Software Code Smells and Defects: An Empirical Investigation

Reuben Brown and Des Greer[a]
School of Electronics, Electrical Engineering and Computer Science, Queen's University, Belfast, U.K.

Keywords: Code Smells, Software Defects, Software Maintenance.

Abstract: Code smells indicate weaknesses in software design that may slow down development or increase the risk of bugs or failures in the future. This paper aims to investigate the correlation of code smells with defects within classes. The method used uses a tool to automatically detect code smells in selected projects and then assesses the correlation of these to the number of defects found in the code. Most existing articles determine that software modules/classes with more smells tend to have more defects. However, while the experiments in this paper covering a range of languages agreed with this, the correlation was found to be weak. There remains a need for further investigation of the types of code smells that tend to indicate or predict defects occurring. Future work will perform more detailed experiments by investigating a larger quantity and variety of software systems as well as more granular studies into types of code smell and defects arising.

1 INTRODUCTION

'Code smells' can be caused by poor design decisions taken when writing code due to, for example, tight deadlines or developer incompetence. The term, first proposed in 1999, describes decisions made in object-oriented systems that are not compoatible with widely accepted principles of good object oriented design (Fowler, 2018). Code smells are not bugs and do not prevent software from functioning; rather they indicate problems in software design or code which makes software difficult to maintain (Kaur, 2020). There are 22 code smells defined by Fowler such as 'Long Method', 'Alternative Classes with Different Interfaces' and 'Message Chains' (Fowler, 2018) and some additional ones that were not initially defined by Fowler, such as 'Dead Code'. Some code smells (such as 'Feature Envy') exist in individual methods, some (such as 'Lazy Class') exist within individual classes, and some (such as 'Inappropriate Intimacy') exist in the relationships between classes. Mantyla et al. (2003) categorised the 22 Fowler-defined code smells, and one additional code smell, into seven categories, namely: 'bloaters', 'object-orientation abusers', 'change preventers', 'dispensables', 'encapsulators', 'couplers', and 'others'. 'Bloaters' refer to methods and classes that have increased to such proportions that they are especially difficult to work with; usually they accumulate over time as the program and codebase evolves, rather than showing up immediately. 'Object-Orientation Abusers' refer to incomplete or incorrect applications of object-oriented programming principles. Descriptions of the other categories can be found in the taxonomy by Mantyla et al. (2003).

Many studies have already been completed relating to code smells and their impact on various quality attributes, including reliability, maintainability, and testability. Code smells are mostly studied in production code (Zazworka et al., 2014; Kaur, 2020; Bán & Ferenc, 2014; D'Ambros et al., 2010; Li & Shatnawi, 2007; Marinescu & Marinescu, 2011; Fontana et al., 2013; Alkhaeir & Walter, 2020; Kessentini, 2019; Olbrich et al. 2010; Aman, 2012; Lanza & Marinescu, 2007; Saboury et al., 2017), but have also been studied in test code (Spadini et al., 2018; Garousi et al., 2018). Surveys have also been conducted to discover the importance of code smells to developers and the reasons for developers placing a high or low importance on them (Yamashita & Moonen, 2013).

The aim of this research is to empirically assess correlation of software code smells with software defects. The following research question will be addressed.

[a] https://orcid.org/0000-0001-6367-9274

Brown, R. and Greer, D.
Software Code Smells and Defects: An Empirical Investigation.
DOI: 10.5220/0011974500003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 570-580
ISBN: 978-989-758-647-7; ISSN: 2184-4895

RQ. Do software modules/classes with more smells tend to have more defects?

Our null hypotheses are therefore:

H1$_0$: There is no correlation between the number of defects and the number of code smells.
H2$_0$: There is no correlation between the number of defects and the severity of code smells.

H2$_0$ is formulated because all code smells are not equal and thus, there is a need to determine the effect on defects, depending on how serious the code smells.

The remainder of this paper is structured as follows. The next section provides a review of existing research articles. Section 3 describes the methodology, including the chosen strategies to detect code smells and defects and the reasoning used in their selection. Details are given of the projects chosen on which to conduct the empirical investigation, the experimental setup and the tool support used for the investigations. Section 4 presents and discusses the results of the experiments. The final section provides a discussion of the findings including threats to validity and some conclusions to the work as well as looking ahead to future work.

2 BACKGROUND

Intuitively, it might be expected that software modules/classes with more smells tend to have more defects and that, on average, the number of defects in a class/module would increase as the number of code smells in it increases. It would be expected that some code smells would have a stronger effect on the rate of defects than others, because there are many different types of code smells of a range of sizes and severities, so it is unlikely that all code smells would be of the same undesirability. It would also be expected that the correlation of defects with code smells would not be impacted by the platform of the system under investigation.

2.1 Literature Review

To gain insight into the areas studied relating code smells with defects, a review of recent papers was conducted. Only articles considering the correlation of code smells with defects were reviewed, using a mixture of reference snowballing on articles found using IEEE Xplore, Web of Knowledge and Google Scholar with a date filter of between 2018 and April 2022. Abstracts of these articles where searched using the search string *"code smell" AND ("defect" or "bug" or "issue")*. This returned 4,840 papers of which 22 were included as being inside the scope of the study. Basic information about some of the selected articles is summarised in Table 1.

Table 1: Selected Studies.

