucia, Mining version histories for detecting code smells, IEEE Trans. Software Eng. 41 (5) (2015) 462–489, doi:10.1109/TSE.2014.2372760.

[37] F. Palomba, M. Zanoni, F.A. Fontana, A. De Lucia, R. Oliveto, Toward a smell-aware bug prediction model, IEEE Trans. Software Eng. (2018), doi:10.1109/TSE.2017.2770122.

[38] F. Petrillo, P. Merle, N. Moha, Y. Guéhéneuc, Are rest apis for cloud computing well-designed? an exploratory study, in: Q.Z. Sheng, E. Stroulia, S. Tata, S. Bhiri (Eds.), Service-Oriented Computing, Springer International Publishing, 2016.

[39] G. Pinto, F. Castor, Y.D. Liu, Mining Questions About Software Energy Consumption, in: 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 22–31, doi:10.1145/2597073.2597110.

[40] G. Pinto, F. Castor, Y.D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014.

[41] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (11) (2015) 867–895.

[42] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, F. Castor, An Empirical Study on the Usage of the Swift Programming Language, in: 23rd International Conference on Software Analysis, Evolution, and Reengineering, 2016.

[43] C. Rosen, E. Shihab, What are mobile developers asking about? a large scale study using stack overflow, Empirical Software Engineering 21 (3) (2016), doi:10.1007/s10664-015-9379-3.

[44] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Lessons from building static analysis tools at google, Commun. ACM 61 (4) (2018), doi:10.1145/3188720.

[45] V.S. Sinha, S. Mani, M. Gupta, Exploring activeness of users in QA forums, in: 2013 10th Working Conference on Mining Software Repositories, 2013, doi:10.1109/MSR.2013.6624010.

[46] D.I. Sjoberg, A. Yamashita, B.C. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Software Eng. 39 (8) (2013), doi:10.1109/TSE.2012.89.

[47] Z. Soh, A. Yamashita, F. Khomh, Y.-G. Gueheneuc, Do Code Smells Impact the Effort of Different Maintenance Programming Activities? in: 23rd International Conference on Software Analysis, Evolution, and Reengineering, IEEE, 2016.

[48] D. Spadini, D. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018.

[49] A. Strauss, J. Corbin, Basics of qualitative research: Techniques and procedures for developing grounded theory, SAGE Publications, 1998.

[50] A. Tahir, S. Counsell, S.G. MacDonell, An empirical study into the relationship between class features and test smells, in: 23rd Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2016.

[51] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, S. Counsell, Can you tell me if it smells?: A study on how developers discuss code smells and anti-patterns in stack overflow, in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, in: EASE'18, ACM, 2018, doi:10.1145/3210459.3210466.

[52] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: areplicated study, Inf Softw Technol 92 (2017).

[53] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Software Eng. 35 (3) (2009), doi:10.1109/TSE.2009.1.

[54] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and Why Your Code Starts to Smell Bad, in: 37th International Conference on Software Engineering, IEEE, 2015.

[55] S. Tushar, S. Diomidis, A survey on software smells, Journal of Systems and Software 138 (2018) 158–173, doi:10.1016/j.jss.2017.12.034.

[56] S. Wang, D. Lo, L. Jiang, An empirical study on developer interactions in StackOverflow, 28th Annual ACM Symposium on Applied Computing, ACM, New York, New York, USA, 2013, doi:10.1145/2480362.2480557.

[57] A. Yamashita, Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data, Empirical Software Engineering 19 (4) (2014), doi:10.1007/s10664-013-9250-3.

[58] A. Yamashita, S. Counsell, Code smells as system-level indicators of maintainability: an empirical study, Journal of Systems and Software (2013).

[59] A. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, in: 20th Working Conference on Reverse Engineering, IEEE, 2013.

[60] A. Yamashita, L. Moonen, T. Mens, A. Tahir, Report on the first international workshop on technical debt analytics (tda 2016)., 2016.

[61] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, Journal of Software Maintenance and Evolution: Research and Practice 23 (3) (2011), doi:10.1002/smr.521.