An Empirical Study of Design Degradation: How Software Projects Get Worse Over Time

Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, Carlos Jensen
School of EECS
Oregon State University
Corvallis, OR, USA
{ahmed, mannanu, gopinath, cjensen} @eecs.oregonstate.edu

***Abstract*— Context**: **Software decay is a key concern for large, long-lived software projects. Systems degrade over time as design and implementation compromises and exceptions pile up.**

Goal: Quantify design decay and understand how software projects deal with this issue.

Method: We conducted an empirical study on the presence and evolution of code smells, used as an indicator of design degradation in 220 open source projects.

Results: The best approach to maintain the quality of a project is to spend time reducing both software defects (bugs) and design issues (refactoring). We found that design issues are frequently ignored in favor of fixing defects. We also found that design issues have a higher chance of being fixed in the early stages of a project, and that efforts to correct these stall as projects mature and the code base grows, leading to a build-up of problems.

Conclusions: From studying a large set of open source projects, our research suggests that while core contributors tend to fix design issues more often than non-core contributors, there is no difference once the relative quantity of commits is accounted for. We also show that design issues tend to build up over time.

Keywords— Software Decay, Design Problems, Project History.

I. INTRODUCTION

Software systems require constant modifications in the form of bug fixes and the addition of new features to satisfy end user needs. Failure to do so might lead to losing users or unsatisfied users [24, 4]. The pressure to keep growing and evolving the software often makes it impossible to refactor and redesign when a requirement changes. This eventually leads to decay in the software design and the growth of technical debt. One outcome of such decay is that code becomes more difficult to extend or understand [47], and as a result the ability to evolve an application tends to decrease over time [33].

Design degradation leads to design debt, which contributes to technical debt [6] and negatively impacts the overall quality of the software. One of the symptoms of design degradation is that code structure drifts away from good object-oriented design principles (e.g. becomes too entangled and difficult to modularize). These bad design decisions leading to technical debt are also known as code smells [13]. This term was coined by Fowler and Beck [13], who gave an informal definition of 22 code smells focused on the maintainability of software systems, and a set of indicators. Each code smell examines a specific kind of system element (e.g. classes or methods), which can be evaluated by its internal and external characteristics. Researchers have used code smells as a measurement of design degradation [7, 25, 39].

In this paper we present the results of an empirical study on the presence and evolution of code smells, used as an indicator of design degradation. To the best of our knowledge, in contrast to previous studies [5, 22, 25, 38, 39, 42] ours is the largest study so far in terms of both the size of programs involved (534 to 100,000 lines), and the number of projects analyzed (220 open-source projects). This allows for stronger and more widely applicable conclusions about the evolution of design degradation and code smells.

The goal of this study is to shed light on how design degradation happens as a project ages, and how traditional quality assurance (QA) activities contribute or fail to contribute towards improving design quality. More specifically we try to answer the following research questions:

1. How code smells evolve over time?
2. Is refactoring aimed at addressing technical debt dominated by specific sub-groups of developers?
3. Does the testedness of a project and the quality of tests show a correlation with design quality?
4. Is there a match between the smells discussed in literature and in tools and the smells projects most commonly struggle with?

The reminder of the paper is organized as follows: We start with a review of research on design degradation and the techniques researchers have used to identify design degradation. Then we discuss how design degradation manifest in the form of code smells and the research related to code smells. Next we describe our methodology, filtering criteria and the demography of FOSS projects we studied. We also explain the tool selection and evaluation criteria. Section 4 describes the results of our study. Section 5 discusses our findings, their implications and how they answer our research questions. Section 6 concludes with a summary of the key findings and future work.

II. RELATED WORK

The informal definition of design degradation provided by Martin states that as software evolves it starts to rot, like "a

piece of bad meat," if dependencies among modules are not adequately managed [33]. This results in a code base that is difficult to maintain, reuse and add new features to.

Researchers have come up with various techniques for identifying design degradation using static analysis techniques, where the degradation is assessed by analyzing single, static versions of software systems [7, 8, 25]. However, software repository mining provides a way to extract the historical evolution of a software system [20]. Researchers have also come up with techniques that rely on evaluation of successive versions of a software system [11, 21, 23] to identify design degradation as a process.

Researchers have looked at the effect of software evolution on design quality. Izurieta et al. found that as systems mature, artifacts that do not have any role in the design pattern tend to build up around the pattern and accumulate, like "grime", which eventually leads to design pattern decay [17]. Another facet of design degradation is design debt, which contributes to technical debt [6]. Design debt builds up as exceptions are made to speed up development, or we deal with edge cases in the development of a product. However, as this debt builds up, it may reach a level where interest payments in the form of difficulties in understanding and maintaining the code and as it deviates from design and documentation outweighs any short-term benefits. Refactoring can be used to address this debt, revisiting exceptions and hacks made, and changing the underlying problem to create a more sustainable, and efficient design. Code smells have been used to measure design debt. For instance Zazworka et al. [47] found that the God class smell is related to technical debt.

The concept of code smells was introduced by Fowler [13]. Code smells are symptoms of problems in source code, and indicators of where refactoring is needed [13]. Although design degradation and code smells are very similar, the distinction between the two is that code smells are defined at a higher level of abstraction and have a negative impact on a larger part of the software value than a localized piece of code. Code smells has been associated with bugs [25, 38] and code maintainability problems [13].

The identification of code smells is typically done during development, testing, and maintenance. Many approaches have been proposed for code smell detection, such as metric based [23] and meta-model based [35]. Metric based measures show that code smells impact software quality [30]. Most of the code smell detection tools are based on metric analysis [23, 29]. This static analysis based approach has its drawbacks. Fowler and Beck claimed that "No set of metrics rivals informed human intuition" [13] when it comes to deciding whether an instance of a code smell should be refactored. Researchers have also categorized code smells based on their impact level or inter component relationship. Examples of such smells include the "Object oriented Abusers", "Couplers", and "Bloaters" [28, 32].

Several studies have looked into the relationship between code smells and change-proneness. Olbrich et al. [38] report that classes infected with the "God class" and "Shotgun surgery" smells are more change prone. Contrary to their finding, Schumacher et al. [42] found that the "God class" is only more change prone if results are not normalized by LOC. However, Khomh et al. [22] found that classes infected with code smells are changed more often overall.