Reference	**Study Type**	**# systems studied**	**# smells studied**
Bán & Ferenc, 2014	Experiment	34	8
D'Ambros et al., 2010	Experiment	6	5
Li & Shatnawi, 2007	Experiment	3	6
Marinescu & Marinescu, 2011	Case Study	3	4
Olbrich et al. 2010	Experiment	3	2
Zazworka et al., 2014	Case Study	1	9
Fontana et al., 2013	Experiment	68	15
Aman, 2012	Experiment	3	1
Alkhaeir & Walter, 2020	Experiment	10	10
Kessentini, 2019	Experiment	3	11
Saboury et al., 2017	Case Study	5	12
Chouchane et al., 2021	Experiment	120	15

Each article in the review focused on various code smells, with some being studied more frequently than others. 'Feature Envy' was the most frequently investigated code smell with eight studies. 'God Class' was next most common covered by seven studies, while 'Data Class' was investigated in six of the papers. These were also the most frequently investigated in an existing systematic literature review (Kaur, 2020). Furthermore, of the 23 code smells described in Mantyla et al.'s taxonomy of bad smells in code (2003), a number of them of them such as 'Switch Statements', 'Primitive Obsession' and 'Incomplete Library Class', were not studied at all in the sample papers. 'Non-Fowler' code smells not defined in Mantyla et al.'s taxonomy (2003) are also rarely studied. Perhaps this is because of a potential lack of tools or techniques to detect them. Some articles focused on multiple different code smells (Fontana, 2013; Kessentini, 2019; Saboury et al. 2017; Chouchane et al., 2021), whereas others focused on just one or two in more detail (Olbrich et al. 2010; Aman, 2012).

A common approach in papers is to use at least one open-source software system for analysis. There did not seem to be any obvious correlation between the number of code smells studied and the number of software systems studied, as shown in Figure 1. Popular systems included 'Eclipse' (D'Ambros et al. 2010; Li & Shatnawi, 2007; Marinescu & Marinescu, 2011; Aman, 2012) and 'Lucene' (Bán & Ferenc,2014; D'Ambros et al;, 2010; Alkhaeir & Walter, 2020; Olbrich et al., 2010). Java systems were the most analysed, but notably there was not much coverage of systems written in Python, for example. Mobile applications were also not analysed to a great extent, with just one study (Chouchane et al., 2021).

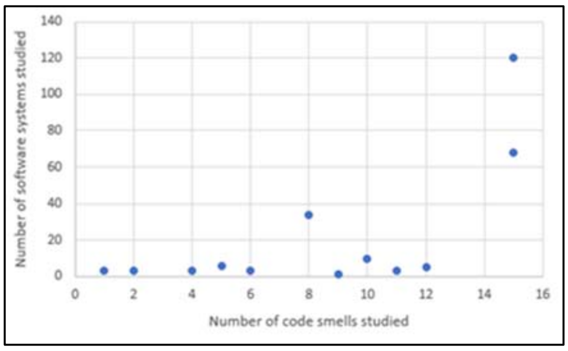


Figure 1: Number of software systems studied and number of code smells studied in research papers.

Articles considered were restricted to those that collected data on code smells and defects for the system(s) they included. Regarding code smell detection, some articles defined their own metrics and models (Bán & Ferenc, 2014; D'Ambros et al., 2010; Fontana et al., 2013; Alkhaeir & Walter, 2020). Others used existing models (Li & Shatnawi, 2007; Kessentini, 2019; Olbrich et al., 2010; Saboury et al. 2017), such as the ones defined by Marinescu & Marinescu (2011). In some cases existing tools such as iPlasma (Fontana et al., 2013), inCode (Alkhaeir & Walter, 2020) or CodeVizard (Zazworka et al., 2014) have been used and paper used a manual detection method (Marinescu & Marinescu, 2011). Some researchers used several of these approaches because certain detection strategies are more suited to particular code smells (Fontana et al., 2013).

In terms of achieving bug or defect detection, some articles gathered bug information from Bugzilla (D'Ambros et a.., 2010; Li & Shatnawi, 2007; Marinescu & Marinescu, 2011; Kessentini, 2019; Olbrich et al., 2010) and some from JIRA (D'Ambros et al., 2010; Olbrich et al., 2010). GitHub is also popular (Saboury et al., 2017), using tools such as FindBugs (e.g. Zazworka et al., 2014). The PROMISE database also features prominently (Alkhaeir & Walter, 2020; Aman, 2012; Bán & Ferenc, 2014). There are various options for data on systems' bugs, defects, or issues to be stored, so the sources used to detect bugs, defects or issues depend on the system under consideration. Some articles also took the recorded severity levels of the bugs into account to weight the issues (Li & Shatnawi, 2017; Kessentini, 2019; Olbrich et al., 2010). On the other hand, D'Ambros et al. (2010) decided against this because of the subjective and potentially biased nature of severity ratings.

To assess the correlation of code smells with defects within classes, each article had to link each defect to a class. The PROMISE database contains the code of numerous open-source applications and their corresponding bug data at a class level (Bán & Ferenc, 2014), so little work is required to link defects to classes. Some articles examined the versioning history for the systems to find commits that addressed certain bugs based on if the ID or the bug issue is contained in the commit message (Zazworka et al., 2014; Mantyla et al., 2003; Olbrich et al., 2010). Defects were then linked to classes based on which classes were modified in the commits. A drawback of this approach has been identified in that it was unable to detect inner classes (Zazworka et al., 2014; D'Ambros et al., 2010). Another approach was to use the changelog of a system to determine which files were changed and which bugs fixed in each release (Li & Shatnawi, 2007).

Each article was analysed for relevance to the research question. Most of the articles found that software modules/classes with more smells tend to have more defects. However, some articles determined that no code smell has more of a correlation with defects in the class containing it than any other code smell (D'Ambros et al., 2010; Alkhaeir & Walter, 2020). Some researchers report that some code smells, particularly 'Shotgun Surgery' and 'God Class', have more of a positive correlation with defects in the class containing them than other code smells (Zazworka et al., 2014; Li & Shatnawi, 2007; Kessentini, 2019). Despite this finding about 'God Classes', Olbrich et al. (2010) found that 'God Classes' and 'Brain Classes' may only correlate positively with defects once they both exceed a certain size and proportion to other classes in the system. Alkhaeir and Walter (2020) determined that the presence of code smells increases the number of defects more than an absence of code smells decreases the number of defects. Marinescu & Marinescu (2011) determined that classes containing the code smells that they studied, including 'God Class', do not tend to have more defects. In an existing literature review Gradišnik and Heričko (2018) determined that software systems with more code smells tend to have more defects, but this does not hold for individual classes. Cairo et al. (2018) concluded that those classes affected by code smells are more disposed to failures than classes that are unaffected by code smells. Although it appears that software modules/classes with more smells tend to have more defects, certain aspects of the existing literature are contradictory.

2.2 Detecting Code Smells and Defects

The detection of code smells in a software codebase can either be performed manually or automatically using a tool. Manual detection is considered to be the most reliable way of identifying code smells (Fowler, 2018), but has disadvantages such as being very time-consuming (Marinescu, 2001; Mantyla et al., 2003) and in that human evaluators have differing perceptions on what is a code smell or the exact categorization of a code smell (Mantyla et al., 2003). One reason for this could be differing experience levels. For example, Mäntylä et al. (2003) found that developers with more experience tended to evaluate software as having fewer code smells than developers with less experience and that differences in detection were larger for some code smells (such as 'Switch Statements') than others. Lead developers tended to identify more structural code smells, whereas regular developers tended to identify more code smells at the code level. Various studies have been performed comparing different automated detection strategies of code smells with manual detection of code smells (Van Emden & Moonen, 2002; Mantyla et al., 2003) and mostly conclude that automated detection is a good alternative to manual detection due to automated detection strategies scaling much better and being of similar precision and recall as manual detection (Olbrich et al., 2010).

For this study, SonarQube[1] was selected to detect code smells. SonarQube supports 29 programming languages including Python and Swift. Mobile applications were identified as a gap in existing research and thus an area to focus on in these experiments, so the support of Swift by SonarQube is attractive since Swift is commonly used to develop iOS mobile applications. SonarQube identifies a code smell as: "A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes. This, alludes to a belief that software modules/classes with more smells do indeed tend to have more defects. Calls can be made to the 'Sonarcloud' API to conduct an analysis of a codebase.

SonarQube identifies code smells and describes them using statements, which can be mapped to code smells described in Mantyla et al.'s (2003) taxonomy. SonarQube identifies code smells by calculating metrics and comparing them to set threshold values, with a code smell being reported if the threshold is exceeded. The statements and the corresponding code smells in SonarQube vary depending on the language of being analysed. Table 2 shows a sample of the statements that can be raised by a SonarQube analysis and which of the code smells from the taxonomy they correspond to, and the severity level ('Info', 'Minor', 'Major', 'Critical' or 'Blocker') assigned to them. Several statements can correspond to the same type of code smell and not every code smell of the same type is assigned the same 'severity' level. SonarQube can return an assessment of different severities of the same code smell and identify the exact cause of it.

Table 2: Example SonarQube statements and code smell meaning.

Statement	**Code smell**	**Severity**
Functions and methods should not have identical implementations	Duplicate Code	Major
Two branches in a conditional structure should not have exactly the same implementation	Duplicate Code	Major
String literals should not be duplicated	Duplicate Code	Critical
Functions should not have too many lines of code	Long Method	Major
Files should not have too many lines of code	Large Class	Major
Functions, methods and lambdas should not have too many parameters	Long Parameter List	Major
Functions and methods should not be empty	Speculative Generality	Critical
Sections of code should not be commented out	Speculative Generality	Major
Unused function parameters should be removed	Speculative Generality	Major
Unused local variables should be removed	Speculative Generality	Minor
Comments should not be located at the end of lines of code	Comments	Minor
Track comments matching a regular expression	Comments	Major

[1] https://www.sonarqube.org/)

With regard to identifying defects in a software module/class, this can be more difficult than detecting code smells. SonarQube can detect "bugs" in classes, which are described in the SonarQube documentation as "An issue that represents something wrong in the code. If this has not broken yet, it will, and probably at the worst possible moment. This needs to be fixed". An issue is described in the SonarQube documentation as "when a piece of code does not comply with a rule" and that "there are 3 types of issue: Bugs, Code Smells and Vulnerabilities". The SonarQube documentation describes a 'vulnerability' as "a security related issue which represents a backdoor for attackers".