Identifying the impact of the code smells [38] and how these impact the understandability and maintainability of code [2, 10] has also been of interest to researchers. Smith et al. found that "God class" and "Switch statements" smells have an impact on software performance [44]. Prioritization of code smells has been identified as an important issue, as large numbers of warnings are often generated for a code-base. Fontana et al. [12] surveyed 6 code smell detection tools and found that none of these prioritized results. Many different ways of visualizing code smells have also been proposed [36]. Murphy et al. [36] identified some guidelines that could be useful to help developers prioritize code for refactoring.

Understanding design degradation is important. Researchers have looked at how the testability of a system is impacted as design patterns decay over time. Izurieta et al. found that design pattern decay leads to reduced modularity, eventually increases the required number of test cases needed to meet test requirements [18]. They also found that design pattern decay leads to testing anti-patterns such as "concurrent-user-relationship" and "self-use-relationship".

III. METHODOLOGY

A. *Project Selection Criteria*

We wanted to make sure that our findings would be representative of the code developed in real world. We decided to use Java as the language of focus. This decision was influenced by 2 factors: First, Java is one of the most popular languages (according to the number of projects hosted on Github and the Tiobe index[1]). The second was the availability of code smell detection tools for Java compared to other programming languages.

We searched for projects in Github written in Java. We randomly selected 500 projects from this list. For ease of build and analysis, we only selected projects using the Maven [1] build system.

We checked for the distribution of commits across the history of the projects and found that the majority (95%) of projects had an active history of less than 200 weeks. Figure 1 has the frequency distribution of commits over time for all projects. Because of the long tail property, we cut off analysis at 200 weeks in order to not skew our findings due to the skewing.

Our aim is to see how code smells evolve over time. To do so we could have used different ways of partitioning time. Some researchers [e.g. 17, 18] have chosen to use releases as the unit of time, others individual commits, or discrete time units (years, months, weeks, days). Though all of these approaches should lead to similar findings, the "resolution" may be different. Furthermore, none of these approaches lead to a true apples-to-apples comparison across projects. Projects work at different phases, projects are of different size, maturity level, and follow different release cycles and policies. Individual commits are the only "level" measure, but would be too fine grained for our purpose. We therefore selected the

[1] http://tiobe.com/index.php/content/paperinfo/tpci/index.html

week as our unit of measure because, while subject to some variation from project to project, did give us fine-grained enough insight into the evolution of projects.

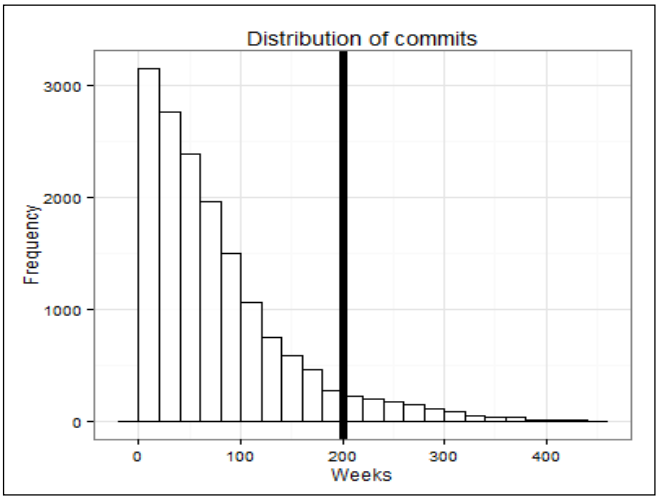

Fig. 1. Distribution of commits over time (vertical line indicating cutoff at 200 weeks)

We removed projects that were too small, had very few files (< 10 files), or few lines of code (< 534 lines of code). This filtering was essential because we wanted make sure that the projects we are analyzing are not too small or too simple for real world projects. We also removed projects that had short lifespan (< 10 weeks) because such projects can skew the results. Our final data set contained 220 projects. Table I provides a summary of features and other descriptive information about the projects that were part of our study.

TABLE I. PROJECT STATISTICS

Dimension	Max	Min	Average	Stddev
Line count	116,238	534	5,837.00	14,511.73
# Developers	105	4	10.78	11.04
Total Code smells	260	1	15.57	30.27
Duration (Weeks)	200	10	41.37	43.18

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [9]. Table II has the summary of the domains of the projects. In the next subsection, we discuss the code smell detection tools used.

B. Tool selection

We chose to use *InFusion* [16] to identify code smells because it has been found to identify the broadest set of smells [12]. Researchers have found that the metric-based approach identified by Marinescu [31] has the highest recall and precision (precision: 71%, recall: 100%) for finding most code smells [42]. *InFusion* uses this same principle and set of thresholds for identifying code smell, which was another reason for using *InFusion*.

Our analysis depends on the smells identified by Infusion and we needed to have some level of confidence about the performance of the tool. There was no such evaluation available for InFusion, so we evaluated the smell detection performance of *InFusion*. We used the oracle constructed by Palomba et al [11]. Palomba et al. mentions that their oracle does not ensure completeness but it provides a degree of confidence about the correctness of the identified smell instances. In the oracle Divergent Change and Parallel Inheritance code smells are "intrinsically historical" and is not identified by *InFusion*. So we evaluated *InFusion's* performances by calculating precision (1) and recall (2) for identifying Blob and Feature Envy code smells from the oracle. We also report the F-measure (3), defined as the harmonic mean of precision and recall as an aggregate indicator of precision and recall [3]. Table IV has the summary of the performance of *InFusion*.