The simplest approach to identify defects could on the surface be to just use an analysis using SonarQube and to look at the 'Bugs' and 'Vulnerabilities' data generated, in a similar manner to detecting code smells using SonarQube. However, this would not be accurate because the SonarQube definitions of 'Bug' and 'Vulnerability' are merely predictions of defects, Many of defects could be false positives. An alternative approach is to use the GitHub REST API to retrieve all the issues related to the code repository for the project in question. However, this approach raises the problem of issues on GitHub not having any direct link to a module/class in the codebase. Pull requests on GitHub do have files linked to them based on the files that are edited as part of the pull request, so one possible approach could be to use the GitHub REST API to retrieve pull requests rather than issues. A problem with this approach is that not every pull request corresponds to a defect; some pull requests correspond to new functionality being added or to other changes that are not defects. Simply using pull requests as a measure of defects could therefore lead to the research accidentally investigating the correlation of changes with code smells rather than the correlation of defects with code smells. Some pull requests have issues linked to them, however, and the impact of this problem can be reduced by only considering pull requests that have issues linked to them and hence are to be used to fix defects. There are still potential problems with this approach because some 'issues' on GitHub can refer to new functionality to be added rather than defects to be fixed.

To reduce the impact of these problems, careful consideration must be taken when choosing the repositories to perform the experiments on. Projects with a large number of issues and pull requests on GitHub as well as having been in existence for a prolonged period of time are more likely to be 'stable' so that the majority of their issues and pull requests are related to defect fixing rather than adding new functionality, meaning that such projects would be a good choice. Some projects might not make use of GitHub to track issues and instead use other bug-tracking systems, and thus wouldn't have any issues 'officially' linked to pull requests – meaning that such projects would be a bad choice in this case. There is also the fact that some repositories might be inconsistent with linking issues to pull requests and might not do it for every issue or pull request. On the other hand, some repositories might not create an issue for every pull request. When considering these scenarios, it can be argued that it is difficult to find an approach for detecting defects and linking them to classes that would work for every repository.

After careful consideration of the benefits and drawbacks of each approach, it was decided to use the approach of using the GitHub REST API to retrieve pull requests and considering a defect to be present in each edited file for each pull request, while assuming that each file corresponds to one class. This assumption has the drawback of not considering inner classes, because they are exist in the same file as their containing class. This drawback was also encountered and considered difficult to fix by D'Ambros et al. (2010). Another drawback of this assumption is that files that contain more than one class on the same level will have all classes within considered as one. However, inspection shows that files containing more than one class are relatively rare, so the impact of this is expected to be negligible. The repositories to perform the experiments on must be chosen carefully to ensure that they are suitable to use this approach of defect detecting with – mainly, they must be projects that started a relatively long time ago and thus have a 'stable' release, and they also must have a large set and history of pull requests and files to allow sufficient and reliable data to be gathered.

Table 3: Software systems investigated.

System	Description
Zulip	A team collaboration tool.
Superset	A business intelligence web application.
ECharts	A charting and visualization library.
Zulip Mobile	The official Zulip mobile client, supporting both iOS and Android.

2.3 Choice of Projects for Study

Four open-source projects from GitHub were selected to conduct the empirical investigations. These software systems are detailed in Table 3.

The choice of projects was driven by the conclusions for suitable projects that were arrived at previously in this paper. These are summarised as follows.

2.3.1 Age of Project

Software repositories that have been in development for a considerable amount of time are more likely to have a 'stable' release and thus be mainly focused on bug fixing and have enough issues and pull requests to provide sufficient data for the experiments. Table 4 shows how many issues (both open and closed) and pull requests (both open and closed) are associated with each of the chosen projects and the date when the oldest pull request of each project was closed, all accurate at the time of writing.

Table 4: Size and age of selected software systems.[2]

Project	Issues	Pull requests	First pull request closed
Zulip[a]	7,095	14,805	25 Sep. 2015
Superset[b]	8,021	10,791	23 Jul. 2015
ECharts[c]	15,495	1,388	3 Jun. 2013
Zulip Mobile[d]	1,914	3,449	14 Aug. 2016

2.3.2 Tracking of Issues

Projects that keep track of their defects and issues using GitHub (rather than other bug-tracking systems such as JIRA or Bugzilla) are preferable to use to gather data for the experiments. All of the selected projects use the 'Issues' section on GitHub as the way to track and record issues.

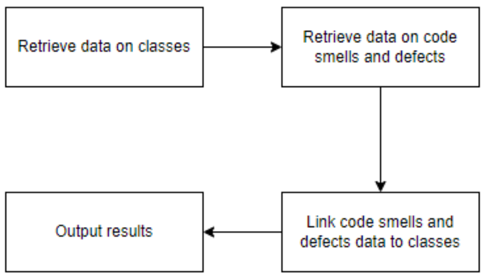

Retrieve data on classes
Retrieve data on code smells and defects
Link code smells and defects data to classes
Output results

Figure 2: Overview of the steps carried out by CodeCorrelator.

2.3.3 Gaps in Research

The identified gaps in research were projects written mainly in the Python programming language, and mobile applications. The majority (61.9%) of the Zulip code is written in Python, and Zulip Mobile is a mobile application and is written mainly in JavaScript. Superset is written mainly in TypeScript (35.1% of the code) and Python (33.2% of the code). The identified gaps in the research are covered by experiments on these repositories. ECharts was also selected for the empirical investigations because it meets the other two recommendations for a suitable project and it was decided to also use a project in a language that has already been thoroughly investigated, namely JavaScript.

3 METHODOLOGY

To answer the research question and provide data collection, a tool, 'CodeCorrelator' was developed,. The tool can be used to calculate the correlation of code smells with defects at a class level. It is a reasonable assumption to make that classes that are larger will have, on average, more code smells and more defects than classes that are smaller, simply because they contain a larger quantity of code and hence have more opportunities for code that is smelly or that causes defects – and this observation could lead to an unintended impression that code smells do correlate with defects. To deal with this, CodeCorrelator can also calculate the correlation on a 'per line' basis.

An execution of CodeCorrelator (Figure 2) on a project first uses the Sonarcloud API to retrieve data on the classes within the project. This requires a Sonarcloud analysis to be performed on the GitHub repository of the project in question beforehand. CodeCorrelator will process the data from the API response to determine the folders that each class is in and how many lines of code are in each class. CodeCorrelator will then use the Sonarcloud API to retrieve data about the code smells within the project. CodeCorrelator will then use the GitHub REST API to retrieve the defects of the project; as mentioned previously, this will be done by retrieving data about pull requests. CodeCorrelator will then process the class data and the code smell data retrieved from the Sonarcloud API to link each code smell to a class. The data on defects retrieved from the GitHub REST API will then be processed to link each defect to a class, by using the GitHub REST API to retrieve more detailed information about each individual pull request that was retrieved previously and identifying which files were edited in each pull request. Finally, the results showing how many code smells and

[2] github.com/{zulip/zulip[a]|superset[b]|echarts[c]|zulip/zulipmobile[d]}

defects are in each class, with or without a comparison to how many lines of code are in the corresponding class, will be outputted to a comma-separated values (CSV) file, and a graph of defects against code smells for each class will be generated.

Manual checking was carried out on the outputted results to ensure that no irrelevant files were detected as classes and hence mixed in with the analysis, reducing the integrity and reliability of the results of the experiments. For example, the experiments are only concerned with production code and hence analysis of test code is not desirable. Some of this checking was automated; CodeCorrelator contains logic to ignore any files that have the string 'test' contained in their name or path so that they are not analysed. Manual checking was performed by opening the generated CSV results files and removing the records for any 'test' files that managed to escape detection. Equally, the records for migrations files or image files were also removed prior to generation of results.

One important note is that some code smells are more serious and undesirable than others. Hence, in addition to using raw numbers of code smells, a weighted score was used based on the 'severity' variable for a code smell as returned via the Sonarcloud API. This weighting is similar to the weights that Olbrich et al. (2010), assigned to weighted defects which were based on the severity levels provided by JIRA (which uses the same terms for severity levels as the Sonarcloud API for code smells, except for the lowest severity being referred to as 'trivial' by JIRA but 'Info' by Sonarcloud), as shown in Table 5.

Table 5: Code Smell Weightings.

Severity Level - Sonarcloud	**Weight**
Blocker	16
Critical	8
Major	4
Minor	2
Info	1

4 RESULTS AND ANALYSIS

Table 6 shows a snippet of the number of code smells and defects per class for the Zulip project. Due to 1,383 files being analysed, only the first five and the last five when sorted alphabetically are displayed as a sample. Figure 3 shows the scatter graph of defects against code smells per class for each of Zulip, Zulip Mobile, Superset and ECharts. All of these plots (a-d) show large clusters of data points in the bottom-left corners of the graphs and much more sparse spreads of data points as they move away from the origin, meaning that the vast majority of classes contain both only a small number of code smells, if any, and only a small number of defects, if any. Indeed, the snippet of results in Table 6 shows six of the ten Zulip classes presented containing zero code smells, with only ten code smells combined between the ten classes shown.

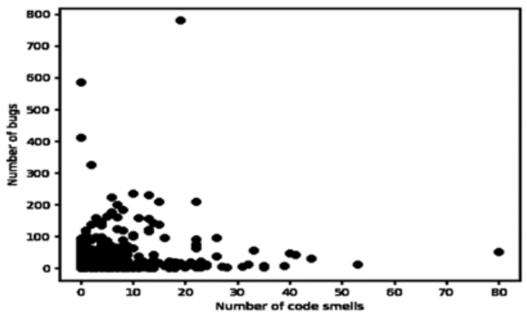
Table 6: Sample Results for Zulip Project.