TABLE II. DISTRIBUTION OF PROJECTS BY DOMAIN

Domain	Percentage
Development	61.98%
System Administration	19.80%
Communications	6.25%
Business & Enterprise	3.12%
Home & Education	3.12%
Security & Utilities	2.61%
Games	2.08%
Audio & Video	1.04%

TABLE III. LIST OF SMELLS IDENTIFIED BY INFUSION

Smells
Cyclic Dependencies
Brain Method
Data Class
Feature Envy
God Class
Intensive Coupling
Missing Template Method
Refused Parent Bequest
Sibling Duplication
Shotgun Surgery
SAPBreakers
Internal Duplication
External Duplication
Blob Class
Blob Operation
Data Clumps
Message Chains
Distorted Hierarchy
Schizophrenic Class
Tradition Breaker
Unstable Dependencies

$$\text{Recall} = \frac{|\text{True positive smells} \cap \text{code smells detected by InFusion}|}{|\text{True positive smells}|}\% \quad (1)$$

$$\text{Precision} = \frac{|\text{True positive smells} \cap \text{code smells detected by InFusion}|}{|\text{Code smells detetctedby InFusion}|}\% \quad (2)$$

$$\text{F-measure} = \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)$$

TABLE IV. INFUSION PERFORMANCE

Precision	Recall	F-measure
84%	100%	91.30%

C. Data collection

We selected data from the Git repositories of the 220 projects, from the project start date until April 25th, 2013. We collected a total of 33,070 commits across the 220 projects. From the initial code commit, we calculated the code smells added or removed by each subsequent code commit to a project. For each commit we also calculated the number of modified lines.

We categorized the code smells into broad categories, as suggested by the code smell literature [28]. These categories were: Bloater, Object oriented abusers, Coupler, Dispensable, Encapsulators and Others. Bloaters are code smells that lead the code to balloon so it cannot be effectively managed. The smells include data clumps, large class, long method, long parameter list and primitive obsession. Object oriented abusers are smells that do not fully exploit the advantages of object-oriented design. Some of the smells include Switch statements, parallel inheritance hierarchies, and alternative classes with different interfaces. The Coupler category contains the code smells that identify high coupling between objects, in defiance of good object oriented design principles. Smells in this category include feature envy and inappropriate intimacy. The Dispensable category contains smells such as the lazy class, data class and duplicate codes. The Encapsulators category contains code smells that deal with the data communication mechanism or encapsulation. This includes message chain and middleman smells. Others is an aptly named catch-all category.

We collected the total number of test cases present for each project after each code commit, an indicator of the testedness of the code. We also calculated the code coverage of the test suites. Coverage metrics such as statement coverage, branch coverage, path coverage etc are the indicators of quality of the test case [46]. We gathered different coverage metrics, such as statement coverage from Emma [45], branch coverage from Cobertura [26], path coverage from JMockit [40], and mutation kills from PIT [15]. Then we checked whether there is any difference between the low (less than 30%) and high (more than 60%) tested (measured using these coverage criteria) projects and total number of code smells present in the project.

IV. RESULTS

In the following section, the collected and observed results for the research questions stated above are presented.

A. How code smells evolve over time

To answer our first research question we collected the total number of code smells after each commit. We normalized the smell count using feature scaling (4), which gives us a score between 0 and 1.

$$\text{Rescaled value} = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (4)$$

Previous studies have shown that normalizing the smell count using the project size reduces the bias of larger projects on the overall smell count [42]. For our study this was not necessary. Our aim was to identify general trends across projects, not to look at differences between them. There was therefore no need to normalize based on project size.

We looked at the code smells change trend for each project. For this purpose we calculated the effect size of week on normalized smell count using a linear regression model, giving us how much smell count changed for each project per week. Then based on the effect size we categorized each project into one of three categories: increasing, decreasing or unchanged. If the effect was positive and larger than 0.005, we marked those projects as increasing. We selected 0.005 as our threshold because this indicates a change of less than or equal to a 0.5% of the number of smells, or at most 1 smell added or removed per week. For negative effect we applied the same threshold and marked the project as decreasing. Any other project was marked as unchanged. In Table V we report the percent of each of these category. We also checked whether the effect size of mean smell count change of increasing and decreasing groups are different. (Welch two-sample t-test, t = -2.1623, df = 34.689, p-value = 0.02411), meaning that there is strong evidence that these two groups differ in their mean effect size.

TABLE V. PERCENTAGE OF DIFFERENT CODE SMELL CHANGE PATTERN

Category	Percentage
Increasing	55.00%
Decreasing	7.85%
Unchanged	37.15%

To have an understanding of the bigger picture we looked at the average number of smells across all projects, and found that it grows monotonically throughout 200 weeks. Figure 2 shows the average project smelliness compared to the smelliest project in the sample.

We also wanted to check whether this same trend holds true for all smell categories. We found that all smells in the Bloaters category (consisting of Blob Class which indicates classes that are very large and complex, Blob Operation which is a very large and complex operation and Data Clumps representing groups of data that appear together over and over again, as parameters that are passed to operations throughout the system) increase over time.

We found that code smells in the Dispensables category had a mixed tendency; Data Classes code smell – which are data holders without complex functionality, but usually heavily relied upon by other classes in the system – increase over time. Internal and External Duplication – which identifies duplication between portions of the same class or module, and duplication between unrelated capsules of the system respectively, tend to increase slowly or even dip over time.

We found that code smells in the OO abusers category follow a mixed pattern also. These include SABreakers, which looks for a mismatch between the subsystem's stability and its level of abstractness and the God class, indicating high complexity classes with a low inner-class cohesion and extensive access to the data of foreign classes. These two classes showed a generally growing pattern.