Class	Code smells	Defects
analytics/__init__.py	0	2
analytics/lib/__init__.py	0	5
analytics/lib/counts.py	4	90
analytics/lib/fixtures.py	2	21
analytics/lib/time_utils.py	0	10
…	…	…
zproject/prod_settings_template.py	2	137
zproject/sentry.py	0	12
zproject/settings.py	0	411
zproject/urls.py	2	327
zproject/wsgi.py	0	10

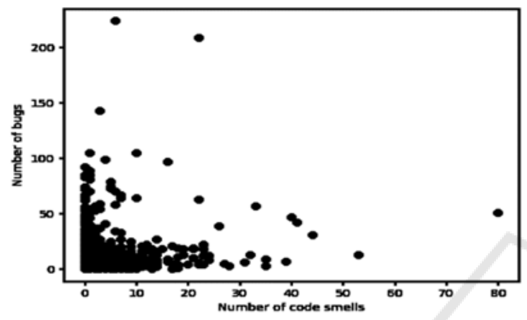
The results show a small positive correlation between the number of code smells within a class and the number of defects within a class for all four analysed projects. The correlation for each project has been calculated using the Pearson Product-Moment Correlation Coefficient, presented in Table 7. A correlation value close to 1 suggests a strong positive correlation, while a correlation value close to -1 suggests a strong negative correlation. A correlation coefficient close to 0 suggests no correlation exists. The correlation values calculated from the experiments conducted average at approximately 0.474, which would indicate a weak positive correlation. This implies that software modules/classes with more smells do indeed tend to have more defects. However, correlation values are not normally considered to be important when the absolute value is less than 0.8. From the experiments performed here, however, it is seen that all four projects have a similar weak positive correlation value.

Table 7: Correlation of code smells and defects.

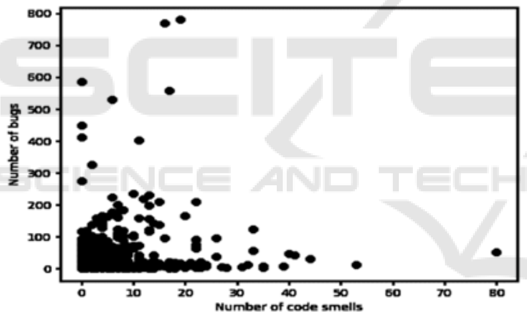
Project	Code smells / defects correlation
Zulip	0.494
Zulip Mobile	0.357
Superset	0.460
ECharts	0.584



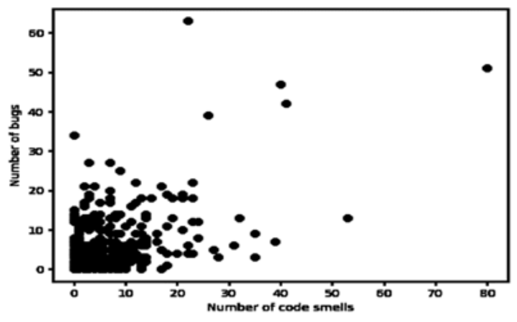
a. 'Zulip'



b. 'Zulip Mobile'



c. 'Superset'



d. 'ECharts'

Figure 3: Number of defects in each class in 4 systems against the number of code smells in the class.

Several hypotheses can be formulated based on the results. The Zulip Mobile project had the lowest correlation between code smells and defects, with a correlation value that was 0.103, or 22.4%, lower than the next lowest correlation value, which was the Superset project. This provides evidence that perhaps code smells are not as impactful for mobile applications as they are for non-mobile applications.

With regards to programming languages, there did not seem to be a significant difference in the correlation of code smells with defects within classes. Zulip, mostly in Python and Superset, with significant code written in Python were the analysed projects with the second and third strongest positive correlations between code smells and defects within classes respectively; hence, Python was neither the language with the most positive correlation or the language with the least positive correlation.

With regards to the sizes of a project in terms of the total number of lines of code that make it up, there did not seem to be a significant difference in the correlation of code smells with defects within classes either. Table 8 shows the number of lines of code within each of the four projects that were analysed.

Table 8: Project Sizes.

Project	# Lines	# Files	Lines per file
Zulip	406,597	1,383	310
Zulip Mobile	126,491	572	221
Superset	1,687,013	3,213	525
ECharts	303,598	731	415

Larger projects with more lines of code, such as Superset from our experiments, do not seem to have a significant difference in the correlation of code smells with defects within classes as smaller projects with fewer lines of code, such as ECharts from our experiments. Furthermore, the size of a class (shown in Table 8 as the number of lines per file) does not seem to have a significant impact on the correlation of code smells with defects either. While it is true that the project with the weakest correlation between code smells and defects within the classes, Zulip Mobile, was also found to be the project with the smallest average class size in terms of the number of lines of code, this relationship did not hold for the other projects. For example, Superset had the largest average class size in terms of the number of lines of code but had the weakest correlation between code smells and defects within each class, apart from Zulip Mobile.

When weighting the code smells on severity level, the results are similar to when looking at the raw numbers of code smells, with all four projects showing a weak positive correlation between code smells and defects within classes. These results are presented in Table 9 and visualized in Figure 4 for each project.

Table 9: Correlation of code smells and defects.

Project	Weighted Code smells / defects correlation
Zulip	0.453
Zulip Mobile	0.405
Superset	0.472
ECharts	0.572

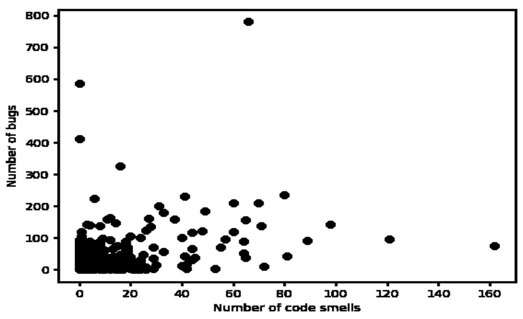
The average correlation value when weighting the code smells was 0.476, which is similar to when the code smells are not weighted. Zulip and ECharts have a weaker correlation when weighting the code smells, while Zulip Mobile and Superset have a stronger correlation when weighting the code smells – but the change in correlation for each of the four projects is minimal. Ordering the projects from strongest positive correlation to weakest positive correlation gives a similar order when weighting the code smells as when not weighting the code smells – with ECharts as the strongest correlation between code smells and defects and Zulip Mobile as the weakest, with Zulip and Superset swapping their orders.