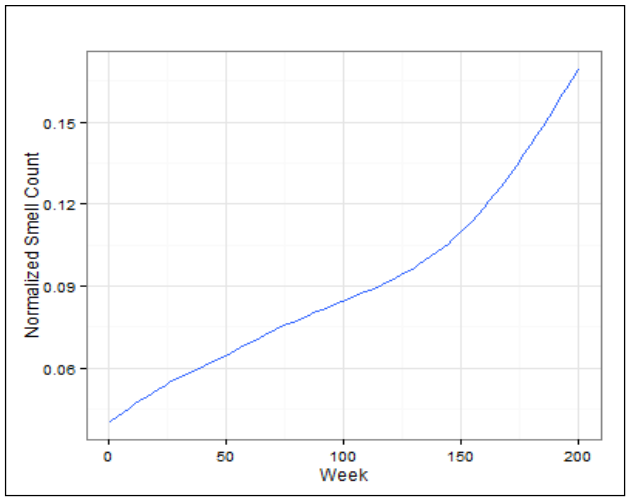

Fig. 2. Week-wise average project smellines compared to the smelliest project

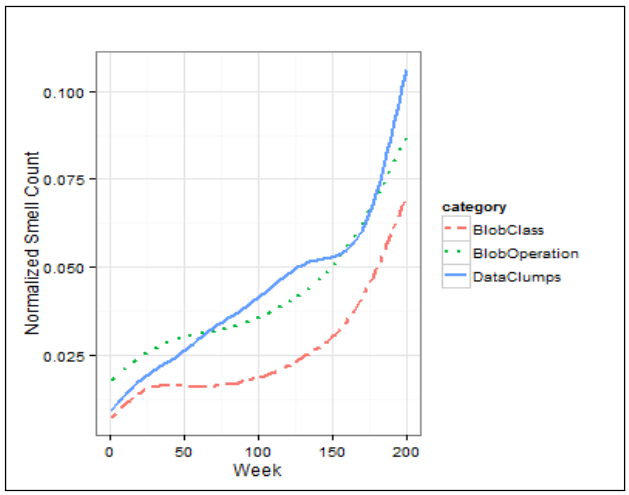

Fig. 3. Week-wise average project smellines compared to the smelliest project of bloater category

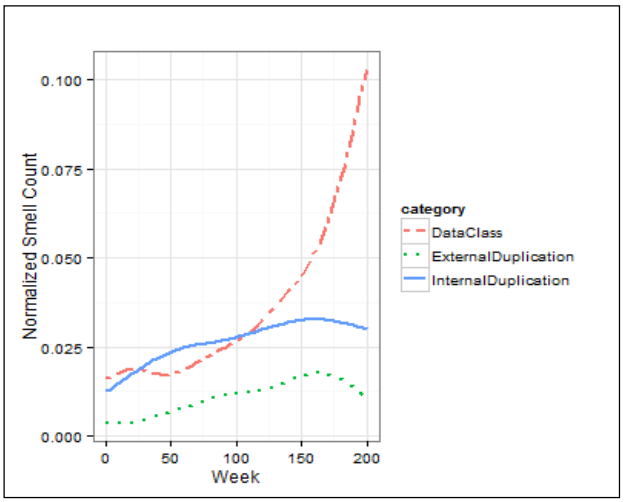

Fig. 4. Week-wise average project smellines compared to the smelliest project of Dispensables category

The other OO abuser smells – the Schizophrenic class, a code smell that captures the scenario where a class has two or more key abstractions, the Refused Parent Bequest code smell – a sign of inheritance relation problems between parent and subclass, and the Distorted Hierarchy – indicative of the inheritance hierarchy being too deep, just like the Internal and External Duplication in Dispensables category, Sibling Duplication – indicative of duplication between siblings in an inheritance hierarchy, display a different growth pattern, with most of them plateauing relatively quickly. The Encapsulator category had too little data to give any meaningful insights.

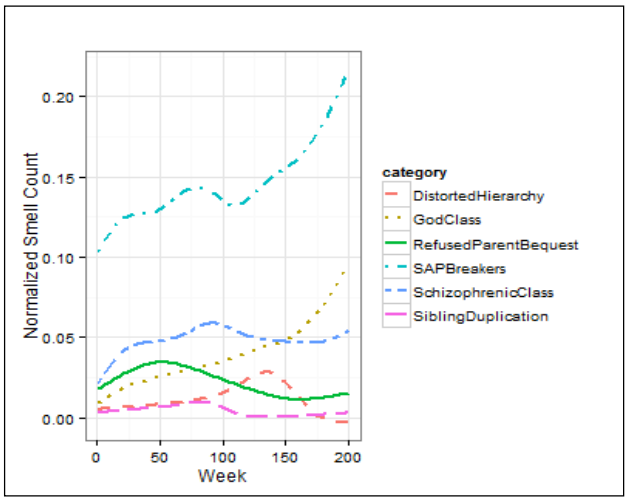

Fig. 5. Week-wise average project smellines compared to the smelliest project of OO abusers category

We also found that in the Other category all smells (Cyclic Dependencies, Feature Envy, Shotgun Surgery, Tradition Breaker and Unstable Dependencies) have a tendency to increase over time, and Intensive Coupling shows an oscillating behavior.

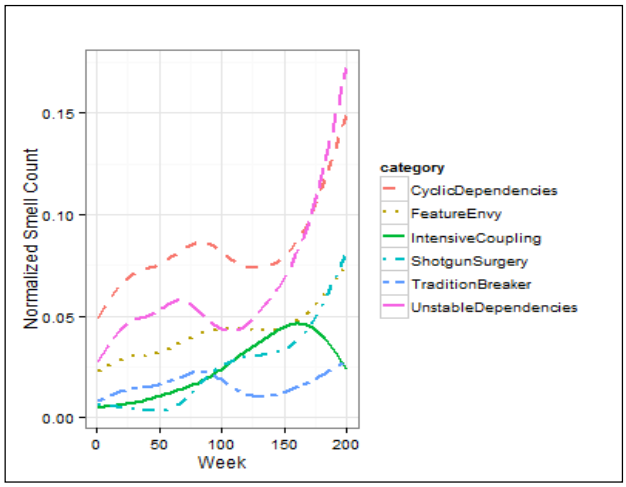

Fig. 6. Week-wise average project smellines compared to the smelliest project of Other category

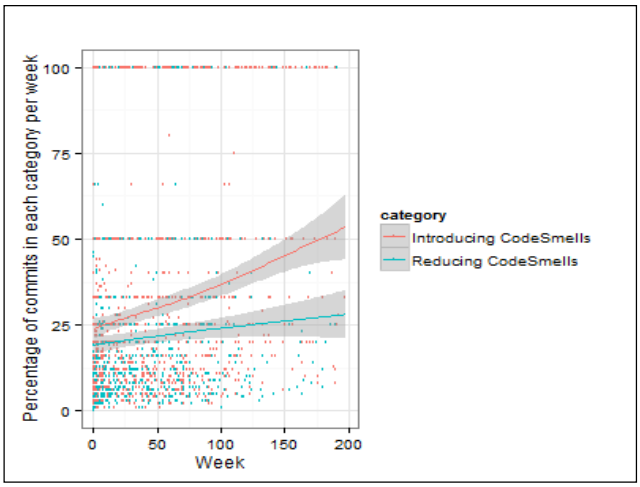

Fig. 7. Breakdown of commits introducing(trend line on top) and removing(trend line on bottom) smells

To gain a better understanding of how design issues evolve over time we classified all code commits into one of three categories; those that introduced at least one smell, those that removed at least one smell, and those that did not impact smells. We then calculated the percentage of commits that fell into each of these three categories over the life of the projects. Figure 7 shows that, as projects progress, the rate of smell introducing commits increases (the trend line on top). Smell reducing commits do increase over time, but not nearly as fast as the smell introducing commits (the trend line at the bottom). The grouping of dots around the 50 percent and 100 percent markers are formed from the many projects that in any given week only see a small number of commits. The shaded band is the 95% confidence interval, indicating that there is 95% confidence that the true regression line lies within the shaded region.