The similarity of the results when weighting the code smells to the results when not weighting the code smells indicates that the hypotheses and observations observed earlier – such as code smells are not as impactful for causing defects for mobile applications as they are for non-mobile applications, Python (and other programming languages too) not having much impact on the correlation of code smells with defects within classes, and the size of the projects not having much impact on the correlation of code smells with defects within classes, and there always being a weak positive correlation between code smells and defects within classes – still exist. It can be additionally argued, based on these results, that the severity of a code smell does not have much impact on how many defects it causes.

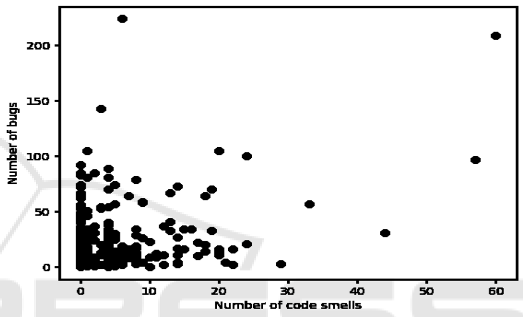
If software classes with more smells tend to have more defects.

5 DISCUSSION

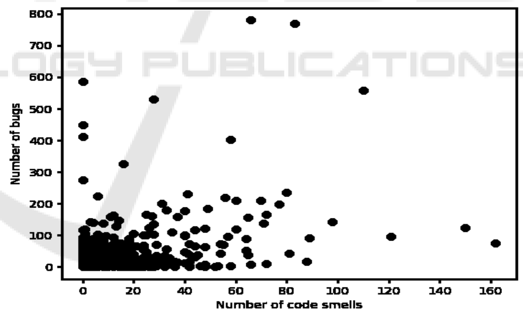
Empirical investigations were conducted using a tool, 'CodeCorrelator', on four open-source software systems to investigate the correlation between the number of code smells in a module/class and the number of defects in the module/class, to determine the relationship and correlation between the number of code smells in a class/module and the number of defects in the class/module. Attempting to answer the research question 'Do software modules/classes with more smells tend to have more defects?'



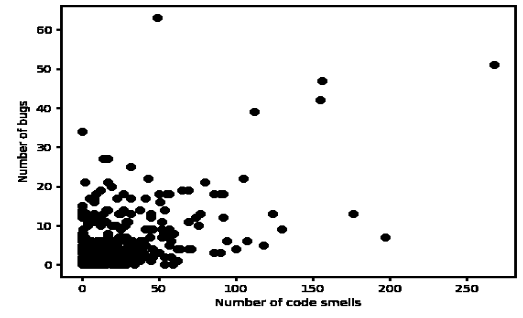
a. 'Zulip'



b. 'Zulip Mobile'



c. 'Superset'



d. 'ECharts'

Figure 4: Number of defects in each class in each project against the weighted code smells in the class.

5.1 Threats to Validity

Construct Validity: As stated previously, because of the file-based nature of Sonarcloud and GitHub, the assumption was made that each file corresponds to one class, which is not always true. This means that the code smell and defect data gathered for one class could be code smell and defect data for several classes merged. This threat and its consequences also apply to inner classes which were not considered. One way to avoid this problem could be to analyse the description of pull requests, issues, or commit messages to determine which class within a file is affected by the defect (or code smell). Another threat is in that each pull request was assumed to correspond to defects, when in reality some pull requests introduce new functionality to a project rather than fixing defects. Even among issues marked as 'bugs', there still could have been many that did not actually correspond to a defect, as shown by Antoniol et al. (2008) that a considerable percentage of problem reports marked as 'bugs' are not related to corrective maintenance. In future it may be a good idea to make use of the approach proposed by Antoniol et al. to filter out the 'non-bugs'.

External Validity: Open-source software systems only were investigated; it is possible that differences between closed and open-source development could alter the results of the experiments, as the rate of code smells and/or defects may differ. Further although choice of systems studied was informed by knowledge of those systems, it cannot be claimed that they are representative of all software systems.

Internal Validity: The usage of only Sonarcloud to detect code smells also raises potential threats to validity, because it might output some false positives or false negatives. Another validity threat relates to the code that was analysed; although both automated and manual approaches were used to remove files from the analysis that were not production code (such as test code, migrations, image files, etc.), it is possible without exhaustive searching that some obsolete or discarded code may have escaped filtering. It is also unlikely that the defect data gathered is complete, because it is possible that there were some defects that were not recorded or discovered. Another threat to validity is that, despite carrying out some experiments in which code smells were weighted due to their severity, the defects were not weighted due to severity levels and instead were just considered as a raw number of defects, with any potential 'minor' defects being treated the same as any potential 'blocker' defects. This is because the defect data was gathered using GitHub issues and pull requests, and there is no 'severity' attribute available. Using data from alternative bug tracking systems, such as Bugzilla or JIRA, could resolve this but requires considerably more effort and is left as future work. In mitigation of this also, Ostrand et al. (2004) have suggested that defect severity levels may be unreliable because they are evaluated by humans and thus can be subjective and inaccurate, sometimes assigned because of political reasons and not related to the actual bug itself. A defect can also have its severity level unreliably increased to boost the reputation of the developer who fixes it or be reported to be more severe than it actually is so that it receives extra focus and a quicker fix (D'Ambros, 2010).