B. *Who takes care of smelly code?*

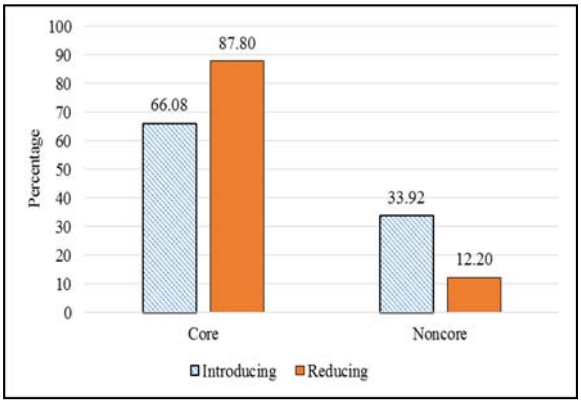

Fig. 8. Origin of commits introducing or reducing code smells

We found that less than 5% of commits removed smells, and that only 10% of the developers were responsible for those commits. Because we know that a typically small core development team is responsible for more than 80% of contributions in any open source project [34] we next needed to see if this group was also responsible for either the insertion or removal of code smells. For the purposes of our paper we defined the core contributors for each project as the top contributors who made 80% of the contributions in the project.

As expected, core contributors, being responsible for the bulk of contributions, both introduce more smells and remove more smells than non-core contributors. We do however see that core contributors appear to remove more smells than they introduce, whereas the inverse is true for non-core contributors. However, we found that there is no statistically significant difference in terms of smell reducing commits and introducing commits for the core developers (Welch two-sample t-test, t = 1.0733, df = 289, p-value = 0.284, Not Statistically Significant). The same was true for non-core contributors (Welch two-sample t-test, t = -0.9976, df = 180.74, p-value = 0.3198, Not Statistically Significant).

C. *Does better testing lead to less smelly code?*

For each week in a project's lifespan we calculated the number of test cases available, and the number of code smells in the code to check if there exists a correlation between them. We found that there is no statistically significant correlation between these two factors (Pearson Correlation Coefficient 0.4051681). Figure 9 shows the trend line found after plotting the normalized total count of test case and code smell count for each week, for all projects.

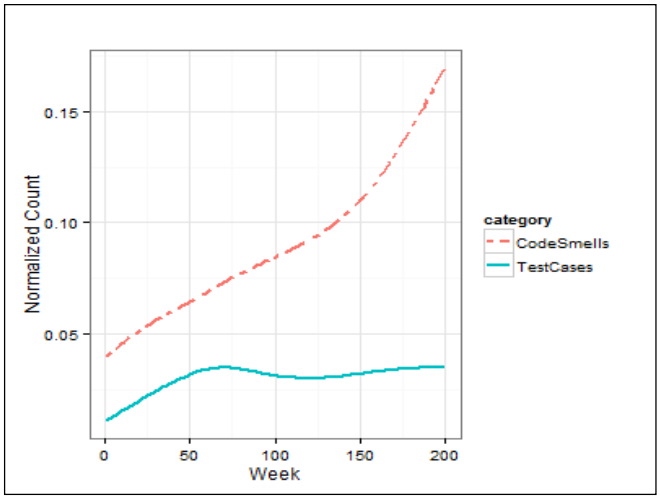

Fig. 9. Comparison of Week-wise normalized total smell and test case count

As test cases are not created equal, we also wanted to check whether there is a correlation between test coverage and smelliness of the project. We selected the last commit for each project and used the existing test suite for coverage analysis. Then for each code smell we checked whether there is any difference between the low (less than 30%) and high (more than 60%) coverage (measured using statement, branch, path and mutation coverage criteria) projects and number of code smells present in the project. We found that for External duplication (t = 2.166, df = 72, p-value = 0.03363, Statistically Significant) and Internal duplication (t = 2.4813, df = 72, p-value = 0.01543, Statistically Significant), low and high coverage groups have statistically significant difference in number of code smells present in the project.

D. Literature v. real-life

To answer our fourth research question we calculated the number of smells being introduced and removed by category. Our goal here was to determine if there was a good match between the practices and problems real programmers deal with, and the concerns of researchers. Figure 10 shows that, SAP Breakers, Data Class and Cyclic dependencies and Feature Envy were the most common smells, constituting almost 50% of the code smells being introduced and removed.

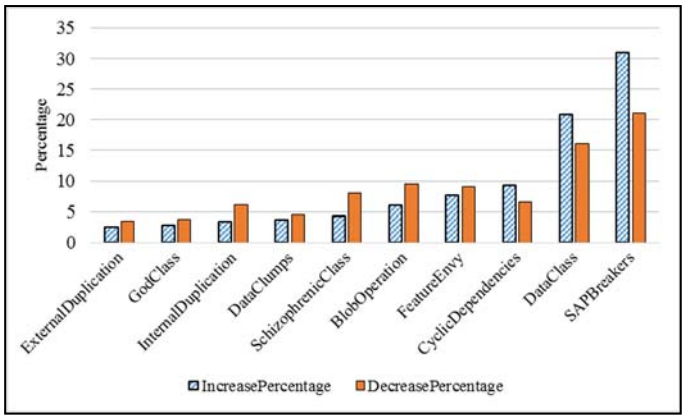

Fig. 10. Breakdown of smells introduced and removed

Next we looked to the research literature to identify which code smells receive most attention from researchers. We used the work of Zhang et al. [48] and Sjoberg et al. [43]. Zhang et al. performed a systematic literature review on code smells published in IEEE and six leading software engineering journals from January 2000 to June 2009 [48]. They identified 39 papers out of 319 papers that could answer the research question about which code smells receive most research. Sjoberg et al. expanded the analysis period from June 2009 to October 2011 and identified 10 additional papers. We ranked the smells based on percent of total smells and compared it against the ranking from the two survey papers. In Table VI we report the percent based ranking.

TABLE VI. COMPARISON OF RANKINGS BASED ON OUR ANALYSIS AND THE NUMBER OF RESEARCH PAPERS DEALING WITH A SMELL

Smell	From Projects		From Literature	
	Rank	*Freq.*	*Rank*	*Freq.*
Data Clumps	1	22.05	7	5
Data Class	2	17.34	4	11
Cyclic Dependencies	3	11.39	10	2
Blob Operation	4	8.02	5	8
Duplication	5	17.84	1	25
Feature Envy	6	5.47	2	13
SAP Breakers	7	4.79	8	5
God Class	8	3.41	9	4
Intensive Coupling	9	2.78	8	5
Schizophrenic Class	10	1.99	8	5
Blob Class	11	1.47	5	8
Unstable Dependencies	12	1.27	8	5
Tradition Breaker	13	1.00	8	5
Refused Parent Bequest	15	0.64	3	12
Message Chains	16	0.38	8	5
Shotgun Surgery	17	0.16	6	8
Distorted Hierarchy	18	0.00	8	5
UnnecessaryCoupling	19	0.00	8	5

V. DISCUSSION

During our analysis we found that the overall number of smells increase over the life of open source projects, as shown in Figure 2. This is not to say that issues of design or technical debt are not addressed over the life of the project, but as Figure 8 shows, it is simply a matter of new issues being introduced faster than old ones are resolved. More importantly, as is also evident from Figure 7, the pace of smell introduction accelerates over the life of the project. This could be an artifact of either projects adding new developers over time, thus having some of the initial core design knowledge watered down, or that as a project progresses and code builds up, it becomes increasingly difficult to unravel fundamental design revisions, or that an artifact of bad design decisions leading to further compromises. While further research would be needed to look into the nature of smells added and removed, we find it likely that there is an element of all three dynamics at play here.

Next we looked at the types of design smells being introduced, and found two general patters; those smells that more or less monotonically increase over the life of the project, and those that plateau at some point (see Figures 3, 4, 5 and 6). For the monotonically increasing smells, the analysis seems straightforward; some mistakes are made throughout the life of the project, or some compromises in design breed other similar compromises to be made later in the code. Most likely though, these represent self-reinforcing patterns with projects; we've used this structure or technique elsewhere in our code, therefore it is OK to do so again. As a code-base grows, this can have serious consequences, as previous research has shown a correlation between smelly code and maintainability and bugs [13, 25, 38]. All code smells in the Bloating category show a growing tendency. This category is associated with centralized control structures in object-oriented languages. Arisholm et al. found that novice developers perform better with centralized control styles [2], it is therefore possible that novice contributors are pushing for these changes, or that they are being introduced by regulars to make it easier for newcomers to participate. Yamashita et al. found that a considerable portion (32%) of developers did not know about code smells [46] and only (4%) used specific code smell detection tools with refactoring tools to remove smells. This could explain the monotonic growth; once a smell is introduced is unlikely to be identified or fixed.

The more interesting pattern is that of the smells that plateau, or even decrease after an initial spike, which included many of the smells in the OO Abusers category (Figure 5). These smells are indicative of poor and unsustainable designs. One possible interpretation of our findings is that they may represent acceptable compromises for prototyping and getting something out the door quickly, but that these design patterns likely present serious roadblocks to the future growth and success of the project. Developers facing such a situation are forced to refactor the code, and moreover, according to Yamashita et al., developers are more aware of this types of smells [46], which likely leads to increased refactoring. On the other hand, this pattern could just as easily be caused by projects making all the OO design decisions early in the projects lifecycle, with few if any such smells being added over time because no OO design changes take place.

The slow increase and then dipping pattern seen for internal and external duplication in the Dispensables category might be caused by developers working under constraints such as imposed deadlines or LOC-driven performance evaluations. Another reasonable circumstance where developers duplicate code is when they do not fully understand the problem or solution. Duplication becomes a safer way of modifying code rather than generalizing. As code-bases grow, it eventually becomes difficult to add new functionality, and developers are forced to refactor and remove duplication. Furthermore, developers are aware of code duplication and its consequences. Yamashita et al. found that duplicate code was the most mentioned code smell in their survey [46]. This can also explain why duplicate codes are refactored more often. We do not really have data to support this, and further research is needed to fully explain such trends.

We also found that some smells are essentially added and removed on a near constant basis. In the Dispensables category for instance, the Data Classes code smell was already found by Khomh et al. as one of the most change prone code smells [22]. Possible reasons for such behavior is that coders are either unaware of the perils of this design pattern, or that when coding they do not realize when they are violating such design practices (disconnect between theoretical knowledge and practical), or that they fall into the trap of thinking that doing this once won't make a difference.

This to us is a clear sign that we need to do a better job integrating smell analysis either into IDE environments or into repository tools, not so much to block developers from using undesirable patterns, but rather as a way of giving developers feedback so they can reflect on the availability of better design patterns and how bad design decisions accumulate over time. We found that developers are not aggressively fixing design issues, which can be explained by the findings of Yamashita et al. who found that a considerable portion (32%) of surveyed developers did not know about code smells [46]. It is therefore conceivable that a majority of developers don't actually know when they are making poor implementation decisions. It's also obvious that these developers are unware of the issues such as bugs [25, 38] and code maintainability problems [13] are associated with code smells.

When tools are used, these do not always provide great feedback for developers. Many of the tools we looked at give a large number of false positives, an inherit issue with any kind of static analysis tool [19]. Lack of visibility of the deduction rules and thresholds of the metrics and context awareness might be other reasons, as identified by Fontana et al. [12], why the developer community remains skeptical and uninterested in code smell analysis. Moreover, developers don't want to have their workflow disrupted by tools that do not integrate well into their development process [19]. Moreover the current tools do not always align with the problems projects struggle with. All these factors along with developer unawareness about smells helps to explain why developers don't use code smell detection tools and also helps to explain our observation why design issues build up over time. Further research should look at making these tools more accessible and relevant to real world programmers.