6 CONCLUSIONS

In this study, empirical investigations were performed using a the CodeCorrelator tool on four open-source software systems to determine the relationship and correlation between the number of code smells in a module/class and the number of defects in the module/class. The findings of the experiments were in line with most of the studies analysed as part of the literature review in Section 2 of this paper, as software classes with more code smells did tend to have more defects, and this held true for all four of the software systems investigated, regardless of platform (mobile or non-mobile), programming language (Python or JavaScript), or size (such as number of classes or number of lines of code). Thus, null hypothesis $H1_0$ can be rejected. The findings remained similar when weighting the code smells using severity, so the null hypothesis $H2_0$ can also be rejected. However, the correlation between code smells and defects within classes was very weak in all of the investigations, so while the answer to the research question was found to be 'yes', it could be argued that the evidence of a relationship between code smells and defects within classes is not particularly strong. This is in line with the findings in Kaur (2020) who found the relationship between code smells and failures inconclusive.

In future work, it would be important to complete empirical investigations on many more software systems, both open-source and closed-source. This would ensure a higher chance of knowing if the results found in this study still stand when performed on a much wider range, variety, type, and quantity of software systems, or if they are simply valid just for the four specific software systems focused on in this study. Alternatives to SonarQube may also be sought since recent work has been critical of its rules and how they translate to severity (Lenarduzzi et al, 2020). Another possibility for future work is assessing the correlation of different types of code smells with defects within classes to see which code smells are the most impactful in terms of defects. Another interesting avenue for exploration might be to explore non-code 'smells' that can be detected earlier in the software process, even as early as project initiation and in requirements models (Greer & Conradi, 2008) and how these might relate to defects.

REFERENCES

Alkhaeir, T., & Walter, B. (2020). The effect of code smells on the relationship between design patterns and defects. IEEE Access, 9, 3360-3373.

Aman, H. (2012). An empirical analysis on fault-proneness of well-commented modules. In 2012 Fourth International Workshop on Empirical Software Engineering in Practice (pp. 3-9). IEEE.

Bán, D., & Ferenc, R. (2014). Recognizing antipatterns and analyzing their effects on software maintainability. In International Conference on Computational Science and Its Applications (pp. 337-352). Springer, Cham.

Cairo, A. S., Carneiro, G. D. F., & Monteiro, M. P. (2018). The impact of code smells on software bugs: A systematic literature review. Information, 9(11), 273.

Chouchane, M., Soui, M., & Ghedira, K. (2021). The impact of the code smells of the presentation layer on the diffuseness of aesthetic defects of Android apps. Automated Software Engineering, 28(2), 1-29.

D'Ambros, M., Bacchelli, A. & Lanza, M. (2010). On the impact of design flaws on software defects. In 2010 10th International Conference on Quality Software (pp. 23-31). IEEE.

Fontana, F. A., Ferme, V., Marino, A., Walter, B., & Martenka, P. (2013). Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In 2013 IEEE International Conference on Software Maintenance (pp. 260-269). IEEE.

Fowler, M. (2018) Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional.

Garousi, V., Kucuk, B., & Felderer, M. (2018). What we know about smells in software test code. IEEE Software, 36(3), 61-73.

Gradišnik, M. I. T. J. A., & Hericko, M. (2018). Impact of code smells on the rate of defects in software: A literature review. In CEUR Workshop Proceedings (Vol. 2217, pp. 27-30).

Greer, D., & Conradi, R. (2009). Software project initiation and planning–an empirical study. IET software, 3(5), 356-368.

Kaur, A. (2020). A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. Archives of Computational Methods in Engineering, 27(4), 1267-1296.

Kessentini, M. (2019). Understanding the correlation between code smells and software bugs. Available at https://deepblue.lib.umich.edu/bitstream/handle/2027.42/147342/CodeSmellsBugs.pdf; accessed 9-Jan.2023.

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

Lenarduzzi, V., Lomio, F., Huttunen, H. & Taibi,D. (2020) "Are SonarQube Rules Inducing Bugs?," In IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE.

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

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. (pp. 381-384). IEEE.

Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. In Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39 (pp. 173-182). IEEE.

Marinescu, R., & Marinescu, C. (2011). Are the clients of flawed classes (also) defect prone?. In 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation (pp. 65-74). IEEE.

Olbrich, S. M., Cruzes, D. S., & Sjøberg, D. I. (2010). Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In 2010 IEEE international conference on software maintenance (pp. 1-10). IEEE.

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

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In 2018 IEEE international conference on software maintenance and evolution (ICSME) (pp. 1-12). IEEE.

Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. In Ninth Working Conference on Reverse Engineering, 2002. Proceedings. (pp. 97-106). IEEE.

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

Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., & Shull, F. (2014). Comparing four approaches for technical debt identification. Software Quality Journal, 22(3), 403-426.