We did find that core developers introduce both more smells and smell fixes than non-core developers, not an unexpected finding given core developers predominance in the world of coding. What was interesting though was that core developers were no more likely to fix code smells (in proportion to the size of their contributions) than non-core developers. This was a surprise to us, as we expected core developers to have a better understanding of both the software's high-level design and of best coding practices. This turned out not to have a significant impact on the outcome of their coding. While difficult to interpret, this to us leads us to think that even among core contributors, understanding of high-level design tradeoffs and/or the time to refactor code may be in short supply.

We also found that the number of test cases does not show any correlation with the design quality of the project. Although the quality of test cases works as an indication of how well the system is tested, it doesn't give any indication about how bad design in the project is. Though this was expected, as test cases are written to identify bugs, not design issues, there was a possibility that testing could be part of a bigger refactoring and review process for code. Such activities would likely catch many of the code smells we were documenting in this study. We did not find evidence to support that testing indeed sparks or goes hand in hand with such review activities.

We found that most of the code smells that were ranked high by our analysis were not highly ranked in the research literature (with the exception of duplication). While understandable to a certain extent, common problems are not always interesting problems, this shows a divide between the world of theoreticians and practitioners which may further drive the later away from the tools and practices we in academia try to promote. More attention should be paid towards analyzing the impact of high frequency real world smells and making the tools more efficient in identifying these. Alignment between the research and real world smell is necessary for making code smell analysis acceptable to everyone.

VI. THREATS TO VALIDITY

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Our samples have been from a single source - Github. This may be a source of bias, and our findings may be limited to open source programs from Github. Github's selection mechanisms favoring projects based on some unknown criteria may be another source of error. However, we believe that the large number of projects sampled more than adequately addresses this concern.

During our analysis we calculated the code smells after each commit and categorized the commits into three categories. There is always a chance that smells get introduced over multiple commits. Categorizing individual commits into these three categories poses the risk that, commits that actually contributed a major portion towards introducing the smell but

did not actually led to crossing the threshold value will not be identified as a smell introducing commit. Though this could add some noise to our data, overall the risk is negligible; eventually this threshold will be crossed and it will impact the average. Over a sample of 220 projects, chances are that these slight variations will have a relatively minor effect.

For our analysis we had to categorize contributors into core and non-core contributors, for this categorization we had to set a threshold based on the number of commits for each contributor. It might be the case that some of the contributors that were categorized as non-core contributor based on our criteria were actually core contributors focusing on large contributions rather than frequent contributions, or simply focusing on architecture and high-level design (high value contributions).

The smell detection tool we used uses a code metric and threshold-based detection strategy. These metrics and thresholds have been evaluated for their efficacy in a number of previous studies. However, it has not been evaluated whether their use is appropriate in all contexts. Hence, the precise metrics and thresholds that it is appropriate to use may vary depending on the context. We did not evaluate their efficacy for use in our study. Hence, it may well be that different metrics and values would have been more appropriate. Moreover the tool we used uses static code analysis to identify smells and research shows that code smells that are "intrinsically historical" such as Divergent Change, Shotgun Surgery and Parallel Inheritance are difficult to detect by just exploiting static source code analysis [11]. So the number occurrence of such "intrinsically historical" smells should be different when historical information based smell detection technique is used.

VII. Conclusion and Future Work

In this paper, we have tried to develop an understanding of how design issues build up in an open source project over time, and whether this build-up can be effectively mitigated or controlled. We found strong evidence that design issues build up over time. As the project grows older and bigger, design issue are fixed less and as a consequence build up.

As expected, core contributors, being responsible for the bulk of code contributions, were also responsible for the bulk of smells being introduced. Though they are also responsible for removing most of the smells that are removed, they were not significantly better at doing so than non-core contributors. This was surprising given core contributors' deeper understanding of the project, and opportunity to remove smells through significant refactoring. This leads us to suspect that rather than due to a concerted effort, a large number of code smells must be removed accidentally or as part of ad-hoc code review. Further qualitative studies should provide insight and conclusive reasons behind the identified patterns.

In line with previous observations, we also found that a project's testedness is not an indicator of design quality. Though this is not unexpected, it is another indicator that developers are paying more attention to removing and identifying bugs rather than refactoring. Current tool-sets and techniques are not tailored towards identifying design issues however, which may in part explain why these problems are difficult to tackle for projects.

We also found that there is a mismatch between many of the most frequently occurring code smells and the most popular code smells in the research literature, with many of the common problems encountered in real life seeing relatively little research. More focus should be given to the code smells that occur frequently if we want to tackle the issue of technical debt in real-world projects.

In our analysis, we didn't consider factors such as the number of contributors or the size of the project, which have been proven to contribute towards design issues. It would be interesting to do a large scale longitudinal study where all these factors are considered in the analysis to try to identify the relationship between these factors and code smells, and see if the resulting models are different from the models identified by analyzing single snapshots of the projects.

Finally, we conclude by mentioning that our finding along with the findings of other researchers provides evidence for the theory that developers in general are not very conscious about fixing design issues. The researcher community should try to make this easier by improving tools, including a focus on the more common code smells, which are sometimes ignored in the research literature.

Acknowledgment

Withheld for review.

References

[1] Apache Software Foundation. Apache maven project. http://maven.apache.org

[2] Arisholm, E., & Sjoberg, D. I. (2004). "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software". In *IEEE Transactions on Software Engineering*, (pp.521-534).

[3] Baeza-Yates, R., & Ribeiro-Neto, B. (1999). Modern information retrieval (Vol. 463). New York: ACM press.

[4] Booch, G. (2006). "Object Oriented Analysis & Design with Application". Pearson Education India.

[5] Chatzigeorgiou, A., & Manakos, A. (2010, September). "Investigating the evolution of bad smells in object-oriented code". In *Proceedings of the Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, (pp. 106-115).

[6] Cunningham, W. (1992, December). "The WyCash portfolio management system". In *ACM SIGPLAN OOPS Messenger, Vol. 4, No. 2*, (pp. 29-30).

[7] Deligiannis, I.; Shepperd, M.; Roumeliotis, M.; Stamelos, I. (2004). "An empirical investigation of an object-oriented design heuristic for maintainability". In *The Journal of Systems and Software 72 (2),* (pp.129-143).

[8] Deligiannis, Ignatios; Stamelos, Ioannis; Angelis, Lefteris; Roumeliotis, Manos; Shepperd, Martin (2003, February). "A controlled experiment investigation of an object oriented design heuristic for maintainability" In *Journal of Systems and Software, Vol.65, No .2,* (pp.127-139).

[9] De Souza, L. B. L., & Maia, M. D. A. (2013, May). Do software categories impact coupling metrics?. In Proceedings of the 10th Working Conference on Mining Software Repositories (pp. 217-220). IEEE Press.

[10] Du Bois, B., Demeyer, S., Verelst, J., Mens, T., & Temmerman, M. (2006). "Does god class decomposition affect comprehensibility?" In

Proceedings of the IASTED Conf. on Software Engineering (pp. 346-355).

[11] F. Palomba, G. Bavota, M. Di Penta, R.Oliveto, A. De Lucia, D. Poshyvanyk.(2013) "Detecting Bad Smells in Source Code Using Change History Information". In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*.

[12] Fontana, F. A., Mariani, E., Morniroli, A., Sormani, R., & Tonello, A. (2011). "An experience report on using code smells detection tools". In *Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 450-457).

[13] Fowler, M. (2002). "Refactoring: improving the design of existing code". Pearson Education India.

[14] Gopinath, R., Jensen, C., & Groce, A. (2014, May). "Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (pp. 72-82).*

[15] H. Coles. "Pit mutation testing". http://pittest.org/

[16] InFusion, http://www.intooitus.com/inFusion.html.

[17] Izurieta, C., & Bieman, J. M. (2007, September). How software designs decay: A pilot study of pattern evolution. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (pp. 449-451). IEEE.

[18] Izurieta, C., & Bieman, J. M. (2008, April). Testing consequences of grime buildup in object oriented design patterns. In *Software Testing, Verification, and Validation, 2008 1st International Conference on* (pp. 171-179). IEEE.

[19] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). "Why don't software developers use static analysis tools to find bugs?". In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 672- 681).

[20] Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". In *Journal of Software Maintenance and Evolution: Research and Practice, Vol.19, No. 2*, (pp. 77-131).

[21] Kagdi, H., Gethers, M., Poshyvanyk, D., & Collard, M. L. (2010, October). "Blending conceptual and evolutionary couplings to support change impact analysis in source code". In *17th Working Conference on Reverse Engineering,* (pp. 119-128).

[22] Khomh, F., Di Penta, M., & Gueheneuc, Y. (2009, October). "An exploratory study of the impact of code smells on software change-proneness". In *16th Working Conference on Reverse Engineering*, (pp. 75-84).

[23] 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.

[24] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997, November). "Metrics and laws of software evolution-the nineties view". In *Proceedings of Fourth International Software Metrics Symposium*, (pp. 20-32).

[25] Li, W., & Shatnawi, R. (2007)." An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". In *Journal of Systems and Software, Vol.80, No.7*, (pp.1120-1128)

[26] M. Doliner and Others. Cobertura - a code coverage utility for java. http://cobertura.github.io/cobertura

[27] Mäntylä, M. (2003). "Bad smells in software-a taxonomy and an empirical study". Helsinki University of Technology.

[28] Mäntylä, M., Vanhanen, J., & Lassenius, C. (2003, September). A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003). Amsterdam, The Netherlands.* (pp 381-384).

[29] Marinescu, C., Marinescu, R., Mihancea, P. F., & Wettel, R. (2005). "iPlasma: An integrated platform for quality assessment of object-oriented design". In ICSM (Industrial and Tool Volume).

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

[31] Marinescu, R. (2004, September). "Detection strategies: Metrics-based rules for detecting design flaws". In *Proceedings of 20th IEEE International Conference on Software Maintenance*.(pp. 350-359).

[32] Marticorena, R., López, C., & Crespo, Y. (2006). "Extending a taxonomy of bad code smells with metrics". In *Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR)*, (pp. 6).

[33] Martin, R. C. (2003). "Agile software development: principles, patterns, and practices". Prentice Hall PTR.

[34] Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(3), 309-346.

[35] Moha, N., Rezgui, J., Guéhéneuc, Y. G., Valtchev, P., & El Boussaidi, G. (2008). "Using FCA to suggest refactorings to correct design defects". In *Concept Lattices and Their Applications* (pp. 269-275). Springer Berlin Heidelberg.

[36] Murphy-Hill, E., & Black, A. P. (2010). "An interactive ambient visualization for code smells". In *Proceedings of 5th international symposium on Software visualization* (pp. 5-14).

[37] Olbrich, S. M., Cruzes, D. S., & Sjoberg, 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 *Proceedings of ICSM,* (pp. 1-10).

[38] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). "The evolution and impact of code smells: A case study of two open source systems". In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement* (pp. 390-400).

[39] Oliva, G. A., Steinmacher, I., Wiese, I., & Gerosa, M. A. (2013, August). "What can commit metadata tell us about design degradation?". In *Proceedings of the 2013 International Workshop on Principles of Software Evolution* (pp. 18-27).

[40] R. Liesenfeld. JMockit - A developer testing toolkit for Java. http://code.google.com/p/jmockit/

[41] Riel, A. J. (1996). "Object-oriented design heuristics", *Vol.* 335. Reading: Addison-Wesley.

[42] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010, September). "Building empirical support for automated code smell detection". In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement* (p. 8). ACM.

[43] Sjoberg, D. I., Yamashita, A., Anda, B. C. D., Mockus, A., & Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, *39*(8), 1144-1156.

[44] Smith, C. U., & Williams, L. G. (2000). "Software performance anti-patterns." In *Proceedings of the Workshop on Software and Performance* (pp. 127-136).

[45] V. Roubtsov and Others. Emma - a free java code coverage tool. http://emma.sourceforge.net/

[46] Yamashita, A., & Moonen, L. (2013, October). "Do developers care about code smells? An exploratory survey". In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, (pp. 242-251).

[47] Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. (2011, May). "Investigating the impact of design debt on software quality". In *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 17-23).I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[48] Zhang, M., Hall, T., & Baddoo, N. (2011). Code bad smells: a review of current knowledge. Journal of Software Maintenance and Evolution: research and practice, 23(3), 179-202